Talk delivered at IFIP WCC 2004, Toulouse, France

Trusting Strangers: Open Source Software and Security

Carl E. Landwehr
Institute for Systems Research
University of Maryland, College Park

Abstract:
The issues of trusting software, certifying security, and the relative merits of open and closed source software as a basis for critical systems are discussed. It is concluded (i) that neither approach in itself can assure the absence of security flaws or sabotage, (ii) that better methods are needed for assuring the properties of products without respect to the people or process used to create them, and (iii) that system architects should exploit what they know they don't know, as well as what they do know, in composing system architectures.

1. Software and Trust

We trust many artifacts that we do not personally investigate or even understand. Most of us probably understand how a rowboat or a bicycle works, and we can usually see all of its functioning parts. But using a car, an airplane, or almost any modern appliance involves relying on technology that we don't examine directly and which in many cases we understand only at some surface level. Few of us understand the detailed calculations and inspections that underlie the safety of public buildings in which we enjoy concerts and films or the microwave ovens in which we may cook dinner. We in fact rely on the anonymous strangers who design, build, deliver, and in some cases maintain that artifact, as well as the processes that are in place in our society to reward or punish them.

Software is an unusual sort of artifact in that it has little physical substance, yet it can convey information, possibly sensitive information, and it can control physical devices. It has a significant cost of design and implementation, yet very low cost of replication. Very minor changes in its physical realization can cause major changes in system behavior. Though a great deal of money is spent on it, it is rarely sold. Usually it is licensed, customarily under terms that relieve the producer from nearly all responsibility for its correct functioning.

2. Certifying Security

There are many perspectives for contrasting open and closed source software: purchase price, maintenance cost, reliability, performance, safety, interoperability, and so on. The focus here is from the security perspective, from which we consider the response of software to potential acts of malice.

Safety critical systems for public use frequently depend on certification of process combined with some degree of inspection and testing. These processes, particularly when

combined with both the threat of legal liability for the consequences of accidents attributed to poorly engineered products, and the possibility of customers avoiding products and companies whose products have proven unsafe, seem to work reasonably well. There are not regular reports of airplanes falling from the sky or trains colliding because of faulty software. Continued vigilance in these domains is of course required.

Unfortunately, these processes do not seem to be operating so well with regard to security. The possibility of malicious use, or the malicious insertion of subtle flaws that might later be maliciously exploited is not in general contemplated by the certification processes applied in safety.

There are, of course, regular and increasing reports of security failures in systems throughout the world. The increasingly common malicious acts of creating and distributing worms and viruses primarily exploit accidentally introduced flaws in widely distributed software. Economic incentives (e.g. for platforms from which to send spam or mount distributed denial of service attacks) motivate the covert installation of malicious, remotely controllable software on vulnerable platforms.

There are also mechanisms for certifying the security properties of software. The primary mechanism at present is the provided by the "Common Criteria" scheme, which provides a somewhat complex means to specify the security properties and assurance level required of some particular component and to evaluate whether those properties are present in some particular target of evaluation.

There are two significant problems with this scheme: one is that at the levels of assurance most commonly sought, the source code of the system is never even looked at by the certifiers; their tasks are primarily to assure that the system's specifications are properly in order (and often these are created strictly for the certification process), to perform some level of testing of security functions, and to see that mechanisms are in place to assure that the software delivered is the same as what was evaluated. Yet the kinds of software flaws most commonly exploited by today's attacks are not likely to be discovered without access to the source code. Second, this scheme remains component-oriented, while security remains a system property. This is not to say that the scheme is without merit, but it is very definitely limited in what it can achieve, and its cost-effectiveness has never been assessed.

But in a world where the headlines are being made by suicide bombers, nations continue to engage in well-funded intelligence gathering activities, and daily electronic transfers of funds in the US Fedwire system alone average more than $1 trillion, simple software flaws are not the only concern in assessing software security. Software that may wind up in critical systems may be the target of specific attacks. Indeed, a recent book asserts that the U.S. made sabotaged pipeline control software available for the Soviet Union to obtain more than twenty years ago, and this software in the end triggered a major pipeline fire in Siberia. While I have no direct knowledge of the truth or falsity of this report, it is difficult to deny that that famous fire might have been triggered in this way. An expert might well be able to hide such software sabotage so that it is not even visible in the

source code, using methods like those documented in Ken Thompson's famous Turing lecture. Software can be highly inscrutable even if the source code is available; the effects of asynchronous operations and feature interactions are notoriously difficult to understand.

3. Open vs. Closed

Do these considerations weigh on one side of the balance or the other for open source software?

Open source software has the advantage that any interested party can apply arbitrary tools to investigate it, to rebuild it, and to modify it to suit specific needs. It may be compiled by different compilers and linked by different linkers and the results compared. It can be examined by an arbitrary third party in as much detail as the sponsor can afford. But liability is unlikely to be present as a potent force in this case, since the user takes on the responsibility of composing, and potentially changing, the software. Control over the input to the source code may be uncertain, since it may have contributions from around the globe. And the fact that the proverbial "thousand eyes" could examine it does not mean they will, or that all of those eyes will be friendly -- they may be looking for holes, or places to insert subtle back doors. There is ample evidence that flaws can persist unseen in source-available software for decades. People may donate effort to creating open source software, but the evidence to date is that, except in a few notable cases (e.g., OpenBSD), they won't donate very much effort to providing competent security review of it.

Closed source software has the benefit of the producer's economic interest in the product. This interest should not be underestimated; it is (or can be) a powerful force for assuring product quality in a competitive marketplace. It can drive strong measures for control of software development, for testing of software prior to release, for configuration control of the released product and for prompt repair of defects. It is the value of the product in the marketplace, and hence its value as a corporate asset, that can justify this investment. Regardless of one's view of the result, it is certainly significant that after years of shrugging off security concerns as irrelevant to the market, since January 2002 Microsoft has apparently invested heavily in improving its programming practices from the standpoint of security and in changing the tradeoffs it makes in determining what features to enable by default. Nevertheless, even lacking the source code, hackers continue to find and exploit security flaws in Microsoft and other closed source software products. Further, commercial software development is increasingly a global enterprise, even if it is conducted within one corporation.

4. Conclusions

I propose three general conclusions from these observations:

1. Caveat emptor. Neither the open source model nor the closed source model has done a very good job of producing "bullet-proof" software except when someone has been

willing to make a significant investment for that purpose. Exposing the source code does nothing per se to improve its security properties. Neither does hiding it.

2. Seek product and architectural assurance rather than process assurance. As software, closed or open source, is developed by groups of people in many diverse locations around the world, it will become increasingly difficult to have confidence that a particular critical system is secure because its software was produced by people you trust or by people using a process you trust. Rather, we need stronger methods for examining the actual code run on critical systems to assure that it has the intended properties, and where we can't gain that assurance, we need architectural methods to limit the damage it can cause.

3. Exploit what you know, and what you know you don't know, about the software in your system. If you are using open source software, take advantage of the opportunity to review it, consider whether you should reconfigure it or rebuild it. If you are dealing with a vendor of closed source software, investigate the development processes used, consider the possible motivations of the developer, take account of any independent evaluations of the software, or the lack thereof. Use this information to develop a system architecture that manages the risks you know about.

ACKNOWLEDGEMENT

Thanks to my Maryland colleague Michael Hicks for helpful comments on a draft of this note.