Open-source versus proprietary software: Is one more reliable and secure than the other?

A. Boulanger

One of the most powerful movements in the information technology community today is the widespread adoption of free and open-source software (FOSS). What was once an idealistic fringe movement conceived and formalized by MacArthur award laureate Richard Stallman has now become one of the most powerful influences in the world of information technology. As FOSS systems grow in popularity, questions of the reliability and security of these systems emerge, especially in comparison with proprietary systems. This paper surveys the arguments presented by proponents of each type of software in published reports and discusses the deployment and reliability figures for both FOSS and proprietary systems as well.

The explosive increase in the number of deployed free and open-source software (FOSS) systems has changed the world of information technology. When the first FOSS systems were developed, many of the users of these early systems were themselves technologists. Moreover, the distribution and use of such FOSS systems was initially limited to academia, research laboratories, and technical user groups. Today, however, FOSS systems are being developed and designed for mass consumption. Most of the businesses on the Internet use FOSS-developed systems, and retail stores such as Wal-Mart are offering to the general public steeply discounted computers that take advantage of FOSS-developed software. As the group of people and organizations that depends on FOSS technologies continues to grow, it becomes increasingly important that FOSS systems be secure and reliable.

Many FOSS systems were originally developed by a loose collaboration of volunteer programmers. The completed systems were then released to the public, and anyone could acquire and use these systems without paying a licensing fee. Free support for these systems was also provided by the volunteer community in the form of mailing lists and Web sites. Currently, however, many FOSS projects are professional efforts in which development is performed by a team of paid programmers, and the system is supported either without charge or through fees and subscriptions. In contrast, traditional proprietary systems are developed by a team of designers,

project managers, programmers, technical writers, and quality assurance engineers. The systems they produce undergo design reviews, development progress reports, and formal quality assurance testing. Once completed, these systems are packaged commodities that are sold or licensed to the public for a fee. Support for the software product is usually provided by the developer of the system.

Which model is more reliable in terms of availability and security? Many papers discussing these issues have been published by proponents of each type of software. This paper examines the arguments presented in these published reports as well as the deployment and reliability figures for both open and proprietary systems.

SECURITY AND RELIABILITY CONSIDERATIONS FOR FOSS AND PROPRIETARY SYSTEMS

The security and reliability of FOSS-based systems are currently topics of an often heated debate. Proprietary vendors are funding, producing, and publishing reports supporting the position that closed-source proprietary systems offer superior security relative to their FOSS counterparts. For every report that is published claiming the superior security of proprietary systems, the FOSS community responds with a report refuting these claims.

Perhaps a significant reason for this heated debate is the fact that widespread adoption of the FOSS model would directly threaten the revenue stream of vendors of proprietary software. In several recent 10-Q quarterly filings with the Securities and Exchange Commission, Microsoft, one of the world's largest software publishers, has stated that the popularization and adoption of FOSS systems pose a significant challenge to its business model.[1] It is not surprising then that proprietary software vendors are on the offensive, attempting to discredit FOSS-developed systems.

Arguments about the relative security and reliability of FOSS and proprietary software typically focus on two key issues: availability of source code and software defect levels. We discuss these issues in the following sections.

Availability of source code

In June 2002, the white paper "Opening the Open Source Debate"[2] was released by the Alexis de Tocqueville Institution, an organization funded in part by Microsoft. Among its most controversial findings was that "Open source GPL [General Public License] use by government agencies could easily become a national security concern. Government use of software in the public domain is exceptionally risky." The basis for this assertion is the assumption that publicly available source code invites "hackers"[3] to examine the code in order first to search for exploitable vulnerabilities and then to develop and deploy Trojan horses and other types of malicious software. Therefore, the study concludes that the availability of source code is a significant security threat to government organizations using FOSS.

There are several problems with this assertion. It is inferred that closed-source proprietary systems are automatically more secure than their FOSS counterparts, a "security through obscurity" approach. If this assertion were true, then the number and rate of published vulnerability reports for closed-source systems should be significantly lower than those of their FOSS equivalents.

