Uninitialized Values

When we execute code by hand, we put a '?' in a box if that location has not been initialized. Similarly, when we execute by hand, we consider it an error to use an uninitialized value. It is indicative of a problem in our program, and we have no idea what will actually happen when we run our code.

However, when you run your program, there is always *some* value there, we just do not know what it is. Therefore, when we run our program, we will get a value (it is just whatever happened to be in that memory location before), and there is not any checking that the value is initialized. We will not know that we have used an uninitialized value, and if we get unlucky, our program will print the correct output anyways.

Use of uninitialized values are a great example of why "silent" bugs are dangerous and need to be fixed. We might think our program is right when we test and debug it because the value in that location is coincidentally correct. However, some change in the execution of the program can cause that "luck" to end, resulting in the program exhibiting an error.

The change that disrupts your "luck" could be due to a variety of circumstances. For example, changing the input to the program could cause a different execution path leading up to the use of the uninitialized value, thus causing it to have a different value in a variety of different possible ways. Another possibility is if you compile your code with optimizations versus for debugging. When the compiler optimizes your code, it may place different variables in different locations relative to the debugging version. Such a bug can therefore lead to the annoying situation where the debug version of your program appears to work, but the optimized version does not (even on the same inputs). We have even seen students experience problems with differences between the values read for uninitialized locations between redirecting the output to a file and printing it directly to the terminal.

Valgrind's Memcheck tool explicitly tracks the validity of every bit in the program, and can tell us about the use of uninitialized values. By default, Memcheck will tell you when you use of an uninitialized value; however, these errors are limited to certain uses. If **x** is uninitialized, and you do **y = x**, Memcheck will not report an error, but rather just note that **y** now holds an uninitialized value. In fact, even if we compute **y = x + 3**, we will still not get an error immediately.

However, if we use an uninitialized location (or one holding a value computed from an uninitialized location) in certain ways that Memcheck considers to affect the behavior of the program, it will give us an error. One such case is when the control flow of the program depends on the uninitialized value—when it appears in the conditional expression of an if statement, or a loop, or in the selection expression of a **switch** statement. In such a situation, Memcheck will produce the error:

```
1
```

```
Conditional jump or move depends on uninitialized value(s)
```

and produce a call-stack trace showing where that use occurred. For example, if we write the function:

```
1
2
3
4
5
void f(int x) {
  int y;
  int z = x + y;
  printf("%d\n", z);
}
```

Valgrind's Memcheck will report the following error (this code is inside of uninit.c, which has other lines before and after those shown above):

```
1
2
3
4
5
==12241== Conditional jump or move depends on uninitialised value(s)
==12241==    at 0x4E8158E: vfprintf (vfprintf.c:1660)
==12241==    by 0x4E8B498: printf (printf.c:33)
```

```
==12241==    by 0x400556: f (uninit.c:7)

==12241==    by 0x400580: main (uninit.c:15)
```

This error indicates that the uninitialized value was used inside of **vfprintf**, which was called by **printf**, which was called by **f**(on line 7), which was called by **main** (on line 15). We may be able to fix the program directly by observing the call to **printf** on line 7 and seeing what we did wrong.

However, if we do not see the problem right off (which could be likely if the uninitialized value has been passed through a variety of function parameters and data structures before Memcheck reports the error), we need more help from Memcheck. In fact, when Memcheck reports such errors, it will helpfully suggest the option we need to get more information from it at the end of its output:

```
1

==12241== Use --track-
origins=yes to see where uninitialised values come from
```

If we run Valgrind again passing in this option (**valgrind --track-origins=yes ./myProgram**), it will report where the uninitialized value was created when it reports the error:

```
1

2

3

4

5

6

7

==12260== Conditional jump or move depends on uninitialised value(s)

==12260==    at 0x4E8158E: vfprintf (vfprintf.c:1660)
```

```
==12260==    by 0x4E8B498: printf (printf.c:33)
==12260==    by 0x400556: f (uninit.c:7)
==12260==    by 0x400580: main (uninit.c:15)
==12260==  Uninitialised value was created by a stack allocation
==12260==    at 0x40052D: f (uninit.c:4)
```

We can now see that the value was created by stack allocation (meaning allocating a frame for a function), and we have the particular line of code that caused that creation (**int y;** inside of **f**).

There are other cases where a use of an uninitialized value will result in a message such as:

```
1
==12235== Use of uninitialised value of size 8
```

Here Memcheck is telling us that we used an uninitialized value in a way it considered problematic, and that the value we used was 8 bytes in size (that is, how many bytes of memory it was accessing). If our uninitialized value is passed to a system call, we will get an error message that looks like this:

```
1
==12362== Syscall param write(fd) contains uninitialised byte(s)
```

All of these indicate the same fundamental problem: we have a value that we did not initialize. We need to find it (probably by using **--track-origins=yes**) and properly initialize it.