RISK PERCEPTION AND TRUSTED COMPUTER SYSTEMS: IS OPEN SOURCE SOFTWARE REALLY MORE SECURE THAN PROPRIETARY SOFTWARE?

by David L. Wilson

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

RISK PERCEPTION AND TRUSTED COMPUTER SYSTEMS: IS OPEN SOURCE SOFTWARE REALLY MORE SECURE THAN PROPRIETARY SOFTWARE?

A Thesis

Submitted to the Faculty

Of

Purdue University

By

David L. Wilson

In Partial Fulfillment of the

Requirements for the Degree

Of

Master of Science

May 2004

ABSTRACT

Wilson, David L. MS, Purdue University, May 2004. Risk Perception and Trusted Computer Systems: Is Open Source Software Really More Secure than Proprietary Software? Major Professor: Eugene H. Spafford.

The purpose of this paper was to examine available evidence to try to find support for the claim that Open Source software is more secure than commercial or proprietary software based on the theory that Open Source software can be examined by many people, and any flaws that they find will quickly be repaired. This paper examined the various theories behind the argument, both pro and con, and then studied the best evidence available on the topic, concluding that there appeared to be no difference in software quality. But this paper concluded that better metrics might provide data that could more readily allow for a reconsideration of this topic with additional research. This paper also used analysis techniques related to risk perception to examine the forces that might encourage people to believe that Open Source software is more secure than commercial software in the absence of any real evidence to that effect.

CHAPTER 1: A HISTORY OF THE OPEN SOURCE MOVEMENT

A Mighty Wind

The development of Open Source software is one of those polarizing movements that sweep across the globe, and, this being the Internet age, the movement has swept across the globe with unusual speed and ferocity. As with all significant ideas that challenge orthodoxy, it has legions of rabid supporters and detractors. This paper is primarily an examination of the security aspects of code produced by the Open Source software movement as compared to more traditional forms of software development. But because much of the research and debate about Open Source has clearly been inspired by ideology and political activism, along with their traditional companions, fear and hatred, it is necessary to provide a bit of background about how the Open Source movement came to be, and why it has come to stand for so much more than simply a way of making software.

Free (as in speech) Software

In 1984, Richard Stallman, who had been a researcher and programmer at the Massachusetts Institute of Technology's Artificial Intelligence Laboratory since 1971, faced what he had come to think of as a purely moral crossroads in his life (Stallman, 1999). Stallman had spent much of his time in the lab over the past 13 years working on an operating system known as the Incompatible Timesharing System, which had been developed within the lab for use with a large computer named the Digital Equipment

Corporation PDP-10 (Levy, 1984). Programmers who developed software at many institutions freely allowed others to use code the programmers had created, in addition to permitting the modification of that code to meet specific needs, and even the rewriting of the code to run on a different type of computer, a process called "porting." MIT administrators had the same sorts of policies for ITS, permitting others to use it and make modifications.

Stallman had enjoyed being part of a community that actively aided its members in problem-solving, largely through sharing resources (King, 1999). If Stallman saw an interesting program, he need only ask its programmer for the software's "source code," and it would be provided. Source code is the raw programming language humans use to create a computer program. The source code is translated – typically through the use of another program called a compiler – into machine, or object code, so the computer can understand and execute, or run, it. When an individual user purchases a program from a store, the user is typically buying the machine code; without access to the source code from which the machine code was made, the user probably will not be able to alter the program to meet the user's own unique needs.

But in 1982, MIT decided to abandon the free ITS that Stallman and others had developed and instead pay Digital Equipment Corporation for a timesharing operating system. As is customary with most software vendors, Digital Equipment Corporation prohibited buyers from redistributing the product or discussing the inner workings of the code. Stallman was appalled at such restrictions (Stallman, 1999). As the years had passed him by, he had grown alarmed over what he perceived as the unconscionable restrictions software developers were forcing upon software users. By barring users from seeing or modifying source code, he believed developers were preventing users from fully exploiting the power of computers. For instance, a program used to control a printer did not work well in the MIT lab environment, which made using the printer extremely difficult. But Stallman was unable to make changes in the program himself because of the nondisclosure agreement. Such agreements, he found, were becoming increasingly common. According to Stallman, “This meant that the first step in using a computer was to promise not to help your neighbor. A cooperating community was forbidden. The rule made by the owners of the proprietary software was, ‘If you share with your neighbor, you are a pirate. If you want any changes, beg us to make them’” (Stallman, p 54).

Stallman eventually concluded that placing restrictions on software that limited whether the code could be shared or changed was both antisocial, because it essentially grants developers a monopoly, and immoral because users become slaves to the whims of software developers. Stallman’s critique anticipated one of the critical debates of the 21st Century: In a world where software developers can unilaterally set their own rules of conduct, statutes become increasingly meaningless. Because, as Lawrence Lessig (1999) wrote, in a digitized world, computer “code is law” (p. 6). East Coast code (laws and regulations) can trump West Coast code (the code that runs computers and the Internet) but only if those in the legislative, executive, and judicial branches have the will, and the knowledge, to do so.

Stallman saw his open and sharing community of software developers being destroyed as one-by-one the programmers – or “hackers” as they usually called themselves before the term generally came to mean a malevolent computer vandal – went to work for people who made proprietary software. Those hackers signed nondisclosure agreements, forever agreeing to stop openly collaborating with users who needed to make changes in the programs. Stallman eventually concluded that computer users needed a solid computing environment that did not have any significant user restrictions attached to it. He quit the MIT lab in January, 1984 to launch his project, which he dubbed GNU: GNU's Not Unix (GNU Project, 2004). The GNU project was based on a concept called "free" software, but Stallman was not intending to provide software at no financial cost to the user. Rather, Stallman used "free" in the political sense of the word. To be a part of the GNU universe, the software must allow users to:

1. Use the software for any purpose.
2. Modify the program (which generally means users have access to the source code).
3. Redistribute copies, either for free or for a fee.
4. Redistribute modified copies of the software, allowing the community to benefit from any improvements.

The critical point is that Stallman envisioned people making money by selling copies of "free" software. He had watched over the years as software that developers gave away – such as the X Window system developed by MIT for the Unix environment – was added to proprietary commercial Unix systems. Those proprietary licenses meant that users did not have the freedom to modify the code, which Stallman considered critical. To address this issue, the GNU used copyright to prohibit anyone from imposing such restrictions, a technique Stallman named "copyleft." Under the copyleft concept, even modified versions of protected works must also remain unfettered. Stallman memorably summed up the distinction by saying the software was free as in "free speech," not free as in "free beer," (Stallman, 1999).

Stallman, and eventually others who found this cause appealing, founded the Free Software Foundation – a nonprofit that derives much of its funds from sales of free software, as oxymoronic as that sounds – and spent the next few years developing various pieces of a complete system aimed at users of Unix, a powerful computing environment. By 1991, however, while Unix users had embraced a number of critical components developed by the GNU project – such as GNU Emacs – the project still had no "kernel," the fundamental core of an operating system. That year, however, Linus Torvalds released a Unix compatible kernel named Linux (Torvalds, 1991), which, combined with the bits of the GNU system that had already been developed let users run a relatively complete Unix-like system dubbed GNU/Linux.

Better Marketing Means Better Software

While Stallman's project continued to grow, a number of key players in the Free Software community became increasingly convinced that Stallman's vision was widely perceived as "anti-business," and that corporations would thus shun these products (Scoville, 2004). A general rejection by corporations would doom any hope the group had of a wide scale adoption of their products; conversely, a broad adoption of their products by corporations would almost certainly lead to wider adoption of their products by people outside the corporate world as well, because individuals would presumably first grow comfortable using the products at work and then embrace them at home.

To combat this issue, in 1998 a small group of people largely associated with Stallman's Free Software movement met to found what they came to call Open Source software (DiBona, Ockman & Stone, 1999). The primary distinction between Stallman's somewhat utopian vision of Free Software and Open Source is that Open Source permits distributors to mix proprietary and Open Source software. For instance, under the Open Source license, a programmer can freely incorporate code used in an Open Source project within a proprietary program.

To be considered Open Source software, the program must adhere to the following criteria (Open Source, 2004):

1. The license cannot keep anybody from either selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. In addition, no royalties or similar fees can be levied on sales. This measure was instituted to discourage Open Source participants from defecting.
2. The program has to include source code, or must let users easily obtain the source code, preferably at no cost. Obfuscated code – that is, source that is deliberated distorted to make it harder for people to interpret it – is prohibited.
3. The license has to permit modifications of the original code, and allow users to create "new" works derived from that code, and those works are allowed to be distributed under the same license as the original software. This encourages rapid evolution of software.

4. The license can limit distribution of modified code only if modified versions of the code are distributed along with the original code (using a technique known as patch files). In this case, the license can also require the modified code to carry a different name or version number. This exception allows individual programmers to protect their reputations and also gives users a clear idea of who has written the software they are using.
5. The license must not discriminate against any individual, group, or organization. In cases where some regulations apply, such as export restrictions, the license can mention the law, but not incorporate that into the license.
6. The license also has to be endeavor neutral. For instance, it cannot prohibit use by a political party, or discriminate between corporations and non-profits.
7. No additional licenses, such as a non-disclosure agreement, can be added.
8. A program can be distributed independently of a product or package without affecting the license.
9. A license cannot place restrictions on other software, such as requiring that an Open Source program only be used in a completely Open Source environment.
10. The license must be technology neutral, neither requiring nor prohibiting uses of specific technology, such as requiring that the program be distributed on CD-ROM.

The Great Success Stories

While Linux – which is distributed in several different varieties, including those labeled SUSE, Mandrake, Red Hat, and Debian – is the poster child of the Open Source movement with millions of users, hundreds of Open Source programs are widely used today. Some notable examples are:

- BIND, or the Berkeley Internet Name Domain, plays a key role in the Internet's Domain Name System, which translates easy-to-remember addresses such as www.name.com into the address numbers used by Internet computers to properly route traffic. The vast majority of name server machines on the Internet use BIND (Internet Systems Consortium, 2004).
- Sendmail is one of the most widely used mail transport agents, used to move e-mail around the Internet (Leonard, 1998).
- Perl, a programming language first released in 1987, is so powerful and widely used that it has been called "the duct tape of the Internet" (Perl.com, 2004).
- Perhaps the greatest Open Source success story is the Apache Web server, which, as of March 2004, was running two out of three Web sites, or 67 per cent. Microsoft's Web server product, Internet Information Server, was a distant second with only 21 per cent. Two years ago, Microsoft servers ran about one third of all Web sites, with Apache holding a bare majority at 58 percent (Netcraft, 2004).
- Mozilla is a redesign of the first mass-market Web browsing application, Netscape (Mockus, Fielding, & Herbsleb, 2002). In what remains an unusual maneuver, developer Netscape Communications Corporation,

> unable to charge users for its Web browsing software after Microsoft Corporation began including its Internet Explorer Web browser with Microsoft's Windows operating system, decided to make its core product completely free in hopes of earning money on its high profit-margin Web servers. That plan proved unsuccessful, however. While Mozilla and related programs have become commonly used alternatives to the Internet Explorer Web browser, their use has not driven developers to embrace Netscape servers, which have continued to decline as Apache has continued to increase its share of the market.

A New, Sometimes Scary, Way of Living

It was not long before thousands of people were working on Open Source projects such as Linux, writing millions of lines of code. While many critics were baffled by the movement, and characterized the Open Source approach to software development as appearing to be "ad hoc" and "chaotic" (Lawrie & Gacek, 2002, p. 35), in many cases Open Source projects were actually highly organized. And indeed, it was the level of organization that was possible with such a disjointed organization that attracted much of the academic analysis of the Open Source movement. While participants seemingly arbitrarily chose what they wanted to work on, resolved whatever problem they had chosen to their own personal satisfaction, and then sent the code they had developed via the Internet to an individual they had oftentimes never met, all the work thus submitted was somehow seamlessly integrated into a complex whole. "Collaborative Open Source projects such as Linux and Apache have demonstrated, empirically, that a large, complex system of code can be built, maintained, developed and extended in a non-proprietary setting where many developers work in a highly parallel, relatively unstructured way and without direct monetary compensation" (Weber, 2000, p. 3).

