Warranties Solve Software Security

How do we cultivate enough security talent? Once we have it, how do we get it applied to the right problems, in the right projects, at the right times, with the right material support, forever?

Apparently not how open source does it now.

The solution to the open security crisis isn't standardized packing slips, a centralized inspection agency, or a sign-off scheme. It's not more software or another foundation. This isn't actually a software problem. It's much bigger than that.

It is also, thankfully, a solved problem. It's just that open source *unsolved* it.

Known Class

The software-security problem is a specific instance in the general class of product-service quality problems. Much as drivers need reliable cars and depositors need dependable banks, users need and want secure software. How secure varies from case to case. Trouble is, users can't tell on their own, in a hurry, if particular software's as secure as they need, or whether it will stay that way.

The software's developers are best positioned to know how secure it is, and what it will cost to keep it secure. Especially with the help of security expertise, if they can afford it. So software users know best how much security they need and how much they can afford to pay for it, but software developers know best how secure their software already is and how much more security they can get at what cost. To allocate resources—the costs of developer attention and even scarcer security talent—we need these signal paths connected.

Fortunately, the general product-service quality problem has a general solution: written guarantees, backed by financial responsibility, enforced by courts. In legal terms, "warranties" and "indemnities"—promises to cover customers financially when bad things happen. For software security, this means promises from software vendors to cover customers financially when vulns and breaches cause them losses, written out in license agreements for software and terms of service for services. The effect, for a customer, is a bit like buying insurance along with software.

Warranties and indemnities encode security signals in price. If users need more security coverage—higher dollar limit, coverage in more situations—they pay more for it. The vendors, receiving the money, and can therefore afford to spend more on security. If the vendors fail to prevent problems, they end up owing customers for the consequences.

Each customer-vendor pair forms a link in a network conveying resources and information about the appropriate level of security. Nodes in that network can be entirely confidential, covered by nondisclosure agreements, but still convey security-relevant signals and resources transitively. A framework vendor can cover a website provider who in turn covers its own user-customers. The end user-customers don't need to see the vendor-provider contract. Each customer, who may in turn be the next customer's vendor, sees only prices and terms with their immediate counterparties.

Market System

Warranties and indemnities depend on public courts. But they aren't themselves any centralized institution. They rely on a universal and mandatory system of justice, but facilitate autonomous, uncoordinated dealmaking upon that foundation. The result in the large is a constellation of self-assembling, self-balancing, many-nodes, many-edges networks of responsibility-resource exchanges. In other words, a market-based supply chain.

Markets have known failure modes, albeit far fewer than public-goods crisis chasing. Many known failure modes also have known workarounds.

For example, market-based supply frequently does a good job balancing risk and reward with warranties and indemnities…until the last link in the chain, where a large company does business with an individual consumer. Individual customers frequently lack the leverage to get any terms but those offered to them. This can lead to outrageously unjust situations where consumers get badly hurt in utterly preventable ways, but their private terms of sale leave their vendor off the hook.

Products liability—a legal field focused on public laws that impose warranty- and indemnity-like terms on deals with consumers—steps in to impose mandatory minimums. Sometimes these laws give consumers the right to sue, either individually or, to overcome collective action problems, collectively, through "class actions". Sometimes these laws empower regulators working for the public to sue on consumers' behalf. Sometimes both. We are beginning to see more such laws for secure software as a service, implemented as security requirements under privacy laws.

Terms of protection aside, we also see that scurrilous vendors sometimes take customers' money, make the right promises, and either drain all the cash out of their businesses or fail to invest in hard-to-gauge qualities, like security. Comes a problem, customers find themselves with shoddy products or strong, worthless legal claims against vendors with no money left to sue for. This tends to happen especially with young, small companies—so-called "fly by night" operations. Thus evolved liability and "cyber" insurance.

If customers suspect a vendor will take their money for security but fail to invest in prevention or pay out for a loss, they can insist the vendor carry insurance to cover. This brings a third party into the picture, for accountability. An insurance company will look at the commitments the vendor makes to the customer, as well as their security practices. They'll compute a premium for the insurance policy accordingly. If a customer has a problem and the vendor has to pay, the insurance company will cover the cost. But they will likely also increase the vendor's premium, or simply refuse to keep insuring them.

Insurance can also help address problems from more innocent misunderstandings and misallocations. Both the value the customer puts on functionality and the value the customer puts on security get encoded in one price. The vendor can't tell from one figure how many dollars the customer attributes to functionality or security. On the return end, the functionality of the software the customer gets back may be clear, but the degree of security may not be. But the customer's insistence on insurance can function as a side channel, expressing security-specific concern and triggering security-specific risk assessment. The vendor may charge the customer more, to cover the cost of insurance. That's fine: the more security the customer needs, the more they should pay for. If multiple customers express the same concern, the cost of insurance can be spread across their accounts.

