Buy/by the Book or Bye-Bye the Game
A talk for LangSec Workshop, 2016

M. Douglas McIlroy

Dartmouth College

ABSTRACT

To banish the ominous scenario of garbage in, evil out, we must reject bad stuff no matter how closely it resembles good. For that, we need to precisely define the good, assure that it can be identified accurately, and handlle it correctly. Formal methods come to the fore. If you call this a platitude, you are right. The hard part is to thoroughly buy into it and keep the faith in the face of myriad temptations and compulsions to do otherwise. Among the challenges are technical subtleties that lurk behind the innocuous adverb "thoroughly".

I questioned Sergey when he invited me to give this talk. What could someone who hasn’t been in the trenches since before the turn of the century offer an audience of folks facing genuine 21st-century problems? Well, maybe I’d been there and done that, or maybe 60-odd years of experience would somehow lend gravitas, or maybe my sympathy for formal methods is especially apt in the setting of LangSec.

In any any event, if I’m not still in the trenches, I do know what it’s like to be in a hole. My first experience of international data theft happened exactly 40 years ago.

I had written a real-time text-to-speech program. It was something of a sensation, and I showed it off many times. It depended on a special computer attachment called the Votrax made by guess what famous electronics company? ... Federal Screw Works. To demonstrate the program live, I would feed input to our lab computer via dial-up modem and get the audio via a speakerphone connection.

Audiences participated enthusiastically as they vied to defeat the program’s heuristics for sounding out the famously non-phonetic spellings of English words. Of course the program melted down on foreign names like Beethoven, which came out "beet oven". But the best challenge came from Professor Saul Gorn at the University of Pennsylvania. He proposed "coworker". The computer gamely responded "cow orker".

But I digress. You folks are supposed to be interested in vulnerability of interfaces, not of pronunciations. Bear with me, though. I’m getting getting there.

One of these talking-computer demmos brought me face-to-face with computer crime. The venue was student computer club in Canada.

So. Unix had good password protection ... in principle. Because I had nothing to hide, I chose the empty password. Sure, I had superuser rights, but those were protected by a real password.A few weeks after the demo, a phone bill forwarded to the computer club listed the calls to our computer and Votrax. Curious students tried it, guessed my easy login name, and were rewarded by a free login.

They promptly looked around for the system source, fetched it, diffed it with the legitimate version they had, and folded interesting changes into their locally modified system. Some time later, a faculty member got wind of the caper, and with great chagrin called us to apologize, for by then there was no way he could unwind the misdeed.

My blithe belief that my password didn't matter led to international software theft. Fortunately the stolen software was soon released, and the misdeed was erased.

The incident was certainly due to user error. But it was developer files that got stolen. It would have been wise to vet the complexity of passwords, much as that might offend the cleanliness of Unix design.

The incident made me gun-shy. Not long after, the office of the CEO of AT&T learned that our lab could do typesetting. Could we teach one of their secretaries to enter speeches for the chairman and have them printed in big type so he wouldn't have to wear glasses in public? Sure, why not? It's always nice to have a channel to the top.

Of course, once they learned to enter documents, they started using our computer for other things. Among them were minutes of board meetings, marked private. As the most visible Unix system in the world, we were regularly poked at. We also had a dozen or so superusers. We couldn't guarantee to protect board minutes. (This was attested by the very fact that we had discovered them.) Not wanting to be burned again, I invited the vice president for public relations to buy a computer for the chairman’s office. And thus Unix got embedded at the apex of AT&T.

The Unix theft was indeed a lesson. Today most everybody in important projects is fully aware of the hazards of shortcutting security on the grounds that ‘‘nobody knows about’’ some subvertible feature. Yet the phenomenon persists in profusion.

Back in the 1980s, a developer of AT&T switching systems showed one of our people how they could patch central offices remotely. This they did by calling up a small Unix system connected to a switch. The onlooker was startled to see that the call was answered not with a login challenge but with a superuser prompt. The developer asssured him that was OK because nobody knew the phone number. The idea that it’s no big deal to check 10,000 phone numbers was news to him.

