There is no better way to start an argument among a group of developers than proclaiming Operating System A to be "more secure" than Operating System B. I know this from first-hand experience, as previous papers I have published on this topic have

Open vs

RICHARD FORD, FLORIDA INSTITUTE OF TECHNOLOGY

led to reams of heated e-mails directed at me—including some that were, quite literally, physically threatening. Despite the heat (not light!) generated from attempting to investigate the relative security of different software projects, investigate we must.

Closed

Which source is more secure?

Open vs. Closed

Understanding why products are (and are not) secure is a critical stepping stone toward building better software.

Before wading into these dangerous waters, we should clarify the question. All too often when comparing open and closed source approaches, the question is unconsciously interpreted as Windows versus Linux. While that's a fantastic question to knock around, doing so is a very narrow way of looking at the world, as it ignores many other projects in both the open and closed source worlds. Although it's foolish to ignore the data points the Windows/Linux world provides, they are simply examples of the process. So, let us first strip away the misconception that the question is about these particular platforms and recognize its real breadth.

With this in mind, our answer requires three crucial definitions in order to have meaning: "What is open source?"; "What is closed source?"; and, surprisingly, "What is security?" The first two we can deal with quickly; the third is a lot subtler, however, so we shall tackle it first.

WHAT IS SECURITY?

Traditionally, we tend to think of security as maintaining the CIA (confidentiality, integrity, and availability) of information. This is a useful taxonomy of security, and because of this, it's pervasive. One limitation of the CIA approach is that it isn't very helpful when we consider how to *measure* security. What does it mean to say that one product is *more secure* than another product? Is C more important than A, and is A more important than I? How does one rank these different aspects of security?

A literature review quickly shows that measuring security is a tricky problem, which, as yet, we haven't gotten our arms around very well. That's a pity, because if we had, it would be tempting to run the simple experiment of measuring the security of various open and closed source projects and see if one methodology is consistently more secure than the other. If closed source, for example, were measurably better from a security perspective, we would have the answer to our question.

There are two obvious ways to measure security:

- What are the chances of any member of the CIA triad being violated?
- How many actual vulnerabilities are there in a product?

Let's take a look at both of these approaches.

The problem is that the former is a combination of the quality of the software under test, the number and type of attackers targeting that software, and how the box is configured, administered, and used. Thus, if "more secure" simply means measuring the probability of compromise, it might be possible to conclude that MS-DOS with a TCP/IP stack is more secure than a fully patched Windows XP box, simply because the number of attackers looking for MS-DOS machines is now vanishingly small. While the measure is pragmatic, it tells us a lot about the ubiquity of the system and the talent and number of its attackers.

Discarding this approach leaves us with the latter of our two approaches: counting vulnerabilities in the code. Even here, it's not obvious how to proceed, as we don't have *direct* measures of actual vulnerability counts; we have information only about the number of vulnerabilities that are *publicly* disclosed. Thus, like the first approach, this one doesn't provide an objective measure of security; it also considers external factors (such as attacker profile).

A variation of this approach is known as "days of risk," which is literally counting the elapsed time between vulnerability disclosure and remediation. Defining remediation is a difficult task. Does turning off a noncritical service count as temporarily "fixing" the problem, or does only a "sanctioned" patch supported by the vendor constitute remediation? This would depend on the service provided and the needs of the user. Even if we can agree on remediation, the number of attackers plays a critical role in determining the total days of risk. Despite this, the approach is tremendously practical because it takes into account the fact that actually exploiting a vulnerability is relatively rare until the vulnerability is publicly known.[1] It's a practical measure, however, and as such, doesn't speak directly to inherent security properties, but pragmatic ones. Note here that days of risk are traditionally counted from the date the *vulnerability* is publicly available, not the date an *exploit* is known. Although one can argue that knowledge of the vulnerability is meaningless in the absence of an exploit, it is often difficult

to determine when an exploit became "public," as many members of the black-hat community keep such information under close guard. Thus, vulnerability date is the most objective—and therefore repeatable—measure (even if it is not as desirable as the exploit date).

Even based on this short discussion, it's clear that accurately measuring security will mean different things to different people. Thus, for the purposes of this article, it's reasonable to accept that we can't (yet) measure the inherent security outcomes of open/closed source processes in an ordinal way. This means that our "experimental" approach to determining which approach leads to better security is off the table: until the science matures, we will have to examine the pros and cons of each approach independently and try to balance them ourselves.

