Schneier on Security

Blog >

Countering "Trusting Trust"

Way back in 1974, Paul Karger and Roger Schell discovered a devastating attack against computer systems. Ken Thompson described it in his classic 1984 speech, "Reflections on Trusting Trust." Basically, an attacker changes a compiler binary to produce malicious versions of some programs, INCLUDING ITSELF. Once this is done, the attack perpetuates, essentially undetectably. Thompson demonstrated the attack in a devastating way: he subverted a compiler of an experimental victim, allowing Thompson to log in as root without using a password. The victim never noticed the attack, even when they disassembled the binaries -- the compiler rigged the disassembler, too.

This attack has long been part of the lore of computer security, and everyone knows that there's no defense. And that makes this paper by David A. Wheeler so interesting. It's "Countering Trusting Trust through Diverse Double-Compiling," and here's the abstract:

> An Air Force evaluation of Multics, and Ken Thompson's famous Turing award lecture "Reflections on Trusting Trust," showed that compilers can be subverted to insert malicious Trojan horses into critical software, including themselves. If this attack goes undetected, even complete analysis of a system's source code will not find the malicious code that is running, and methods for detecting this particular attack are not widely known. This paper describes a practical technique, termed diverse double-compiling (DDC), that detects this attack and some unintended compiler defects as well. Simply recompile the purported source code twice: once with a second (trusted) compiler, and again using the result of the first compilation. If the result is bit-for-bit identical with the untrusted binary, then the source code accurately represents the binary. This technique has been mentioned informally, but its issues and ramifications have not been identified or discussed in a peer-reviewed work, nor has a public demonstration been made. This paper describes the technique, justifies it, describes how to overcome practical challenges, and demonstrates it.

To see how this works, look at the attack. In a simple form, the attacker modifies the compiler binary so that whenever some targeted security code like a password check is compiled, the compiler emits the attacker's backdoor code in the executable.

Now, this would be easy to get around by just recompiling the compiler. Since that will be done from time to time as bugs are fixed or features are added, a more robust form of of the attack adds a step: Whenever the compiler is itself compiled, it emits the code to insert malicious code into various programs, including itself.

Assuming broadly that the compiler source is updated, but not completely rewritten, this attack is undetectable.

Wheeler explains how to defeat this more robust attack. Suppose we have two completely independent compilers: A and T. More specifically, we have source code S_A of compiler A, and executable code E_A and E_T. We want to determine if the binary of compiler A -- E_A -- contains this trusting trust attack.

Here's Wheeler's trick:

Step 1: Compile S_A with E_A, yielding new executable X.

Step 2: Compile S_A with E_T, yielding new executable Y.

Since X and Y were generated by two different compilers, they should have different binary code but be functionally equivalent. So far, so good. Now:

Step 3: Compile S_A with X, yielding new executable V.

Step 4: Compile S_A with Y, yielding new executable W.

Since X and Y are functionally equivalent, V and W should be bit-for-bit equivalent.

And that's how to detect the attack. If E_A is infected with the robust form of the attack, then X and Y will be functionally different. And if X and Y are functionally different, then V and W will be bitwise different. So all you have to do is to run a binary compare between V and W; if they're different, then E_A is infected.

Now you might read this and think: "What's the big deal? All I need to test if I have a trusted compiler is...another trusted compiler. Isn't it turtles all the way down?"

Not really. You do have to trust a compiler, but you don't have to know beforehand which one you must trust. If you have the source code for compiler T, you can test it against compiler A. Basically, you still have to have at least one executable compiler you trust. But you don't have to know which one you should start trusting.

And the definition of "trust" is much looser. This countermeasure will only fail if both A and T are infected in exactly the same way. The second compiler can be malicious; it just has to be malicious in some different way: i.e., it can't have the same triggers and payloads of the first. You can greatly increase the odds that the triggers/payloads are not identical by increasing diversity: using a compiler from a different era, on a different platform, without a common heritage, transforming the code, etc.

Also, the *only* thing compiler B has to do is compile the compiler-under-test. It can be hideously slow, produce code that is hideously slow, or only work on a machine that hasn't been produced in a decade. You could create a compiler specifically for this task. And if you're *really* worried about "turtles all the way down," you can write Compiler B yourself for a computer you built yourself from vacuum tubes that you made yourself. Since Compiler B only has to occasionally recompile your "real" compiler, you can impose a lot of restrictions that you would never accept in a typical production-use compiler. And you can periodically check Compiler B's integrity using every other compiler out there.

For more detailed information, see Wheeler's website.

Now, this technique only detects when the binary doesn't match the source, so someone still needs to examine the compiler source code. But now you only have to examine the source code (a much easier task), not the binary.

It's interesting: the "trusting trust" attack has actually gotten easier over time, because compilers have gotten increasingly complex, giving attackers more places to hide their attacks. Here's how you can use a simpler compiler -- that you can trust more -- to act as a watchdog on the more sophisticated and more complex compiler.

Tags: computer security, malware, trust, vulnerabilities

Posted on January 23, 2006 at 6:19 AM • 73 Comments

Comments

gnu • January 23, 2006 6:46 AM

I made the experience that compiling the same source with the same compiler two times produces two diffrent binarys. That should make this test nearly impossible.

Jonathan • January 23, 2006 6:49 AM

I don't buy it.

In order for the countermeasure to work, the infected compiler must be infected in such a way that it can detect the compilation of _any_ other compiler. IMO, this assumption is far too strong.

Isn't it far more reasonable to make the assumption that the "trusting trust" malware can only detect the compilation of a compiler significantly similar to itself?

Jonathan • January 23, 2006 6:52 AM

@gnu: GCC's "make bootstrap" actually does build itself with itself twice, in such a way that it can compare the two stages to be identical.

davidg • January 23, 2006 7:02 AM

What if the two compilers were once (way back in their history) initially compiled with a common compiler?

This is quite common with bootstrapping - create an initial compiler using an off-the-shelf one.

Anonymous • January 23, 2006 7:05 AM

@gnu,

that's not the point. You are trying to check that RedHat's GCC is correct according to the source (compiler A). The point is that compiler A compiled with compiler B should produce the same binary as compiler A delivered by RedHat.

or put another way

if

compilewith(compilewith(A,source),(source))
== compilewith(compilewith(B,source),(source))

where compilewith(x,y) compiles y with compiler x,

Then there can only be an attack hidden in the binary A if the binary B is cooperating with A.