Thirty years on, *c’est la même chose.* A few weeks ago I attended a talk [by Dan Tentler, atenlabs.com] that presented dozens of examples from the internet of things that were vulnerable due to the same blasé attitude, that what you think they don’t know can’t hurt you. All kinds of control systems, especially heating, ventilation, and air-conditioning systems, but also industrial and public-utility control systems, are sold with laughable protection against remote meddling. A crawl of pertinent IP ports is bound to be rewarded.

There was a lesson, too, in the affair of the CEO’s eyesight. No matter how benign expected applications may seem, sensitive uses may arise. It may seem rather harmless if an idle kid can tinker with the temperature of a faraway building. It can always be reset. But who knows what other mischief can be done at the controls? And in some buildings the temperature may really be critical, as in a biology lab, where a spike of heat or cold could ruin a long-running experiment.

Formal methods to the fore

So what do these stories have to do with LangSec? They all exemplify human inattention to potential threats or misapprehension of potential threats. The old saying, "to err is human", applies just as much to developers as to anybody else. The vulnerability of internet-connected control systems is a case in point.

Developers need all the help they can get to assure a sound product. All too many forces are arrayed against them. At the top of the list is marketing: the urge to get out there first with the most features. Performance is another driver; customers want more bang for the buck.

These demands come up against cost constraints: every project has limited resources, and the drive to allocate time and money to marketing objectives usually works at cross-purposes with less visible aspects of quality.

Safety and security tend to be early losers in the battle for resources. The short-term outcome of this tendency can be rewarding. It served Takata well for many years. Volkswagen pushed beyond mere dismissal of the issues into outright fraud. There’s hell to pay now.

These systemic obstacles arise partly due to insufficient imagination about the downside of the tradeoffs. They are essentially other manifestions of the belief that "what you think people don’t know can’t hurt you."

The pressures to produce more features and more performance bring further dangers. One is feature interaction. Roughly speaking, one needs to test each new feature against an ever-growing background of preexisting features. If only pairwise interactions matter, the marginal cost of testing a new feature grows linearly with the feature count.

I submit, though, that serious pairwise feature-testing is rarely done. What, for example, is the point of testing for interactions between supposedly unrelated features? For one thing, all too often nominal independence may have been violated, particularly in the interest of efficiency. For example, some basic service may have been surreptitiously accessed via a side door, leading to an inconsistent view of system state. Moreover if concurrency is involved (and when is it not?) testing may not be very revealing anyway. Finally there is the possibility of inherently multiway feature interactions, testing for all of which may be completely infeasible.

Marketing will almost always want more features while engineering will prefer less. The best hope that can be held out for reconciliation is that with a broader view one may find generalizations that cover a multitude of features. The rewards of doing so may well be worth the pain of major refactoring.

Performance demands bring another danger: optimization. Dijkstra likened optimization to walking on the edge of a cliff. I would go further. Successive optimizations are like fractal refinments: the more you optimize the hairier the edge of the cliff becomes. Almost any direction in which you might try to step leads into the abyss.

The LangSec vision foresees a shield against such dangers. More automatic methods of providing defensive measures can save development time, and thereby overcome management incentives to cut corners. At the same time they help ensure against developers' outright mistakes.

Thus a central concern of any approach informed by Langsec must be formal methods and their embodiment in mathematically based tools. Use them thoroughly. Use them early. Use them often.

Of course we all use lots of quality-assurance tools every day, for both generation and analysis of programs. Few people attempt to hand-code parsers any more. And largely automatic generators of device drivers have been around for decades, though they are far from well known.

On the analysis side, many compilers offer lots of diagnostics about things like possibly unintended type conversions and code with no effects. Optimizing compilers often give program-flow diagnostics like detecting program paths that use variables before they are set. These services are free side benefits of something you had to do anyway.

