NICHOLAS M. STOUGHTON

USENIX Standards Activities

Nick is the USENIX Standards Liaison and represents the Association in the POSIX, ISO C, and LSB working groups. He is the ISO organizational representative to the Austin Group, a member of INCITS committees J11 and CT22, and the Specification Authority subgroup leader for the LSB.

nick@usenix.org

2005 was a busy year for me as the USENIX standards representative. There are three major standards that I watch carefully:

- POSIX, which also incorporates the Single UNIX Specification
- ISO-C
- The Linux Standard Base (LSB)

In order to do that, USENIX funds my participation in the committees that develop and maintain these standards. Throughout 2005, the Free Standards Group (FSG) also helped fund these activities. For each of these, let's look at the history of the standards, then at what has happened over the past 12 months or so, and, finally, what is on the agenda for this year. Each of these standards is critical to a large proportion of our members. Without these standards, open source software as we know it today would be very, very different!

POSIX

The POSIX family of standards was first developed by the IEEE, arising from earlier work from /usr/group and the System V Interface Definition (SVID), and was published as a "trial use" standard in 1986. In 1988, the first full-use standard was published. The difference between "trial" and "full" use is principally in the use of the term "should" rather than "shall" in the requirements for any interface.

In 1990, the 1988 API standard was revised, clarifying a number of areas and expanding them. At the same time, the API standard became an ISO standard. At this point in history, there were about 10 separate POSIX projects under development, ranging from the basic OS system calls and libraries, through commands and utilities, to security, remote file access, super-computing, and more. In 1992, the second part of POSIX was published (the Shell and Utilities volume), and it became a second ISO standard. Amendments to these standards were also under development, and led to the addition of real-time interfaces, including pthreads, to the core system call set. Many of the other projects died away as the people involved lost interest or hit political roadblocks (most of which were reported in *;login:* at the time).

Until the end of the twentieth century, POSIX was developed and maintained by IEEE exclusively. At the same time, the Open Group (also known as X/Open) had an entirely separate but 100% overlapping standard, known as the Single UNIX Specification. This specification started from the same place in history, and many of the participants around the table at an X/Open meeting were the exact same people who had met a few weeks before at an IEEE POSIX meeting to discuss the same set of issues!

This duplication of effort became so annoying that a new, collaborative, group was formed to produce a single document that would have equal standing for each of ISO, IEEE, and the Open Group. That group held its first meeting in Austin, Texas, in 1998, and was therefore named the "Austin Group." The Austin Group published a full revision of the POSIX and Single UNIX specifications as a single document in 2001. It was adopted by all three organizations and is maintained by the same team, which represents the interests of all three member organizations.

Since the 2001 revision, work has been steadily progressing maintaining this 3762-page masterpiece. Every week, there is a steady stream of "defect reports," which range from typos in the HTML version (the document is freely available in HTML on the

Web; see http://www.unix.org/single_unix_specification), through major issues with ambiguous definitions, and so on. Some of these defects can be quickly and cleanly fixed, and two "Technical Corrigenda" documents have been approved, which alter the wording for some of the interfaces to clarify their meanings.

Every ISO standard (and every IEEE standard, too) has a five-year "reaffirm/revise/withdraw" process, where the document is examined to see if it is still relevant, whether it needs revision to meet current needs, or whether it is now outdated and should be withdrawn. For POSIX, the Austin Group has elected to revise the specification during 2006.

Under the Austin Group rules, the group as a whole cannot invent new material. One of its sponsor groups (IEEE, ISO, and the Open Group) must have prepared a document and had it adopted under its own organization rules before it can be presented to the group as a whole. Therefore, the Open Group has been developing, and is now in the final stages of approving, a number of documents which include new APIs to become a possible future part of a UNIX branding program. Once approved, these documents can then be examined by the Austin Group (OK, so it's still the same group of people who developed the set in the first place) for inclusion into the POSIX revision.

The new interfaces under consideration are ones that have been popular in the GNU-C library (glibc) and Solaris for some time, but have not been formally standardized before. They include support for standard I/O functions to operate on memory buffers as well as external files, getline and getdelim, some multibyte string-handling functions, robust mutexes, and versions of functions that take pathnames relative to a directory file descriptor rather than plain pathnames (this helps avoid certain race conditions and helps with really long pathnames).