Aze • January 23, 2006 7:07 AM

anon post was me, and I should just add, of course,

s/Then there can only be an attack/Then there can only be a _trusting_trust_attack_ /

It's still possible to have lots of other direct attacks hidden in A or B.

Adam • January 23, 2006 7:13 AM

"Since X and Y were generated by two different compilers, they should have different source code but be functionally equivalent."

Shouldn't that be different binaries, rather than different source code?

Anonymous • January 23, 2006 7:25 AM

"Also, the only thing compiler B has to do is compile the compiler-under-test. It can be hideously slow, produce code that is hideously slow"

That is not correct. If B is producing a result that is "hideously slow", as compared to A then the binary is different. You have to have the same optimizers to produce the same binary.

My guess is that you need to hand craft a disassembler/decompiler on trusted hardware and then test the suspect binary. The resulting source would then be hand inspected. If found to be clean it could then be used to test untrusted compilers. The new binary should match what was fed into the trusted decompiler.

---

arl • January 23, 2006 7:32 AM

Last anon was by me.

Paul Crowley • January 23, 2006 7:33 AM

This works so long as running the optimizing compiler twice over the same inputs produces the same outputs. Sadly, this isn't usually the case, but it can probably be arranged with suitable operating system tricks.

The anonymous comment "That is not correct" (*please* sign these comments, even if pseudonymously) is mistaken - read the description more carefully.

D • January 23, 2006 8:47 AM

I'm thinking, under an Open Source UNIX (Linux, *BSD) on Intel processors, use GCC and Intel's compilers to perform such comparisons. The problem is that GCC's engineers probably don't think like Intel's engineers and the compilers will produce different binaries anyway.

I suppose, however, the analyst can focus on the differences between the binaries rather than having to manually compare the entirety of both -- still a daunting task.

I still don't see this paper as providing a viable defense against the attack.

NS • January 23, 2006 8:59 AM

@ Jonathan

"In order for the countermeasure to work, the infected compiler must be infected in such a way that it can detect the compilation of _any_ other compiler. IMO, this assumption is far too strong.

Isn't it far more reasonable to make the assumption that the "trusting trust" malware can only detect the compilation of a compiler significantly similar to itself?"

Agreed. I've always thought that the significance of the "trusting trust" was perceived to be much greater than it really is.

I haven't read Wheeler's paper, but based on what others have written, it seems to have similarity to an idea I came up with in the late '90s.

I observed that in order for the "trusting trust" attack to work, the rigged compiler must be able to, perhaps just minimally, understand the semantics of the program under compilation. To detect the compilation of a compiler, maybe it can detect the presence of a login system call. It became obvious to me that such a compiler cannot possibly understand the semantics of all possible future programming languages, so it cannot intelligently detect and subvert programs that implement interpreted languages that have yet to be invented.

So if you implement a Java Virtual Machine, a P-code interpreter, or some other processor emulation, *and then* execute a compiler written in that interpreted language, I don't see how the "trusting trust" attack could be successful.

I thought the idea must have been obvious to computer scientists -- so much so that nobody bothered to write it up.

Fred Page • January 23, 2006 9:05 AM

@Paul
-If optimizations are a problem, turn them off. However, that does not appear to be a problem with the paper that's referenced above; optimizations should be a non-issue.

@arl
"You have to have the same optimizers to produce the same binary."
-This seems irrelevant for the above paper. Y !=X; They merely need functional equivalency.

Secure • January 23, 2006 9:09 AM

Another example why monocultures are bad: Imagine there would be only one compiler...

NS: "I thought the idea must have been obvious to computer scientists"

Yes, it is obvious, and it is more or less used all over the place. You compare a fingerprint over the phone to avoid man-in-the-middle attacks when exchanging public keys over a public network. You could even exchange the keys personally (write your own compiler). You can use operating systems on live-CDs to detect viruses on the harddisk without the viruses being able to interfere, and you can check the integrity of system files to detect rootkits.

The general pattern is: Use a second, trusted resource to check the integrity of the first.

---

sul • January 23, 2006 9:10 AM

this strikes me as a bit of a bandaid. At the end of the day should the actual executing object of the binary (kernel) be functioning as an audit point? Make the source and binary available to the kernel and have it randomly do a check on the validity of it (i.e. compile the source into a temp binary and compare)? This would also allow for the inclusion of a micro-compiler into the kernel and a check against the validity of the binary object...
Step 2 would be obvious...

Fred F. • January 23, 2006 9:15 AM

I don't agree that optimizations are a non-issue. They are a big issue. I mainly know about GCC but there are MANY options. For example one can make a binary that works on the all of the x86 processors or one optimized for the 586 family for example. There will be instructions used in this one that will not be in the first one. Then there is the obvious case of -Os optimizations where it will optimize for size, the file will be smaller than say a -O2 optimized binary. Add on top of that things like using the registers instead of the stack and you end up with very different results. I think this is more of academics not registering with the real world.

denis bider • January 23, 2006 9:18 AM
This is a nice solution, but how can one rely on byte-wise comparison between two files on a machine that may contain a compromised compiler?

denis bider • January 23, 2006 9:30 AM
If one plants you a rogue compiler binary, that binary doesn't have to compile its own source code into another rogue compiler binary. Instead, it can generate a clean compiler binary as long as it has already compiled any system binaries (a) so that a vulnerability is introduced, and (b) so that the system will modify even a clean version of this compiler in-memory to repeat tasks (a) and (b) whenever compiling system binaries.

The rogue compiler binary can in fact be totally erased, and the compromised system will ensure it remains compromised by patching up even a clean compiler in-memory.

paul • January 23, 2006 9:32 AM
I'm still not sure about buying the "bit-for-bit identical" part of the argument. Even without the issue of optimization settings, it's a given that compilers producing functionally equivalent code don't necessarily produce bit-for-bit identical object code. Plenty of different machine-code sequences will give you the same result out the other end, either as a result of different register decisions, or of different coding of certain primitive operations, and so forth. There may be some limited language and OS choices for which this makes sense, but I don't see how it would work in anything approaching wider practice.

Dan • January 23, 2006 9:43 AM
to all who think this method doesn't work...

I think alot of people are missing the point.

1. you have the source code of the compiler you don't know whether to trust. (say GCC)

2. you have the compiler which you don't know whether to trust (say GCC) and another compiler you don't know whether to trust (say VC++)

3. now compile the GCC source with both GCC and visual studio. You will now have 2 new compilers, both GCC which behave in the same way but will have different binaries since they have been compiled by different compilers with different optimisations. But they will both be GCC compilers and so will create the same binaries when compiling the same source.