However, the available data does not support this assertion. In fact, many FOSS systems have substantially lower rates of published vulnerabilities than their closed-source counterparts. For example, a recent report[4] showed that Apache**, a FOSS Web server whose history and development[5] will be discussed in more detail later in this paper, suffered from substantially fewer published vulnerabilities than Microsoft's IIS (Internet Information Server) and marginally more vulnerabilities than Netscape** Enterprise Server. If the de Tocqueville Institution's assertion were true, then Apache should have had significantly more published vulnerabilities than the closed-source Web servers.

Hiding the source code for a system does not provide any additional security. People searching for vulnerabilities do not require source code to discover software defects. For example, a common way to locate a software defect is to send a program unexpected and unusual data and then monitor how the system responds.[6] If the system fails or behaves erratically as a result of the input, this might indicate a flaw in the system that would warrant further investigation.

With the prevalence of sophisticated software monitoring, debugging, and disassembly tools,

much of the source code can be derived from the binary version of the executable program. Anyone interested in obtaining the source code would simply have to apply one of many widely available tools to the program. The output from these programs, while not perfect, would deliver sufficient information to make it fairly easy to understand the internal working of the system.

Moreover, source code for many deployed systems has often been inadvertently leaked. Many software producers outsource code development or exchange source code with other organizations for a variety of reasons. Anyone with access to the systems involved, authorized or not, can obtain a copy of the source code and distribute it. It is a very common practice in the hacker community to traffic in source code packages. In the course of investigating systems intrusions, our organization has discovered copies of the source code for nearly all of the major operating systems on hacker systems. By examining the source code, hackers hope to locate new vulnerabilities and produce what are known as *zero-day* exploits, attacks against vulnerabilities which the software vendor has not yet found. Zero-day exploits are the most prized exploits in the hacker community because the targeted systems are defenseless. On the surface, the possibility of such exploits appears to support the most common rationale for keeping source code secret, namely preventing miscreants from using the code to discover software defects and attack vulnerable systems. However, the real problem may not be that the source code is being used to discover vulnerabilities. Rather, it may be that only two groups of people have access to the source code, the small group of developers, who are tasked with developing and maintaining the system, and the potentially larger community of hackers, who are motivated to discover and exploit vulnerabilities. When the source code is widely available to everyone, as is the case with FOSS systems, there are more opportunities for people outside of the hacker community to examine and correct potential software defects before they can be exploited.

There are, of course, examples of FOSS systems that have been notoriously insecure. Software defects in *sendmail*, a FOSS mail transfer agent, have been a favorite target of attack for years. In November of 1988, the first Internet worm was released by Cornell graduate student Robert Morris.[7] The worm quickly spread throughout the emerging Internet, in part by exploiting vulnerabilities in the *sendmail* and *fingerd* routines, causing widespread outages.

■ The security and reliability of FOSS-based systems are currently topics of an often heated debate ■

According to one report, the Internet worm of 1988 impacted 6,000 out of 60,000 systems, or 10 percent of the systems on the emerging Internet. Morris discovered these vulnerabilities while analyzing the source code for *sendmail* and *finger*. However, it was the availability of the source code that enabled the Internet community to respond quickly and mitigate this threat. In their paper "An Analysis of the Internet Virus of November 1988,"[8] Mark Eichin and Jon Rochlis remarked, "Source availability was important. All of the sites that responded quickly and made progress in truly understanding the virus had the UNIX** source code."

Having the source code widely available is a multi-edged sword. The hacker community can use the code to analyze systems and locate vulnerabilities. The developer community can maintain and improve the code through analysis and can use formalized testing methodologies to discover and remedy software defects before they can be exploited. The user community can also use the source code in response to an attack to discover and correct the vector through which the attacker has exploited the system. After the flaw has been discovered, the user community can respond rapidly by patching the system, removing the vulnerability, and then sharing the patch with the public. With a closed system, users are completely dependent upon the vendor's ability to discover the vulnerability, and then develop and distribute fixes.

