The Software Industry /S the Problem

The time has come for software liability laws.

Poul-Henning Kamp

One score and seven years ago, Ken Thompson brought forth a new problem, conceived by thinking, and dedicated to the proposition that those who trusted computers were in deep trouble.

I am, of course, talking about Thompson's Turing Award lecture—"Reflections on Trusting Trust."² Unless you remember this piece by heart, you might want to take a moment to read it if at all possible.

The one sentence in Thompson's lecture that really, REALLY matters is: "You can't trust code that you did not totally create yourself."

This statement is not a matter of politics, opinion, taste, or in any other way a value judgment; it is a fundamental law of nature, which follows directly from pure mathematics in the general vicinity of the works of Turing and Gödel. If you doubt this, please (at your convenience) read Douglas Hofstadter's classic *Gödel, Escher, Bach*, and when you get to the part about "Mr. Crab's *record player*," substitute "Mr. Crab's *laptop*."

GÖDEL, ESCHER, BACH

Hofstadter's book, originally published in 1979, does not in any way detract from Ken Thompson's fame, if, indeed, his lecture was inspired by it; 1979 was a long time ago, and it's possible that not every reader may know of—much less have read—this book. My editor proposed that I summarize or quote from it to make things clearer for such readers.

Considering that Gödel, Escher, and Bach are all known for their intricate multilayered works and that Hofstadter's book is a well-mixed stew not only of their works, but also of the works of Cantor, Church, Gantör, Turing, and pretty much any other mathematician or philosopher you care to mention, I will not attempt a summary beyond: "It's a book about how we think."

The relevant aspect of the book here is Gödel's incompleteness theorem, which, broadly speaking, says that no finite mathematical system can resolve, definitively, the truth value of all possible mathematical conjectures expressible in that same mathematical system.

In the book this is illustrated with a fable about Mr. Crab's "perfect record player," which, because it can play any and all sounds, can also play sounds that make it resonate and self-destroy—a vulnerability exploited on the carefully constructed records of Mr. Crab's adversary, Mr. Tortoise.

Mr. Crab tries to protect against this attack by preanalyzing records and rearranging the record player to avoid any vulnerable resonance frequencies, but Mr. Tortoise just crafts the sounds on his records to the resonance frequencies of the part of the record player responsible for the rearrangement. This leaves Mr. Crab no alternative but to restrict his record playing to only his own, preapproved records, thereby severely limiting the utility of his record player.

Malware-scanning programs try to classify executable code into "safe" and "unsafe," instead of mathematical conjectures into "true" and "false," but the situation and result are the same: there

invariably is a third pile called “cannot decide either way,” and whatever ends up in that pile is either a security or a productivity risk for the computer user.

Amusingly, malware scanners almost unfailingly classify malware-scanner programs, including themselves, as malware, and therefore contain explicit exemptions to suppress these “false” positives. These exemptions are of course exploitable by malware—which means that the classification of malware scanners as malware was correct to begin with. “Quis custodiet ipsos custodes?” (Who will guard the guards themselves?)

BACK TO THOMPSON

In 1984, the Thompson lecture evoked wry grins and minor sweating for Unix system administrators at universities, because those were the only places where computers were exposed to hostile users who were allowed to compile their own programs. Apart from sporadic and mostly humorous implementations, however, no apocalyptic horsemen materialized in the sky.

In recent years, there have been a number of documented instances where open source projects were broken into and their source code modified to add backdoors. As far as I am aware, none of these attacks so far has reached further than the lowest rung on Ken Thompson’s attack ladder in the form of a hardcoded backdoor, clearly visible in the source code. Considering the value to criminals, however, it is only a matter of time before more advanced attacks, along the line Thompson laid out, will be attempted.

The security situation with commercial closed-source software is anyone’s guess, but there is no reason to think—and no credible factual basis for a claim—that the situation is any different or any better than it is for open source projects.

The now-legendary Stuxnet malware incident has seriously raised the bar for just how sophisticated attacks can be. The idea that a widely deployed implementation of Java is compiled with a compromised compiler is perfectly reasonable. Outsourced software development does not make that scenario any less realistic, likely, or scary.

WE HAVE TO DO SOMETHING

We have to do something that actually works, as opposed to accepting a security circus in the form of virus or malware scanners and other mathematically proven insufficient and inefficient efforts. We are approaching the point where people and organizations are falling back to pen and paper for keeping important secrets, because they no longer trust their computers to keep them safe.

WHAT CAN WE DO?