Even if you're leery of radical optimization, it may be worth using it merely for the diagnostics that may issue from deeper analysis. More generally, If you don't always dial the warning level to the max, you're not with the program.

Programs like Lint look for for other questionable practices. Code-coverage analysis helps tune the effectiveness of regression tests. Microsoft subjects released software to even deeper analysis, such as automated recognition of potential buffer overflows.

Small tweaks are a well known source of grief. If a small change accomplishes what it was supposed to and passes a standard battery of regression tests, it's likely to be blessed as ready to go. The acceptance test for the change is added to the regression suite and the product is shipped. Other things you do with big new code, like checking the regression tests for code coverage, may seem to be unnecessary.

That’s just what I thought about a tiny enhancement I once installed in a main-line program. The program had never failed in the field, but this time I got an error report almost overnight.

When I first created this quite complicated program, I labored hard to construct tests that got 100% code coverage. Sure enough, the dozen-or-so lines of new code contained a branch that the regression tests never exercised. It was a corner case that I had thought of, but got wrong, and inadvertently installed without testing. If I had kept up my previous practice, my error would have been spotted by the coverage tool, and not by a customer.

The lesson is clear. No change is small.

No change at all? What could go wrong with something as trivial as correcting the spelling of a diagnostic message? Well, it might become longer and lead to buffer overflow or truncation.

In the matter of spelling, lots of software ‘‘gracefully’’ handles overlong names by truncating them. Eleven-digit mnemonic phone numbers like 800-CALL-AJAX count on the telephone system to do so. But all kinds of things go wrong when parties to a transaction have different ideas of what’s being said.

I remember one website-maintenance program that quietly truncated file names to 23 characters. This had the effect of mangling file extensions. With a file’s extension unknown, browsers would have to guess how to render it. The results could be wildly different depending on the heuristics used in guessing. Imagine, for example, what happens when you treat PDF as plain text.

And speaking of heuristics, browsers all contain AI in deference to Postel's notorious ''robustness'' principle. The hope that senders will be exacting in their conformance to standards and receivers should be tolerant of deviations might well be expressed as ''always trust strangers''. Way back when, the principle was commended in the HTML standard. It has long since been removed, but its legacy of chaos lives on.

LangSec's manifesto is the antithesis of Postel's principle. For LangSec robustness means intolerance for garbage, not tolerance, because who knows what diseases it may carry.

And yet, we really want to make sense of what people we want to hear have to say. At the same time, we know that computers lack judgment about whether or not to act on the basis incoming information. Is there no safe middle ground?

In fact there often is: garbage gets corrected by feedback, ranging from low-level ack-nak protocols up to customer-service hot lines. Here is a social-science LangSec problem. Garbage, of which nonconforming web pages are a prime example, should be corrected at the source, not the receiver. Can mechanisms be devised that will encourage correcting rather than perpetuating the trouble?

Protocol analysis and scaling up

Formal methods have been most thorougly adopted in the field of protocol design. Protocols are usually fairly small. Yet they are hard to reason about because of concurrency and nondeterminism.

Wise designers submit new protocols for automated model-checking before doing a line of implementation. As always, the sooner errors are found, the easier a less costly they are to fix. Thus the design should be rechecked after every tweak and refinement, no matter how trivial.

So far so good. But how closely connected is a checked model to the associated product? Is the model kept up to date with the product? How is it verified that the model actually represents the product?

Modern build systems generally assure that every change at least gets a regression test. But that's for the code. Model-checking is prebuild. Does your build system know anything about models? Is the only connection of models to the build system indirect? Is that indirection via parenthood, in which a coder works from the model? Or is it merely cousinhood, in which both model and product are created from the same informal spec—probably by different people?

In the hardware world, much implementation works straight from a model—logic equations. But there's a further trick that so far has almost no analog in software. That is logic extraction. To confirm that the tool chain has behaved faithfully, logic is inferred from the geometric layout of a chip and compared against the original equations.