A differentiator then between widely deployed FOSS systems and their proprietary counterparts is the value of source code in responding to security threats. Having the source code available was critical in assisting systems administrators to respond to the Internet worm of 1988. Technical staffs reviewed the source code of the affected systems and were able to understand the vulnerability and develop defenses against both the immediate threat

and future attacks. This advantage of having source code available was outlined in the government-funded research report "A Business Case Study of Open Source Software," published in July of 2001 by MITRE**.[9] This report was the result of a publicly funded study to help program managers evaluate open-source software and development methodologies, as well as their potential application within the managers' technical programs. The MITRE report stated that in the FOSS community "popular open-source products have access to extensive expertise, and this enables the software to achieve a high level of efficiency," and that software patches, or corrections, happen "potentially an order of magnitude faster than those of proprietary software."

■ Widespread adoption of the FOSS model would directly threaten the revenue stream of vendors of proprietary software ■

Software defect levels

Software defects are a fact of life, and any software package, whether FOSS or proprietary, is likely to have a substantial number of flaws. A percentage of these defects will directly impact the security of the system. According to the Software Engineering Institute, an experienced programmer produces approximately one defect per 100 lines of code, or an average defect rate of 1 percent.[10] If, during the software development life cycle, 99 percent of those defects are discovered and remedied, then approximately 1000 software defects will remain in a software package consisting of one million lines of source code, a modest code base by current standards. For example, the Red Hat Linux** 7.1 distribution consists of approximately 30 million lines of code (LOC), and Microsoft Windows** XP contains approximately 40 million LOC. Even with extensive testing and defect remediation, there will still remain a very significant number of software defects in these systems. Using the defect rate statistics noted earlier, Windows XP and Red Hat Linux would be estimated to have approximately 40,000 and 30,000 undiscovered defects respectively. In addition, both proprietary and FOSS systems are constantly evolving. As they progress through the software life cycle, features are added to the system and defects are discovered and remedied. Because most software packages are in this constant state of evolution, it is difficult to estimate the number of software defects a particular system will contain. When code is added to a system, there is also the possibility that the developers will introduce a new defect into the system. There are even examples where a developer has produced modifications to correct one security problem but inadvertently introduced another. Software defects can even occur due to problems having nothing to do with changes in the source code. For example, there have been instances where a defect in a development tool has introduced a defect into a system build.

Software defects are an unavoidable problem that can reduce the overall reliability of a system. When the reliability of a system is reduced, the overall security of the system is also reduced. Reliability and security are inextricably intertwined. A highly reliable system has fewer software defects and, in general, is more secure than an unreliable system. To use an analogy from the physical world, a padlock built with an excellent locking mechanism (security) but constructed with defective steel (a defect) offers very little security. The same is true for a lock using excellent construction materials (security), but having a defect in the locking mechanism (reliability) that makes the lock vulnerable. A classic example of the latter involved the published demonstration that a particular high-security bicycle lock could be quickly opened with an ordinary Bic** pen.[11]

Every software package release, whether FOSS or proprietary, contains a number of defects. The available data suggest, however, that when a software flaw is discovered, the FOSS community responds more rapidly than proprietary software vendors. According to an article published by e-Week Labs, FOSS organizations in general respond to problems more quickly and openly, while proprietary "software vendors instinctively cover up, deny, and delay."[12] The FOSS organizations produce small fixes that are available publicly through mailing lists and Web sites. Users of the affected software can download the update, apply the patch, and rebuild the system. The patch itself is in turn a segment of source code that is available for inspection by the general public. In contrast,

proprietary software vendors tend to wait and roll out large cumulative releases known as *service packages*. These service packages usually contain only binary code, which is not open for public scrutiny and can potentially introduce new problems into the system. The users of proprietary software products must blindly trust the integrity and competence of the software publisher; whereas, FOSS users know that the patches they have installed can be (and will be) publicly scrutinized.