I would expect to see official drafts of this new revision this summer, and the final version in 2008.

POSIX has long had support beyond the C language world. There are Ada and Fortran official "bindings" to POSIX. However, there has never been a real connection between the C++ world and the POSIX world; C++ programs can use C to call POSIX functions. But this leads to all sorts of complications for C++ programmers and, more seriously, to much reinvention of the wheel in providing mappings between C++ constructs and those of POSIX. The Austin Group has received several defects from C++ programmers who want to know why they can't do x, to which the traditional answer has been "don't use C++, use C"! And to make matters worse, the C++ language committee is also going through a revision at present, and they want to add all sorts of features to the language that might make it harder to access some of the fine-grained features of POSIX (since they want the language to work on other platforms, they deliberately try to be OS-neutral).

All that may change soon. A study group has recently been formed to look into the need for, and desire to build, a C++ binding to POSIX. USENIX is hosting the wiki for this group, and you are welcome to join: http://standards.usenix.org/posix++wiki.

ISO C

The first version of the ISO C standard, then known as ANSI-C, was published in 1989. It took the original language from Kernighan and Ritchie's book and tightened it up in a number of places. It added function prototypes and considerably improved on the standard C library. The first versions of POSIX used this language as the underlying way to describe interfaces, and included a c89 command to invoke the compiler.

Between 1989 and 1999, the C committee added wide character support and addressed several language "defects"—internal discrepancies in the way various features were described. The committee included a number of compiler vendors, who were also keen to have the language permit ways to guide an optimizer: features such as constants, volatile variables and restrict pointers were added to the language for this purpose.

In 1999, a new revision came out which included several new features such as these, along with major rework for floating point support (including things such as complex numbers).

At this point, the committee is fairly happy with the state of the core language and is fighting back against proposals to change it. However, they have not stopped working! They are currently preparing several technical reports that optionally extend the C language in a number of directions. Of these, by far the most significant to most USENIX members is the report formerly known as the "Security TR." I say formerly because the term "Security" (and it turns out, many other related words) are so overloaded and charged with meaning that by far the most objections to the document were to its title.

The report formerly called the "Security TR" actually attempts to deal with the fairly common problem of buffer overflow. It does so in a very simple fashion: every interface in the ISO-C standard library that takes a buffer has a secure variant which includes the size of the buffer. Now, while that is the meat of the original concept, it isn't all that the report currently proposes. The report introduces the concept of runtime constraints, that is, various things that must hold true when an interface is invoked. The original standard library simply had undefined behavior when you passed a null pointer to an interface that expected a pointer to a buffer. So

```
char *p = malloc(10);
gets(p);
```

could fail in a variety of ways, despite being well-formed, legal C.

The new "secure" library version of this,

```
char *p = malloc(10);
gets_s(p, 10);
```

will invoke a runtime exception handler (analogous to a signal handler) if p is null (because the malloc failed) or if there are more than 10 characters on the next line of standard input.

According to its current stats, this document proposes a library that might be of benefit to someone going over thousands or millions of lines of existing code and trying to find and plug all of the possible buffer overflow spots. It is likely to end up obfuscating some of the code. It is also possible that if the buffer size is not well known, it could end up hiding bugs where the programmer simply guesses at a buffer size but is wrong; now the code looks as if it has been retrofitted to prevent buffer overflows, but it hasn't!

It will also likely change the ABI of third-party libraries that want to use this; they must now have a way of receiving the size to check against. This suggests to me that this library will have little uptake as it stands, though Microsoft has implemented it and has updated all of its core programs to use it (is this a good thing?).

The core of the problem is that memory handling in C is complicated and error-prone. Nobody will doubt that improvements in the supporting APIs are useful, but the existing APIs already provide the means to write correct programs. It is just cumbersome to do so. The proposed interfaces won't change that; on the contrary, they could make programs even more complex. An alternative approach is to take as much of the memory handling away from the programmer as possible.

To that end, I am preparing a second part to this technical report that uses dynamic memory allocation instead of static buffers. For new programs (rather than retrofits of old code), this approach leads to a cleaner, more robust application, with fewer possibilities for problems. For example, instead of reading data from an input with gets into a static buffer (that might be too small), the getline function allocates a buffer big enough to hold the entire input line, however long it was (or returns NULL if there was insufficient memory). The only problem with such an interface is that the programmer must remember to release the memory when he or she is done with it, by means of a call to free. Some have argued that this, too, can lead to unexpected bugs, as programmers forget to free these buffers, and the application slowly leaks memory. However, I believe this is a smaller problem than the use of static buffers with guessed sizes.

