Overflowing the stack on Linux x86

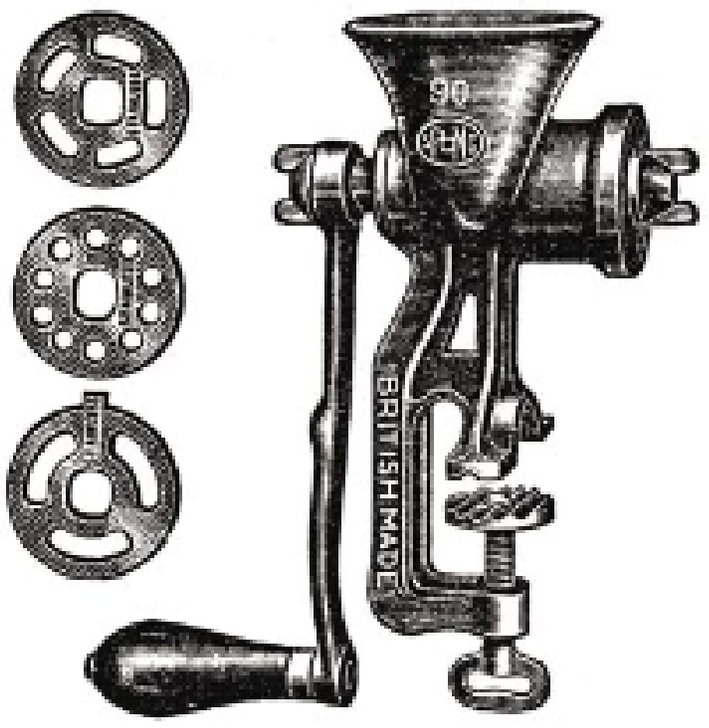
Piotr Sobolewski

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).