Ken Thompson’s statement –“You can’t trust code that you did not totally create yourself”—points out a harsh and inescapable reality. Just as we don’t expect people to build their own cars, mobile phones, or homes, we cannot expect secretaries to create their own text-processing programs nor accountants to create their own accounting systems and spreadsheet software. In strict mathematical terms, you cannot trust a house you did not totally create yourself, but in reality, most of us will trust a house built by a suitably skilled professional. Usually we trust it more than the one we might have built ourselves, and this even when we may have never met the builder and/or when the builder is dead. The reason for this trust is that shoddy construction has had negative consequences for builders for more than 3,700 years. “If a builder builds a house for someone, and does not

construct it properly, and the house which he built falls in and kills its owner, then the builder shall be put to death.” (Hammurabi’s Code, approx. 1700 BC)

Today the operant legal concept is “product liability,” and the fundamental formula is “if you make money selling something, you’d better do it properly, or you will be held responsible for the trouble it causes.” I want to point out, however, that there are implementations of product liability other than those in force in the U.S. For example, if you burn yourself on hot coffee in Denmark, you burn yourself on hot coffee. You do not become a millionaire or necessitate signs pointing out that the coffee is hot.

Some say the only two products not covered by product liability today are religion and software. For software that has to end; otherwise, we will never get a handle on the security madness unfolding before our eyes almost daily in increasingly dramatic headlines. The question is how to *introduce* product liability, because just *imposing* it would instantly shut down any and all software houses with just a hint of a risk management function on their organizational charts.

A SOFTWARE LIABILITY LAW

My straw-man proposal for a software liability law has three clauses:

Clause 0. Consult criminal code to see if any intentionally caused damage is already covered. I am trying to impose a civil liability only for unintentionally caused damage, whether a result of sloppy coding, insufficient testing, cost cutting, incomplete documentation, or just plain incompetence. Intentionally inflicted damage is a criminal matter, and most countries already have laws on the books for this.

Clause 1. If you deliver software with complete and buildable source code and a license that allows disabling any functionality or code by the licensee, then your liability is limited to a refund. This clause addresses how to avoid liability: license your users to inspect and chop off any and all bits of your software they do not trust or do not want to run, and make it practical for them to do so.

The word *disabling* is chosen very carefully. This clause grants no permission to change or modify how the program works, only to disable the parts of it that the licensee does not want. There is also no requirement that the licensee actually look at the source code, only that it was received.

All other copyrights are still yours to control, and your license can contain any language and restriction you care to include, leaving the situation unchanged with respect to hardware locking, confidentiality, secrets, software piracy, magic numbers, etc. Free and open source software is obviously covered by this clause, and it does not change its legal situation in any way.

Clause 2. In any other case, you are liable for whatever damage your software causes when used normally. If you do not want to accept the information sharing in Clause 1, you would fall under Clause 2 and have to live with normal product liability, just as manufacturers of cars, blenders, chainsaws, and hot coffee do. How dire the consequences and what constitutes “used normally” are for the legislature and courts to decide.

An example: A salesperson from one of your longtime vendors visits and delivers new product documentation on a USB key. You plug the USB key into your computer and copy the files onto the computer. This is “used normally” and should never cause your computer to become part of a botnet, transmit your credit card number to Elbonia, or send all your design documents to the vendor.

The majority of today’s commercial software would fall under Clause 2. To give software houses a reasonable chance to clean up their acts and/or to fall under Clause 1, a sunrise period would make

sense, but it should be no longer than five years, as the laws would be aimed at solving a serious computer security problem.

And that is it, really. Software houses will deliver quality and back it up with product liability guarantees, or their customers will endeavor to protect themselves.

WOULD IT WORK?

There is little doubt that my proposal would increase software quality and computer security in the long run, which is exactly what the current situation calls for.

It is also pretty certain that there will be some short-term nasty surprises when badly written source code gets a wider audience. When that happens, it is important to remember that today the good guys have neither the technical nor the legal ability to know if they should even be worried, as the only people with source-code access are the software houses and the criminals.

The software houses would yell bloody murder if any legislator were to introduce a bill proposing these stipulations, and any pundits and lobbyists they could afford would spew their dire predictions that “this law will mean the end of computing as we all know it!”

To which my considered answer would be: “Yes, please! That was exactly the idea.”

REFERENCES

1. Hofstadter, D. 1999. *Gödel, Escher, Bach*. Basic Books.
2. Thompson, K. 1984. Reflections on trusting trust. *Communications of the ACM* 27 (8): 761-763;
<http://m.cacm.acm.org/magazines/1984/8/10471-reflections-on-trusting-trust/pdf>.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

POUL-HENNING KAMP (phk@FreeBSD.org) has programmed computers for 26 years and is the inspiration behind bikeshed.org. His software has been widely adopted as “under the hood” building blocks in both open source and commercial products. His most recent project is the Varnish HTTP accelerator, which is used to speed up large Web sites such as Facebook.

© 2011 ACM 1542-7730/11/0900 \$10.00