Evangelists described this new process as the distinction between a cathedral – the traditional way of doing business – and a bazaar (Raymond, 2000). They suggested that a new world had dawned, one free of the shackles of corporate domination. This "flat" development structure was the wave of the future, they argued.

Others disagreed, however, in part because there were many Open Source projects that were run as little fiefdoms. Bezroukov (1999) quoted Jordan Hubbard, a leader in the FreeBSD Unix project as saying, "Despite what some free-software advocates may erroneously claim from time to time, centralized development models such as the FreeBSD Project's are hardly obsolete or ineffective in the world of free software. A careful examination of the success of reputedly anarchistic or "bazaar" development models often reveals some fairly significant degrees of centralization that are still very much a part of their development process."

Indeed, Torvalds himself made no secret that he maintained a firm grip on the process of Linux development, once posting the following message in an online newsgroup: "If you still don't like it, that's ok: that's why I'm boss. I simply know better than you do. Linus "what, me arrogant?" Torvalds" (Torvalds, 1996).

Even so, traditional developers and academics spent a great deal of time analyzing exactly what was going on with the Open Source movement, this despite the fact that to many people from a traditional business background, the individuals writing Open Source code for free sounded dangerously close to what some might call communists; Steve Ballmer, chief executive of Microsoft Corporation, made headlines when he actually used that word to describe the Open Source movement (Lea, 2000). Open Source participants, a loose affiliation of programmers located around the world with little in common beyond their coding skills and Internet connections, were demonstrating a new economic paradigm that had left Microsoft, still reeling from a series of protracted antitrust battles that threatened its dominance in the market for desktop computers, worried about its future (Gomes, 1986). Open Source products such as Linux appealed to people on a number of different levels. It was available for free. It could be customized to meet specific needs. In addition, Open Source software came to be perceived as a more secure alternative to Microsoft Corporation's Windows family of operating systems. This paper will focus on the security aspects of Open Source software.

CHAPTER 2: BACKGROUND AND DEFINITIONS

Proponents of Open Source software argue that it provides a number of advantages over proprietary systems. While these advantages are said to include better performance, lower cost, and improved scalability, this paper will be focused on the claim by Open Source advocates that proprietary, closed programs and operating systems are less secure than Open Source code (Raymond, 2001; Wheeler 2003). Open Source proponents make two arguments for this claim, one theoretical and one experiential.

The experiential argument is based on the fact that many of the most widely publicized computer security failures since the year 2001 have been directly tied to proprietary programs and operating systems, primarily those made by Microsoft. In July 2001, a worm – a malevolent bit of software capable of seeding itself among other computers with no human intervention – known as Code Red spread around the world (CERT advisory, 2001b; Pethia 2002). Two months later, the Nimda worm caused significant damage less than an hour after the first report of infection (CERT advisory, 2001c; Pethia 2002). On January 25, 2003, a worm known as Sapphire or Slammer was released onto the Internet. Analysts were stunned by the speed with which the program distributed itself: More than 90 percent of all vulnerable computers in the world were infected within 10 minutes. Analysts believe the infection rate doubled every 8.5 seconds. By comparison, the Code Red worm launched less than two years before only doubled every 37 minutes (Moore, Paxson, Savage, Shannon, Staniford, & Weaver, 2003). Each of these incidents, which affected Microsoft products, caused widespread inconvenience, and cost billions of dollars in lost business, repair costs, or preventative maintenance. There have been no incidents on a similar scale affecting Open Source products, such as Linux.

The theoretical argument supporting the claim that Open Source is more secure than proprietary software is based in part on a standard tenet of cryptographic security: People trying to protect the integrity of encrypted data must always assume that their opponents understand the underlying methods used to encrypt the data. While the cryptographic key used in the process of encryption can be kept a secret, any system based on keeping the method or mechanism of encryption a secret is inherently insecure. This principle, formulated by Kerckhoffs (1883), is axiomatic within the cryptographic community (Schneier, 1996). Open Source advocates argue that proprietary software distributors who do not let users see the underlying source code of their products are practicing the flawed model discredited by Kerckhoffs, engaging in a kind of "security through obscurity" and assuming that would-be attackers will be prevented from mounting effective attacks because they do not have access to the source code (Schneier, 1999).

In addition, software testing that its practitioners intend to use to find bugs and security issues using the whitebox or, as it sometimes called, glassbox method – where testers have access to the source code – is generally more likely to uncover programming problems than blackbox-type testing, in which testers typically have no access to the source code (Freeman, 2002).

And finally, Open Source advocates argue that thousands of people examine Open Source code, so it is therefore less likely to contain flaws or deliberately inserted malevolent capabilities. The official Web site of the Open Source Initiative (http://www.opensource.org/advocacy/faq.php) declares, "Open source promotes software reliability and quality by supporting independent peer review and rapid evolution of source code."

Definitions

There are many different versions of Open Source software, but this paper will be using the term in its broadest sense to simply refer to a computer program whose source code can be inspected by anyone who uses it. This paper will use the term Commercial Off-the-Shelf Software, or COTS, to refer to software that does not offer users that option. Use of the terms in this fashion is not always entirely accurate, because some commercial software allows users to see the source code, but for clarity's sake this paper will avoid referring to such exceptions.

Security vulnerabilities are problems with a computer system that can be exploited by someone, either locally or remotely via a network connection. Vulnerabilities can be software based – often referred to as a bug – and are often repaired with a software "patch" that can be applied to a vulnerable system or program and close that security hole. In addition to vulnerabilities based entirely on software bugs, there are vulnerabilities that exist because of a system's fundamental design. Sometimes the two are closely related; a good design can keep a bug from becoming an exploitable problem, but even good software in a bad design can break down and pose a grave risk to everyone who must use that system.

And finally, this paper will on occasion use the term “trust” in reference to security issues. That is because there is no such thing as a completely secure system, only systems that deserve to be trusted to a greater or lesser extent.

CHAPTER 3: LITERATURE REVIEW, WHAT MAKES SECURE SYSTEMS

A Working Definition

What exactly does it mean to say that a computer is secure? One operational definition (Garfinkel, Spafford, & Schwartz, 2003) simply suggests that a secure system offers the user no surprises: “A computer is secure if you can depend on it and its software to behave as you expect.” (p. 5).

Later, this paper will delve further into the meaning of computer security, but that operational definition will serve for now as the starting point for this paper’s literature review of the significant early work aimed at designing secure computer systems.

Computer security has been a source of concern ever since it was first raised as a serious issue 34 years ago in a then-classified report produced by the Rand Corporation for the Defense Department (Ware, 1970). That document, which declared that computer security could be achieved only through following specific system designs, noted that resolving security issues on any given system required input from a large team of skilled individuals, including programmers, hardware and software engineers, and security specialists.

Amazingly, the field has not progressed much beyond that conclusion today. One could argue that the situation is much worse, because the democratization of computing has forced untrained, unknowing, and uncaring individual computer users to service machines and make security decisions that demand the skills of an enormous team of experts. The vast majority of computer users today must secure their systems by themselves. In this environment, systems that continue to fail should not be considered an anomaly so much as the systems that are operated by untrained users and yet remain secure.

Two years after the Rand study, a significant paper on computer security mechanisms was released (Anderson, 1972). This document was prepared largely because the Air Force foresaw increasing pressure to link its computers into government networks, and because of the growing need to develop computer systems that could protect information from individual users who might exhibit a multitude of different security clearances while all using the same computer system. Such "resource sharing" systems offered profound efficiency benefits, but also threatened the privacy and security that had come to be expected from what were effectively single-user "batch" systems. The new time-sharing systems allowed multiple programs to take up residence in memory simultaneously, allowing a commingling of operations and potential access to data and processes by unauthorized personnel. All this was coupled with a move away from physical confirmation that processes were separate – by mounting different tapes, for example, or running different stacks of punch cards – with a reliance on electronic, or logical, protections that could not be observed by even the most cautious operators.

To make matters worse, experts were horrified to discover that, while the operating system of a time-sharing environment would do its best to prevent an accidental incursion caused by a flawed program, these operating systems did not, as a rule, have any defense against a deliberate attack by a malicious user. For instance, it would be theoretically possible for one user to see another user's password, because passwords are typically reused. Air Force operators had thus been forced to run their computers based on the security level of everything on the system; a high-level classified file, for example, could never reside on a system for lower-level confidential documents because otherwise someone who should only have access to confidential documents might somehow get access to that classified data.

One primary issue underscored by the report was an "all or nothing approach to memory protection" (p. 13). Most significantly, attempts to develop memory protection did not offer complete control, leading to "monolithic, totally privileged" (p. 14) operating systems easily manipulated into granting privileges to objects that should be restricted. The fundamental problem identified over three decades ago is the same issue that haunts us today: Operating systems were not designed to be secure. The paper said a securely designed operating system would have three attributes:

1. A mechanism to control system access.
2. A mechanism for authorization.
3. A controlled sharing mechanism for the execution of all programs, utilizing the other mechanisms.

The paper also made a number of recommendations for further development and deployment of tools such as encryption and specific hardware recommendations. What is most striking about the paper is that its analysis and recommendations could easily be applied to many modern computing systems.

By 1973, the first of the basic modern design principles (Schell, Downey & Popek, 1973) were laid out. The paper, essentially the collected notes from the Air Force's Computer Security Branch from August to November 1972, set down principles that its authors argued should be used in a secure computer system. The report presciently laid out three requirements for such systems:

1. Complete Mediation. The operating system must stand securely between any reference to data and access to that data. Hardware and software in charge of security have to validate all references.
2. Isolation. The items responsible for validation have to be tamperproof and isolated, in addition to providing a unique and foolproof identifier for each user.
3. Simplicity. The security system has to be as small and uncomplicated as possible, so that it can be easily understood, and reasonably, thoroughly tested.

These three principles would later be incorporated into larger, more thorough lists of system design principles that would be developed in the years to come, but history demonstrates that simply understanding the proper principles to designing a secure computer system is no guarantee that anybody is going to implement those principles in practice.

Butler W. Lampson weighed in on data leakage within an operating system, an issue generally related to all these concerns (Lampson, 1973). He theorized that leakage could be limited by eliminating the ability of a confined program to save information inside itself from one call to another, along with several other rules. This is more complex than it sounds, but Lampson clearly illustrated several basic techniques that could be used to steal data. For example, a service running within the operating system might create a temporary file, which, while it would typically be destroyed at the conclusion of that process, could be read while the process is running. Lampson's simple principles, if used, could eliminate these sorts of hazards.

In 1974, two Air Force officers, Paul A. Karger and Roger R. Schell, issued a report in which they evaluated a computer system named MULTICS (1974). They concluded that while the MULTICS system was not certifiably secure – it was not immune to deliberate attack – its design principles were far better than other systems available, and that MULTICS could be used as a base on which to construct what they referred to as a certifiably secure, open use, multilevel system. As an example of the flaws in the system, their team managed to place a "trap door" – what today would be described as a backdoor – into the system. The paper notes that "Tiger Team" exercises in which experts attempt to compromise systems cannot prove those systems are secure, they can only identify vulnerabilities.

The following year, a significant work appeared arguing that systems could and should be made provably secure (Neumann, Robinson, Levitt, Boyer, & Saxena, 1975). This paper, and others that followed (Feiertag & Neumann, 1979; Neumann, Boyer, Feiertag, Levitt, & Robinson, 1980) described a way to use tools such as the Hierarchical Development Methodology to easily and completely describe a computer system in precise detail – as opposed to more abstract proposals, such as those put forth by Bell & La Padula (1975) – thereby avoiding the standard difficulties involved during system design. The Provably Secure Operating System (PSOS) offered a way to make otherwise extremely complicated designs easy to understand and construct, using concepts such as encapsulation and hierarchical development tools. By compressing the process into relatively small modules whose descriptions could fit on several of pages of text instead of hundreds of pages, the PSOS system offered a formal methodology for developing a highly secure system on an architectural level, not by attempting to "patch" something that was already broken.