4. now compile the original GCC source with your two new GCC compilers, this should produce the same binary. If it produces different binaries then you know that there is a high chance that one of the original compilers (GCC or VC++) has a back door in it. If they produce the same binary, then either they are both non malicious, or they are both malicious in the exact same way (which we are guessing is relatively unlikely since they don't share any of the same code)

please say you understand now!

Jim Hyslop • **January 23, 2006 9:48 AM**

Wheeler's whole logic fails at this point: "Since X and Y are functionally equivalent, V and W should be bit-for-bit equivalent."

As others have mentioned, optimizations could affect the result. Even if you turn off all optimizations, though, many C statements (assuming C is the languaged used, which seems a likely assumption) can be rendered in several different ways.

Suppose the compiler needs to cache a value temporarily, such as in an integer post-increment:

x++

The compiler can store the pre-incremented value of 'x' in a register. Suppose compiler A uses register B, but compiler T uses register C. The two sequences are functionally identical, but the two programs fail Wheeler's test because they are not bit-for-bit identical.

That's just one example. There are dozens, if not hundreds, of areas where compiler writers have a choice of ways to generate code for a particular statement. For example, in a statement such as

someFunc(x++)

the compiler is free to choose whether to increment x before the call to someFunc, or after the call. Optimizations don't enter the picture at all - it's a design choice the compiler writers are allowed to make.

It seems to me the only way to defend against this attack is to use a generic binary editor (not a disassembler) and manually disassemble and reverse-engineer the program. You won't have to do the whole program, just the portions vulnerable to the threat.

---

philip • **January 23, 2006 9:48 AM**

Aaargh! Read the post, people. Carefully.

To recap:
- You have the Source for compiler A.
- You compile this twice, once with compiler A, once with compiler B.
- This gives you compilers X and Y, which are functionally equivalent (to each other and to A) but almost certainly not bit-for-bit identical.
- Now compile the Source for A with X and with Y.
- The results of *this second* round of compilation will be bit-for-bit identical *unless* A has recognized *its own* source code and inserted something special.

Nicholas weaver • **January 23, 2006 9:54 AM**

One possible glitch:

The compiler being compiled MUST BE deterministic.

If Simulated Annealing or another stochastic optimization technique is used anywhere, you must be able to specify a seed.

Secure • January 23, 2006 9:56 AM

"The results of *this second* round of compilation will be bit-for-bit identical *unless* A has recognized *its own* source code and inserted something special."

And unless the compiler is non-deterministic and selects equivalent code-sequences based on a random number generator. Then you should drop it anyhow, no matter if it is manipulated or not.

Jim Hyslop • January 23, 2006 9:59 AM

@Dan: "3. now compile the GCC source with both GCC and visual studio. You will now have 2 new compilers, both GCC which behave in the same way but will have different binaries since they have been compiled by different compilers with different optimisations. But they will both be GCC compilers and so will create the same binaries when compiling the same source."

Ah, that's the step I missed. OK, I can see how Wheeler's defense works now.

MarkW • January 23, 2006 10:03 AM

Very simple and clever. I'm astonished Ken Thompson didn't know of this.

There is a small typo.

So all you have to do is to run a binary compare between U and V; if they're different, then EA is infected.

should be

So all you have to do is to run a binary compare between V and W; if they're different, then EA is infected.

Carlo Graziani • January 23, 2006 10:04 AM

Two bits of evasive action which should also be checked concomitantly with Wheeler's procedure:

(1) cmp, ls, stat, diff and friends are also very likely compromised on any machine that has been attacked with this level of meticulousness. The bit-by-bit comparison should be carried out on another machine with known-good utilities, and if the files are transfered by ftp, scp, or NFS, those systems should be scrutinized with the greatest suspicion. Probably the safest thing would be to disconnect the hard drive and connect it to the known-good machine. Alternatively, a boot disk with a live cd (Knoppix, for example) would do nicely, and defend against the possibility that the kernel was also compromised.

(2) Compile and install are two different phases of the build. A naive implementation of the Wheeler procedure would compare binaries built in the build directory.

It would be straightforward for the malicious code to build *two* binaries (which would, after all, only differ by a few modules containing the malicious payload), and present the innocent one for inspection in a Wheeler test.

When the system administrator is satisfied that the test is passed and types "make install", the Makefile proceeds to install the infected binary, which was built in some obscure sub-directory with a misleading name.

Anyone who has ever had to read through a bowl of autoconf/automake/make spaghetti can imagine how easy it would be to obscure this additional layer of sneakiness. Detection would require either an extremely experienced and clinically paranoid sysadmin, or some kind of automatic makefile analyzer imported from another machine.

Fred Page • January 23, 2006 10:15 AM

@Nicholas
"The compiler being compiled MUST BE deterministic."
- Correct. The paper mentions this issue (briefly). In my personal opinion, a compiler that you intend to verify that can't be made to be deterministic is one that you can't trust.

Ari Heikkinen • January 23, 2006 10:31 AM

I'm just not sure what's the point. If the attacker already has root access to your system he can do whatever he wants with the system. It's just one possible attack (using "terrorists" analogy, it's actually similar to trying to protect against certain "movie plot threats" as you put it).

Clive Robinson • January 23, 2006 10:40 AM

@Dan

No, they might still be different...

The way GCC and MS compilers optomise will affect the way the GCC compilers are compiled (which you agree with).

This gets passed down to the way the CPU registers etc are used. I suspect that even though the resulting code compiled with the two GCCs may be functionaly equivalent, I suspect that they might still not be bit for bit equivalent, when optimisations are applied, as the optomised compilers could take a diferent paths based on their own register optimisations.

So I suspect you need to do a functional test not a diff on the binary, that's going to be a whole lot harder but not impossible :)

---

pdf23ds • January 23, 2006 10:40 AM

Someone may have already said this...

"Since X and Y were generated by two different compilers, they should have different source code but be functionally equivalent."

I think you mean "they should have binary differences", since X and Y are binaries.

Andrew Prock • January 23, 2006 10:41 AM

I saw it mentioned in the text of the explanation, and alluded to in some of the comments, but the real problem with this method is that it assumes independence of compilers.

Given that you've been compromised, and that you've been compromised by an attack with this level of sophistocation, you may have a hard time getting your hands on an independent compiler. That is, a compiler which doesn't have the same backdoor.

---

pdf23ds • January 23, 2006 10:51 AM

Clive, let's be clear. Are you saying that X and Y will not be bit-for-bit equivalent, or are you saying that V and W will not be? Since X and Y are functionally equivalent (let's say they're deterministic), they should produce the same output given the same input, period. X and Y are both given the same source to produce V and W, so V and W should be the same.

