Fixing C strings

Originalartikel

Backup

<html> It’s well-known that null-terminated C strings are bug-prone and unsafe to use. They’re the stereotypical footgun. I’ve been tinkering in a bare-metal environment recently, writing all code myself, including the common types and routines you find in

highlighter-rouge"

```
libc
```

or similar. In all the code I wrote, there is not at single null-terminated string, and I have yet to encounter a bug related to bounds checking on strings or buffers. This is a quick rundown of what I’m doing and how it holds up. This is what the string type looks like. I came across this idea in a post by Chris Wellons. <div class=„language-c highlighter-rouge highlight“><pre>struct str { char *dat; sz len;};struct str_buf { char *dat; sz len; sz cap;};#define STR(s) \ (struct str) \ { \ .dat = (s), .len = lengthof(s) \ }</pre> I use the following functions to create strings and string buffers. You can imagine how functions implementing common string operations would look like. The post I linked also has some suggestions. <div class=„language-c highlighter-rouge highlight“><pre>struct str_buf str_buf_new(char *dat, sz len, sz cap);struct str str_from_buf(struct str_buf buf);struct str str_from_range(char *beg, char *end);struct str str_new(char *dat, sz len);</pre> Note that you can turn a string buffer into a string, but not the other way around. String buffers are meant to be read from and written to, while strings are only meant to be read from. So it makes sense to have strings be a “subtype” of string buffers (this can also be seen in the definition of the types;

highlighter-rouge"

```
struct str_buf
```

includes all fields from

highlighter-rouge"

```
struct str
```

). It seems reasonable to make the pointer in

highlighter-rouge"

```
struct str
```

highlighter-rouge"

```
const
```

, given what I just said. I’ll come back to that later. You might complain that writing

highlighter-rouge"

```
STR(...)
```

everywhere is not as nice as using a plain string literal. But I found it’s not an issue in practice, and you get used to it quickly. The only place I really use the

highlighter-rouge"

```
STR
```

macro is when calling print functions. Here’s what it looks like: <div class=„language-c highlighter-rouge highlight“><pre>print_str(STR(„blah\n“));print_fmt(STR(„0x%lx\n“), 0xdeadbeef);</pre> Consider using

highlighter-rouge"

```
S
```

or similar if you really want to save characters here. But this is C, so we accept being a little verbose and move on. I did not encounter a single bounds-checking-related bug since adopting this pattern six months ago. This is the main point: using this string type makes my code safer. I make extensive use of wrapper functions to do simple things such as creating or casting strings. Or, for example, to allocate a new string buffer from an arena: <div class=„language-c highlighter-rouge highlight“><pre> *The buffers I use for formatting are string buffers, hence the name.struct str_buf fmt_buf_new(struct arena *arn, sz cap){ struct str_buf buf; buf.dat = arena_alloc_array(arn, cap, sizeof(*buf.dat)); buf.len = 0; buf.cap = cap; return buf;}</pre> At no point do you have to touch the fields inside these types, making sure their invariants are kept intact. You can be pretty sure that you won’t footgun yourself if you only combine these functions. Of course, you could always touch the fields inside any of the structures, but why would you if you have functions to do everything for you. This improves safety, as you only need to figure out the correct routine to, say, allocate a sting buffer once. Later, you can be instantly suspicious of any code touching the fields inside these types. As mentioned before, the pointer in <code class=„language-plaintext highlighter-rouge“>struct str</code> is not <code class=„language-plaintext highlighter-rouge“>const</code>. I try to avoid <code class=„language-plaintext highlighter-rouge“>const</code> pointers as they are not well-enforced by the C language, and modifying <code class=„language-plaintext highlighter-rouge“>const</code> pointers is UB. Instead of dealing with the mess of <code class=„language-plaintext highlighter-rouge“>const</code>, I’m leaving it out entirely. You are not allowed to*