Over the years, a number of approaches to developing or evaluating secure computers have appeared, such the Defense Department's Orange Book (1985) and the Generally Accepted Systems Security Principles (Ozier, 1997). These fundamental concepts became the foundation for subsequent work in developing secure systems. Unfortunately, however, the issues uncovered, discussed and largely addressed by work done in some cases decades ago remains a largely academic pursuit; the solutions laid out by researchers have been almost completely ignored by the people who design and sell systems. As Peter Neumann (2003) put it in a presentation, it is not solutions to known problems that drive advances in computer development, but rather market forces. For this reason, people who develop computer systems are slow to adopt the solutions developed by security researchers.

What do users want from a secure system?

One standard definition of computer security (Bishop, 2003) holds that it is based on three attributes: confidentiality, integrity, and availability (CIA). Confidentiality refers to the need to keep information or resources a secret from unauthorized parties or systems. This concept includes not only, for example, data itself, which can often be protected with tools including cryptography, but also to knowledge that the data exists. Integrity describes the need to keep data or resources from being altered in some secret or unauthorized fashion using systems designed to prevent or detect alteration. The concept encompasses not only keeping someone from making changes to a database but in guaranteeing that the source of the data is not fraudulent, a concept known as authentication. And finally availability deals with the idea that data and systems must not become inaccessible. A system that reliably provides this CIA triad (Harris, 2003) is said to be secure.

Parker (1998) expands on that triad, offering six foundational elements. In addition to availability, integrity, and confidentiality, Parker puts forth utility (if the information is not in useful form it is worthless), authenticity (the information is valid, as distinguished from integrity, meaning unaltered), and possession (the information is in the owner's possession, as distinguished from confidentiality, which encompasses only what an individual both possesses and knows, but not what an individual can "possess without knowing" (p. 236).

Guaranteeing all this is quite complex, although researchers believe they understand the basic security principles that must be followed when designing an operating system that has a chance of providing those guarantees. Those principles are based largely on work by Saltzer and Schroeder in 1975, drawing in no small measure on the work that went into developing the MULTICS system (Bell & La Padula, 1975). A discussion of those principles follows, relying on their work along with its interpretation by others (Bishop, 2003; Pfleeger & Pfleeger, 2003).

Basic Security Principles:

1. Complete Mediation. Every attempt to access a protocol or memory on the system must be checked to be sure it is allowed. The mechanism developed for this has to be configured so that it cannot be circumvented, and must check for both specific requests and attempts to circumvent the checking. Most systems check whether access is authorized the first time, but do not check on subsequent access attempts, caching the data from the initial access and allowing subsequent requests to the cached data.
2. Economy of Mechanism. The more complex a system, the more difficult it is to find all the problems within it. The security technology should be built to be as small and as simple as possible, which allows for better understanding and therefore fewer surprises.
3. Fail Safe Defaults. The standard rule for granting access to an object should be "deny." That is, no access will be granted unless specifically authorized.
4. Least Common Mechanism. Items used to get to resources should not be shared. Separating resources – either physically or logically – can reduce the risk that sharing powerful resources brings to systems. A Web site flooded with so many requests that customers cannot use it – known as a Denial of Service Attack – is an example of the trouble that failing to understand this rule can cause: Disabling a single aspect of the system incapacitates the entire system because of shared resources.

5. Least Privilege. Programs and users should function with the lowest, fewest, and most separate privilege levels possible, operating with only those needed to finish a specific chore. For instance, on Windows systems, the "administrator" account – the default configuration for a Windows installation – has no access control restrictions; Unix systems have the same vulnerability for users running as "root," although that is not typically a default configuration on many systems
6. Open Design. The utility of a security mechanism cannot depend on its design or implementation remaining a secret. As mentioned in the previous section, security through obscurity cannot be counted on forever. At the same time, however, if the mechanism is strong, keeping it a secret can enhance security, which is discussed later in this paper.
7. Psychological Acceptability. Protection mechanisms or policies that are cumbersome to use will be rejected. For instance, requiring users to change passwords every hour would likely lead to those users choosing poor, easily guessed passwords.
8. Separation of Privilege. The system should not grant access to objects or resources based on a single attribute or condition. For instance, access to a system based on a password combined with a smartcard is more secure than access based on a password alone.

Obviously these principles are idealized and some even contradict each other. For instance, a pure interpretation of the Principle of Complete Mediation would suggest that users should enter a password each time their e-mail application attempted to log in to the mail server to check for new mail; in such instances it is likely that the Principle of Psychological Acceptability would take precedence to avoid angering users.

The genesis of these principles dates back to some of the earliest literature on the subject. Indeed, Neumann (2004) points out that, despite all the years of research that have specifically identified preventative techniques to keep security issues from having an impact on us, the fundamental problems remain. For instance, identification and authentication remain significant issues on our systems, with no serious efforts underway to resolve that issue, despite the fact that many solutions – such as non-reusable passwords – are known and have even been implemented in many different settings. The problem of monolithic systems and access controls remains and continues to make it difficult to employ techniques such as context-sensitive authorization. Insufficient run-time validation, using techniques such as boundary checking, allows the installation of nefarious tools such as Trojan Horses and sniffers onto systems, corrupting an operating system. Lack of encapsulation allows leakage of sensitive material. Simple programming errors, such as off-by-one, in which, for example, a programmer erroneously thinks he has written code that starts counting at zero, but actually starts counting at one, remain distressingly common.

While experts have developed better tools that let people avoid making some of these mistakes in the development process and can help people find such errors once they have already been made, the real answer to problems of this nature is to offer better designs for the system. Fundamentally, the flaws we are most concerned about these days fail on a systemic level, and must be attacked on that level as well if experts are to successfully create systems that users can trust will offer no surprises.

Finally, it is worth noting that many of what are considered the most highly secure systems, which actually implemented some of these principles, were not Open Source products. Such systems include the SCOMP system developed by what is today named Honeywell International Incorporated, Gemini Computers Incorporated’s GEMSOS, and MLS LAN from the Boeing Company, were written to highly secure specifications. These systems were all either proprietary or classified (Smith, 2001).

CHAPTER 4: THEORETICAL ARGUMENTS FOR OPEN SOURCE

Having explored the basic ideas behind making software and operating systems more secure, defined what is meant by secure software, and explored existing issues that tend to make software and operating systems insecure, this paper will turn back to the argument that Open Source software is more secure than commercial-off-the-shelf software (COTS).

The theoretical aspects of this position, according to dozens of works (Asiri, 2003), can be stated as: Open Source software is more secure than COTS products because thousands of skilled programmers are able to examine Open Source code both for flaws created accidentally by bad programming and for the deliberate insertion of malicious code, such as a backdoor. Once these flaws are identified, the Open Source community develops patches for these problems – typically far more rapidly than companies selling COTS products – and makes those patches available for users to apply to their systems. In contrast, no one really knows what is going on inside COTS software, because users cannot see inside it.

As an example, Cisco Systems announced on April 7, 2004, that hidden within some of its products, including a widely used wireless networking device, is a default username and password. Anyone who knows those two pieces of information can gain complete control over the device, which means they could compromise the integrity of any system or network on which it is running. Even more shocking, however, was the declaration from Cisco that this backdoor cannot be removed or disabled (Cisco, 2004).

Open Source advocates point to such incidents as evidence that COTS software simply cannot be trusted, while Open Source software is inherently trustworthy. For example, Wheeler (2003) relates the tale of Borland Software Corporation's Interbase Server database product. At some point between 1992 and 1994, the company placed a "backdoor" into the product, inserting, as did Cisco, a username and password combination. Such capabilities are usually built in to products to allow customers to access data in the event of, for example, a lost password, but backdoors are considered severe security risks because a single breach – an angry employee giving the password away via the Internet, for example – can compromise everyone using that technology. The Borland product was sold and used for six years with no public acknowledgement of this situation, and, because Borland did not initially make the source code available to those outside the company, no one who did not work for Borland knew about it. But in July 2000, the company decided to release the project's source code under a standard Open Source license. Five months later, an Open Source product review team identified the flaw and on January 10, 2001, the Computer Emergency Response Team Coordination Center posted an alert (CERT advisory, 2001a). The Open Source community soon developed a patch for the problem, securing the software.

Searching for bugs in software code that could lead to surprising behavior is a fairly complex undertaking; some bugs are deep, or hard to find, and some are shallow, or easy to spot. Open Source advocates say that "Given enough eyeballs, all bugs are shallow" (Raymond, 2000, p. "Release Early, Release Often"), by which they mean that there are thousands of people around the world who examine Open Source code, searching for flaws, reporting them, and helping writing patches for those holes, which makes Open Source software more bug-free and, thus, more secure.

Despite these claims – which remain something of a religious test among many in the technology community – the general consensus among experts is that, at least right now, Open Source software is not more secure than COTS products, though the vast majority of these individuals believe that Open Source does have the potential to provide more secure software to everyone (Lawrie & Gacek, 2002).

To understand why, it may be useful to refer to this old maxim: In theory there is not much difference between practice and theory, but in practice, there are many substantial differences.

Marcus Ranum, President of NFR Security, argues that the "many eyes" theory breaks down in practice (Lawton, 2002). "In my experience, not many people actually read through the code. They just look at the readme files. When I wrote the first public firewall toolkit, there were maybe 2000 sites using it to some degree, but only about 10 people gave me any feedback or patches. So I am way unimpressed with the concept of Open Source." (p. 19). Many say it is the quality, rather than the quantity, of people who look for security issues that helps create secure code (Lawton, 2002).

Even individuals committed to the Open Source movement say Open Source software can be as vulnerable to programming errors that lead to security lapses as programs designed using the traditional methods used by COTS developers. For instance, John Viega built Mailman, a mailing list manager used in the Open Source GNU universe. According to Viega, the program had substantial and obvious problems with its code, but no one ever reported the errors, or, to his knowledge, even noticed that there were problems (Whitlock, 2001). The program was downloaded thousands of times and was included in many versions of the Linux environment that people actually paid for. According to Viega's analysis, Mailman users believed that because the application's source code was available for inspection, it had therefore been inspected and could then be relied upon as secure.

"The benefits open source provides in terms of security are vastly overrated, because there is not as much high-quality auditing as people believe, and because many security problems are much more difficult to find than people realize" (p. 3), Viega said. Steven Lipner (2000), agrees, saying of his time with a company that shipped a commercial product along with source code, "Our experience was that people were too busy using the product to review its source code" (pp. 124-125), although it must be noted that Lipner is a Microsoft Corporation executive. Microsoft has a long history of hostility toward the Open Source movement (Halloween Document, 1998), because the company views the movement as a threat to its continued dominance of the desktop computer market in which it currently holds a 95 percent share (Geer, D., Bace, R., Gutman, P., Metzger, P., Pfleeger, C. P., Quarterman, J. S. & Schneier, B. 2003). Even so, Lipner's basic points cannot be reasonably challenged: "There is no question that source code review is a key component of building a secure system. However, simply making source code available for review does not guarantee that an effective security review will occur. Builders of secure systems must ensure that their security test and review teams have the resources and the motivation to perform a thorough and effective test of the system" (Lipner, p. 125).

Viega argues that, to the extent that Open Source software users examine the code at all, they usually do so to only to make sure that it meets their specific needs, not to do a broad analysis that would be useful to anyone but them or their organization (Whitlock, 2001). While this process can and does lead to discovery and repair of security flaws useful to the broader community of people who use that particular bit of software, in general that is not how things work out. In large part, Viega argues, that is because many of the people doing the code review are rank amateurs, unfamiliar with even basic security issues. (Viega, 1999). As Viega puts it, what good are many eyeballs if they cannot even see a bug, whether shallow and thus supposedly easy to spot, or deep and therefore difficult for even a skilled programmer to ferret out?

In his examination of the sociology of the Open Source movement, Glass (2003) notes that at the core of the Open Source ideology is the notion of self-reliance, the idea that if a user thinks the software the user needs to run is insecure, the user can fix it. "This works just fine if the users are programmers, but if they are not this is simply a technical impossibility" (p. 22). Glass goes on to note that even when skilled programmers peruse Open Source code, make changes, and distribute those changes to the rest of the world, that may not mean a lot to the vast majority of computer users. That is because the largest share of programs used by programmers – and thus scrutinized for problems – are what are generally known as "system" programs, such as compilers, or, more generically, the tools and environments used to develop other programs. Glass points out that programmers do not usually make use of what is by far the largest category of computer software programs: business applications related to tasks such as bookkeeping and payroll data. Nor are programmers likely to use the next largest category, scientific and engineering applications. Glass argues convincingly that, with the exception of the operating system, programmers generally do not look at any code in any software that is likely to actually be used by consumers. "Thus the percentage of software likely to be subject to Open Source reading is at best quite small" (p. 23).