VENDOR-NEUTRAL STUDIES

If the FOSS community is more responsive to software defects, then it would be reasonable to conclude that FOSS products should show higher reliability statistics than their proprietary counterparts. Although each side of the FOSS issue has offered anecdotal evidence supporting its respective position, there are several vendor-neutral studies that favor popular FOSS-developed systems.

In 1990, Professor Barton Miller from the University of Wisconsin developed the *fuzz* system, a system which produced random data streams that were then fed as input to programs from several proprietary versions of the UNIX system.[6] Dr. Miller discovered that 24–33 percent of the programs tested failed when fed the fuzz-generated data. Each of these failures can be directly attributable to a software defect. A properly written program should not fail when it receives unexpected or random data. In 1995, Dr. Miller revisited this work and included in his test both Linux-derived and other freely available GNU utilities. (GNU is a recursive acronym for "GNU's not UNIX.") The new study tested over 80 utility programs from nine different versions of UNIX. Of these UNIX platforms, seven were from proprietary vendors and two were from the FOSS community. The major results of the study were published in the paper "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services."[13] Since the original study, the proprietary systems had improved, and their overall failure rate dropped from 24–33 percent down to 18–23 percent. For the FOSS systems, the failure rate for the Linux utilities was second lowest at 9 percent failures, and the failure rate for the GNU utilities was the lowest at 6 percent. The results of the study demonstrate that popular FOSS implementations of UNIX utilities can have significantly better resilience to unexpected input than their proprietary counterparts. This in turn could indicate higher quality and reliability for the FOSS utilities. This is not to say

■ Hiding the source code for a system does not provide any additional security ■

that all FOSS systems are more reliable than proprietary systems. However, this experiment does demonstrate that the quality of software developed under the FOSS model can equal or exceed the quality of commercially developed software.

Another study conducted in 1999 by Bloor Research, an independent IT analysis and consultancy organization based in the United Kingdom, pitted Windows NT** against GNU/Linux in a head-to-head comparison.[14] The test was conducted for one year and rated each of the platforms according to nine criteria. Overall, GNU/Linux was rated as superior to Windows NT in seven of the nine categories. In particular, in the OS (Operating System) Availability category, GNU/Linux had a significantly better reliability rating than Windows NT. During the year of testing, the GNU/Linux system did not experience a single outage that was attributable to software. However, a hardware failure resulting from a hard disk malfunction did cause an outage for the GNU/Linux system that lasted four hours before service was restored. The resulting availability figure for GNU/Linux was thus 8756 hours out of 8760 hours, or 99.95 percent uptime. During the same period of time the Windows NT system suffered a total of 68 failures. Of the 68 failures, one failure was a hard disk failure; twenty-six failures were attributed to memory management faults; eight failures were attributable to file system faults; and the remaining failures were of unknown origin. The amount of lost time due to these outages was reported as 65 hours, resulting in an overall availability of the Window NT system of 99.26 percent or 8695 hours out of a total of 8760 hours. The results of this one-year study are impressive and demonstrate that FOSS-developed systems, at the very least, can compete well with their proprietary counterparts, providing stable and reliable server platforms.

Additional information comes from companies that develop automated software-inspection services. Some background will help explain the role of these

companies in the software industry. In large projects the people responsible for maintaining a system are often not the same people who originally developed the system. Unless the maintainers are careful and

■ Reliability and security are inextricably intertwined ■

fully understand the system, it becomes very easy to make a mistake that can affect the overall quality of the system code. One of the ways to increase the reliability of a system is to review the source code for defects and remedy them before the system is released. Typically the inspection process is performed through formal code reviews and evaluations. This process is very labor-intensive and time-consuming. Historically, as systems grew and it became more expensive to perform formalized code reviews, researchers developed ways to automate this process and make code reviews less labor-intensive than manual inspection. Several companies now offer automated software-inspection services, allowing software publishers to outsource their code reviews. One such company, Reasoning, Inc., has been assisting organizations to improve the quality of their systems through automated software inspection for almost 20 years and is considered a leader in this field.