No differences in the way that CPU registers are used in X and Y could affect their output such that V and W would have different binaries, unless one of the compilers that produced X and Y had a bug.

Clive Robinson • January 23, 2006 11:30 AM

@pdf23ds

As far as the compiler goes they should produce functionaly identical intermediate code.

When you move to producing the machine code output (ASM / GAS) it can order any number of asembler instructions in different orders and make no difference what so ever to the functionality of the code or for that mater it's execution speed under most circumstances. However occasionaly it will affect the use of a branch or jump statment. This is a given with asembler level programers who can had optomise different branch/jump instructions etc. and can often save quite a few bytes in loops etc.

Also with the Intel platform it is also known that various differnet assembler instructions produce the same results. It is up to the compiler which it picks (it is known that atleast one asembler developer used this to put a watermark on code produced by his assembler).

However the use of the registers can and does affect the way the code executes and unless the original source code nails it all down then the usage may well be different in your two end GCC tool chains. You can see this by looking at the different ways the MS and GCC compilers deal with setting up the stack frame etc..

This then affects what happens at the asembler, which may well optomise to differnet instructions that are functionaly equivalent.

The result is functionaly identical code but with different bit patterns.

If this will happen in practice I do not know but there is sufficient degrees of fredom to allow it to happen.

---

arl • January 23, 2006 11:48 AM

@Paul Crowley

We cross posted after I noticed I forgot to include my sig on the first post.

I see my mistake now, I thought "slow" was talking about the object emitted from B, not B itself.

Eric Blom • January 23, 2006 12:10 PM

Ari,

I'd say you're right this would not be a worthwile exercise for most people; presumably someone has already done this, for instance, with the OpenBSD compiler binaries. But I'd imagine it would be a very important exercise for programmers compiling security-critical code, like encryption programs. There is still the problem that a very clever and motivated person could affect the compiler binaries of multiple compilers to affect output the same way, but this seems unlikely.

Gauss • January 23, 2006 12:26 PM

@Clive

All the optimizations and instruction options you describe are still done deterministically. If they were done non-deterministically, then identical input to that compiler would not produce the same output twice in a row. By "identical input" I mean both the source code AND the command-line options AND any other state-representing config-files the compiler program is using.

If a compiler, or more properly a compilation process, uses additional programs, such as an external assembler program, then those also have to be included in the double-compilation trust-checks. Presumably, the same criteria apply: the assembler (or linker or whatever) is deterministic and produces the same output for a given set of inputs (and "inputs" again includes source, options, and all other configurable state).

Under those deterministic conditions, if there is any difference in the final binaries, compared bit-for-bit, then there is no alternative explanation for that difference than that one compiler is changing the code.

Whether it's doing it because it's malicious or because it's picking register assignments, binary instructions, etc. using a non-deterministic algorithm doesn't matter. The factual outcome is that two outputs that should be identical are not, and at least one of the compilation chains is defective.

I am, of course, assuming that non-deterministic algorithms are considered to be a defect in this situation. Although the algorithms for choosing registers, instructions, etc. do vary across tools, I know of no tools that actually use non-deterministic algorithms in order to assign registers, pick binary instructions, etc. Furthermore, I know of no reason to use such algorithms in those cases, so the chance of them appearing as some future optimization is slim.

Fortinbras • January 23, 2006 12:37 PM

@NS

I recall similar strategies from even earlier. Forth, in particular, is a very small language with a very small set of trusted op-codes (words). So small, that in many cases the assembler or even the native binary is auditable. Given that small trusted base, other words are defined, which either refer to the trusted words or create new native words. The source for the native code is known, auditable, and presumably trusted.

I mention Forth in particular because it has been used off and on over the years in creating systems that can be audited "all the way down".

What this double-compilation is doing is basically using a series of compilers as a simple "virtual machine", or more specifically, two virtual machines, and confirming they produce identical outputs. The "instruction set" of the virtual machine is limited to compiling a source language and options, but the arrangement is still confirming that two virtual machines have identical outputs.

Ryan Russell • January 23, 2006 12:39 PM

I think the disassembler bit was glossed over a bit too quickly. If one were suspicious of one's compiler output, I don't think the change would survive unnoticed for long. I've probably got half a dozen disassembler lying around, and that's not counting any I write myself, or doing it by hand.

I believe much of the original idea was from a time when you were in a captive environment on a single system. I believe rootkits have obsoleted this attack in that situation some time ago.

Secure • January 23, 2006 12:43 PM

The only difference that could occur even for deterministic compilers are timestamps derived from the current system time - even if you modify the timer it should be hard to catch the exact millisecond the timer is accessed. Thus there could be some small differences - just not big enough to contain a complete infection engine including a compile-myself detection system.

Nyarly • January 23, 2006 12:53 PM

This is quite an interesting procedural suggestion, as well as valid as far as it goes.

In terms of practical application, complexity rapidly abounds. Ultimately, you can only trust the compilation mode that you test. So each unique set of optimization techniques should essentially be treated as it's own compiler. Likewise each possible seed for a non-deterministic compiler needs to be treated as a it's own compiler. Likewise each compilation platform. A malicious compiler might be rigged only to insert it's backdoors under certain circumstances - like -O3, or in non-deterministic mode without a chosen seed.

It seems that if one were to want to use Wheeler's method in securing a system, then a) you'd want two clean-room-level unrelated compilers that you have source to. b) Subject both compilers to Wheeler's test. c) Perform all comparisons of result binaries with two unrelated programs compiled from source with both compilers under test - total of 8 compares. d)Perform the whole test under unrelated (hah!) OSes - because if the kernel is rigged, nothing is trustworthy.

Now, if both compilers, both comparators, and both kernels are all benign or all malicious in the same way, the ultimate result will be a pass. If any are differently malicious, the result will fail. Once you have a pass, though, keep in mind that each compiler is only trustworthy in the mode it was used in the tests.

Joe Buck • January 23, 2006 1:02 PM

It's not difficult to determine whether or not Red Hat's compiler has been subverted.

During the bootstrap process GCC is built three times. In stage 1, it is built by some arbitrary compiler. In stage 2, the compiler is built again, this time with optimization enabled, by the compiler built from stage 1. Finally, in stage 3, the compiler is built a third time, using the compiler built from stage 2. Now the generated object files are compared, and the object code produced in stage 2 must be bit-for-bit identical with the object code produced in stage 3. This is to test that the compiler correctly builds itself.