And simply reading the code for potential flaws in the absence of a rigorous security testing program can be a pointless exercise, as Kenneth Thompson's seminal paper of two decades ago illustrates. Thompson modified a compiler – used to translate source code into machine code – and used this Trojan Horse modification to further modify other programs written in the C programming language and run through the altered compiler, thereby using a "trusted" compiler to create a security hole in certain types of programs run through the compiler (Thompson, 1985). Thompson's conclusion was directly applicable to the current discussion. Looking at source code created by someone else does not help identify security issues, he wrote:

> No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well installed microcode bug will be almost impossible to detect. (p. 763)

Would better software testing help?

Even so, few would want to return to the days when running a program on a computer meant the user had to first create the program. So users rely on programs developed by other people but also assume that procedures such as software testing have demonstrated that code meets security requirements (Bishop, 2003; Pfleeger & Pfleeger, 2003). While testing cannot prove the absence of flaws, it can demonstrate that the product is performing in agreement with a specification. Software tests are performed from two perspectives: Blackbox, or functional testing, and whitebox, sometimes referred to as structural or glassbox testing. Blackbox testers do not have access to the source code. In that sense they are similar to outsiders testing COTS software. And whitebox testers, along with those who use Open Source software, do have access to the source code. A good software development team tests all aspects of software, but as Bishop notes, security testing, whether blackbox or whitebox, is special because of its "focus, coverage, and depth" (p. 534). While typical software testing focuses on the sorts of things that are most likely to happen, security testing must focus on the things that are least likely to happen, along with the least-used capabilities, devices, and mechanisms of the software and whatever environment it may be expected to run in. According to Drabick (2004), too many developers look on testing as a phase, instead of a carefully crafted, well-managed system.

Whitebox testing is demonstrably more effective at identifying software flaws than blackbox testing (Pfleeger & Pfleeger, 2003), but blackbox testing is more likely to be used because it is less demanding and less expensive (Freeman, 2002). All types of testing require a methodical, systemic program and knowledgeable testers (Whittaker, 2003) because those testers must know exactly what sort of behavior they are trying to identify, and be able to spot it when it occurs.

Largely because of competitive pressures that force rapid distribution of new products – and an apparent disinterest in the market for paying extra for more secure software – the vast majority of commercial software is only tested for features, not security (Hoglund, 2004). As a result, the increasing complexity, extensibility and connectivity of software are leading to more significant security issues appearing with greater frequency; flaws that are trivial in a standalone computer become critical if that computer is connected to the Internet, an example of how increasing connectivity makes security harder to obtain. Extensibility refers to the ability of programs to run what is known as untrusted mobile code, such as Sun Microsystems Incorporated’s Java, or Microsoft Corporation’s .NET software; other, less savory examples of mobile code include viruses and worms.

Turning now to complexity, the easiest predictor of how many bugs exist in a given piece of code is to simply find out how many lines of code are in the program. A system that has undergone a very rigorous quality assurance testing program might have perhaps five bugs for every 1,000 lines of code, but most commercial code has, on average, about 50 bugs for every 1,000 lines of code (Hoglund, 2004).

Put simply, the more lines of code in a program, the more bugs there are, and the more surprises for users. When Microsoft Corporation introduced Windows 3.1 in 1990, the product was a relatively large 3.1 million lines of code, but by 2002, Windows XP had grown to enormous proportions, containing 40 million lines of code. What all this means is that even people who are actually interested in performing a rigorous security test are finding it increasingly difficult to methodically investigate every possible line of attack. Remember that performing an adequate security review involves more than simply carrying out random attacks on the program being tested; without a solid methodology, testers are wasting time and money. Successful testing, according to Pfleeger & Pfleeger (2003) lets the testers spot trapdoors, Trojan Horses, viruses, worms, and a host of other security issues, especially when the components are tiny and encapsulated. Unfortunately, current systems are so large and so complex that simply trying to design a valid test procedure has become extremely difficult.

In his article, “What is software testing? And why is it so hard?” Whittaker (2000) says there are a number of answers to those questions. In some cases, the code itself is fine, but the order in which the code is executed makes all the difference; this is yet another reason why simply looking at the source code will not eliminate all possible surprises. In addition, there are so many variables – users can input billions of different combinations – that there is simply no way to test them all. He argues that the only reasonable way to perform a valid test is to set up a four phase process, consisting of modeling the environment the software will run in, selecting test scenarios, running and evaluating those scenarios, and finally measuring the progress of the test.

This, as Whittaker makes clear, takes a great deal of work. When using a blackbox method because the source code is not available, he says, it is likely that whatever security test is performed will not be especially reliable.

Fortunately, the Open Source community does have access to the source code – that is a defining characteristic of the movement – so Open Source participants could be running a large number of labor intensive, but highly effective, white box security tests on their software.

Only they are not. Attempts to set up formal, structured review processes – the only kind that are actually effective at doing security testing, according to Whittaker – have come to naught. For example, the Linux Security Audit Project was supposed to help coordinate security reviews of the Linux system among the diverse group of people all over the world who contribute to Linux development (Lemos, 2002). In the end, however, little code was actually reviewed. The site's database of programs that have been tested contains no data (LSAP, 2004).

A federally-funded initiative named the Sardonix Audit Portal and bankrolled by the Defense Advanced Research Project Agency – the lineal descendant of the same organization that funded research into the technologies that led to the Internet – was supposed to establish a formal testing structure for Linux. But Sardonix completely collapsed two years after its launch because of lack of interest, or more precisely, a lack of productive interest (Poulson, 2004). Sardonix was set up for one purpose, according to one of its founders, Crispin Cowan. "It is my belief that the programs are getting audited a lot less than people think" (Lemos, p.1), he said at the time the project was launched.

"The promise of open source is that it enables many eyes to look at the code, but in reality, that doesn't happen," (Lemos, p. 1). The Sardonix project offered, not only an organizational system, but a suite of tools to help participants examine code for security issues and to help participants write better code. This was important because a part of the problem in the Open Source movement is a lack of "traditional software engineering tools, methods, and techniques" (Lawrie & Gacek, 2002, p. 35) that can be used for such endeavors.

In the end, Crispin argued, the project failed because the Open Source development community, which prides itself on being a largely philanthropic endeavor, is actually a rather narcissistic enterprise (Poulson, 2004). Participants seek, not the good feeling that comes from knowing that they have helped make the world a better – or safer – place, but recognition and even a measure of fame for that contribution. Crispin said he dealt with an enormous number of people debating the design and eventually implementing the structure built to recognize top performers, but got "squat" in terms of actual code reviewed.

But others argue that Sardonix might have been more successful if the project had gotten more publicity; certainly some people who might have participated never even heard of the project during its two-year existence (Flynn, 2004). More to the point, the system was never designed to take advantage of the forces that drive Open Source programmers: Solving a software problem either for a business or for themselves. Many of these people do indeed publicly distribute information they uncover about vulnerabilities online and bask in the recognition from their peers. "But many sit on their findings, because kudos on a mailing list or a software auditing website can never compare to the reward of unauthorized access to a high-profile system. Sardonix had nothing to offer either variety of auditor," (Flynn, p. 1). Flynn concludes his analysis by writing, "The success of Sardonix would have proved a key argument that open-source advocates have used to lend validity to the cause since time immemorial: that open-source software is more secure because the source is available to the world to be audited. The project's failure is a reminder that the statement is a myth" (p. 1).

And finally, some argue that testing is not really the answer. They say that testing is inefficient, expensive, and, because it cannot prove that no flaws exist, ultimately futile. Rather than testing for bugs, software developers, whether open-source or COTS, should adopt new techniques that would prevent bugs from ever being created in code. For example, the most common type of software bug that can be exploited by an intruder is called a buffer overflow, which can break through standard security techniques if a malevolent user feeds a program a larger input than was expected. Rather than using code inspection to ferret out such issues, Schneider (2000) points out that use of better programming languages would simply prevent such bugs from ever being successfully set into code at all. He argues that the issue of whether or not to make code public is a red herring. For instance, Microsoft Corporation's Windows XP operating system comes with a firewall – a tool designed to prevent outsiders from breaking in to the system – but that feature is turned off by default because the firewall can cause some programs that rely on access to the Internet to malfunction if the firewall is not configured to allow that behavior.

As Schneider writes:

Systems today are compromised not only because hackers exploit bugs but for prosaic reasons: initial configurations are shipped with defenses disabled, administrators misconfigure their systems, and users find it inconvenient to set protections for their files or encrypt their data. Some of these vulnerabilities would be addressed with easier-to-use security functionality while others require a new culture of caution and care amongst users. None has anything to do with whether a system is open-source or closed-source. Thus if bugs in the code base are not today the dominant avenue of attack, then embracing Open Source software is applying grease to the wrong wheel (p. 126).

CHAPTER 5: COULD MAKING CODE AVAILABLE HARM SECURITY?

While there are theoretical arguments in favor of allowing source code to be examined by users – the primary one being that the code can be examined for security – there are theoretical arguments against it as well. The primary argument against allowing people to see source code is that while it gives people whose only interest is in making software secure a chance to review and fix any security problems, it also offers would-be intruders a chance to see and exploit those vulnerabilities. As a controversial study issued by the Alexis de Tocqueville Institution put it: Why offer potential intruders the opportunity to study the basic blueprint of one's information security structure by using computer code that the entire world has seen? (Brown, 2002).

Neumann (2000) argues that if a computer system is already relatively secure, opening up the code to public inspection will not do anything but help the good guys and will not provide any meaningful help to would-be bad guys. But he glumly notes that most systems today are not secure, victims of both abysmal designs that ignore everything researchers have come to understand about building solid programs, and dreadful implementation with specifications that are both unfinished and simply unworkable.

Schneider (2000) argues forcefully that the very nature of the Open Source development process makes it more likely that code designed to offer users unpleasant surprises will inevitably be inserted by nefarious participants in the Open Source development process. Our methods of code analysis remain inadequate at best, so one way commercial software developers try to minimize the chance that a rogue employee will insert something such as a Trojan Horse or a backdoor into a product is by exerting significant control over the actual process of constructing the software, being careful who they hire to code, and doing more than paying lip service to employee morale. The unstructured nature of the Open Source development process, where anybody with a good idea is welcome to submit code to the project, means that strangers are building systems that may someday be trusted to perform extremely sensitive functions. “Suggesting that software be inspected for Trojan Horses misses the point,” (p. 127) Schneider dryly concludes.

Indeed, there is ample evidence that people have repeatedly attempted to insert nefarious code into Open Source projects, though to the best of anyone’s knowledge, none of these attempts has been successful on a wide scale. But in 2003, security around Linux development projects was repeatedly breached, apparently with the intention of altering the software’s code, presumably to insert some sort of backdoor (Lemo, 2003). It is largely for these reasons that some people continue to fear that Open Source programs or operating systems could have a backdoor inserted into them (Lawton, 2002).

And finally, despite the concern over “security through obscurity” as a primary means of ensuring safety, there is some evidence to suggest that at least trying to keep source code a secret does indeed make exploitation of software vulnerabilities more difficult, or at least less likely. In February of 2004, portions of the source code from older versions of Microsoft Corporation’s Windows operating system were distributed on the Internet. Within weeks of the release, an exploitable security flaw was found in the code (Seltzer, 2004) and a tool designed to exploit the flaw was posted on the Internet, although it was apparently never used because the flaw had been repaired in modern versions of the software and so was not actually exploitable, at least not on a wide scale. That flaw had been in existence for more than three years, but nobody noticed it – at least not publicly – until the source code was released. While Open Source advocates might say that flaw was a time bomb waiting to go off and would likely have been caught if the code had been released to the public years ago, Seltzer argues convincingly that, at the time the code was released, the hazards of that particular vulnerability were not recognized, and thus the code would likely have passed scrutiny from even the most skilled examiner, whether or not the Open Source community had access to the code.