In 2003, Reasoning conducted a study of the implementation of the Internet protocol code in the 2.4.19 version of the Linux kernel and in five proprietary operating systems.[15] The purpose of the study was to use automated code inspection techniques to compare the quality and defect rate of each implementation of the TCP/IP (Transmission Control Protocol/Internet Protocol) networking software. Reasoning discovered that the defect rate for the Linux code was 0.1 reported defects per 1000 lines of code (KLOC). The defect rate for proprietary implementations was reported to be 0.55 defects per KLOC. Reasoning concluded that the FOSS implementation of TCP/IP had a significantly lower defect density compared to the implementations in the five proprietary operating systems. The study also concluded that the overall quality of the FOSS package rated in the top third of all source-code projects that had been inspected by Reasoning.

In July of 2003, Reasoning analyzed the popular Apache Web server software package.[16] The Apache Web server is a FOSS system developed and maintained by the Apache Software Foundation, a membership-based not-for-profit corporation.[5] The Apache server is the dominant HTTP (Hypertext Transfer Protocol) server package on the Internet today, according to a recent survey by Netcraft.[17] This survey, conducted in June 2004, reported that of the 51.6 million identifiable servers on the Internet at that time, Apache had over 67 percent of the market, followed by Microsoft with a 21 percent market share. With so many organizations relying on FOSS technology for their Internet presence, it would obviously be valuable for IT managers to have a vendor-neutral software-quality metric to assist in a decision whether to deploy FOSS or proprietary systems. Reasoning concluded in their study that the defect density for the 2.1 release of the Apache system was 0.53 defects per KLOC. To put that figure into perspective, Reasoning compared the defect density of the Apache system to the 200 other projects Reasoning had analyzed at that time, both FOSS and proprietary, involving a total of 33 million analyzed lines of code. The top third of these 200 projects showed defect densities of less than 0.36 defects per KLOC; defect densities of the middle third ranged from 0.36 to 0.71 defects per KLOC; the bottom third had defect densities greater than 0.71 defects per KLOC. Given these statistics, the defect rate for the Apache system falls somewhere in the middle compared to the rest of the industry and slightly above the average defect density Reasoning has found for proprietary software (0.51 defects per KLOC).

Reasoning then continued their study of FOSS quality by inspecting the source code for the 4.0.16 version of MySQL**, the leading open-source database system.[18] In this December 2003 study, Reasoning examined 236,000 lines of MySQL source code and detected 21 software defects in the system. The study determined that the code quality of the MySQL system was six times better than that of comparable proprietary code, with a defect density for the MySQL system of 0.09 defects per KLOC compared to the average defect rate of 0.51 defects per KLOC for proprietary code. Not incidentally, as a result of this study the maintainers of the MySQL package promptly corrected the problems Reasoning had found and produced a maintenance release,

Version 4.0.17, that was made available for download from their distribution site.

These studies suggest that FOSS systems can meet, or even exceed, the quality of their proprietary counterparts. Furthermore, the Reasoning studies only inspected defect rates in source code. Their studies did not include other aspects of software packages such as usability, compatibility, features, and support costs, all important aspects that need to be considered in the decision-making process when deploying systems. The studies from Reasoning do suggest that FOSS-developed systems offer viable alternatives to proprietary systems in terms of software quality and reliability.

COMPARISON OF DEVELOPMENT PROCESSES

The empirical evidence from these studies suggests that popular FOSS-developed systems are, at the very least, as secure and reliable as their proprietary counterparts. Obviously, proprietary software is expensive to develop and support. How can a disparate loose-knit group of developers produce software of comparable quality for free? A look into how open-source software is produced provides some insight into the success of FOSS projects.

Most traditional proprietary software projects use a variant of the waterfall model[19] in their software development process. The waterfall model has five well-defined phases:

1. *The requirements phase*, in which the problem and the requirements of the proposed system are defined.

2. *The system and software design phase*, in which a technical solution is applied to the problem.

3. *The implementation and unit-testing phase*, in which the components of the technical solution are developed and individually tested.

4. *The integration and system-testing phase*, in which all of the individual components are aggregated and tested as a whole unit and compared to the defined requirements.

