7 Attaining Robust Open Source Software

Peter G. Neumann

"Is open source software inherently better than closed-source proprietary software?" This is a question that is frequently heard, with various intended meanings of "better." As a particularly demanding case, let us consider critical applications with stringent requirements for certain attributes such as security, reliability, fault tolerance, human safety, and survivability, all in the face of a wide range of realistic adversities—including hardware malfunctions, software glitches, inadvertent human actions, massive coordinated attacks, and acts of God. In addition, let's toss in operational requirements for extensive interoperability, evolvability, maintainability, and clean interface design of those systems, while still satisfying the critical requirements. In this context, we are interested in developing, operating, and using computer systems that are robust and easily administered.

To cut to the chase, the answer to the simple question posed in the first sentence is simple in concept, but decidedly not so simple in execution: Open source software is not intrinsically better than closed-source proprietary software. However, it has the potential for being better if its development process addresses many factors that are not normally experienced in mass-market proprietary software, such as the following:

- Well-defined and thoroughly evaluated requirements for system and application behavior, including functional requirements, behavioral requirements, operational requirements, and—above all—a realistic range of security and reliability requirements.
- System and network architectures that explicitly address these requirements. Sound architectures can lead to significant cost and quality benefits throughout the development and later system evolution.
- A system development approach that explicitly addresses these requirements, pervasively and consistently throughout the development.

- Use of programming languages that are inherently able to avoid many of the characteristic flaws (such as buffer overflows, type mismatches, wild transfers, concurrency flaws, and distributed-system glitches) that typically arise in unstructured, untyped, and error-prone languages and that seem to prevail over decades, through new system releases and new systems.
- Intelligent use of compilers and other development tools that help in identifying and eliminating additional flaws. However, sloppy programming can subvert the intent of these tools, and thus good programming practice is still invaluable.
- Extensive discipline on the part of designers, implementers, and managers throughout the entire software development process. This ultimately requires better integration of architecture, security, reliability, sound programming techniques, and software engineering into the mainstream of our educational and training programs.
- Pervasive attention to maintaining consistency with the stated requirements throughout operation, administration, and maintenance, despite ongoing system iterations. Some combination of formal and informal approaches can be very helpful in this regard.

Conceptually, many problems can be avoided through suitably chosen requirements, architectures, programming languages, compilers, and other analysis tools—although ultimately, the abilities of designers and programmers are a limiting factor.

The answer to the initially posed question should not be surprising to anyone who has had considerable experience in developing software that must satisfy stringent requirements. However, note that although the same discipline could be used by the developers of closed-source software, marketplace forces tend to make this much more difficult than in the open-source world. In particular, there seems to be an increasing tendency among the mass-market proprietary software developers to rush to market, whether the product is ready or not—in essence, letting the customers be the beta testers. Furthermore, efforts to reduce costs often seem to result in lowest-common-denominator products. Indeed, satisfying stringent requirements for security and reliability (for example) is generally not a goal that yields maximum profits. Thus, for practical reasons, I conclude that the open-source paradigm has significant potential that is much more difficult to attain in closed-source proprietary systems.

The potential benefits of nonproprietary nonclosed-source software also include the ability to more easily carry out open peer reviews, add new functionality either locally or to the mainline products, identify flaws, and fix them rapidly—for example, through collaborative efforts involving people irrespective of their geographical locations and corporate allegiances. Of course, the risks include increased opportunities for evil-doers to discover flaws that can be exploited, and to insert trap doors and Trojan horses into the code. Thus a sensible environment must have mechanisms for ensuring reliable and secure software distribution and local system integrity. It must also make good use of good system architectures, public-key authentication, cryptographic integrity seals, good cryptography, and trustworthy descriptions of the provenance of individual components and who has modified them. Further research is needed on systems that can be predictably composed out of evaluated components or that can surmount some of the vulnerabilities of the components. We still need to avoid design flaws and implementation bugs, and to design systems that are resistant to Trojan horses. We need providers who give real support; warranties on systems today are mostly very weak. We still lack serious market incentives. However, despite all the challenges, the potential benefits of robust open-source software are worthy of considerable collaborative effort.

For a further fairly balanced discussion of the relative advantages and disadvantages with respect to improving security, see five papers (Lipner 2000; McGraw 2000; Neumann 2000; Schneider 2000; and Witten et al. 2000) presented at the 2000 IEEE Symposium on Security and Privacy. The session was organized and chaired by Lee Badger. These contributions all essentially amplify the pros and/or cons outlined here. Lipner explores some real benefits and some significant drawbacks. McGraw states flatly that "openish" software will not really improve security. Schneider notes that "the lion's share of the vulnerabilities caused by software bugs is easily dealt with by means other than source code inspections." He also considers inhospitability with business models. The Witten paper explores economics, metrics, and models. In addition, Neumann's Web site includes various papers and reports that can be helpful in achieving the goals of system development for critical requirements, with particular attention to the requirements, system and network architectures, and development practices. In particular, see Neumann 2004 for a report for DARPA (summarized briefly in Neumann 2003a) on the importance of architectures in attaining principled assuredly trustworthy composable systems and networks, with particular emphasis on open source but with general applicability as well. That report is part of the DARPA CHATS program on composable high-assurance trusted systems, which is seriously addressing many of the promising aspects of making open-source software much more robust. Furthermore, see the archives of the ACM Risks Forum (http://www.risks.org), a summary index (Neumann 2003b) to countless cases of systems that failed to live up to their requirements, and an analysis of many of these risks cases and what needs to be done to minimize the risks (Neumann 1995). It is an obvious truism that we should be learning not to make the same mistakes so consistently. It is an equally obvious truism that these lessons are not being learned—most specifically with respect to security, reliability, survivability, interoperability, and many other "-ilities."