One of the most interesting arguments about whether distributing source code helps or harms security comes from Cambridge scientist Ross Anderson, who, with his paper presented at the Open Source Software: Economics, Law and Policy in Toulouse, France (2002) laid out a framework of mathematical analysis that found that both Open Source and COTS programs are about equally secure, at least in the idealized mathematical representation he set up to analyze each. Laying out a system in which, essentially, bugs are easier to find in Open Source software and more difficult to find in COTS, Anderson concluded that from a security perspective, there really is no difference between Open Source and COTS. Distributing source code, which should make it easier to find bugs, does indeed help good guys find and repair bugs but also helps bad guys exploit them. At the same time, keeping the source code a secret can hinder the bad guys, but it also hurts the good guys. "In a perfect world, and for systems large and complex enough for statistical systems to apply, the attack and defense are helped equally. Whether systems are open or closed makes no difference in the long run" (p. 10). But Anderson cheerfully concedes that his analysis presents an ideal world, and the real world is far from ideal. The next and final section in this chapter moves this paper away from theoretical analysis of these issues and examines a series of attempts to perform empirical research on these questions.

What the data show

Researchers have been trying to measure whether Open Source or COTS software is more secure, using various schemes, for several years. Many of these attempts have stumbled in part because of difficulties in data interpretation. For example, James Middleton's work (2002) declared that "Windows suffered fewer vulnerabilities than Linux last year," when Microsoft Corporation's Windows NT/2000 operating system had 42 vulnerabilities, while most of the other Linux distributions, such as those distributed by Red Hat, had much fewer. In this analysis, as in several others, vulnerabilities in different Linux distributions appeared to have been counted at least twice, once for each time they appeared in any Linux distribution (Wheeler, 2003). Thomson (2002) appears to make a similar error, counting all vulnerabilities across every Linux variant as a separate vulnerability, thereby counting each Linux bug at least twice.

Others have tried to take a different approach, arguing that all software has vulnerabilities and that thus one practical measure of how secure any given system is would be to measure how quickly those vulnerabilities are patched after they have been discovered.

John Reinke and Hossein Saiedian (2003) used the advisories put out by the Computer Emergency Response Team Coordination Center at Carnegie Mellon University, which were only issued for vulnerabilities thought to present serious risk. The two researchers collected data for a four year period, from 1999 to 2002, and cataloged each announcement as affecting an Open Source product or a COTS product, and then noted how long it took for a patch to be released. Their data showed that 61 percent of flaws related to advisories for COTS software were fixed, versus 71 percent for Open Source software. In addition, it took COTS vendors over 52 days on average to provide those fixes, versus nearly 29 days for the Open Source developers.

While these numbers would seem to suggest that indeed the Open Source community can and does produces software patches for security issues far more rapidly than does a COTS vendor, the numbers used are in reality somewhat dubious. Up until Oct. 9, 2000, CERT/CC did not post any kind of missive about any vulnerability unless a patch or some jury-rigged repair was available. After that date, CERT/CC began disclosing vulnerabilities 45 days after receiving the initial report, whether there was a patch available or not (CERT, 2002). CERT/CC changed this policy after years of increasingly angry protests from the security community, which had come to see the public acknowledgment of bugs in software as the only way to pressure software developers to release patches for their code (Levy, E., personal communication, Dec. 6, 2001). As the years went on, many security experts simply stopped reporting bugs to

CERT/CC, because many times nothing ever seemed to happen, or else did not happen for months or years; many of those who disaffected started contributing to full disclosure lists such as BUGTRAQ. CERT/CC finally agreed to publish vulnerabilities after a suitable waiting period to give vendors time to prepare a patch, but by the time that policy change was made, CERT/CC had largely lost its central position as most important disseminator of software vulnerability information that it had once enjoyed. For these reasons, the data used in that study, and thus the study's conclusions, should not be relied upon.

A study released by Forrester Research Inc. (Koetzle, 2004), addressed the same sorts of question, using a wide variety of data sources, and developed a number of metrics including:

1. How many days passed between the time a vulnerability was publicly disclosed and the group responsible for maintaining the platform issued a fix? This figure, named "All Days of Risk," was calculated for Microsoft products and for various varieties of Linux. (A similar metric dubbed "Distribution Days of Risk" was used to take into account the unique nature of the Linux software development and distribution networks).

2. "Flaws fixed" was calculated based on the number of vulnerability holes plugged.

3. "Percentage of high-security vulnerabilities" was an attempt to calculate the relative severity of a vulnerability. A vulnerability was classified as "high" if it met one of three criteria: It would allow a remote hacker to open an account on a system; it allowed a local user to gain complete control over the system; or the Computer

Emergency Response Team Coordination Center issued an advisory. These criteria are used by the U.S. National Institute for Standards and Technology's ICAT project, a searchable index of security vulnerabilities.

The evaluation period ran from June 1, 2002 to May 31, 2003. Analysts cross-referenced all security bulletins issued by every major software developer in the study: Debian, MandrakeSoft, Microsoft, Red Hat, and SUSE. And they also used an astonishing amount of data, available from public sources, including:

> The various Full Disclosure mailing lists (i.e., Bugtraq, NTBugtraq, Vulnwatch, Vulndiscuss, etc.), the various SecurityFocus mailing lists (i.e., FOCUS-MS, FOCUS-Linux, Incidents, Vuln-dev, etc.), the Neohapsis mailing list archives, the MARC mailing list archives at marc.theaimsgroup.com, LWN.net, bugzilla.org, freshmeat.net, rpmfind.net, the CVE dictionary and CVE reference maps (available at http://cve.mitre.org), the ICAT project at NIST, (available at http://icat.nist.gov), CERT/CC at Carnegie Mellon University, the security bulletins released by Debian, MandrakeSoft, Microsoft, Red Hat, and SUSE, the security and bugs mailing lists run by Debian, MandrakeSoft, Red Hat, and SUSE, component maintainers' security bulletins (i.e., kde.org for the Linux KDE GUI components, the Internet Systems Consortium (ISC) for BIND, tcpdump.org for tcpdump, the Massachusetts Institute of Technology for Kerberos, etc), security research organizations' security bulletins (i.e., @stake, iDEFENSE, Internet Security Systems (ISS) X-Force, etc.), and the change logs for the software packages the platform maintainers released. (Koetzle, p. 11)

All this data was cross-checked and verified by the various vendors. The study found that Microsoft had the lowest "days of risk," with an average of 25 days between disclosure and fix. The company also resolved all 128 of the vulnerabilities announced during the time of the study, topping that category as well. But nearly 70 percent of Microsoft's vulnerabilities were in the high severity category, meaning Microsoft had the worst performance of any other group in that category. While researchers found that Red Hat users were exposed to 57 days of risk – tying with fellow Linux variant Debian – it tied Microsoft by resolving all but one of its 229 vulnerabilities, and only 56 percent of those were high severity.

The Linux developers took strong exception to the conclusions, however, issuing a joint statement challenging the Forrester report (Common statement, 2004). The statement argued that the Forrester report did not take into account the fact that their developers deal with security issues based on their severity, and pointed out that for extremely important problems patches have been developed and made available for distribution in a few hours. Less important vulnerabilities, the statement said, are often delayed to work on more pressing matters.

Thus, there has not been a great deal of highly reliable scientific analysis performed that can support the claim that Open Source software is more (or less) secure than COTS software. But a study prepared by Steven M. Christey (2002) does offer up some useful information. Christey used data from the CVE, or Common Vulnerabilities Exposures, a list or dictionary that tries to establish common names for publicly recognized security vulnerabilities to develop a list of the ten most commonly reported vulnerability types, ranked by overall risk, and cross referenced by their incidence in both Open Source and COTS products (Table 1).

Table 1:

Ten Most Commonly Reported Vulnerability Types, by overall rank, rank in Open Source software and rank in COTS software

Overall Rank	Flaw type	Overall Percent	Open Src. Rank	COTS Rank
1	Buffer overflow	21.8%	1	1
2	Directory Traversal	6.8%	11	14
3	"Malformed input"	5.8%	6	2
4	Shell Metacharacters	4.4%	5	7
5	Symlink Following	3.6%	2	10
6	Privilege Handling	3.5%	4	3
7	Cross-site scripting	3.1%	8	13
8	Cryptographic error	2.9%	13	11
9	Format strings	2.8%	3	12
10	Bad permissions	2.4%	7	5

Note that by far the most common overall type of security flaw, a buffer overflow, is the most common in both Open Source programs and COTS programs. This is the first solid evidence to suggest that there is no significant difference in security between Open Source and COTS software, at least in terms of the vulnerabilities they are prone to. After that first entry, however, the frequency of the type of vulnerability begins to diverge. Christey notes that some of these differences are most likely caused by distinct architectural issues, such as those between the Windows and Linux platforms. Christey also points out that issues such as malformed input errors could be reported more frequently in COTS because of the techniques used in blackbox testing. "This *migh*t suggest that the auditing of open source products relies more on source code analysis than dynamic testing, but it is unclear" (Christey). Christey suggests that additional research be undertaken to better understand exactly what is occurring.

What these studies have in common is their reliance on the best objective information available, usually databases that classify vulnerabilities by type – such as buffer overflow – and "severity." There are obvious difficulties with the nature of classification type, but the more compelling questions revolve around the concept of severity.

At the moment, severity is classified by the amount of damage a vulnerability exposes the user to, if that vulnerability were fully exploited. In general, vulnerabilities that would, for instance, allow an intruder to take complete control of a system, permitting the destruction of data or even the reformatting of a hard drive, are considered more critical than vulnerabilities that might perhaps expose a user to, for example, a privacy invasion.

But what is missing from the concept of severity, as it is currently constructed, is any attempt to evaluate how likely it is that the vulnerability will be exploited at all. This paper submits that a flaw that could conceivably allow a user's hard drive to be reformatted, but which, for various reasons is unlikely to ever be actually exploited – at least in its most severe form – is not as severe a threat as a vulnerability that could be easily exploited yet would not expose the user to as much damage. Thus, the entire "severity" metric commonly used to label what the greater threats are, which patches for which holes should get priority, and what should get patched is flawed.

What needs to be done is to use the concept of risk analysis to more precisely identify and explain what is being examined. Utilizing the traditional methodology (Peltier, 2001), once a vulnerability is identified, risk assessment involves a calculation of the likelihood that an attack exploiting the vulnerability would be successful. Accurately quantifying such things is quite difficult, but not impossible. Hogland & McGraw (2004) suggest that instead of using actual numbers, which could grow to be imprecise to the point of pointlessness, categorize risks as High, Medium, and Low, based on the chances of a successful exploitation of the flaw.

This paper would also propose weighting the scale depending on the nature of the population using the assets. Damage potential is higher with a group of non-technical users or young children, for example, than with a group of well-trained and responsible adults. Add in three categories of users to the equation – call them Expert User, Knowledgeable User, and Novice User – and attach numbers to them. For example, the highly skilled Expert User gets a value of .2, Knowledgeable User is assigned .6, and Novice User, who likes to do things such as set his password to "password," gets a weighting of .9.

These weightings are proposed in no small measure because of the comparative difficulty that Open Source products pose for users. As Glass (2003) notes, Open Source products are often built by extremely skilled programmers for use by extremely skilled programmers. As a result, many Open Source products are difficult even for a knowledgeable user to properly configure. Even Open Source evangelist Eric S. Raymond (2004) has noted that just doing something as trivial as installing a printer in a Linux environment can prove extremely challenging. Such tasks are typically much easier using COTS products, such as Microsoft Corporation's Windows operating system. If installing a printer is difficult on Linux, imagine how challenging it would be for a less-knowledgeable user to properly secure the system.

CHAPTER 6: EXPERIENTIAL ARGUMENTS

Market Forces

This bulletin went out over the Reuters news service at 3:55 PM in the afternoon on April 13: “SEATTLE - Microsoft Corp. (NasdaqNM:MSFT - news), the world's largest software maker, warned on Tuesday that three "critical"-rated flaws in the Windows operating system and other programs could allow hackers to sneak into personal computers and snoop on sensitive data,” (Stevenson, 2004). The story was mentioned in most major newspapers, by television journalists, and on the radio. Despite all the attention, such announcements are not unusual. Microsoft typically announces several critical vulnerabilities each month.