5. *The support and maintenance phase*, which begins when the tested system, having met the defined requirements, is deployed and maintained.

The entire process is iterative, in that at any phase in the process the project may be forced to return to an earlier phase as new problems and requirements are defined. This is the developer feedback loop.

■ FOSS systems can meet, or even exceed, the quality of their proprietary counterparts ■

Problems that are detected downstream in the testing and maintenance phases are communicated back to the programmers. The programmers research a solution to the problem, develop a correction, and resubmit the component for testing and integration.

FOSS development is often less structured, but still shares features of traditional development. The main difference is the addition of a consumer feedback loop, wherein the users of the systems are encouraged to directly participate as part of the development community. In a proprietary environment, consumers may report software defects and offer suggestions, but they are unable to directly participate in the development process because they lack access to the system source code. In FOSS development, every consumer has access to the source code and can thus directly participate in the continuous improvement of the software package. In reality only a small percentage of the user base will have the desire or expertise to actively participate in the project. However, when the user base grows large enough, that small percentage of users can swell into a substantial number of contributors.

The development of the Apache Web server demonstrates this process.[5] The Apache system emerged in 1995 and was derived from a set of patches that were applied to the then-popular NCSA (National Center for Supercomputing Applications) Web server source code (leading to the name "Apache" server, a play on the words "a patchy" server). These patches were contributed by frustrated users of the then-largely-unsupported NCSA server who required additional functionality. These users had access to the server source code and were able to modify the system to meet their requirements. They then submitted their patches to a group of volunteers who maintained the system. Eventually the group abandoned the initial set of patches and completely redesigned the system. In December 1995, Apache

Version 1.0 was formally released and, as noted previously, quickly became the dominant Web server on the Internet. As the Apache system grew in popularity, more organizations became dependent on this server technology to support their missions, and as a result, more people contributed to the project. The group of volunteers who reviewed submissions and maintained the Apache source code continued to grow in size, eventually forming the Apache Group in 1995, and then the Apache Software Foundation in 1999.

■ FOSS-developed systems have a distinct advantage in their ability to respond to security threats ■

The Apache Software Foundation currently consists of 22 core developers who contribute to the development and maintenance of the Apache system. Users of the system submit patches, bug reports, and suggestions for improvement to this core team of developers. The Apache development community communicates through Internet mailing lists and Web sites. Any piece of software that is submitted is peer-reviewed and extensively tested before being included in the source code repository. Every Apache source code repository contains status information on changes and plans for improvements, as well as references to outstanding issues, so that all developers can stay informed. This collaborative effort is synchronized through status information and identification of the developers who are responsible for specific parts of the project. The actual software development and module testing are performed on the developer's machine. When a module has been tested and is ready for release, the developer posts the software patches to the developer mailing list. Subscribers to the developer list then review the proposed modifications and test them against their own systems. Only when the proposed changes have been peer-reviewed and tested are they incorporated into the Apache source repository. If there are problems with the proposed code changes, such as incompatibility issues or software defects, the developer community has the opportunity to detect and remedy these defects before they can impact the rest of the system. This is the developer feedback loop of collaborative software development: people responsible for developing the system reviewing one another's work.

In the consumer feedback loop, users of the published system uncover defects and submit bug reports and suggestions for improvement to the core developers. Because the source code is publicly available, users are able to locate defects, submit suspect code fragments, and contribute patches to correct the problem. Once a patch is submitted from the consumer loop, it undergoes rigorous peer review and testing in the developer loop before making its way into the system source code repository. This is a remarkably efficient system for distributing the burden of code review across both the developer and user domains. Every user can potentially become a developer and contribute to the overall success of the software package. One caveat is that this system may only work well with popular FOSS projects. A small orphaned FOSS project could lack the critical mass of resources that a larger, more popular project enjoys with its extensive user base. However, for large FOSS projects this system appears to work very well. There are many examples of FOSS-developed systems that enjoy massive user bases who continually work to improve the reliability and functionality of the system. Once a critical mass of developers and users emerges, the open-source development project can achieve the same reliability and security standards as proprietary systems.