Open Source

Open source licenses break the chain of security responsibility. Customers don't pay developers for open source. But neither do customer's get security assurances. Security resources do not flow through open source distribution channels, and neither do price-encoded security signals.

I am a ridiculous person to be writing this on the Internet. In the late 1990s and early 2000s, I was one of the kids with dial-up flaming the business meanies harping on open source as unprofessional, negligent, and unwise without “a vendor standing behind the product”.

The idea that people would use software without paying for it wasn’t crazy, naive, or unethical. I didn’t have any money. It was awesome. The idea that people would use software without getting any assurance of quality, safety, or reliability wasn’t reckless, demeaning, or unprofessional. I was doing nothing important, and my time spent was time spent learning, not time wasted. As long as “user beware” meant more code for zero dollars, it was all good for me.

Every popular open source license says something like this, usually in `SHOUT CAPS`:

> The software is provided “as is”, without warranty of any kind, express or implied.... In no event shall the developers be liable for any claim....

Here it is is modern translation, from the Blue Oak Model License:

> ***As far as the law allows, this software comes as is, without any warranty or condition, and no contributor will be liable to anyone for any damages related to this software or this license, under any kind of legal claim.***

There are no warranties. There are no indemnities. There is no liability. Definitely none about security.

If the terms didn’t mention liability at all, what would aggrieved users have to point to in court? Why does a license have to say this at all? Because the law, in its labyrinthine wisdom, sets some reasonable defaults. When things get bought and sold, the law implies warranties to match reasonable expectations.

When people buy stuff, they tend to expect it to work, at least for a while. They expect sellers to refund or replace if it doesn’t work, at least for a while. If a salesman helped them decide what they needed to buy, they expect what they bought to meet their need. All this in the context of a “sale”, which could be a straightforward trade of money for wares, but could also involve freebies in some greater context of overall exchange. Hence the language in open source licenses, even though money often isn’t directly involved.

Some jurisdictions, including a few states of the USA, and, I’m told, some foreign countries like Germany, actually insist on default warranties for software *no matter what software license terms may say.* The warranties aren’t defaults but requirements—the products-liability approach. There have been big pushes, even in the United States, to apply more products-liability-style laws to software, at least when individual consumers are buying. Under such laws, the common open source “no warranties” and “no liability” language can’t actually do all that it says. You can’t zero out responsibility, even if you zero out price. Which is why the Blue Oak Model language starts “as far as the law allows...”.

Missing Middle

We see two extremes: total disclaimers, on one hand, and mandatory protections, on the other. As usual with extremes, mismatch errors of all types abound.

Under open licenses, nobody gets security assurances, even when software flows from Fortune 500 to Fortune 500, even in the context of clear business exchange. All these firms know how to do that deal, and used to insist on those terms, before open source “ate the industry”.

Under products-liability law, everyone gets security assurances, even when software passes from mendicant hacker to broke student, even in the context of anti-FAANG collaborative activism. Some people don’t need protection, and can’t afford to pay for it.

The go-to justification for wiping out warranties on open source is that developers aren’t charging. Why take the risk for no reward? Of course, we might also ask: Why make and maintain the software in the first place?

There's no reason warranties couldn't become just another form of loss-leading. If my company wants to own a market segment by loss-leading a complement, no rule stops your company from swooping in and offering your own, competing project, plus a strong warranty of security. You could even post bond, or take out insurance, and brag about that publicly. Still a third competitor could move in, pointing out how much bigger their bank balance is than both of ours, and therefore how much better they can afford to "stand behind" their competing project.

Whether compensation is paid "for software" or called "a license fee" is irrelevant. Whether comp takes the form of cash is orthogonal. Selling software cheap can be an act of charity. Think student discount or emerging-market pricing. Giving it away can be the cruelest commercial ploy. Think "commodify your complement" or malicious product dumping. In the end, people want software to do useful things. They also want it to not leak their data, lend their hardware to malware gangs, or cause myriad other security mayhem.

The argument for using open source without guarantees where security matters boils down to "we get away with it"—an argument not from reason but experience. We hear this reprised whenever assembly programmers bump into C programmers, C programmers bump into interpreted-language programmers, and just about anybody bumps into JavaScript programmers. How do you trust all this code you're importing from somebody—anybody—and sell it customers. Like you do.