First, to test that Red Hat builds their compiler from the supplied source, download their source RPM package, build it, and compare the object code to their binary RPM package.

Next, download the free-30-day-trial version of Intel's ICC and repeat. When again everything works, you know that either Red Hat is clean or Intel has been subverted. Then you can build a cross-compiler starting from Sun Solaris, and test their compiler. Or you could use a 7-year-old GCC version. You will quickly conclude that either there is no Thompson hack, or every compiler in the known universe has been corrupted.

Then, when you think about the problem of creating a Thompson hack that can affect every known compiler, you'll quickly see the difficulty; the code that the compiler must recognize to do its magic has changed radically over time.

Nevin ":-)" • January 23, 2006 1:54 PM

One of the underlying assumptions is that the same compiler run over the same source code will result in a reproducible build.

There are language features which make this impossible. As an example, here is a trivial case in C:

```
#include
int main() {
printf("I was compiled on %s\n", __DATE__);
}
```

And that is one of the easy cases to find...

Nevin ":-)" • January 23, 2006 1:58 PM

In my above posting, I stated "the same compiler run over the same source code". A compiler built by two separate compilers also exhibits this issue.

Nevin ":-)" • January 23, 2006 1:58 PM

In my above posting, I stated "the same compiler run over the same source code". Two versions of the same compiler (themselves built by two separate compilers) also exhibits this issue.

Anonymous • January 23, 2006 2:22 PM

Is there any evidence that such an attack has ever been realized?

Secure • January 23, 2006 3:33 PM

"Is there any evidence that such an attack has ever been realized?"

If there is no evidence it could either mean that there never was such an attack - or it could mean the attack was successful and is still undiscovered.

Smoke • January 23, 2006 7:15 PM

@Secure

"If there is no evidence it could either mean that there never was such an attack - or it could mean the attack was successful and is still undiscovered."

Or it could mean there was an attack, it was discovered and remedied, but it was felt better not to reveal that such an attack had been discovered.

Discretion is the better part of valor.

Anonymous Coward • January 24, 2006 12:19 AM

I know I'm picky, but wow! Such a long
article for laying out such a simple

concept.

Maybe it is newsworthy that someone took this idea and formalized it. But then on the other hand, finding the "bad ones" by compaing it against the "other/good ones" is what I do each week in the supermarket to get the good fruits. The "trusted" or "good" fruit is then in my mental picture.

just my 2 cents

False Data • January 24, 2006 12:41 AM

Interesting technique. From a practical standpoint, though, has anyone tried to compile, say, GCC using a non-GCC compiler and non-GNU toolset? I would not be too surprised to learn that compilers drift from strict language standards over time, especially given a compiler's greater-than-normal need to work with the underlying hardware.

Since file comparison tools would be a natural target to compromise, you might need to use a non-standard comparison tool that's simple enough to audit, and compile a copy with each compiler. Presumably the tool would also know enough about the binary's format to skip obvious red herrings like embedded timestamps.

Finally, given the size and complexity of a modern compiler, the risk of the compromise being buried somewhere in the sources rather than the binary is probably increasing over time. We've come a long way from the smallc compiler.

Richard Veryard • January 24, 2006 2:59 AM

A similar pattern arises in social trust. A relies on the "fact" that B and C are unable to collaborate against A. This pattern is often found in international relations, where A, B and C are countries.

Even if the pattern isn't absolutely reliable, it may still be worth having if it increases the level of trust.

This raises the question of the algebra of trust - how do we get higher levels of trust by composing heterogeneous elements - the opposite of the "weakest link" principle?

Ari Heikkinen • January 24, 2006 3:23 AM

I think this whole discussion is also wrong from the standpoint that a compiler don't generate binaries. A compiler generates assembly. An assembler generates objects. A linker generates the final binary (this is how it's generally done). Now, even though you use how many compilers as you like from whatever sources or where-ever systems, if you run them on the compromised system and the linker (which the compiler binaries then run on the compromised system) always adds the attack code at the same place (say at the beginning with the startup) this check proposed would be useless. All the binaries and all the compiled (and assembled) objects would always match even with the attack code inserted.

Nix • January 24, 2006 4:30 AM

The ability to compile GCC using a non-GCC ISO C compiler and non-GNU toolset is part of the release criteria for GCC, and is regularly tested. (You *do* need GNU make, but you can build that using whatever compiler you like, as well.)

If this property wasn't maintained (and wasn't maintained for GNU binutils, GNU make, and GNU sed as well) then it would be impossible to bootstrap the GNU toolchain from a foreign system. This is obviously undesirable.

Ari: Wheeler discusses the 'compromised other binaries' point, and the fix is trivial: consider the assembler, linker et al part of the 'compiler' whose integrity you must test. (This means that you'd need a trusted assembler, linker et al, too.)

---

aikimark • January 24, 2006 5:58 AM

Wheeler's technique seems like a lot more work than is necessary. It would be MUCH SIMPLER for the trusted source to produce a decent hash (SHA-512) for folks to use as a comparison against their compiler executables.

One could also compare hash values at the source code level.

David Thomas • January 24, 2006 6:03 AM

The only thing here that I could see being a problem is if any part of compiler in question is nondeterministic to the source-code, either using poorly defined language constructs or pseudo-randomness. Of course, differences produced may well be limited enough that they can themselves be examined by hand...

To those who have a problem with the "bit for bit identical" (and, perhaps, reading comprehension...), you are not producing code to be compared on two different compilers, but on two different compiles of the same compiler. Modulo what was mentioned above, the functional equivalence of these two compiles establishes the bitwise equivalence of their output.

And point the last, most interestingly this countermeasure *does not* require a trusted compiler - two compilers can be checked against each other assuming you can reasonably trust that there was no cross-contamination. Two attacks are quite unlikely to take precisely the same form, and will otherwise be found. Of course, you cannot then tell which of the tested compilers is bad without inspecting the code produced. It does require a trusted diff, but it is significantly easier to verify the bytecode of a simplistic diff program than of a compiler.

Nix • January 24, 2006 6:08 AM

The trusted source can't do that, because it isn't a known non-malicious version of the compiler under test: it's a *completely different program*. Its *only* necessary property is that it takes the source of the compiler under test (which is, remember, innocuous: it's the *binary* that's been subverted), and produces a compiler from it which has the same *semantic* effects as the original.