CONCLUSIONS

Which is more secure: closed or open-source software? Unfortunately the answer is not that clear. In general, both FOSS and proprietary systems are roughly equivalent in terms of security and reliability. Neither is inherently more secure or reliable than the other. Analytical arguments made in favor of either approach are not conclusive. Empirical studies have suggested that FOSS can potentially outperform proprietary systems. Nonetheless, any system that was not developed to be secure invariably will not be. There are certainly proprietary systems deployed that are more secure than their FOSS counterparts (e.g., S/COMP or GEMSOS** versus Gnu/Linux), just as there are FOSS-deployed systems that appear to be more secure than their proprietary equivalent (e.g., Apache versus Microsoft IIS).[20] One problem with attempting to quantify the security of proprietary and FOSS systems is the fact that verifiably trustworthy systems are very

difficult and expensive to develop and certify. To date there are only seven operating systems that have achieved a Common Criteria Certification evaluation assurance level (CC EAL) rating of 4 (meaning "methodically designed, tested and reviewed").[21] The highest rated FOSS-developed operating system is the SUSE** Enterprise Server V8 distribution of Gnu/Linux with a rating of CC EAL 3+ (meaning "methodically tested and checked"). This does not necessarily mean that FOSS operating systems with lower ratings are less reliable than proprietary systems. This can also mean that the funding required for formal certification has not been made available. Because the CC certification is expensive to obtain, only larger organizations can afford to sponsor a CC evaluation, especially at the higher levels of certification.

Another problem is that every software system mentioned in this article, both open-source and proprietary, requires frequent patching to remediate defects. Any system that requires frequent patching is inherently insecure. Using patch counts as a metric for security is misleading. A system that requires a security patch every six months is not twice as secure as a system that requires patching every three months. They are both insecure. To use another analogy from the physical world, a car that explodes once every 1000 miles cannot be considered twice as safe as a car that explodes once every 500 miles. As in the software example, both vehicles should be considered unsafe. Another issue with using published vulnerabilities as a security metric is that software systems are under constant change. Whenever old software defects are discovered and remedied, new software defects may be introduced into the system. Because systems are constantly evolving, there is no easy way to determine the absolute number of defects in a system at a given instant. However, FOSS-developed systems have a distinct advantage in their ability to respond to security threats. As noted previously, the organizations that responded most successfully to the Internet Worm of 1988 had access to the UNIX source code. Having the source code enabled technical personnel to understand the immediate threat and then share that information with other affected organizations.

It is this sharing of information that is the key strength behind the FOSS movement. FOSS developers have the ability to analyze how previous systems were constructed and "stand on the shoulders of giants." As in the scientific research community, this free exchange of information promotes innovation and advances the field. The FOSS movement can use this shared information to encourage participation from a global talent pool. When the number of users of a FOSS project increases, so too will the number of developers who can potentially participate in the project. Once a critical mass of users has formed, the momentum from this combined effort will yield quality systems that meet and exceed the security and reliability metrics of their proprietary counterparts—at a much reduced cost.

The FOSS movement is gaining traction. What was once an idealized concept espoused by hackers, hobbyists, and academics is now formalized and organized and is the dominant technology behind the Internet. As FOSS-based technologies continue to gain market share, proprietary software publishers will be forced to innovate to remain competitive and survive. It will be interesting to watch and see where FOSS technology takes us in the future. It will be even more interesting to participate.

**Trademark or registered trademark of Aesec, Linus Torvalds, Microsoft Corporation, MySQL AB Company, Netscape Communications Group, Red Hat, Inc., Société BIC, SUSE LINUX AG, The MITRE Corporation, The Apache Software Foundation, or The Open Group.

