# You still need the "safe" functions even if you check string lengths ahead of time

devblogs.microsoft.com/oldnewthing/20120628-00

June 28, 2012

Raymond Chen

Commenter POKE53280,0 claims, “If one validates parameters before using string functions (which quality programmers should do), the ‘safe’ functions have no reason to exist.”

Consider the following function:

```
int SomeFunction(const char *s)
{
  char buffer[256];
  if (strlen(s) ≥ 256) return ERR;
  strcpy(buffer, s);
  ...
}
```

What could possibly go wrong? You check the length of the string, and if it doesn’t fit in the buffer, then you reject it. Therefore, you claim, the `strcpy` is safe.

What could possibly go wrong is that the length of the string can change between the time you check it and the time you use it.

```
char attack[512] = "special string designed to trigger a "
                   "buffer overflow and attack your machine. [...]";
void Thread1()
{
 char c = attack[256];
 while (true) {
  attack[256] ^= c;
 }
}
void Thread2()
{
 while (true) {
  SomeFunction(attack);
 }
}
```

The first thread changes the length of the string rapidly between 255 and 511, between a string that passes validation and a string that doesn't, and more specifically between a string that passes validation and a string that, if it snuck through validation, would pwn the machine.

The second thread keeps handing this string to `SomeFunction`. Eventually, the following race condition will be hit:

- Thread 1 changes the length to 255.
- Thread 2 checks the length and when it reaches attack[256], it reads zero and concludes that the string length is less than 256.
- Thread 1 changes the length to 511.
- Thread 2 copies the string and when it reaches attack[256], it reads nonzero and keeps copying, thereby overflowing its buffer.

Oops, you just fell victim to the Time-of-check-to-time-of-use attack (commonly abbreviated TOCTTOU).

Now, the code above as-written is not technically a vulnerability because you haven't crossed a security boundary. The attack code and the vulnerable code are running under the same security context. To make this a true vulnerability, you need to have the attack code running in a lower security context from the vulnerable code. For example, if the threads were running user-mode code and `SomeFunction` is a kernel-mode function, then you have a real vulnerability. Of course, if `SomeFunction` were at the boundary between user-mode and kernel-mode, then it has other things it needs to do, like verify that the memory is in fact readable by the process.

A more interesting case where you cross a security boundary is if the two threads are running code driven from an untrusted source; for example, they might be threads in a script interpreter, and the toggling of `attack[256]` is being done by a function on a Web page.

```
// this code is in some imaginary scripting language
var attack = new string("...");
procedure Thread1()
{
 var c = attack[256];
 while (true) attack[256] ^= c;
}
handler OnClick()
{
 new backgroundTask(Thread1);
 while (true) foo(attack);
}
```

When the user clicks on the button, the script interpret creates a background thread and starts toggling the length of the string under the instructions of the script. Meanwhile, the main thread calls `foo` repeatedly. And suppose the interpreter's implementation of `foo` goes like this:

```
void interpret_foo(const function_args& args)
{
 if (args.GetLength() != 1) wna("foo");
 if (args.GetArgType(0) != V_STRING) wta("foo", 0, V_STRING);
 char *s = args.PinArgString(0);
 SomeFunction(s);
 args.ReleasePin(0);
}
```

The script interpreter has kindly converted the script code into the equivalent native code, and now you have a problem. Assuming the user doesn't get impatient and click "Stop script", the script will eventually hit the race condition and cause a buffer overflow in `SomeFunction`.

And then you get to scramble a security hotfix.

Raymond Chen

**Follow**