Again: the trusted compiler is not necessarily a version of the compiler under test (in fact, it's better that it isn't). It is not a program that produces the same bitwise output for any input that the compiler under test does. It *just* has to be able to produce a working compiler binary with the same semantics as those of the compiler under test given the source of the compiler under test. That compiler binary will not contain the trusting-trust attack, because it wasn't produced by the subverted binary: but it won't be bitwise identical to the original compiler, because it was produced by a different compiler with different optimizers (or none!)

The trick is to recompile the source of the compiler under test *with the binary which was itself produced by the trusted compiler*. As long as the compiler under test is deterministic and nonsubverted, you'll get exactly the same output from that step as you do from compiling the trusted compiler with itself.

And *that* is what we're checking for. (And Wheeler puts it more clearly than that in his paper. Read it!)

David Thomas • January 24, 2006 6:16 AM

"In terms of practical application, complexity rapidly abounds. Ultimately, you can only trust the compilation mode that you test. So each unique set of optimization techniques should essentially be treated as it's own compiler. Likewise each possible seed for a non-deterministic compiler needs to be treated as a it's own compiler. Likewise each compilation platform. A malicious compiler might be rigged only to insert it's backdoors under certain circumstances - like -O3, or in non-deterministic mode without a chosen seed."

Think of it as a method of obtaining a compiler that you are certain is built from the source. Presuming you can be sufficiently certain that the source itself is correct, the final result here is now a functioning compiler in all it's glory, and does not require further testing or examination regarding whether it conforms to the source - doesn't matter the options or architecture it is then applied to.

Marko • January 24, 2006 6:59 AM

My comment. If crack are able to crack exe files of several programs without ever knowing the siurcecode, then it would be possible to detect anomalies in the behaviour of a program, which was compiled using a malicious compiler.

Andy • January 24, 2006 7:27 AM

So, we've verified the compiler. Now we need to verify the linker and the loader, to make sure they're not adding their own little bits. And the filesystem drivers, too.

Next comes the microcode in the processors. Makes a good argument for RISC, no?

What about the firmware in the disk interfaces? The BIOS or equivalent?

Oh-oh. What did Apple put in my iPod? RIM? Palm? The printer manufacturers? And, what's really running in that router, proxy, and firewall?

Can't we just all switch from MD5 to SHA1 and pretend it's all okay?

Ari Heikkinen • January 24, 2006 7:47 AM

As my final comment, as far as practicality goes, this just seems too much hassle to me. Like someone else already suggested, taking sums (pick anything considered reasonably secure) of the system files you're concerned with and checking them would seem much more practical to me.

Aze • January 24, 2006 10:15 AM

@Nevin ":-)"

by manipulating system call results, it's also possible to allow this. For example, we could replace time(2) with a call that always returns 42 or returns a monotonically increasing sequence of integers increasing by one for each call.

@Ari

Check sums are completely useless against such an attack. Your checksums will be against an already compromised binary. You are lucky, however, all you need to know is that _someone_ does this on your compiler and you can know that you are safer.. In fact, the mere threat that they might is a deterrent.

On your earlier post; the word "compiler" is being used in a simplified form. The compiler is (should be) really a complete system including computer and all programs needed and a CDR drive.

You create a big bunch of CDR(not W)s with source code. You compile on the "compiler" and generate a CDR with your binary. Really paranoid people build an entire minimalist operating system and treat that as the "compiler"

@ALL

I don't think anyone has stated the)really (_really_REALLY_) important point about this.

Until now, us Free Software fanatics claim to be "fundamentally more secure" could be laughed at by saying that, somewhere in their history, they used a proprietary compiler and so it's possible that all their systems were compromised. The "only" difference was the amount of proprietary code that we trusted. KT himself said "No amount of source-level verification or scrutiny will protect you from using untrusted code."

Now it's possible to say that we have a higher level of trust. The source code really can tell you what the system does... Note that this benefit only really applies to 100% clean, no proprietry software, systems. It also only applies as long as you fully audit the hardware or use multiple different, independently produced computers for each calculatoin. :-).

Clive Robinson • January 25, 2006 4:38 AM

@Gauss @pdf23ds

Gauss, you are correct in that if all inputs are identical, and all algorithums are determanistic (and probably also single threaded) then the output should be the same.

THe exception to this is unless some one writing part of the tool chain has deliberatly decided to put some kind of watermark in the system which can identify the machine etc. I know this might sound paranoid but it is known that a shareware asembler did indead watermark the binary output, and it is also known that MS Word in Office 97 put the MAC address of the machine in it's files.

However it was a posting about the use of the MS compiler that reminded me of a company I was working at some time ago.

Basically they developed embeded controler devices using Arm processors and various applications running on them.

They used the GNU tool-chain on Intel desktops to do the cross compiling. The toolchain was compiled from scratch on the desktops as it was being used for cross compiling.

Some of the engineers used MS desktops and others used GNU/linux, therfore some used the MS product for the tool-chain compile others used the GCC. As development progressed more of the tool chain was moved onto the target proccessor.

During regression testing they did indead find that binaries did differ depending on the desktop used. Which obviously caused some concern at the time. From what I was told the differences where small and appeared to be a re-ordering of code segments, where functionalty was not effected.

They used VMware as a tempory work around,I know they where investigating the problem however as I was on contract on another project that finished I did not find out if they fully got to the bottom of it.

The lesson I took from it at the time was "everybody on a project should use the same tool chain". Which is the oposit advice of David Wheeler's work.

David A. Wheeler • January 27, 2006 12:18 PM

Hi, I'm the author of the "Countering Trusting Trust..." paper, and I'm delighted to see so many people discussing it!! Some of the questions or comments are answered by the paper or my website, so let me point you to some of the answers.

gnu said: "I made the experience that compiling the same source with the same compiler two times produces two diffrent binarys. That should make this test nearly impossible."

This is nondeterminism, which can be easily handled or not depending on WHY the nondeterminism happens. Timestamps are usually handled easily. If the variation depends on a random number generator, you need control of the random number generator's seed, but that's not weird -- gcc ALREADY has an option to control the random number seed, for example. Some compilers, like gcc, include the check for determinism as one of their tests for validity. In gcc it's "make bootstrap", and Fedora Core's rpmbuild routinely does a "make bootstrap" as part of the compiler build -- so having a deterministic compiler is NOT an unusual circumstance. See the start of section 7 of the paper - many people believe uncontrolled nondeterminism is a bug ANYWAY, because it makes testing nearly impossible.

Clive Robinson observed: "The exception to this is unless some one writing part of the tool chain has deliberatly decided to put some kind of watermark in the system which can identify the machine etc... a shareware asembler did indead watermark the binary output..." and later on noted some differences based on environments in some cases.