The popular press has, not unjustifiably, painted Microsoft Corporation’s flagship product, the Windows XP operating system, as a disaster waiting to happen. “Windows XP on the Internet amounts to a car parked in a bad part of town, with the doors unlocked, the key in the ignition and a Post-It note on the dashboard saying, "Please do not steal this"” (Pegoraro, 2003, p. F6). Pegoraro, of *The Washington Post,* correctly notes that XP ships with several unnecessary “ports” – essentially doorways through which data can come and go between the computer and network – activated and open to traffic by default. In addition, while XP comes with a firewall, which can control port activity and thus help prevent intruders from engaging in nefarious behavior, the firewall is off by default; users must find it and activate it.

The point here is not that Windows cannot be configured to operate in a relatively secure fashion – it can – but that users must make an effort to set up a secure configuration. Accomplishing that, however, requires the average consumer to make choices that many are ill-equipped to decide, because even specialists do not always completely understand the implications of security settings on their computers.

Microsoft avoided doing things such as turning on the firewall by default and closing those ports because doing otherwise can cause programs to fail, which then causes users to complain to Microsoft. But the frequent barrages of negative publicity about the difficulties that the average person has trying to safely run Windows XP in a networked environment have caused the company to rethink that strategy. In 2004 Microsoft will release Windows XP Service Pack 2, a modification for the XP operating system designed largely to enhance security for users by doing things such as turning on the firewall automatically (Windows XP, 2004).

The media have also become adept at alerting the population to outbreaks of new worms and viruses that regularly sweep through computers equipped with Microsoft's products. This problem is again abetted in no small part by Microsoft's design choices, such as the default configuration in Windows allowing the user to run as "administrator" of the system, making any kind of infestation much easier for any would-be intruder. Current estimates suggest that perhaps a dozen new viruses and worms are released onto the Internet each day, though the vast majority do not spread (Chen, 2003).

One would think, based on the popular press, that Windows is the only operating system available. While it is quite possible that eliminating other operating systems is the ultimate goal of Microsoft, which is currently fighting antitrust battles on several continents, today there are desirable alternatives to that company's operating systems and software. But that software is not immune to the same issues that plague Microsoft's products.

Open Source software, such as the Linux system, is regularly found to have security flaws. As with Windows, when a vulnerability is discovered, developers create patches for those holes; users download and install the patches to repair the vulnerability in their systems. In general, however, the media almost entirely ignore security issues relating to Open Source software, such as Linux.

There are exceptions to this rule, of course. For example, a series of attacks in February and March 2004 on many non-Windows machines received extensive coverage in *The Washington Post.* (Krebs, 2004). While in general the rule in newspapers is that consumers do not care about computer security issues that are not directly related to Windows, the Krebs story skirted that prohibition by suggestion that the compromised machines at federal laboratories and universities with Defense Department contracts were being taken over in order to mount an attack on Windows machines. The article notes that many of the attacks were accomplished through known security holes in systems such as Linux, though the paper had never mentioned them before. It had, however, given prominent play to the Windows vulnerabilities announced by Microsoft Corporation in April 2004 (Musgrove, 2004).

This paper is not suggesting some vast conspiracy to undermine confidence in the Microsoft Corporation or downplay security issues in Linux. At least not among the media. The principle of Occam's Razor suggests that all other things being equal, when considering two theories as explanations for behavior, the simplest explanation is likely to be correct. Based on that principle, a vast conspiracy among hundreds of media organizations and thousands of their employees is less likely than the more prosaic explanation: Media companies respond to market forces, and market forces drive their computer security coverage. It is important to remember that when this paper refers to media, this paper is referring to mass media, the members of which are enterprises – television and radio networks, large newspapers, and magazines – whose owners hope to attract the largest share of potential customers they can entice to use their wares. Thus the mass media are more inclined to follow stories relevant to the computers that 95 percent of their prospective readership are likely to use, Microsoft Corporation's Windows operating system, than a comparatively obscure operating system such as Linux.

While certainly many worms and viruses are targeted at Windows systems over other software because it is easy to attack, by aiming malicious software at the environment that currently controls 95 percent of the desktop computer environment – Microsoft Corporation's Windows operating system – the creator of a malicious program increases the mathematical odds that the program will successfully propagate onto other systems.

While many partisans claim that attacking Unix-like systems is much harder than writing a tool that exploits Windows code, the enormous number and widespread use of what are known as "root kits," primarily aimed a Unix and Unix-like systems suggests otherwise. While there are no firm numbers available, anecdotal evidence strongly suggests that while a Windows system may commonly fall victim to a worm or a virus, systems such as Linux are frequently compromised through the use of automated attack tools such as root kits, and indeed such compromises in the Linux environment are far more common proportionally than the sort of compromise that occurs in a Windows environment caused by a worm or a virus.

Got Root?

"Root" is the most powerful position to hold for a user on a Unix computer. Allowing an intruder to "get root" on a system means the intruder is completely unconstrained by setting or policy. Root kits are a bundle of computer programs that make up a kind of automated attack tool box which, when launched at a targeted system, will systematically probe for weaknesses, begin an attack, cover up the evidence of the intrusion, and usually install backdoors that will give anybody who knows about the intrusion complete control over the system. Winfried E. Kuhnhauser (2004) says such attacks have "high success probability" (p. 12).

According to Kuhnhauser, a typical attack, including the break-in, insertion of backdoors, and removal of evidence, takes only a few seconds, after which the system looks to be perfectly normal, even if a knowledgeable person examines it. The root kits succeed because operating systems and other types of code either have flaws in the specification or implementation errors, the same issues the field of computer security has been wrestling with for over three decades. "As long as such exploitable flaws and errors persist there will be no safe protection against root kits" (p. 16).

While both *The Washington Post* and *The New York Times* frequently discuss the threat of viruses, especially as they relate to Windows users, neither publication has ever published any material about root kits, according to several searches of their online databases.

Again, this should come as no surprise. The mass media audience would not know what to make of such information, even if it were provided to them in the daily paper. A root kit can successfully compromise a system in a few seconds and remain undetectable, using commonly available systemic vulnerabilities. This information is of little immediate interest to the 95 percent of computer users (and mass media consumers) who run Microsoft Corporation's Windows operating system. Thus, members of the mass media are far more likely to discuss new patches available through the Windows Update utility to take care of those "critical" vulnerabilities, and urge everyone to keep their anti-viral software definitions updated than to mention the widespread use of root kits to attack a Linux system.

This paper argues that the natural tendency of the mass media to avoid talking about issues that would seem to be too exotic for their audience is likely part of the reason that Open Source software advocates persist in their belief that COTS software is less secure than Open Source software. To understand why, this paper will rely on significant studies that explain how we perceive risk.

Perception of Risk

The seminal paper, “The Social Amplification of Risk: A Conceptual Framework (Kasperson, Renn, Slovic, Brown, Emel, Goble, Kasperson, & Ratick, 1988), attempted to go beyond the traditional measurement of risk – multiplying the probability of an event by the magnitude of the expected outcome – to try to understand why people do not perceive risk accurately. For instance, many people erroneously assign an invalid low level of risk to driving without a seat belt (or, conversely, an equally incorrect high level of risk of the seat belt trapping one in the car after an accident).

Classic communications theory embraces the concept of amplification, in which the influence or power of information is increased as it passes from an intermediate source to the ultimate recipient (the same process can occur in reverse, as attenuation decreases the effect). Kasperson, et al., theorized that “information processes, institutional structures, social-group behavior and individual responses shape the social experiences of risk, thereby contributing to risk consequences” (p. 180). This social amplification of risk influences behavioral responses that ultimately lead directly to what that paper terms secondary impacts. These include enduring mental attitudes (such as fear or distrust of technology) and social apathy; political and social pressures; lower levels of public acceptance for things such as the use of technology and a general decline in the trust of the public. These secondary impacts themselves have impacts further out, like ripples across a body of water.

Individuals learn a great deal about correct risk assessment in their everyday lives from personal experiences, such as laying out for a bit too long in the sun and getting a burn. Individuals learn about risks that have not been experienced directly from other people or information disseminators such as the mass media. While there are a number of factors that bear on amplification, for the purposes of this paper the critical vector is this: Regardless of the accuracy or context of the information, large volumes of information flow can act as a risk amplifier. That is, simply being bombarded with information about a subject over and over can increase the sense of risk about a specific subject. “However balanced the coverage, it is unclear that reassuring claims can effectively counter the effects of fear arousing messages” (p. 185). In addition, the news media, by definition, tend to focus on extraordinary events, a trend that has tended to skew the public perception of things such as the safety of airline travel, for example, which remains statistically a much safer way to travel than by car.

Slovic (2000), argues that many of the difficulties society has had managing fears of technological hazards – such as the ongoing fear that genetically engineered foods are somehow different in nature than foods created using the more traditional genetic engineering known as crossbreeding – can be directly related to a lack of trust. He argues persuasively that acceptance of risk, such as a deadly animal in a cage at the zoo, is entirely dependent on the level of confidence we have in the people assigned to manage that hazard.

Slovic makes a key point when he says that trust is quite asymmetric. That is, it is easy to lose trust in someone; it is much harder to gain trust. The field is tilted in the direction of distrust for a number of reasons. For one thing, it is much easier to identify behavior that destroys trust than it is to recognize behavior that should encourage trust. Accidents, lies, and errors are all easy to spot as evidence that somebody cannot be trusted. In contrast, a day without drama, without an incident of any kind, is simply another day; it is not clear how to interpret it and thus it carries less weight in making the evaluation. In addition, people naturally have a psychological predisposition to put more emphasis on the negative events that destroy trust than the positive events that build trust.

Thus, people often, or even usually, misjudge the level of risks for behaviors they engage in. Individuals tend to overestimate the likelihood that they will die from an accident and underestimate the odds that they will succumb to diabetes (Morgan, et al, 1985). This situation is compounded by the fact that "news" is something unusual; newspapers do not typically print stories about how 145 planes landed safely at the local airport today because most people would not care about such a story. Presentation, it turns out, is everything. Morgan et al tested people for their knowledge about the risks involved in living near high voltage electrical lines and then tested them again after they had studied a summary of possible concerns that had been raised by some studies, but were then refuted. The refutation did no good, however; simply hearing about such concerns left people concerned. And when the presentation was offered up to participants as a kind of worst case scenario – a favorite of journalists – participants became downright agitated, unable to distinguish between what was possible (but highly unlikely) and what was probable (and considerably less threatening).

Thus it is clear that the way the media cover security issues has had a profound effect on the perception of security in Open Source software versus COTS. The popular press is constantly presenting stories about Windows vulnerabilities, because that is the system that the vast majority of consumers are most likely to use, and thus that is what those consumers will be interested in hearing stories about. The incessant drumbeat of noise on the issue, following the theory of amplification, further stimulates concern among the population and the media, creating an ongoing feedback mechanism. At the same time, virtually no mention of Open Source software is made in the popular press; it is a wonder that we have not all mysteriously stopped using this Windows software that people are constantly telling us is unsafe and started using something – anything – else. It seems given the forces outlined here that is exactly what would have happened, but for the fact that Microsoft has monopoly power in the market for desktop computer operating systems (Auletta, 2000). The economists who testified at that trial were right; Microsoft's Windows is so powerful and society has become so dependent upon it that users cannot give it up, not even after listening to those such as Pegoraro who essentially warn that there are two types of people who use Windows: Those that have had their machines compromised, and those who will someday have their machines compromised. Economists enjoy speaking about tipping points, a time and place at which a market suddenly slides quite rapidly in favor of a specific product or competitor. As an example, a single telephone is not worth much, but if someone else gets a telephone too, that phone suddenly becomes quite a bit more valuable. Eventually, instead of having to coax people to take a leap of faith by investing in a phone so that telephone owners can talk with them, consumers realize how valuable a phone is all by themselves, because so many other people have them. In an instant, the market tips. Microsoft has allegedly benefited from this phenomenon to achieve its market dominance; it is possible that the increasing perception of their products as fundamentally untrustworthy will eventually tip the market in another direction altogether.

