Jonathon McDonald
April 04, 2014

Fuzzer Analysis

This is a line by line analysis of why my fuzzer did not cause the execution of certain lines. My overall percentage was 63%. I'm confident that if I had more time I could have hit a higher number.

Lines 145 - 150, 160:

```
      break; case ULONG:      arg->i = va_arg(*ap, unsigned long);
#endif
      break; case ULLONG:     arg->i = va_arg(*ap, unsigned long long);
      break; case SHORT:      arg->i = (short)va_arg(*ap, int);
      break; case USHORT:     arg->i = (unsigned short)va_arg(*ap, int);
      break; case CHAR: arg->i = (signed char)va_arg(*ap, int);
```

Line 160 excluded.

My fuzzer did not produce an appropriate input range, as well as enough input types. I do not believe I had any shorts generated. I ran into some issues with the ruby binary available on the CADE machines that limited my generation of random inputs.

Line 177:

```
      for (; l >= sizeof pad; l -= sizeof pad)
            out(f, pad, sizeof pad);
```

Line 176 included for clarity.

When printing strings I did not ever use padding.

Lines 226 - 240:

```
      } else if (fl & MARK_POS) {
            prefix+=3;
      } else if (fl & PAD_POS) {
            prefix+=6;
      } else prefix++, pl=0;

      if (!isfinite(y)) {
            char *s = (t&32)?"inf":"INF";
            if (y!=y) s=(t&32)?"nan":"NAN", pl=0;
            pad(f, ' ', w, 3+pl, fl&~ZERO_PAD);
```

```
        out(f, prefix, pl);
        out(f, s, 3);
        pad(f, ' ', w, 3+pl, fl^LEFT_ADJ);
        return MAX(w, 3+pl);
}
```

Again my range of inputs (especially for floating point) was incredibly lackluster. This was due to some unfortunate limitations with Ruby 1.8.7 and my limited time looking for a work around. I'm confident I could have hit code coverage here had I produced a better range of inputs.

Lines 246 - 293:

```
        long double round = 8.0;
        int re;

        if (t&32) prefix += 9;
        pl += 2;

        if (p<0 || p>=LDBL_MANT_DIG/4-1) re=0;
        else re=LDBL_MANT_DIG/4-1-p;

        if (re) {
              while (re--) round*=16;
              if (*prefix=='-') {
                    y=-y;
                    y-=round;
                    y+=round;
                    y=-y;
              } else {
                    y+=round;
                    y-=round;
              }
        }

        estr=fmt_u(e2<0 ? -e2 : e2, ebuf);
        if (estr==ebuf) *--estr='0';
        *--estr = (e2<0 ? '-' : '+');
        *--estr = t+('p'-'a');

        s=buf;
        do {
              int x=y;
              *s++=xdigits[x]|(t&32);
              y=16*(y-x);
              if (s-buf==1 && (y||p>0||(fl&ALT_FORM))) *s++='.';
        } while (y);
```

```
if (p && s-buf-2 < p)
      l = (p+2) + (ebuf-estr);
else
      l = (s-buf) + (ebuf-estr);

pad(f, ' ', w, pl+l, fl);
out(f, prefix, pl);
pad(f, '0', w, pl+l, fl^ZERO_PAD);
out(f, buf, s-buf);
pad(f, '0', l-(ebuf-estr)-(s-buf), 0, 0);
out(f, estr, ebuf-estr);
pad(f, ' ', w, pl+l, fl^LEFT_ADJ);
return MAX(w, pl+l);
```

Again formatting floating points was my main crux. The range of inputs was not sufficient, and I did not use any of the capital inputs such as %A, %G, or %F.

Line 336 & 385:

Code is not shown for brevity. These lines belonged in the floating point format function, and again signal an insufficient range of inputs for floating points.

Line 447:

```
for (i=0; isdigit(**s); (*s)++)
            i = 10*i + (**s-'0');
```

Simply put, the `isdigit` function always returned zero.

Lines 471-472:

Code is not shown for brevity. My inputs never caused overflow, which led to an error never occurring.

Lines 485-487:

```
                l10n=1;
                argpos = s[1]-'0';
                s+=3;
```

This code appears to be handling a case recently added by POSIX. My code did not generate material to test this case.

Lines 495-523 & 537:

Code is not shown for brevity. Again this appears to be a case where my fuzzer did not generate modifiers. I also always passed in a valid argument so the validity checker never proved false.

Lines 559-572:

```
            case 'n':
                  switch(ps) {
                  case BARE: *(int *)arg.p = cnt; break;
                  case LPRE: *(long *)arg.p = cnt; break;
                  case LLPRE: *(long long *)arg.p = cnt; break;
                  case HPRE: *(unsigned short *)arg.p = cnt; break;
                  case HHPRE: *(unsigned char *)arg.p = cnt; break;
                  case ZTPRE: *(size_t *)arg.p = cnt; break;
                  case JPRE: *(uintmax_t *)arg.p = cnt; break;
                  }
                  continue;
            case 'p':
                  p = MAX(p, 2*sizeof(void*));
                  t = 'x';
                  fl |= ALT_FORM;
```

I never passed in the `%n` or `%p` formats. While I should have been able to fuzz `%n`, I'm not sure I could easily fuzz `%p`.

Lines 586 & 588:

Code not shown for brevity. My code did not exploit sub specifiers.

Line 605:

I was not aware of the `%m` formatter, which is an extension to `printf` provided by the GNU glibc to print out the standard error number.

Lines 616-631:

```
            case 'C':
                  wc[0] = arg.i;
                  wc[1] = 0;
                  arg.p = wc;
                  p = -1;
            case 'S':
                  ws = arg.p;
                  for (i=l=0; i<0U+p && *ws && (l=wctomb(mb, *ws++))>=0 &&
l<=0U+p-i; i+=l);
```

```
if (l<0) return -1;
p = i;
pad(f, ' ', w, p, fl);
ws = arg.p;
for (i=0; i<0U+p && *ws && i+(l=wctomb(mb, *ws++))<=p; i+=l)
      out(f, mb, l);
pad(f, ' ', w, p, fl^LEFT_ADJ);
l = w>p ? w : p;
continue;
```

I was unfamiliar with the `%C` and `%S` modifiers, but it turns out they are Microsoft specific. My fuzzer does not test this.

Lines 652-658:

```
if (!l10n) return 0;

for (i=1; i<=NL_ARGMAX && nl_type[i]; i++)
      pop_arg(nl_arg+i, nl_type[i], ap);
for (; i<=NL_ARGMAX && !nl_type[i]; i++);
if (i<=NL_ARGMAX) return -1;
return 1;
```

Simply put, my printfs always printed successfully and this code was never reached.

Lines 672-689:

```
// FLOCK(f);
if (!f->buf_size) {
      saved_buf = f->buf;
      f->wpos = f->wbase = f->buf = internal_buf;
      f->buf_size = sizeof internal_buf;
      f->wend = internal_buf + sizeof internal_buf;
}
ret = printf_core(f, fmt, &ap2, nl_arg, nl_type);
if (saved_buf) {
      f->write(f, 0, 0);
      if (!f->wpos) ret = -1;
      f->buf = saved_buf;
      f->buf_size = 0;
      f->wpos = f->wbase = f->wend = 0;
}
```

These lines appear to be for putting the output of printf somewhere other than `stdout`, though I could be wrong.