Yes, clearly if the toolchain is INTENTIONALLY inserting variations then they won't have the same result. But the process will detect that, in most cases in stage 0.... and you can see exactly what varies. You might want to know that!

An Anon asked, "Is there any evidence that such an attack has ever been realized?"

Well, Thompson implemented it in 1984 and sent his malicious compiler to a sister Bell Labs unit. His malicious compiler subverted the compiler, login program, and disassembler. The victim even looked at disassemblies and never saw the attack! Now it's true that Thompson wasn't malicious, but there's every reason to believe that someone has tried it or will try it someday. There are more people who like to attack computers (and have the technical skill to pull this off), the money value of attacking computers is much greater, and compilers are larger (making this easier to hide). Section 3.1 discusses attacker motivations -- with this attack, you could 0wn every computer in the world: the entire banking system, infrastructures, whatever you want. And until now, it was undetectable. Think that might be worth something? I don't think people appreciate just how POWERFUL this attack is... you can subvert whole classes of systems, undetectably, and they stay subverted.

Smoke said: "finding the "bad ones" by comparing it against the "other/good ones" is what I do each week in the supermarket to get the good fruits."

But how would you do that with a compiler? Let's say that you have a gcc binary and a Microsoft C compiler binary. Which one has malicious code in it - do either of them? How do you prove it? In theory, you could read every object code byte - but they'd release a new version before you were done. Trying to directly compare two different binaries created by two different compilers has been tried, but it's a hairy-hard research project with no REAL solution. This process finds the malicious code without requiring that.

Jonathan said: "In order for the countermeasure to work, the infected compiler must be infected in such a way that it can detect the compilation of _any_ other compiler. IMO, this assumption is far too strong. Isn't it far more reasonable to make the assumption that the "trusting trust" malware can only detect the compilation of a compiler significantly similar to itself?"

Er, that's backwards, if I understand you correctly. The countermeasure will succeed UNLESS the trusted compiler is infected the same way. The compiler under test doesn't NEED to be infected in a way that it can detect the compilation of any other compiler, though it certainly could be, and it's not even hard to do. And as far as being "significantly similar", there's absolutely no reason that has to be true. If you stood to make a billion dollars by controlling the worldwide banking system, you could probably find time to insert attacks against 5 compilers instead of just one. The hardest part of the attack is getting started... once you've figured out how to attack one compiler, it's all to easy to add more attacks :-(.

NS said: "in order for the "trusting trust" attack to work, the rigged compiler must be able to, perhaps just minimally, understand the semantics of the program under compilation.... such a compiler cannot possibly understand the semantics of all possible future programming languages, so it cannot intelligently detect and subvert programs that implement interpreted languages that have yet to be invented. So if you implement a Java Virtual Machine, a P-code interpreter, or some other processor emulation, *and then* execute a compiler written in that interpreted language, I don't see how the "trusting trust" attack could be successful. I thought the idea must have been obvious to computer scientists -- so much so that nobody bothered to write it up."

That's been written up; I cited the papers in the 1980s that suggest that very thing. It's not that the malicious code "understands", they just need to match a pattern. The idea is sound, but doesn't help as much you might think. So you execute a compiler in the interpreted language... every time? Not likely. When you're all done, you'll compile it... and then I subvert THAT. The problem is that you have no DETECTION technique, so you have no way to know when you need to try to counter it. See section 2.2. The idea of using interpreters is good though.. and in fact I suggest doing that very thing as part of the process.

Andy said: "So, we've verified the compiler. Now we need to verify the linker and the loader, to make sure they're not adding their own little bits. And the filesystem drivers, too."

Right. And you do that by... compiling. That's all. Sure, it's millions of lines of code, but you don't need to UNDERSTAND it, you just need to compile it twice and see if you get the original answer. It might take overnight or longer to do the recompile twice, but so what, it's just compute time & compute time is cheap. And when there's a new binary release, you can do the double-recompile again. Once you have a working setup, it should be fairly straightforward. As long as it's software (any level), AND you have the source code, you're fine.

Several folks thought I meant that two different compilers would produce the same binaries. Not so, that practically never happens. It's a common misperception, which I try to counter in both my paper and my website.

Aze said: "KT himself said "No amount of source-level verification or scrutiny will protect you from using untrusted code."... Now it's possible to say that we [Free-Libre software developers] have a higher level of trust. The source code really can tell you what the system does... Note that this benefit only really applies to 100% clean, no proprietry software, systems."

You've got it: this is the only known test for the Trusting Trust attack, and it is only beneficial to people who have the source code. With only a binary, you can't use this test.

Aze said, "It also only applies as long as you fully audit the hardware..."

True, malicious hardware can subvert you even if the software is fine. See my website at http://www.dwheeler.com/trusting-trust -- I believe you could even apply DDC to hardware. Countering hardware is still hard, but I think countering the software problem is still an improvmenet.

One impossible problem at a time, okay :-)?

DigiLife • February 17, 2006 9:07 PM

honeypots
honeynets
and coming soon ....
honeycode!

a source code containing security code like password checks. it's kept small and includes only what is necessary to cause itself to be infected. it doesnt actually have to be *usefully* functional...

and to analyze it, a live-cd with a decompiler and debugger. which of course have to be determined to be clean. still need a standalone computer for all this.

some pen-testing of any computer os compiled with an untrusted compiler may also be helpful if done from a computer os that can be trusted in it's binary form.

seems the only way to stay clean is by starting clean. to do that a standalone that's compiles the source for your compiler with a compiler you wrote in binary (ouch!!) could be needed.....think about it. which came first? the compiler or the source (was the source created by a binary editor program that was compiled?)? gotta start somewhere. can you be sure that where you are starting from is clean? (assuming you go back to a day prior to any recorded use of such an attack but after there were binary compilers already done).

if i remember what i read somewhere else correctly, the first such attack was discovered with a clean compiler/decompiler/debugger that was obtained externally (compared to where the infected compiler came from). with that in mind there are archives of clean dev. env. on the internet as well as enough different compilers as to be able to detect such attacks in the future.

hopefully there will never be a worm that infects them all. something i doubt possible considerring the variety of platforms and operating systems (OS) and languages and compilers available in all thier versions with copies somewhere (old cd laying around?) in addition to being archived on the internet. this and some of those operating systems are more secure and harder to infect than others. this makes no mention of using hashes of course (due to off chance that hashes are not an accurate way of determining infection because either no 2 compiles are *that* identical or the infection doesnt have a "virus definition" yet or whatever theory you choose.)

i do suppose that the day may come when "trusted computing" advocates will make any computer that isn't trusted computing compliant to become illegal to use or even posses. they would then buyout (as a hostile takeover of sorts) or shutdown everyone who produces a secure platform that doesn't include trusted computing. ie: OpenBSD. that will be a sorry day for us all indeed. an Orwellian nightmare for sure. harder to prevent worm infection in your computer if your computer does not follow your orders and additionally tells on you.