OPEN SOURCE, CLOSED SOURCE

Put simply, the open source process can be thought of as an approach where the source code to products/executables is provided. In contrast, closed source approaches restrict source-code access to just the developers of the product and other chosen individuals (usually under the constraints of a nondisclosure agreement). In both worlds, many finer distinctions can be made. For example, some open source projects restrict development to a small cadre of programmers; others allow anyone to contribute. Source code access, however, is the key distinction between the approaches. Note also that neither case *requires* software to be free nor "for fee"—though the open source world is generally friendlier in terms of licensing.

Perhaps appropriately for the open source community, a more precise definition of open source varies from person to person. At its simplest, open source refers to the practice of providing the source code for programs. Furthermore, most proponents of the open source approach would agree that the distributed source code should be legally modifiable and redistributable (with some license restrictions). Thus, users have the ability to inspect and modify programs they use. (A far more complete definition is provided at http://www.opensource.org.)

In contrast, the closed source approach seals the program code. As such, derivative works are usually legally prohibited. Proponents of both camps may object to the simplicity of my definitions: they do capture the essence of both approaches but fail to capture the *culture* that surrounds them.

Culturally, closed source represents traditional corporate software developers. When we think of open source, however, we tend to think of volunteers working as a collective, free software, and community projects. Open source structures are fluid; closed ones rigid. While this is something of a caricature, like all good sketches, it does catch some of the "feel" of the movement.

INHERENT SECURITY PROPERTIES

Armed now with an understanding of the question, it is time to examine the relative merits of the two approaches from a security perspective. Clearly, others have undertaken this process (for a slightly different perspective, for example, see Ross Anderson[2]); however, there are *many* issues that are not addressed completely. As such, we begin by considering the most basic difference between the development methodologies: one can examine the source code of an open source project. Pragmatically, this is of use to both the attacker and the defender.

From the attacker's point of view, code availability means that there is complete disclosure on *how* a particular feature is implemented. Furthermore, it means that discussion of weaknesses and design decisions often happens in the open (see the "Disclosure Models" section later in this article). Thus, open source products allow the attacker a white-box view of the product and, potentially, associated problems. When a security patch is made available, it is trivial for the attacker to determine exactly what was fixed.

From the perspective of the defender, open source also has advantages. Perhaps most importantly, it allows for code inspection. Thus, if the defender really wants to know that a particular feature is secure, he or she can simply examine the code—provided, of course, that the defender has the necessary security knowledge to spot a problem. Second, there is a sense that because many people can review the code, the code is inherently higher quality—as framed by Eric S. Raymond in his now-famous quote, "Given enough eyeballs, all bugs are shallow."[3] Finally, features that are problematic in a particular environment can be turned off by a sufficiently skilled programmer. Thus, when a vulnerability is found, the user doesn't have to wait for a sanctioned patch: anyone can make the requisite changes to the code base.

From an attacker's perspective, closed source means that only a small part of a given community has access to the code. Thus, to understand the internals, the attacker must reverse-engineer the binary; such a process is time consuming and, in the case of software that has been protected from such reverse engineering, nontrivial.

Furthermore, design mistakes may be harder to spot, as grasping the entire form of a large application is quite

Open vs. Closed

difficult when working only with compiled code.

Things are equally double-edged for the defender. When using a closed source product, the user is left entirely at the mercy of the code developer in terms of functionality changes or security patches. Thus, when a vulnerability is announced, the options for the defender are limited. Once again, differences in disclosure models help mitigate this somewhat, but ultimately, the user is left trusting the vendor. Self-help is not a practical option; code cannot be screened internally for structures that are worrisome in particular environments. Of course, these issues are compounded if the code to a closed source product is leaked; then the attacker has many of the benefits of the approach, with few of the downsides.

These fundamental properties are painted with a fairly broad brush, but in essence they encapsulate the systematic differences between the techniques in terms of attacker and defender. Space precludes a thorough examination of these differences, so we will turn our attention to the two that seem to have the most impact: vulnerability disclosure models and trust/validation.

DISCLOSURE MODELS

One key difference between open and closed source processes is the vulnerability disclosure model that is typically shared within them. As open source's nature is openness, when vulnerabilities are repaired it is trivial for an attacker to see exactly what was repaired and work back to the vulnerability and (probably) a working exploit. In the closed source world, it might not even be clear that a vulnerability existed or was fixed.