CITED REFERENCES

1. See, for example, *Securities and Exchange Commission Form 10-Q Quarterly Report For the Quarterly Period Ended March 31, 2004*, Microsoft Corporation (May 3, 2004), p. 35, http://www.microsoft.com/msft/download/FY04/MSFT_3Q2004_10Q.doc.
2. *Opening the Open Source Debate: A White Paper*, Alexis de Tocqueville Institution (June 2002), http://www.adti.net/opensource.pdf.
3. The term "hacker" is used here to mean "a person who illegally gains access to and sometimes tampers with information in a computer system," as defined in *Merriam-Webster's Collegiate Dictionary, Eleventh Edition*, Merriam-Webster, Inc., Springfield, MA (2003), p. 559. Others prefer to use the term "cracker" to describe such a person.
4. CERT® Coordination Center, Software Engineering Institute, http://www.cert.org/.
5. Apache HTTP Server Project, The Apache Software Foundation, http://httpd.apache.org/ABOUT_APACHE.html.
6. B. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM* **33**, No. 12, 32–44 (1990).
7. K. Hafner and J. Markoff, *Cyberpunk: Outlaws and Hackers on the Computer Frontier*, Simon & Schuster, Inc., New York (July 1992).

8. M. Eichin and J. Rochlis, "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988," *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1–3, 1989, IEEE, New York (1989), pp. 326–343.
9. C. A. Kenwood, *A Business Case Study of Open Source Software*, MITRE Corporation (July 2001), http://www.mitre.org/work/tech_papers/tech_papers_01/kenwood_software/kenwood_software.pdf.
10. W. S. Humphrey, *The Quality Attitude*, Software Engineering Institute (2004), http://www.sei.cmu.edu/news-at-sei/columns/watts_new/watts-new.htm.
11. J. S. Clark, "Kryptonite Bic-picking," *New Cyclist* (October 1992).
12. J. Papoza, *eWeek Labs: Open Source Quicker at Fixing Flaws*, Ziff-Davis Media, Inc. (September 30, 2002), http://www.eweek.com/article2/0,3959,562226,00.asp.
13. B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*, Technical Report, Computer Science Department, University of Wisconsin (November 1995), ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf.
14. *Linux versus Windows NT: The Verdict*, Bloor Research (October 1999), http://www.bloor-research.com/research_library.php?productid=245.
15. *Linux TCP/IP Inspection Report*, Reasoning, Inc. (2003), http://www.reasoning.com/downloads.html.
16. *Apache Open Source Inspection Report*, Reasoning, Inc. (2003), http://www.reasoning.com/downloads.html.
17. *June 2004 Web Server Survey*, Netcraft, Ltd. (June 6, 2004), http://news.netcraft.com/archives/2004/06/06/june_2004_web_server_survey.html.
18. *How Open Source and Commercial Software Compare: MySQL White Paper*, Reasoning, Inc. (2003), http://www.reasoning.com/downloads.html.
19. W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *1970 Western Electronic Show and Convention (WESCON) Technical Papers* **14**, Los Angeles, CA, August 25–28, 1970, IEEE, New York (1970), pp. 1–9; reprinted in *Proceedings of the Ninth International Conference on Software Engineering (ICSE'87)*, Monterey, CA, March 30–April 2, 1987, IEEE Computer Society Press, Los Alamitos, CA (1987), pp. 328–338.
20. D. A. Wheeler, *Why Open Source Software/Free Software (OSS/FS)? Look at the Numbers!* (November 7, 2004), http://www.dwheeler.com/oss_fs_why.html.
21. Consumers—List of Evaluated Products, Common Criteria Project, http://www.commoncriteriaportal.org/public/consumer/index.php?menu=4.

Accepted for publication October 27, 2004.
Published online April 12, 2005.

Alan Boulanger
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (boulange@us.ibm.com).
Alan Boulanger joined IBM in October 1995 as a member of the Thomas J. Watson Global Security Analysis Laboratory. His research interests include network security, intrusion detection and remediation, applied penetration testing techniques, data forensics, telephony-related security, and emerging threat analysis. Since joining IBM, Mr. Boulanger has filed several information-security-related patents and has provided security-related technical assistance to the business community and to federal government agencies. He is an active member of both the New York and New England Electronic Crimes Task Forces. ■