Code Overload

Gerard J. Holzmann

I spend much of my time analyzing code. In most cases that code was written by someone else, but it could just as easily have been written by an evil earlier version of me from many years ago. Today, any non-trivial code base is typically hundreds of thousands of lines of code, and for large companies it often reaches into the mega-millions. This makes the code review process feel like you're exploring caves with myriads of little passages leading nowhere in particular.

The most readily available tool that we have in our arsenal for finding flaws is of course the compiler. At JPL (NASA's Jet Propulsion Laboratory) we adopted the rule that all code, and not just the safety critical code, must be compileable with all warnings turned on and without generating warnings. If a new or rewritten piece of code triggers warnings when first compiled, those warnings are now easily spotted and addressed. In this way it is impossible to miss the potentially critical issues that could otherwise hide in a sea of otherwise benign warnings that a good compiler will flag.

Not every organization follows this rule though, which means that if you attempt to compile a large enough code base you may be overwhelmed with thousands of warnings, with a few zingers mixed in that you are unlikely to spot. If you run a static source code analysis tool, and you don't have the patience to carefully tune it, you could hit the same problem of being overwhelmed with an avalanche of output – most of which you are likely to ignore.

A typical trigger for an in-depth code review is that you just learned about a dangerous issue, either by reading about it on a website, or by stumbling upon it in your own code base. The immediate question then is: where, or where else, does this issue appear in the code we are responsible for? Was it flagged already in the thousands of lines of output from the compiler or from a code analyzer? Good luck finding it. What we need in cases like these is a simple and fast way to check a large code base for suspect patterns. Since I don't know of any tools that can do this well, I built my own a couple of years ago [1]. The tool is freely available at GitHub (https://github.com/nimble-code/Cobra). I'll say a little bit more about what you can do with a tool like this.

Suspicious Patterns

As a first example, I'll use a security vulnerability, one of many such vulnerabilities that were recently discovered by researchers at Semmle (see https://lgtm.com/security). The vulnerability in question is a potential buffer overflow in the code for rsyslog/librelp version 1.2.14 that can open a pathway for code injection and remote code execution. The Semmle researchers used a sophisticated query language to find the flaw, but the same issue can also be found, and more quickly, with just a few keystrokes using a pattern-based search tool.

The key issue is related to the use of library routine **`snprintf`**. Most of us know that using the basic routine **`sprintf`** for writing formatted text into a buffer without a preset bound is unsafe and risks overwriting the target buffer. The typical remedy, other than to carefully check the sizes of the arguments of the call, is to use the safer function **`snprintf`**, which limits the maximum number of characters that can be written to a user-set bound.

There is however a quirk here that can catch you by surprise. When a successful call to **sprintf** completes, it returns the number of bytes that were written. The routine **snprintf** also returns a number, but this time it is not the number of bytes actually written, but the number of bytes that **sprintf** would have written. Although this seems like an odd choice, it does allow us to check if the resulting string was unintentionally truncated at the bound that was set in the call.

Consider, for example, the following code fragment:

```
char s1[16], s2[16];
int  n1, n2;

n1 = sprintf(s1, "hello world");
n2 = snprintf(s2, 8, "hello world");

printf("%d '%s'\n", n1, s1);
printf("%d '%s'\n", n2, s2);
```

The output is as follows:

```
11 'hello world'
11 'hello w'
```

First note that while the return value of the first call was 11, the function actually wrote 12 bytes into string s1, because the null byte at the end of the string isn't counted in the return value. Similarly, for the second call, the bound was set at eight bytes, but only seven bytes were written. The eigth byte is again the null terminator.

This gets tricky if we want to use the return values a series of **snprintf** calls to keep track of the length of a string, as we append to it. We may, for instance, see the following type of statement in a loop somewhere:

```
n += snprintf(s1+n, …, "…%s…", …);
```

If at least one of the calls to **snprintf** results in a truncated result, writing fewer bytes than the return value suggests, this will leave a gap in the target buffer, that can be exploited in a code injection attack.

How can we quickly find this pattern in a code base? Text-based search tools are of little use here, so we need something stronger. But we don't really need the full power of a static analyzer either. Something in between that is fast and programmable can do the job.

The Cobra tool I mentioned earlier handles cases like this with easy. It breaks down an input sequence into lexical tokens, with some minimal types of annotations and markings. One useful type of marking is the pairing of parentheses, so that we can easily search for token sequences that are enclosed in matching pairs of round, curly, or square brackets. The tool also supports a simplified form of regular expression patterns, as well as variable bindings to keep track of matching names within a pattern.