Affect, Perceived Risk, and Perceived Reward

Recall that a good deal of the opening chapter of this paper was spent discussing the evolution of the Open Source movement, together with the political sensibility behind it, along with the way its opponents characterized it as downright un-American, or smacking of communism.

That information helps set the stage for a discussion about the influence that the concept of “affect” – how well one likes something – has on all the issues this paper has examined so far.

Here is a quotation that sums up the point at which that part of the analysis will begin: “Linux, for instance, attracts legions of developers who have a direct interest in improving an operating system for their own use. However it scratches another important itch for some of these folks: It is creating a viable alternative to Microsoft Corporation’s products. Throughout our discussions with groups and individuals, this anti-Microsoft Corporation sentiment was a recurring theme.” (Hissam, Weinstock, Plakosh, & Asundi, 2001, p. 57).

If Microsoft Corporation executives such as Steve Ballmer look upon participants in the Open Source movement with the sort of sour gaze one might otherwise use when perusing material clinging to the bottom of one’s shoe, Open Source developers generally look upon Microsoft Corporation as a vast demonic presence frantically trying to finish swallowing the world before anybody important actually notices. Many Open Source participants see projects such as Linux as nothing less than pre-game for the apocalypse, a titanic struggle between good and evil that will in the end decide the fate of civilization. Microsoft Corporation’s bullying tactics – well-documented during the antitrust trial and eventually ruled illegal – together with its lust for power (Auletta, 2000), have almost certainly drawn more people into the Open Source movement than has the opportunity to create a few lines of code designed to make a better looking interface for a printer installer.

Using this well-documented antipathy for Microsoft by Open Source enthusiasts and risk perception evaluation techniques (Finucane, Alhakami, Slovic, & Johnson, 2000) should offer one more clue about why so many members of the Open Source movement continue to insist that Open Source code is more secure than COTS code, despite the lack of any real evidence to support that claim.

Put simply, research shows that if people “like” something, they tend to downplay the risks associated with it, and inflate the possible benefits associated with it. If something is disliked, not surprisingly, that opinion is reversed; people overplay risks and unfairly dismiss benefits related to the loathsome object, activity, or person.

It seems quite clear that this phenomenon is directly related to the perception of Open Source software as “safer” than COTS products.

CHAPTER 7: CONCLUSION

This final section opens with a quote from Theo de Raadt, project leader for OpenBSD, a variant of Unix based on the Berkeley Software Distribution, one of several varieties of Unix currently available to users. OpenBSD has developed a reputation as an extremely secure system, but it did not get that way by assuming that “many eyes” would be hunting down problems. Unlike those running other “laissez faire” Open Source efforts, de Raadt is a firm believer in a carefully controlled development environment; OpenBSD participants get assignments, they have a plan to follow, and those who do not like it are unceremoniously told they are no longer welcome. Theo de Raadt firmly rejects the notion that security will take care of itself in an Open Source project. “These open-source eyes that people are talking about, who are they?” he asks. “Most of them, if you asked them to send you some code they had written, the most they could do is 300 lines long. They're not programmers” (Koetzle, 2004).

And in the end this paper would argue that it is not the number of eyeballs that have stared at code that makes a difference in software quality. What matters are design, specification, implementation, and the education, talent and commitment of everyone associated with the project.

The basic question to be answered by this paper was, is there any real difference in the security between Open Source and COTS software, as Open Source advocates claim? The truth is that there appears to be little difference, though it is difficult to say for certain because the metrics used to attempt to measure such things are questionable at best. This paper suggest that any reasonable measure of how secure a system is cannot be based on gross measurements such as counting the number of critical security alerts put out for each program or system, but must rather rely on more sophisticated risk analysis techniques, which calculate, not how severe the damage would be if the vulnerability were exploited, but also how likely or even possible it is that the vulnerability will be exploited. In addition, we argue that to be valid, such an estimate must also take into consideration the skill levels of those using it. Knowledgeable computer users are at a decreased risk of a security breach in no small measure because they would be less likely to do something hazardous, such as open an e-mail attachment they were not expecting, which is the sort of thing that typically compromises a Microsoft Corporation Windows operating system these days.

In addition, this paper applied techniques of Risk Perception analysis to examine the forces that influence the way Open Source supporters evaluate the world. Based on that analysis, it seems likely that the influences and forces identified in this paper may be responsible for the continued belief that Open Source code is safer than COTS code, despite the absence of any real evidence to support that position. The risk perception techniques were easily applied, provided significant insight, and should become a regular tool used in risk analysis.

In closing, it seems clear that there are many excellent reasons to consider using Open Source products. Their current advantages are manifest: Cost, flexibility, and transparency.

Certainly Open Source software also has the potential to become a model of good security practices, but at the moment, one should no more trust Open Source code than one should trust any other programming not written by the user.

This paper would argue that both COTS development environments and Open Source development environments can and should create systems that are more secure. Unfortunately, it is not at all clear that those systems will be produced.

Companies such as Microsoft that create COTS have a vast amount of resources – including cash reserves, armies of trained specialists, and decades of experience – that could enable them to introduce computer systems that are far more secure than those being sold today. But a company such as Microsoft has an enormous financial incentive to avoid doing that: Introducing safer designs runs the risk of alienating their current customers. Each new generation of Microsoft's operating system offers "backward compatibility" with the previous generation. This encourages users to upgrade to the new operating system, since the user can typically rely on existing software applications that had been running in the old environment to also run in the new environment. But that backward compatibility forces Microsoft to continue to produce operating systems with known security issues. Producing a completely new operating system without backward compatibility would not encourage – some would say force – users to upgrade to the next generation operating system. Existing customers who felt abandoned by Microsoft might seek out – or even create – an alternative.

In contrast, the Open Source movement can choose to move ahead, unfettered by a focus on the bottom line or concerns about how the stock market might react to a proposal to introduce an entirely new operating system designed with security in mind. The dedicated coders who have devoted countless hours to helping build a new vision of the future via the Open Source model can truly achieve that dream, but only if they are willing to walk away from the mistakes of the past that are firmly embedded in the products they are working on today. The greatest promise that Open Source software holds is that it has the ability to adopt an entirely new paradigm, a secure paradigm based on sound design and implementation principles. Because building secure systems is not just about crude metrics such as how many bugs were found in code. Developing a secure system requires designers to rediscover the fundamental computational principles and to implement architectural solutions that address these issues of designing secure systems.

The most significant contribution that the Open Source movement could ever make would be to create an entirely new platform that had all the current advantages of the current offerings, but was not chained to the mistakes of the past.

But a radical redesign of existing systems such as Linux that would incorporate the best security principles is only half of what is needed for the Open Source movement to fulfill its promise. The real issue, as this paper has documented, is the lack of any systematic testing program for many Open Source products. Failure to implement such a program will ultimately doom Open Source products to a marginal role in the information technology infrastructure.

REFERENCES

Anderson, J. P. (1972, October). Computer security technology planning study. James P. Anderson & Co., for the United States Air Force.

Anderson, R. (2002, June 20-21). Security in open versus closed systems: The Dance of Boltzmann, Coase, and Moore. Paper presented at Open Source Software: Economics, Law and Policy, retrieved Feb. 16, 2004, from http://www.idei.asso.fr/ossconf.html

Auletta, K. (2000). *World war 3.0: Microsoft and its enemies.* New York: Random House.

Asiri, S (2003, March). Open source software. *Computers and Society,* Volume 32, Issue 5. Retrieved March 22, 2004 from http://doi.acm.org/10.1145/966498.966501

Bell, D. E. & La Padula, L. (1975) Secure computer system: Unified exposition and Multics interpretation, The Mitre Corporation.

Bezroukov, N. (1999, October 8). Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism). *First Monday,* Volume 4, Number 10. Retrieved on April 12, 2004, from http://www.firstmonday.dk/issues/issue4_10/bezroukov/

Bishop, M. (2003). *Computer security: art and science*. Boston, MA: Addison-Wesley.

Brown, K. (2002, June). Opening the open source debate, Alexis de Tocqueville Institution.

Cisco Advisory (2004, April 7). Cisco security advisory: A default username and password in WLSE and HSE devices. Retrieved April 10, 2004, from http://www.cisco.com/warp/public/707/cisco-sa-20040407-username.shtml

CERT (2001a, January 10). CA-2001-01 Interbase server contains compiled-in back door account. Retrieved on March 25, 2004 from http://www.cert.org/advisories/CA-2001-01.html

CERT (2001b, July 19). CA-2001-19 "Code Red" worm exploiting buffer overflow in IIS indexing service DLL. Retrieved on March 25, 2004 from http://www.cert.org/advisories/CA-2001-19.html

CERT (2001c September 18). CA-2001-26 Nimda worm. Retrieved on March 25, 2004, from http://www.cert.org/advisories/CA-2001-26.html

CERT (2002). CERT/CC Vulnerability disclosure policy. Retrieved on April 11, 2004, from http://www.cert.org/kb/vul_disclosure.html

Chen, T. M. (2003, September). Trends in viruses and worms. *The Internet Protocol Journal,* Volume 6, Number 3, 23-33.

Christey, S. M. (2002, Nov. 27). On vulnerabilities in open and closed source products. Online posting, retrieved on January 27, 2004 from http://www.securityfocus.com/archive/1/301455

Colen, L. (2001, January). The best code reviews are the ones that get done. *Linux Journal,* Volume 2001, No. 81es. Retrieved March 28 from http://delivery.acm.org/10.1145/370000/364693/a11-colen.html?key1=364693&key2=7912881801&coll=ACM&dl=ACM&CFID=20213566&CFTOKEN=17458470

Common statement (2004, April 6). GNU/Linux vendors Debian, Mandrake, Red Hat, and SUSE have joined together to give a common statement about the Forrester report entitled "Is Linux more Secure than Windows?" Retrieved on April 7, 2004, from http://lxer.com/module/newswire/view/9986/index.htm

DeFleur, M. L. (1966). *Theories of Mass Communication.* New York: D McKay.

DiBona, C., Ockman, S. & Stone, M. (Eds.) (1999). *Open sources: Voices from the open source revolution.* Sebastapol, CA: O'Reilly.

Drabick, R. (2004). *Best practices for the formal software testing process: A menu of testing tasks.* New York: Dorset House.

Evers, J. (2004, March 5). Will XP's Service Pack Break Existing Apps? Some software may not work on PCs with SP2 installed. *PC World,* retrieved on April 12, 2004, from http://www.pcworld.com/news/article/0,aid,115093,00.asp

Freeman, H. (2002, September). Software testing. *Instrumentation & Measurement Magazine, IEEE.* Volume 5, Issue 3, 48 - 50.

Feiertag, R. J. & Neumann, P. G. (1979). The foundations of a provably secure operating system (PSOS). *Proceedings of the National Computer Conference,* 329-324.

Ferris, P. (2003, July/August). The age of corporate open source enlightenment, *Queue.* Volume 1, Issue 5, 34-44.

Finucane, M. L., Alhakami, A., Slovic, P. & Johnson, S. M. (2000). The affect heuristic in judgments and risks and benefits. *Journal of Behavioral Decision Making,* Volume 13, 1-17.

Flynn, H. (2004, February 4). Why Sardonix failed, *SecurityFocus,* retrieved on April 1, 2004, from http://www.securityfocus.com/columnists/218

Garfinkel, S., Spafford, G., & Schwartz, A. (2003). *Practical Unix & Internet security (3rd ed.).* Sebastapol, CA: O'Reilly.

Glass, R. L. (2003, November). A sociopolitical look at open source. *Communications of the ACM,* Volume 46, Issue 11, 21-23.

Geer, D., Bace, R., Gutman, P., Metzger, P., Pfleeger, C. P., Quarterman, J. S. & Schneier, B. (2003, Sept. 24). Cyberinsecurity: The cost of monopoly, Computer & Communications Industry Association, retrieved February 5 from http://www.ccianet.org/papers/cyberinsecurity.pdf

GNU Project (2004). GNU operating system: Free Software Foundation, retrieved on February 8, 2004, from http://www.gnu.org/home.html

Gomes, L. (1998, November 3). Microsoft acknowledges growing threat of free software for popular functions. The Wall Street Journal, B6.

