The Ken Thompson Hack

In 1984 KenThompson was presented with the ACM TuringAward. Ken's acceptance speech *Reflections On Trusting Trust* (http://cm.bell-labs.com/who/ken/trust.html) describes a hack (in every sense), the most subversive ever perpetrated, nothing less than the root password of all evil.

Ken describes how he injected a virus into a compiler. Not only did his compiler know it was compiling the login function and inject a backdoor, but it also knew when it was compiling itself and injected the backdoor generator into the compiler it was creating. The source code for the compiler thereafter contains no evidence of either virus.

Ken wrote, *In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well installed microcode bug will be almost impossible to detect.*

Ken does not mean bug in the sense of error, but in the sense of listening device. And it is "almost" impossible to detect because TheKenThompsonHack easily propagates into the binaries of all the inspectors, debuggers, disassemblers, and dumpers a programmer would use to try to detect it. And defeats them. Unless you're coding in binary, or you're using tools compiled before the KTH was installed, you simply have no access to an uncompromised tool.

In fact, given the amenability of microcode to the KTH, not even then.

All manner of controls and monitors could be secreted this way in the OSes of all the devices we all use day to day. It isn't very far fetched to suggest that the hack, in software, can create an updatable backdoor. This way every piece of software on the planet can be KTH bugged without any possibility of detection by any mortal engineer anywhere.

Well, maybe with the diligent use of an electron microscope.

Given last week's horrifying revelations concerning the US government's TotalInformationAwareness? of every US domestic phone call, it is difficult to imagine that the ThreeLetterAgency?'s KTH-hacked binaries are not omnipresent. I mean, can you really imagine AdmiralPoindexter? would pass up an ability like this?

Doesn't Ken's virus depend on the fact that the compiler (a pre-existing binary version thereof) was used to bootstrap itself? Had a different compiler been used to compile the compiler, the virus would have been killed.

No. Ken notes "I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode." Put it in the linker, say, and all your compilers are affected.

You could use a debugger to inspect the compiler's operation.

Perhaps a debugger toggled in via the patch panel on the front of a PDP-10 ... otherwise s/login/debugger/ in

what Ken said and paste in another trojan or two.

How about modification of the compiler sources for another reason? Might have fooled the "am I compiling myself" algorithm, or might have caused the Trojan to have been inserted incorrectly.

There are no C compilers out there that don't use yacc and lex. But again, the really frightening thing is via linkers and below this hack can propagate transparently across languages and language generations. In the case of cross compilers it can leap across whole architectures. It may be that the paranoiac rapacity of the hack is the reason KT didn't put any finer point on such implications in his speech ...

These examples are not given to rebut Thompson's point (which is a valid one), but instead to remind of good practices to prevent this sort of thing.

What practices? I can't think of any way to do it without the resources of a major TLA. KTH demonstrates that we are not able to fully trust any binary even when we have compiled it ourselves from trustworthy sources on a system we have compiled ourselves. The OpenSourceMovement? might have an off switch ... but your cell phone probably does not.

It's been more than twenty years since I read Thompson's marvelous paper, but I believe I correctly recall his fundamental point: UNIX, and every system like it, can NEVER be "secure". It doesn't matter how many layers of anti-virus software, "internet worm protection", "firewall" or any other buzzword -- systems like UNIX (including all versions of Linux, Macintosh OSX, and all versions of WinXP) will NEVER be secure. Thompson published his paper and revealed his hack in order to demonstrate this point.

Whereas what systems are secure? Ones that are not compiled from source code, presumably. And certainly not, say, Windows which you left of your list by sheer coincidence.

In August 2009 a virus utilizing the Ken Thompson hack was seen in the wild. It infected Delphi 4 through 7 and applications genereated with it. http://www.h-online.com/security/Virus-infects-development-environment--/news/114031

Although non-trivial, an effective counter has been identified: http://www.schneier.com/blog/archives/2006/01/countering_trus.html.