The generic Cobra call for a search pattern is very similar to way one would use the familiar Unix grep tool for search text files:

```
$ cobra -pat 'pattern' *.c
```

The pattern for quickly finding all suspicious uses of **snprintf** calls, even in large code bases, is written as follows:

```
'x:@ident += snprintf ( ^,* :x .* /%s .* )'
```

The search identifies the suspected lines of code in the (unfixed) version of the **rsyslog/librelp** code in a fraction of a second. There are two such uses in the file **tcp.c**, on lines 1216 and 1228, both spanning two lines of source text. The first match, reformatted as three lines here to make it fit the width of this column, looks as follows:

```
src/tcp.c:1216-1217:
iAllNames += snprintf(allNames+iAllNames,
       sizeof(allNames)-iAllNames,
       "DNSname: %s; ", szAltName);
```

Let's look in a little more detail at the pattern we used. The pattern is defined over lexical tokens in the input, with each subsequent token to be matched separated by spaces from the other tokens. The first is an identifier, which is matched with the Cobra type specifier **@ident**. We then bind the name of whatever identifier we find here to a variable, which is called "**x**" here, by following the variable name with a colon and the description of the token to be matched. Next, we match the single token "**+=**" followed by the call to **snprintf**. The latter is another identifier, but this time we can specify the exact name that we are looking for.

The next token in the sequence to be matched is an open round brace, and at this point we want to check if the bound variable we just assigned appears anywhere in the first argument of the call. That first argument is everything up to the first comma, though not necessarily immediately following the open round brace token itself. We indicate that with a regular expression **^,*** which matches "any sequence of tokens other than a comma." The hat operator "**^**" indicates negation here, and the star "*****" is the standard Kleene star to indicate "zero or more matches." Next in the pattern we want to see at least on occurrence of our bound variable "**x**," which is indicated with the syntax "**:x**."

Another argument to the **snprintf** calls that we want to match is the format string, skipping over some more tokens that we're not interested in here. The format string itself is a single token, but we can use an embedded regular expression on this token to match on the textual contents of the string, and make sure that it contains at least one plain "**%s**" format specifier. The notation "**/%s**" formalizes that. The remainder of the pattern matches everything else up to the matching closing round brace at the end of the call.

Does It Appear Anywhere Else?

Finding this dangerous pattern in the older versions of the **rsyslog** code is of course no surprise, but now that we have the check available, we can also quickly scan other code bases. Doing so I found two other matches in the 1.3M lines of code for **ghostscript**. The version I checked was **ghostpdl-9.26**. The additional matches appear in file **ip-support.c**, on lines 684 and 2231, which both start with the now familiar:

```
bufptr += snprintf(bufptr, bufend – bufptr + 1,
```

The Cobra check requires no C preprocessing, so it takes very little effort to scan all source files you may have sitting on your disks for possible matches.

As you can probably tell, the design of the Cobra tool itself is simple, which means that it can be quite fast, even when scanning millions of lines of code. When the tool starts up, it reads the source files and builds the basic token stream for the input in core. It does not attempt to parse the code or build a

symbol table. It only does a small amount of preprocessing on the token stream, for instance to match open and close braces of various types at the right level of nesting. The tool also assigns types to tokens, which in the case of C source code will include keywords, identifiers, types, qualifiers, modifiers, constants, etc. If you're especially ambitious, you can define your own map of token categories to be used, which makes it possible to target the tool at a range of additional languages – even non-programming languages. The keywords for C, C++, Java, and Ada, and Python are predefined in the tool.

Bug Hunting

Having a capability to search code bases quickly can be useful to find examples of good coding patterns, but perhaps more frequently of bad ones. We can, for instance look for cases where a **`goto`** jump is followed immediately by the target label. The pattern

```
'goto x:@ident ; :x :'
```

will do the job. Note that the final colon in this pattern matches the token that follows a label name. This check finds 204 matches in the 18.3 Million lines of code from the Linux 5.0 distribution. Using 12 cpu-cores to speed things up, this check takes just 14 seconds on my desktop PC. Other simple checks that produce somewhat unexpected results include looking for **`else-if`** chains that do not end with a final **`else`** clause, which is a violation of the well-known MISRA guidelines (see https://www.misra.org.uk/). In the Linux code base this finds a little over 10,000 matches, matching about one in five cases where an **`else-if`** sequence appears. The pattern I used is again quite simple

```
'else if ( .* ) { .* } ^else'
```

restricting the check here to cases where the **`then`**-clause is properly enclosed in curly braces.

It is similarly easy to find, for instance, **`switch`** statements that do not have a default clause, or that do not contain more than just one case or default clause. Both of these are again in violation of the MISRA guidelines and surprisingly common in mainstream code.

The quick pattern checks can be wonderfully addictive. The typical way I use the Cobra tool is not with command-line calls as I've shown so far, but in interactive sessions. In that case I startup the tool by first reading in all the source code from a large code archive, and then quickly type a series of queries at the token sequence that was constructed in core to home in on areas of interest, or to put it more plainly: to home in on likely bugs. After all, spelunking dark caves for hidden treasure can be a lot more fun and productive if you can bring along a really good search light.

References

[1] Code Mining, IEEE Software, March/April 2019, Vol. 36, No. 2, pp. 25-29.