Because of this, open source tends to do badly from the perspective of "days of risk," where one counts the time between the disclosure of a vulnerability and an "approved" fix. Some may find this unfair, but pragmatically history shows that the window between the public availability of a vulnerability/exploit and its patch is a difficult and dangerous time. In addition, while it is entirely possible (and practiced in several open source communities) to embargo security bug disclosure until a patch is available, the practice of no disclosure is still rarer in the open source community than the closed source community. In addition, the problem is compounded by the many different Linux distributions that contain open source components. If a component is updated by its creators, it is impractical to wait until *all* distributions that use it are ready to issue a validated patch.

The difference in disclosure models is a difficult problem for open source processes to solve. While one can argue that users can fix problems as they arise (thus, as soon as the problem is disclosed, the user writes a patch for his or her own use), this is a little far-fetched. Most users aren't programmers, and those who are usually aren't security experts. Thus, closed source benefits from its "closed" nature in this aspect—its worldview centers on keeping certain "secrets" secret.

Conversely, the open source world is based around information exchange. Changing the open source worldview on this matter with respect to security is really the crux of the solution but is somewhat in contradiction to the culture. Despite the solid progress several open source projects are making in this area (bugs are increasingly discussed in private, not in public forums), as soon as a patch is released it is trivial to determine the exact details of the patch. This makes developing an exploit for the previous version *much* simpler.

TRUSTING TRUST

Ken Thompson's paper "Reflections on Trusting Trust" is as important today as it was when first penned in 1984.[4] Thompson illustrated the trust assumptions we make when deciding on security-related issues. Ultimately, he argues, we're trusting far more than we might realize. The same argument holds when considering open/closed source security.

Classically, security people tend to think of the attacker as either a malicious insider or a third party. It's also possible, however, to think of the software vendor—in its entirety—as untrustworthy (because one suspects the vendor is either malicious or incompetent). What then?

This change of focus in terms of trust can be a little startling, but isn't entirely far-fetched. It doesn't even require malfeasance on the part of the vendor. Consider a well-meaning (but foolish) vendor who, during an

install, disables a critical piece of security software, with the intent of restoring it at the end of the install. Such a vendor could be unwittingly placing the user at risk. Incidents such as the Sony rootkit, used for DRM (digital rights management) purposes, also emphasize the sometimes misplaced trust placed in vendors. In each case, the closed source nature of the project put the user in jeopardy because there was no way—aside from reverse engineering—to determine the real functionality of the software.

There is also the issue of unethical vendors deliberately sneaking adware onto your computers under the guise of a "utility." Vendors aren't inherently trustworthy, and anyone who *blindly* makes the assumption that they are is either in denial or naïve.

In the case of an untrustworthy vendor, open source provides at least a mechanism by which a concerned entity can verify (within reason—remember the implications of Thompson's paper) that all is well. Going to the trouble of auditing the entire code base for a project isn't justified in many cases, but I can provide an example that is difficult to refute: voting software.

The idea of trusting a single vendor with the legitimacy of elections is, frankly, terrifying. With so much at stake, voting software must be verified by source inspection—who would trust a black-box approach to voting? Clearly, in the case of such software, an open source approach provides at least a mechanism by which the software's veracity can be verified. Does one vote entered tally up with one vote counted in all scenarios? Although the process is nontrivial in an open source world, it's really *very* challenging in a closed source scenario where one must resort to reverse-engineering the system. Thus, in some cases, it seems the open source approach clearly has the edge.

An interesting counterpoint can be found in security software. Consider antivirus software. While much antivirus software is signature-based, many different incarnations of *generic* virus protection exist that attempt to apply different techniques to stop new viruses. Such software is important, as it provides a first line of defense against rapid worms, which can become pandemic minutes after their initial release. Generally, such software is not *theoretically* secure—it is heuristic in nature and can be bypassed by an attacker with sufficient knowledge. This being the case, an open source approach is probably less attractive than a closed source one. Let's at least make the life of the attacker a bit harder. If that sounds like security through obscurity, hold on to your seat for a moment: it is.

SECURITY THROUGH OBSCURITY?

The idea of "security through obscurity" has a horrible reputation among software engineers. I can still remember mentors through the years drumming into my head the idea that security by obscurity is no security at all (I expect that some of those fine scientists will contact me as they read this article to see where they went wrong in my education), but my belief is that the entire argument is highly contextual. For example, passwords are the perfect example of "acceptable" security through obscurity: they are useful only if the attacker doesn't know them.

