A Look at the Buffer-Overflow Hack

Mr. Harari dissects the buffer-overflow hack, thereby giving us the necessary information to avoid this problem.

by Eddie Harari

The best system administrator is not always enough to take care of site security. Sometimes a nice program such as mount can be exploited by a user to gain a higher system permission or remote access to an unauthorized location on the World Wide Web.

This article explains the logic behind a popular hack to exploit a program's code so it executes different code then was intended. This hack is known as the buffer-overflow hack and can be used to exploit a program with suid set to gain better permissions on a Linux machine—sometimes even root or remote access. (The examples are taken from `aleph-one` with his permission and have been somewhat modified by me.)

![Figure 1. Virtual Memory Layout](image)

First, let's have a look at Figure 1 and see how a process organizes its virtual memory. The TEXT area is where the actual code of the program resides. The DATA area is where the initialized and uninitialized data of the program resides.

The STACK area is a dynamic area which becomes bigger as data is pushed into it and smaller as data is popped from it. It is called a stack because it works in the LIFO way (last in, first out). The stack is used to hold temporary data for the process and helps the processor in its implementation of high-level functional programming. To understand exactly how the processor makes use of the stack, look at the following example:

```c
void func(int a, int b) { /* This function does nothing */
    main() { int num1; int num2;  func(num1,num2);  printf("This is the next instruction after ");
    }
}
```

The instructions of the `main` function are executed until the processor needs to "break" the normal flow of the program and go to the `func` instructions. When this step of "jumping" to `func` is executed, the parameters to `func`, `num1` and `num2` are transferred with the help of the stack. That is, they are pushed to the stack, and `func` can pop them from the stack and use them. Immediately after pushing these values on the stack, main should push the address to which `func` will return on completion. (In our example, this is the address of the `printf` instruction.) When `func` is finished, it knows to read this return address from the stack and go back to the "normal" flow of the program.

One other value on the stack is called a frame-pointer, since the processor refers to values on the stack by their offset from the stack pointer (SP). Whenever the SP value changes, the processor saves the current value on the stack. (The Intel does not have a dedicated frame pointer (FP), so it does it with the help of the EBP register.) The frame pointer is pushed to the stack following the return address.

To clarify this, let's look at another example:

```c
void func(int a, int b) {  int *p;  main() { int num; num = 0;  func(num);  num = 1;  printf("num is now %d \n",num);  }
}
```

Let's compile it with the -S option to get assembly output using this gcc command:

```bash
gcc -S -o ex2.S ex2.c
```

We see that main's code is actually:

```
main:  pushl %ebp  movl %esp,%ebp    /* Save the SP before changing    */
          pushl %eax    /* Set up the function signature */
          subl $4,%esp  /* SP should subtract 4 so it points
to the function */
```

The main code pushes the arguments for `func`, then calls it. The call instruction puts the return address on the stack, then moves on to the `func` code. `func` puts the four-byte frame pointer immediately following the return address, then pushes the `p` pointer onto the stack. Thus, if we dump the stack's status now, we get the structure shown in Figure 2.
Since this is not the Linux "hacking HOWTO", I will not go into details on these three stages.

The actual buffer-overflow hack works like this:

- Find code with overflow potential.
- Put the code to be executed in the buffer, i.e., on the stack.
- Point the return address to the same code you have just put on the stack.

Since this is not the Linux "hacking HOWTO", I will not go into details on these three stages.
The first stage is very easy, especially in a Linux system, since a huge amount of open-source code applications are available for Linux. Some of these applications are in use on almost every Linux system. Good examples of such programs were mount and some early versions of innd. mount did not check the length of the command-line arguments the user entered and its permissions set to 4555. innd did not check all of the news message headers, so by sending a specific header, a user could get a remote shell on the server.

The second stage has two parts. The first one is to find how to represent the code to be executed; this can be done using a simple disassembler. The second part depends on where the program reads the buffer: in some cases, a mail header; in others, an environment variable whose length goes unchecked; in still others, some alternate means.

The third stage is not so simple, as one cannot know the exact address of the code to be executed. Basically, it is done by guessing the address until the correct address is found. Several ways can be used to make this guessing more efficient; thus, after only a few guesses, we can specify the right address and the code gets executed.

**Conclusions**

The fact that an application is used all over the Web does not mean it is secure, so take care when installing a new application on your machine. In fact, WWW applications are more likely to be searched deeply for security holes by crackers with bad intent. System administrators should read the security newsgroup and related web pages in order to keep applications known to have security holes off the system and to upgrade them when patches become available. Application programmers should take care to write tight code containing proper checks for array and variable lengths in order to foil this type of hack.

Finally, I would like to briefly mention three other things. One, a kernel patch is available that makes the stack memory area a non-executable one. I have never tested it, since applications do exist which count on the fact that the stack is executable, and these applications will most likely have problems with this patched kernel. Two, a special mode to the Intel processor is available that has the stack grow from the lower memory addresses to the higher memory addresses, thus making a buffer overflow almost impossible. Three, a set of libraries available on some systems helps the programmer write code with no such errors. All the programmer has to do is tell the library functions the assumptions about a variable and these functions will verify that the variable meets the specified criteria.

---

**Eddie Harari** works for Sela Systems in Israel as a lecturer and a consultant. He is currently involved in networking security projects and can be reached via e-mail at eddie@sela.co.il.