That's only (and trivially) effective if you already have a trusted compiler executable. The problem is generating one of those in the first place. Living in the era of PRISM, we can guarantee you can't get one online - KTH in the network will see to that. You sure didn't get one shipped with your OS - all the commercial OS mfrs are compromised and the open source ones all went through PRISM's internet.

The only way you're going to get a clean compiler executable now is to build your computer yourself out of TTL logic you salvage from some museum somewhere, then code up a set of basic binary tools sufficient to assemble assembler. Then bootstrap from the source of all the modern tools.

This actually sounds like a worthy project until you realize that the only person who could possibly trust your clean compiler executable is you. No one else will know for sure that you're not secretly working for PRISM. Or that you're not just too careless, or that black hats didn't hack their way into your garage and add the KTH while you weren't looking. And even if you got far enough to prove something, the great conspiracy would dose you with mind-altering chemicals or blackmail you into silence or discredit you or similar. Face it, little dude, against TheKenThompsonHack there simply is no defense.

Little dude? Why the put down? Who are you putting down, all of us? Please see GoodStyle.

{I don't think it was intended to be a put-down so much as a colloquial generic diminutive used as a rhetorical invective to imply weak superiority. It suggests the writer was not confident in his/her claim that "against TheKenThompsonHack there simply is no defense", and subconsciously sought to blunt the full force of the assertion by diminishing the reader.}

- [Dude. Print out the hexdump of the disassembler, disassemble it manually, see that it works, then use it as the first step in analyzing the compiler. Oh, and TheKenThompsonHack is trivially visible in the .s file if you're looking for it.]

- {You're assuming the disassembler, hexdump tool, editor you use to read the .s file aren't all compromised.}

- *Yep. The weakness in TheKenThompsonHack isn't there. It's in the fact that it will have to make mistakes in identifying the disassemblers, hexdump tools, editors, compilers, etc. Somewhere, there's a program whose output it will change but it shouldn't have, or a program whose output it should have changed but didn't.* See TuringIncompletenessTheorem.

"'I've never been able to understand why simply doing some programmatic transformation of your compiler first isn't considered a good defense. Basically just obfuscating the code. For such a backdoor to work the injection point needs to be recognized by the injector. So if one ... Started with a corrupted compiler and then obfuscated it's source. Then compiled the obfuscated but "clean" source with the corrupted compiler. One should end up with a clean and uncorrupted compiler. Then compile the unobfuscated source with the "washed" compiler (if the obfuscation significantly degraded the performance of the compiler). If in my paranoia I needed to bootstrap an entire toolchain this way. I'd start with an FPGA. Use it to emulate an FPGA (at horrendous loss of efficiency). Then use the emulated FPGA to run a custom simple soft-CPU. Preferably hand written. Ad then bootstrap a build toolchain using some expensive obfuscations for the first pass after which the tools are considered clean. One would probably always need to be careful of the CPU the toolchain runs on though, but similar obfuscation arguments should apply..."

How do you know that the compromised compiler won't recognize the obfuscation you used? If it does, your "clean" compiler is no longer clean. (The FPGA and CPU solution has the same problem.)

Wouldn't it be extremely hard (in any sense I can think of) to actually be able to recognize every program that does a particular operation? Say you wrote a program that runs a JVM implementation of a MIPS emulator that can make host system calls that runs a program that implements login by reading whatever files contain the passwords in backwards and converting them to EBCDIC before doing processing on them. I find it hard to believe that the KTH-implementing compiler would actually recognize anything that program is doing.

Yes, it will have to make mistakes somewhere. This is because any implementation that was always correct could be used to solve the HaltingProblem. But this doesn't make any particular trick undetectable. What they detect is mostly a matter of whether or not they thought of it and whether or not they considered checking for it worth the effort.

See also QuineProgram, HyperBug

CategoryPaper CategorySecurity

View edit of August 18, 2014 or FindPage with title or text search