Back to the name of this report: as I said , "Secure Library" got a ringing "no" vote. This report does not address any of what many people regard as security issues. The name "Safer Library" was suggested, but the owners of a product called "Safer-C" objected. In the end it has come down to "Extensions to the C Library—Part 1—Bounds Checking Functions."

THE LINUX STANDARD BASE

The LSB is an Application Binary Interface (ABI), rather than an Application Programming Interface (API). As such, it covers details of the binary interfaces found on a given platform, providing a contract between a compiled binary application and the runtime environment that it will execute on. The first version was published in 2000 and has developed rapidly since then. It now consists of a Core specification (including ELF, Libraries, Commands & Utilities, and Packaging), a Graphics module (including several core X11 libraries), and a C++ module.

Each specification has a generic portion that describes interfaces that are common across all architectures and seven architecture-specific add-ons that spell out the differences between the architectures.

For the past year or more, I have been acting on behalf of the Free Standards Group as the technical editor for the ISO version of this standard. ISO 23360 was unanimously approved last September by the national bodies that contribute to the subcommittee responsible for programming languages and their runtime environments.

We have had to jump through a few hoops in the final publication phase, but now it looks as though the document is ready. You will soon be able to buy a CD from ISO with the LSB on it (see http://www.iso.org), or you can just download the PDF for free from the Free Standards Group (though the copyright notice is subtly different, as are the running headers and footers—see http://refspecs.freestandards.org.

What now for the LSB? Are we done? Of course not! The LSB workgroup has a new chair, Ian Murdock (the Ian of Deb*ian*). A new subgroup is developing a desktop specification, with an increased focus on libraries needed by desktop applications such as GTK, Qt, PNG, XML, more X, imaging, etc.

And with the pace of development in the open source community, it is necessary to continually revise the specification to match current practice. For example, until recently the pluggable authentication modules (PAM library) had no symbol versioning, but the upstream maintainers have now decided to add that (which makes maintaining an ABI possible). The LSB now has to be updated to discuss which version of which symbol you should be using to get the promised behavior.

Additionally, the LSB Core Specification is mostly a superset of the POSIX APIs. However, there is a small handful of places where the two specifications are at odds. For the most part, these differences won't bother most programmers most of the time, but there are corner cases you can creep into where you'll find your application isn't portable between an LSB-conforming platform and a POSIX-conforming platform. For example, POSIX requires the error EPERM if you attempt to unlink a directory, while the LSB requires this error to be EISDIR.

A document describing these differences is now available from ISO as Technical Report 24715.

During the revision of POSIX this year, and as a part of any future LSB development work, we will review these changes to see if there is any way that either specification can accommodate the behavior of the other in some deterministic fashion.

A well-supported standard for Linux is a necessary component of Linux's continued success. Without a commonly adopted standard, Linux will fragment, thus proving costly for ISVs to port their applications to the operating system and making it difficult for end users and Linux vendors alike. With the LSB, all parties—distribution vendors, ISVs, and end users—benefit as it becomes easier and less costly for software vendors to target Linux, resulting in more applications being made available for the Linux platform.

It is important for the LSB workgroup not to slip into the comfortable feeling that the job is now done. If the workgroup does not remain focused on the core document, that core document will quickly become irrelevant, overtaken by the pressures of distribution vendors to have their product be the de facto standard in the absence of a good de jure base.

My work with the LSB over the past year has not just been as the technical editor of the ISO standard, although this has been a major part of my work. I have also been one of the principal technical editors of the specification as a whole. With the completion of the submission of the initial core specification to ISO, the sources of funding for this critical project have largely dried up. I end with a plea: if your organization believes that standards for UNIX, Linux, and C are important, consider donating money to USENIX to help fund the development and maintenance of these standards.

ANNUAL MEETING OF THE USENIX BOARD OF DIRECTORS

The Annual Meeting of the USENIX Board of Directors will take place at the Boston Marriott Copley Place during the week of the 2006 USENIX Annual Technical Conference, May 30–June 3, 2006. The exact location and time will be announced on the USENIX Web site.