# The Fortify Source Code Analyzer

*Sourceanalyzer* is a program that analyzes other programs for vulnerabilities. This is a very brief explanation of its output.

## The Program

This C program copies a string into buffer and quits. It's clearly a demonstration program!

```
1  #include <strings.h>
2  #include <stdio.h>
3
4  #define MAX_SIZE 128
5
6  void doMemCpy(char* buf, char* in, int chars) {
7    memcpy(buf, in, chars);
8  }
9
10 int main() {
11   char buf[64];
12   char in[MAX_SIZE];
13   int bytes;
14
15   printf("Enter buffer contents:\n");
16   read(0, in, MAX_SIZE-1);
17   printf("Bytes to copy:\n");
18   scanf("%d", &bytes);
19
20   doMemCpy(buf, in, bytes);
21
22   return(0);
23 }
```

It has a couple of security problems, were it to be installed setuid and set so anyone could run it. Can you find them before going any further?

## The Analysis

We run the *sourceanalyzer* program over this program, as follows:

```
/opt/HP_Fortify/HP_Fortify_SCA_and_Apps_3.80/bin/sourceanalyzer gcc stackbuffer.c
```

You can set your search path to look in the directory `/opt/HP_Fortify/HP_Fortify_SCA_and_Apps_3.80/bin` to avoid typing the full path name, and from here on we will assume you did this.

Here is the output:

```
[/home/bishop/ecs153/fortify]

[D10CB5094B2FB1C2C6AC8AD7CADECA30 : low : Unchecked Return Value : semantic ]
stackbuffer.c(16) : read()

[4940AB43F66960894026F18AF2032001 : high : Buffer Overflow : dataflow ]
stackbuffer.c(7) :  ->memcpy(2)
    stackbuffer.c(20) :  ->doMemCpy(2)
    stackbuffer.c(18) : <- scanf(1)
```

The analyzer has identified two poor programming practices that may lead to security problems.

The function read at line 16 returns a value that is not checked. The danger from this is low. It is a semantic problem; that is, it results from the semantics of read returning a value.

On line 18 of the program, the function *scanf* reads something into its second argument (the first argument in a parameter list is argument 0, so argument 1 is the second one). The arrow "<-" means "input". This quantity is than passed to the function *doMemCpy* as argument 2, the call occurring on line 20. The arrow "->" means "passed to". This argument is then passed to the function memcpy on line 7, as the third argument. This means that an input number controls how many bytes *memcpy* copies, and if set incorrectly could cause a buffer overflow.

## Potential Exploits

Given these problems, let's see how exploits might work.

1. Unchecked Return Value
   This is marked "low", so it will be difficult to find a security flaw from it. Basically, it requires that the read system call on line 16 of *stackbuffer.c* either fail (hence returning $-1$) or return fewer characters than typed. In that case, the number entered will be larger than the number of characters read, which could cause a problem. The word "semantic" means that the irregularity is from the semantics of the call (that is, no return value used).

2. Buffer Overflow
   This is marked "high" because the source code analyzer asserts it is easy to exploit. This indicates that user input enters the program through the *scanf* call on line 18, which reads data into argument 1. (Arguments are 0-indexed, so argument 1 is the second argument to *scanf*, which is `&bytes`.) This data is then passed as argument 2 to *doMemCpy*(), which in turn sends the data to argument 2 of *memcpy*(). This allows a user to cause an arbitrary amount of data to be written to the 64-byte buffer `buf`, potentially overflowing that buffer.

## More Complex Programs

Some (most) programs are much more complex, and require a *Makefile*. The source code analyzer can handle these, but the procedure is a bit more complicated.

As an example, let's say we're in a directory with a complex program that is compiled using a *Makefile*. You need to create a *build*, and then analyze that. So, first type

```
sourceanalyzer -b mybuild make
```

and add any flags or targets after the "make". This compiles the program, and while doing so creates a build named *mybuild*. You then have to analyze the build, as follows:

```
sourceanalyzer -b mybuild -scan
```

The argument after the "-b" option must be the same as in the previous command.