*modify the string data in a <code class=„language-plaintext highlighter-rouge“>struct str</code> and the API is designed to discourage this. Writing reliable C code depends a lot on the programmer’s discipline, this is the best one can do. Providing functions for all common operations on strings makes it easy to avoid forbidden operations. A downside of using <code class=„language-plaintext highlighter-rouge“>struct str</code> everywhere is that the compiler can’t check <code class=„language-plaintext highlighter-rouge“>printf</code> strings. I initially tried to avoid the use of <code class=„language-plaintext highlighter-rouge“>printf</code> strings and variadic functions altogether, but I was missing their ease of use, so I ended up writing a small <code class=„language-plaintext highlighter-rouge“>printf</code> implementation that uses a <code class=„language-plaintext highlighter-rouge“>struct str</code>. Current compilers warn you if the format string doesn’t match its arguments. But this only works on functions that have the same signature as <code class=„language-plaintext highlighter-rouge“>printf</code> so it doesn’t work on my implementation. Overall, I think this is an acceptable tradeoff because format strings are easier to reason about than arbitrary code and all possible issues are localized in calls to print functions. For my current project, correctness and safety have higher priority than performance (both in time and space). Still, I want to give this some consideration. Compilers are pretty good at optimizing small structures and are usually able to pass these in registers instead of using the stack. See this function for example. (You will note that I touch the fields in the <code class=„language-plaintext highlighter-rouge“>struct str</code> here, which should make you suspicious. Fortunately, this is a small function, and it’s easy to reason about the correctness of this code.^{1}) <div class=„language-c highlighter-rouge highlight“><pre>struct result com_write(u16 port, struct str str){ if (!str.dat || str.len <= 0) return result_error(EINVAL); while (str.len–) { while (!(inb(port + OFFSET_LINE_STATUS) & LINE_STATUS_TX_READY)) ; outb(port, *str.dat++); } return result_ok();}</pre> <code class=„language-plaintext highlighter-rouge“>struct result</code> is another small structure I like using. It’s much nicer than using an <code class=„language-plaintext highlighter-rouge“>int</code> with errnos passed as negative values. Most of the arguments in this post apply equally to <code class=„language-plaintext highlighter-rouge“>struct result</code>. This is the disassembly of the function compiled with <code class=„language-plaintext highlighter-rouge“>gcc 14.2.1</code> and <code class=„language-plaintext highlighter-rouge“>-mgeneral-regs-only -O2</code>: <pre class=„language-assembly“>0000000000000090 <com_write>: 90: mov %edi,%r8d 93: test %rdx,%rdx 96: jle d4 <com_write+0x44> 98: test %rsi,%rsi 9b: je d4 <com_write+0x44> 9d: lea 0x5(%rdi),%ecx a0: lea (%rdx,%rsi,1),%rdi a4: data16 cs nopw 0x0(%rax,%rax,1) af: nop b0: mov %ecx,%edx b2: in (%dx),%al b3: test $0x20,%al b5: je b0 <com_write+0x20> b7: movzbl (%rsi),%eax ba: mov %r8d,%edx bd: out %al,(%dx) be: add $0x1,%rsi c2: cmp %rdi,%rsi c5: jne b0 <com_write+0x20> c7: xor %eax,%eax c9: xor %edx,%edx cb: shl $0x10,%edx ce: and $0x1,%eax d1: or %edx,%eax d3: ret d4: mov $0x1,%eax d9: mov $0x16,%edx de: shl $0x10,%edx e1: and $0x1,%eax e4: or %edx,%eax e6: ret</pre> As you can see, both arguments are passed in a total of three registers. This is what you would expect were the function defined as <code class=„language-plaintext highlighter-rouge“>struct result com_write(u16 port, char *dat, sz len)</code> (indeed, the assembly that’s generated is the same ^{2}). The return value is also passed only in registers. The story is similar for all of these small functions. They are mostly defined <code class=„language-plaintext highlighter-rouge“>static inline</code> in headers, and the compiler does a good job eliminating function calls. Code size increases if you use a lot of inlined functions. So you’re mostly paying for this up front with compile time and with larger code size. <code class=„language-plaintext highlighter-rouge“>struct str</code> takes up more space than using null-terminated*

*strings, but it’s common to use a length variable along with a null-terminated string so it might not make a big difference. I didn’t take any measurements, but for code that’s not performance-critical, the overhead is clearly not too bad. I’s a tradeoff worth making for the ease of use and correctness benefits that these types bring. <li id=„fn:1“ role=„doc-endnote“> It would be quite embarrassing were there a bug here … ↩ <li id=„fn:2“ role=„doc-endnote“> See for yourself. This is the updated code (no change except for removing <code class=„language-plaintext highlighter-rouge“>struct str</code>): <div class=„language-c highlighter-rouge highlight“><pre>struct result com_write(u16 port, char *dat, sz len){ if (!dat || len <= 0) return result_error(EINVAL); while (len–) { while (!(inb(port + OFFSET_LINE_STATUS) & LINE_STATUS_TX_READY)) ; outb(port, *dat++); } return result_ok();}</pre> And the disassembly (same settings as before): <pre class=„language-asm“>0000000000000090 <com_write>: 90: mov %edi,%r8d 93: test %rsi,%rsi 96: je d5 <com_write+0x45> 98: test %rdx,%rdx 9b: jle d5 <com_write+0x45> 9d: lea 0x5(%rdi),%ecx a0: lea (%rsi,%rdx,1),%rdi a4: data16 cs nopw 0x0(%rax,%rax,1) af: nop b0: mov %ecx,%edx b2: in (%dx),%al b3: test $0x20,%al b5: je b0 <com_write+0x20> b7: add $0x1,%rsi bb: mov %r8d,%edx be: movzbl -0x1(%rsi),%eax c2: out %al,(%dx) c3: cmp %rdi,%rsi c6: jne b0 <com_write+0x20> c8: xor %eax,%eax ca: xor %edx,%edx cc: shl $0x10,%edx cf: and $0x1,%eax d2: or %edx,%eax d4: ret d5: mov $0x1,%eax da: mov $0x16,%edx df: shl $0x10,%edx e2: and $0x1,%eax e5: or %edx,%eax e7: ret</pre> ↩ </html>*