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

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

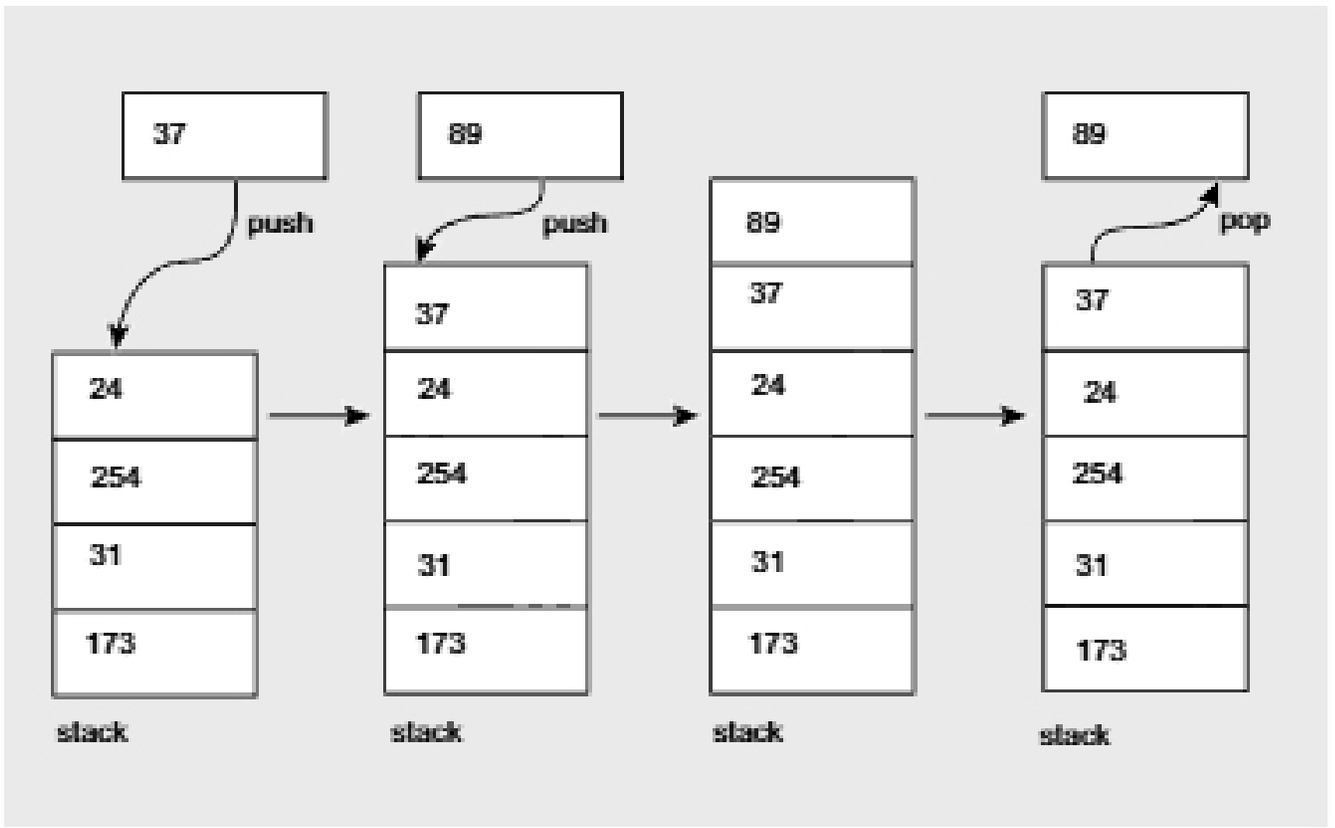


Figure 1. *Basic stack operation is pushing elements onto its top and popping them off the top. The figure illustrates pushing the number 37 onto the stack first, then pushing the number 89. If a number is then popped off the stack, it is the one that was last pushed, ie. 89. To get the number 37, another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

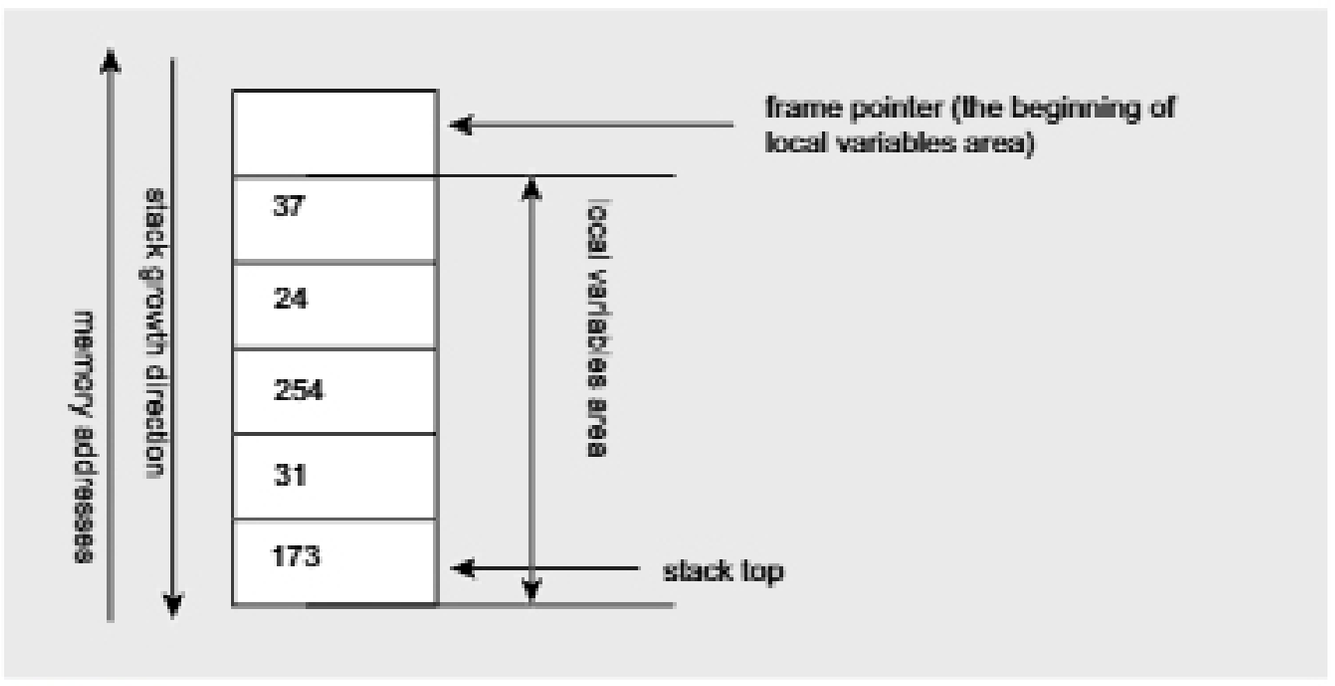


Figure 2. *In Linux on x86, the stack grows downwards (see description in text)*

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.