Halloween Document (1998, August 11). Retrieved on March 4, 2004, from http://www.opensource.org/halloween/halloween1.php#quote1

Harris, S (2003). *CISSP certification all-in-one exam guide (2nd ed.).* Emeryville, CA: McGraw-Hill.

Hissam, S., Weinstock, C. B., Plakosh, D., & Asundi, J. (2001). Perspectives on open source software, technical report, Software Engineering Institute, Carnegie Mellon University, retrieved January 18, 2004 from http://www.sei.cmu.edu/publications/documents/01.reports/01tr019.html

Hoglund, G. & McGraw, G. (2004). *Exploiting software: how to break code.* Boston: Addison-Wesley.

Internet Systems Consortium, (2004). Retrieved on April 2, 2004 from http://www.isc.org/index.pl?/sw/bind/

Johnson-Eilola, J. (2002). Open source basics: definitions, models, and questions. Proceedings of the 20th annual international conference on computer documentation, ACM Special Interest Group for Design of Communications, 79-83.

Karger, P. A. & Schell, R. S. (1974, June) MULTICS security evaluation, volume II: vulnerability analysis. Electronic Systems Division, Air Force Systems Command.

Kasperson, R. E., Renn, O, Slovic, P., Brown, H. S.. Emel, J., Goble, R., Kasperson, J. X. & Ratick, S. (1988). The social amplification of risk: A conceptual framework. *Risk Analysis,* Volume 8, Number 2, 177-187.

Kerckhoffs, A., (1883, January). La Cryptographie Militaire, (French) *Journal des Sciences Militaires.* 5-38. Retrieved an English summary of the original paper and a link to scanned versions of the paper on March 18, 2004 from http://www.petitcolas.net/fabien/kerckhoffs/#english

King, J. J. (1999, August 18). Free software is a political action, retrieved Jan. 16 from http://www.heise.de/tp/english/special/wos/6469/1.html

Koerner, B. I., (2000, August 14). A lone Canadian is reshaping the way software gets written. Is the world paying attention? *The Industry Standard*, retrieved on April 1, 2004, from http://www.landfield.com/isn/mail-archive/2000/Aug/0099.html

Koetzle, L. (2004, March 19). Is Linux more secure than Windows? Forrester Research Inc.

Krebs, B. (2004, April 13). Hackers Strike Advanced Computing Networks. *The Washington Post,* retrieved April 15 from http://www.washingtonpost.com/wp-dyn/articles/A8995-2004Apr13.html

Kuhnhauser, W. E. (2004). Root kits: An operating systems viewpoint. *ACM SIGOPS Operating Systems Review,* Volume 38, Issue 1, 12-23.

Lampson, B. W. (1973, October). A note on the confinement problem. *Communications of the ACM,* Volume 16, Number 10, 613-615.

Lawrie, T. & Gacek, C. (2002, May). Issues of dependability in open source software development. *ACM SIGSOFT Software Engineering Notes,* Volume 27, Issue 3, 34-37.

Lawton, G. (2002, March). Open source security: opportunity or oxymoron? *Computer,* Volume 35, Issue 3, 18-21.

Lea, G. (2000, July 31). MS' Ballmer: Linux is communism. The Register. Retrieved April 15, 2004, from http://www.theregister.co.uk/2000/07/31/ms_ballmer_linux_is_communism/

Lemos, R. (2002, February 6). Site to pool scrutiny of Linux security, NEWS.COM, retrieved April 2, 2004 from http://news.com.com/2100-1001-830130.html

Lemos, R. (2003, December 9). Developers take Linux attacks to heart, NEWS.COM, retrieved April 12 2004, from http://news.com.com/2100-7344-5117271.html?tag=fefd_hed

Leonard, A. (1998, Dec. 21). You've got sendmail, SALON.COM, retrieved on April 15, 2004 from http://cobrand.salon.com/21st/feature/1998/12/cov_11feature.html

Lessig, L. (1999). *Code and Other Laws of Cyberspace.* New York: Basic Books.

Levy, S. (1984). *Hackers: Heroes of the computer revolution.* New York: Penguin.

Lipner, S. B.(2000, May) Security and source code access: issues and realities, Security & Privacy, IEEE Symposium on Security and Privacy, May 14-17 2000, 124 - 125

LSAP (2004). Linux Security Audit Report, Retrieved on April 14, 2004 from http://lsap.org/

Middleton, J. (2002, April 2). Windows more secure than Linux?, VUNET.COM, retrieved on April 13, 2004 from http://www.vnunet.com/News/1128907

Mockus, A., Fielding, R. T. & Herbsleb, J. D. (2002, July). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions of Software Engineering and Methodology,* Volume 11, No. 3, 309-346.

Morgan, M. G., Slovic, P., Nair, I., Geisler, D., MacGregor, D.G. Fischoff, B., Lincoln, D. & Florig, K. (1985). Powerline frequency electric and magnetic fields: A pilot study of risk perception. *Risk Analysis,* Volume 5, 139-149.

Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., & Weaver, N. (2003). The Spread of the Sapphire/Slammer Worm. Cooperative Association for Internet Data Analysis.

Musgrove, M. (2004, April 14). Microsoft Finds New Windows Security Flaws. *The Washington Post,* E05.

Netcraft (2004). Latest Web server survey, retrieved April 12 from http://news.netcraft.com/archives/web_server_survey.html

Neumann, P. G., Robinson, L., Levitt, K. N., Boyer, R. S. & Saxena, A. R. (1975). A provably secure operating system. Stanford Research Institute.

Neumann, P. G., Boyer, R. S. Feiertag, R. J., Levitt, K. N., & Robinson, L. (1980). A provably secure operating system: The system, its applications, and proofs (2nd ed.). Stanford Research Institute.

Neumann, P. G. (2000, May 14-17). Robust nonproprietary software. Proceedings 2000 IEEE Symposium on Security and Privacy, retrieved on April 11, 2004, from http://www.csl.sri.com/papers/i/e/ieee00/ieee00+.pdf

Neumann, P. G. (2003, December 19). PSOS revisited. Presentation, 19th Annual Computer Security Applications Conference, Las Vegas, Nevada.

Neumann, P. G. (2004, March 15). Principled assuredly trustworthy composable architectures. Evolving draft for SRI Project 11459, Contract number N66001-01-C-8040 as part of DARPA's Composable High-Assurance Trustworthy Systems (CHATS) program, SRI International.

Open Source (2004). The Open Source project, retrieved on Jan. 19, 2004 from : http://www.opensource.org/docs/definition.php

Orange Book (1985, December). Department of Defense Trusted Computer System Evaluation Criteria (TCSEC). National Computer Security Center.

Ozier, W. (1997, June). GASSP: generally accepted systems security principles. Technical report, International Information Security Foundation, retrieved on April 8 from http://web.mit.edu/security/www/gassp1.html

Parker, D. B. (1998). *Fighting computer crime: A new framework for protecting information.* New York: Wiley.

Pegoraro, R. (2003, August 24). Microsoft Windows: Insecure by design. The Washington Post, F7.

Peltier, T. R. (2001). *Information Security Risk Analysis*. New York: Auerbach.

Perl.com (2004). Retrieved on March 12, 2004, from http://www.perl.org/about.html

Pethia, R. D. (2002, November 19). Testimony of Richard D. Pethia, director, CERT/CC, before the House Committee on Government Reform Subcommittee on Government Efficiency, Financial Management and Intergovernmental Relations.

Pfleeger, C. P. & Pfleeger, S. L. (2003) *Security in Computing, (3rd ed.).* Englewood Cliffs, NJ: Prentice-Hall.

Poulson, K. (2004, January 30). DARPA-funded Linux security hub withers, SecurityFocus, retrieved on April 1 from http://www.securityfocus.com/news/7947

Raymond, E. S. (2000) The cathedral and the bazaar. Retrieved on March 15, 2004, from http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html#catbmain

Raymond, E. S. (2004, April 4). The luxury of ignorance: An open-source horror story. Retrieved on April 27, 2004, from http://www.catb.org/~esr/writings/cups-horror.html

Reinke, J. & Saiedian, H. (2003). The availability of source code in relation to timely response to security vulnerabilities. *Computers & Security,* Volume 22, Number 8, 707-724.

Saltzer, J. H. & Schroeder, M.D. (1975, September). The protection of information in computer systems. *Proceedings of the IEEE,* Volume 63, Number 9, 1278-1308. Retrieved on March 28, 2004, from http://cap-lore.com/CapTheory/ProtInf/

Schell, R. R., Downey, P. J., & Popek, G. J. (1973 January). Preliminary notes on the design of secure military computer systems. The MITRE Corporation.

Schneier, B. (1996). Applied Cryptography: Protocols, Algorithms, and Source Code in C. (2nd ed.). New York: John Wiley and Sons.

Schneier, B. (1999). Open source and security. Crypto-Gram. Counterpane Internet Security, Inc., retrieved on April 11, 2004, from http://www.counterpane.com/crypto-gram-9909.html

Schneider, F. B., (2000, May 14-17). Open source in security: visiting the bizarre, Proceedings 2000 IEEE Symposium on Security and Privacy, 126-127.

Seltzer, L. (2004, February 22). How closely is open source code examined? EWEEK.COM, retrieved on March 3, 2004 from http://www.eweek.com/article2/0,1759,1536426,00.asp

Scoville, T. (2004). Untangling the open source/free software debate, retrieved on March 12, 2004 from http://opensource.oreilly.com/news/scoville_0399.html

Slovic, P. (2000). Perceived risk, trust and democracy, *The Perception of Risk.* London: Earthscan, 316-326.

Smith, R. E. (2001, February). Cost profile of a highly assured, secure operating system. *ACM Transactions on Information and System Security,* Volume 4, Issue 1, 72-101.

Stallman, R. (1999). The GNU operating system and the free software movement. From DiBona, C., Ockman, S. & Stone, M. (Eds.) *Open sources: Voices from the open source revolution.* Sebastapol, CA: O'Reilly.

Stevenson, R. (2004, April 13). Microsoft warns of 3 “critical” flaws. Reuters.

Thomson, I. (2002, September 30). Honeymoon over for Linux users, VUNET.COM, retrieved on April 1, 2004, from http://www.vnunet.com/News/1135481

Thompson, K. (1984, August). Reflections on trusting trust. *Communications of the ACM,* Volume 27, Number 8, 761-763.

Torvalds, L. (1991, August 25). What would you like to see most in Minix, retrieved on Feb. 2, 2004, from http://lwn.net/2001/0823/a/lt-announcement.php3

Torvalds, L. (1996, July 22). Wake UP! (Re: SAVE LINUX! VOTE AGAINST THE PENGUIN!). Message posted to comp.os.linux.advocacy and comp.os.linux.misc, retrieved on March 15, 2004, from http://groups.google.com/groups?selm=4sv02t%24j8g%40linux.cs.Helsinki.FI

Viega, J. (1999, September 1). Open source software: Will it make me secure? Alleviating the risks of opening up your source can pay off in improved security. Paper from IBM Corporation Web site, retrieved March 28, 2004 from http://www-106.ibm.com/developerworks/security/library/s-oss-security.html

Ware, W. (1970, Feb). Security controls for computer systems: report of Defense Science Board Task Force on Computer Security, Rand Report R609-1.

Weber, S. (2000, June). The political economy of open source software, The Berkeley Roundtable on the international economy, retrieved April 15, 2004 from http://repositories.cdlib.org/cgi/viewcontent.cgi?article=1011&context=brie

Wheeler, D.A. (2003).Why Open Source Software / Free Software (OSS/FS)? Look at the Numbers! http://www.dwheeler.com/oss_fs_why.html#security

Whitlock, N. W. (2001, March 1). The security implications of open source software: Does open source mean an open door? Paper from IBM Corporation Web site, retrieved March 28, 2004 from http://www-106.ibm.com/developerworks/linux/library/l-oss.html

Whittaker, J. A. (2000, January-February). What is software testing? And why is it so hard? *Software, IEEE,* Volume 17, Issue 1, 70-79.

Windows XP (2004, March). Microsoft Windows XP Service Pack 2 Release Candidate 1 Fact Sheet. Microsoft Corporation press release, retrieved April 11, 2004, from http://www.microsoft.com/presspass/newsroom/winxp/WindowsXPSPFS.asp