Again, let me illustrate my position by using an example: DRM software. Any time one is attempting to protect software from unauthorized copying, one runs into the idea of security through obscurity. Essentially, if the computer can run the software, it's almost certainly going to be possible to copy it. Similarly, with a copy-protected document, if all else fails, I can always take a picture of my screen. Almost all DRM software is, at some level, security through obscurity: the bar is set only so high. The trick is making sure it is *high enough* to deter most attackers. Similarly, the protection offered by Microsoft Windows Vista's much-discussed Kernel Patch Protection is of far less value if the source code is available. This would allow attackers to chart the fastest route around it.

A counterpoint once again highlights the context I'm talking about: encryption. As computer scientists, we can make encryption arbitrarily difficult to break given currently known technology. If breaking the code involves factoring a very large number, I can make good predictions of how much effort an attacker needs to spend, and that time doesn't really depend on the attacker's knowledge of my software or algorithm. For such software, the best route to security is to publish the algorithm and let it be independently verified. So, what's the difference?

The difference between these cases is simple: determinism. In the case of the encryption software, the outcome is deterministic. Knowing everything about the mechanism doesn't compromise the security of the outcome. In contrast, for antivirus software the system is heuristic. As such, some things benefit from disclosure, and some things don't. In these two cases, it's obvious. Unfortunately, that's the exception, not the rule. The problem is that many systems contain aspects that are heuristic and aspects that are deterministic.

For a word processor, the question is different. You might like your word processor to work reliably, but the truth is that it contains bugs, and, potentially, security vulnerabilities. The closed source approach makes it expensive for anyone other than the developer to find

Open vs. Closed

those bugs. The open source approach means it's easy for anyone trained in secure coding practices to find weaknesses. Both of these properties are double-edged, and it's not clear which provides the best long-term outcome.

CONCLUSION

Part of the reason why this topic is interesting is because it is difficult: there are arguments on both sides that are compelling. By being able to understand the nuances of the question better, different aspects begin to become clear. Both development methodologies have intrinsic properties: which set of properties most appropriately fits for a particular application is contextual.

Unfortunately, the cases where one is clearly better than the other are few and far between. Most software sits somewhat uncomfortably between the two. In such cases, the makeup, philosophy, and training of the team behind the software are far more important than whether the project is open or closed source. Both methods can be done well, and both can be done badly.

Understanding where each method is strong and where it is weak is the first step toward process improvement. Instead of focusing on either/or decisions, perhaps it is ultimately more fruitful to follow both, using each where appropriate. Software engineering is a young discipline; time will answer if we approach the question with full knowledge of our assumptions and shortcomings. Q

REFERENCES

1. Arbaugh, W. A., Fithen, W. L., McHugh, J. 2000. Windows of vulnerability: A case study analysis. *IEEE Computer* 33 (December): 52-59.
2. Anderson, R. J. 2002. Security in open versus closed systems—the Dance of Boltzmann, Coase and Moore. Presented at Open Source Software Economics.
3. Raymond, E. S. 1999. *The Cathedral and the Bazaar*. Sebastapol, CA: O'Reilly.
4. Thompson K. 1984. Reflections on trusting trust. *Communications of the ACM* 27(8): 761-763.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

RICHARD FORD graduated from the University of Oxford in 1992 with a D.Phil. in quantum physics. Since that time, he has worked extensively in the area of computer security and malicious mobile code prevention. Previous projects include work on the Computer Virus Immune System at IBM Research and development of the world's largest Web hosting system while director of engineering for Verio. Ford is an associate professor at Florida Institute of Technology, where he is the director of the Center for Security Sciences. His research interests include malicious mobile code, behavioral worm prevention, security metrics, and computer forensics. Ford is executive editor of Reed-Elsevier's *Computers and Security, Virus Bulletin* and co-editor of a column in *IEEE Security and Privacy.*

MORE

Related articles in ACM's Digital Library:

Joshi, A., King, S. T., Dunlap, G. W., Chen, P. M. 2005. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05* (October).

Mercuri, R. T. 2005. Security watch: Trusting in transparency. *Communications of the ACM* 48(5).

Neumann, P. G. 1999. Inside risks: robust open-source software. *Communications of the ACM* 42(2).

Viega, J. 2005. Security—problem solved? *ACM Queue* 3(5).

These articles will be available online at www.acmqueue.com for an eight-week period beginning Feb. 1. Want full access? Join ACM today at www.acm.org and sign up for the Digital Library.