In the messier world of software, Jet Propulsion Lab has made big strides towards assuring that the model reflects the product. They extract a model, at least at the level of process interactions from the code itself. I find the idea mind-boggling, but who can quibble with the Mars rover? ‘‘Opportunity’’ is still operating after 50 times its planned life span, with all of its code—not just protocols—having been verified by this technique.

Of course model extraction wasn't the sole program analysis tool on the JPL project. The code was written in C, and they had stringent rules about style. If I remember correctly, they insisted that function returns be used.

Even the most stringent of the customary style-checking programs I know do not go so far as to report on unused function returns. Presumably this is because the practice is so widespread—in particular in connection with error returns from I/O functions.

From a LangSec standpoint this is unpardonable How can a program that doesn't know it has misread data pretend to understand whether the data is good?

In terms of the title of this talk, even if you buy into the doctrine of LangSec, you won't really be living by it if you ignore the low-hanging fruit of error returns. That's one service of formal methods—to sweat the small stuff.

Dr. Richard Carlson says it's all small stuff and we shouldn't sweat it. He's right. As a psychotherapist, he says forget about it. Do that in LangSec's world and you've lost the game. Keep calm and rely on automated formal methods.

I have already mentioned optimizing compilers. They are certainly the most widely used program-analysis tools. Many, especially compilers for functional languages, have a substantial formal basis. But if you're like me, you shrink from using the highest optimization levels. These are monumental products whose capability has evolved by accumulating features for decades. Adverse feature interaction is a real worry. On the other hand they have had the bejeesus tested out of them.

Yet they still have bugs. Bill McKeeman subjected one production compiler to devilishly constructed random tests for two years. During that interval the rate of bug discovery remained fairly constant at about one per day. Not encouraging for LangSec. But look again at JPL. They set out a discipline of programming and as far as possible enforced it mechanically. They carried out extensive formal verification. When something turned out to be unverifiable (which isn't the same as being wrong), they would try to fix it so it could be verified or flag it for more detailed scrutiny.

It's the same way with buffer overflows at Microsoft. The specter of having to rewrite to shut off an automated complaint induces better writing practices in the first place.

Management

Deadlines and performance goals always loom over this Utopia of formal methods. We know in our hearts that cutting corners to get things done is likely to make a shabby product with maintenance problems down the road. But do we behave as if we believe it?

One symptom that we don't is premature optimization. Just because you know a fast way to do something is insufficient reason to do so. The Linux kernel, I am told, is full of binary-search table lookups. A corollary is that tables must be kept in order—a fact with far-reaching consequences, and an unnecessary flirtation with Dijkstra's cliff. One shouldn't take on the extra commitment until one is sure that it will make a real difference.

An example is the ISO protocol stack. Fear of a fully layered implementation running slow may spur one to handcraft a collapsed stack, keeping the separated concerns all in mind simultaneously. It was long ago shown that an orderly implementation of the stack can be collapsed automatically to meet the needs of special applications. [Optimizing Layered Communication Protocols, in Proceedings of the Sixth IEEE

International Symposium on High Performance Distributed Computing, 1997. 169 - 177] Work done once to make such a tool pays off handsomely in avoiding the labor and bugs of doing it repeatedly by hand.

And writing such tools is more fun than writing everyday code. The only reason not to is featherbedding, a term that describes the practice of requiring firemen on diesel locomotives, despite the fact that diesels have no boilers to stoke.

Still, what do you do when management complains you’re late because you’re too careful. Ideally—and I think practically, too—the methods should be justified up front. Bad product will come home to roost. But more to the point, if the methodology is thoroughly adopted you will probably win on both quality and schedule.

So it was in one trial back in my days at Bell Labs. A small team convinced management to let them work off to the side to build a competitor for a product being developed by traditional methods by a team at least five times the size. The competition was halted after the small team had found and fixed so many bugs in the specs that it would have been counterproductive to let the traditional team continue in the dark. [http://spinroot.com/gerard/pdf/hamburg94.a]

LangSec promises that kind of success. Go for it.