---

aikimark • February 23, 2006 3:49 PM

Related paper on trusting and verifying executables.

WYSINWYX: What You See Is Not What You eXecute

http://www.cs.wisc.edu/wpis/papers/wysinwyx05.pdf

OPITEZEXT • March 17, 2010 8:12 AM

Todd Cowle Municipal Bond Credit Report synthesizes, analyzes and presents aggregate credit information and trends in the municipal bond market. The report includes municipal bond rating information from the three major rating agencies – Moody’s Investor Services, Standard and Poor’s and Fitch Ratings.

Paul • November 30, 2015 9:41 PM

So what if you also backdoor "cp" into recognizing when it's working on one of the given utilities in the rootkit?

At a fundamental level, if you can't trust your core toolset, the only option is a full nuke and reload from a trusted source. And if you don't trust your hardware isn't lying to you, you need to nuke that too (down to the firmware level).

You can take mitigation steps like this one, but they don't eliminate the threat, they only elevate it to the level of BadBios-style paranoia.

---

fedes • May 20, 2016 2:02 PM

Sorry, this is a very old post, still I think I didn't see this approach ever mentioned.
This is what I'd call a *stochastic approach*:

1. Make a sufficiently big number of copies of the binary compiler to check.
2. Randomly change a big enough number of bits on all of them. (All will be different)
3. Disasemble with your regular disassembler. The mutations of step 2 should make it hard for the disassembler+trojan recognize the trojan parts.
4. Statistically determine the disassembled shape of the original binary (I'm guessing this should be easy).

The biggest problem is with disassembly of a *corrupt* binary. I'm not sure how that part works, but I've read somewhere how disassembling tends to "self-heal" (meaning that the disassembler may output trash for some bytes, but it ends up finding the right thing after a while).

If you ever try this and works, you can buy me a beer!

Chad M • June 10, 2016 9:57 AM

Today it's telemetry, tomorrow it's... what, exactly?

Reviewing Microsoft's Automatic Insertion of Telemetry into C++ Binaries

Reddit

> Recently Reddit user "sammiesdog" posted claims that Visual Studio's C++ compiler was automatically adding function calls to Microsoft's telemetry services. The screenshot accompanying their post showed how a simple 5 line CPP file produced an assembly language file that included a function call titled “telemetry_main_invoke_trigger”.

---

ianf • June 10, 2016 2:08 PM

@ fedes,
interesting multi-step *stochastic approach* of yours for validating purity of binaries [love then grown-up words]. You forgot to add an estimate of the time the outlined procedure is expected to take per stochastic validation session per kilobyte of decompiled binary. Calculate it, post here, I'll think about it – no promises but, hey! you ought to know by now that expecting "the Internet" to test your hypotheses without a sexy Indiegogo/ equiv. video and AT LEAST bumper stickers in a tiered reward setup, is a non-starter.

@ Chad M ruminates on Microsoft's plans for insertion into C++ binaries… “*today it's telemetry, tomorrow it's... what, exactly*?”

Assuming they've already been there done that, there'd but be one avenue still open to them: grafting of **telepathometry** of course — that is, provided that boffins have ironed the self-reinserting randomly intermittent GOTCHACK::NOACK errors in the TTCP/IP Telepax® suite (sadly, the lead dev died last year, some say not entirely unassisted by those who'd rather keep this tech under wraps).

Jim Dines • May 27, 2017 5:23 AM

There is **at least** one major flaw in this that nobody else seems to see. You need two compilers that can both compile the same source code successfully. That is NEVER going to happen for anything as non-trivial as a compiler. One often can't even get code to compile successfully with different versions of gcc! *(for example)* Thinking you will get Microsoft's and GNU's compilers to be able to do it simultaneously is a major cognitive distortion! :-)

This is the problem with computer security today in my not so humble opinion. There is a boatload of this kind of *Trust us, this is possible because it happens in my mind* kind of security. If Wheeler or anyone else can provide some exploit code in the form of a Ken Thompson Reflections on Trust style compiler and actual *anti-exploit tools and procedures that prove it empirically when followed* then it is all just meaningless mental masturbation.

---

russian typewriters • September 11, 2017 3:18 AM

Consider what Mr Thompson said:

"The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. 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."

Even a hardware CPU can be described as "code". It really is turtles all the way down.

How are you going to trust your two compilers when you can't really trust the hardware they run on? The hardware is all imported and could have multiple backdoors in it.

Pretty much no organization, company, or govt in the world takes software and data security seriously. They all say it is their top priority after they get cracked open but who can believe that.

Once the next war starts, all the computers and internets might suddenly become "enemy" and have to be turned off. This would have a drastic effect on most modern countries.

Russian govt now uses typewriters for stuff they don't want other people to know. They simply don't trust computers that much any more.

---

mos ano ni mos • October 13, 2017 4:47 AM

I remember this Russian typewriters news . This propagated in English language news mention they get set of "electric typewriters" . Why electric ? With possible backdoor? Perhaps because nobody manufacture mechanical ? Or it was kind of planted in news message (hope) we can get them either-way. It does not make sense and is just kind of wash like most of above discussion.

There is a way of having secure system - just abort parasites sucking people veins - to have trust by trusted sagacious society. But for this, in a country on his knees during the anthem is perhaps long way to dawn.

ps(see also Elbrus ru.made computers, chips ... sand for silicon cant carry info viruses, at least not in my universe)

b) most probably the Thomson virus is more probably implemented in go golan g stolen land , mostsad as ziombie spook zbuk who promised to "Don't be evil" and with growing bigger thour creed simply retrieved.

Subscribe to comments on this entry

Leave a comment

Login

Name (required):

E-mail Address:

URL:

☐ **Remember personal info?**

Fill in the blank: the name of this blog is Schneier on ___________ (required):

Comments:

Allowed HTML: <a href="URL"> • <em> <cite> <i> • <strong> <b> • <sub> <sup> • <ul> <ol> <li> • <blockquote> <pre>

Preview Submit

← Friday Squid Blogging: Poetry

Reading RFID Cards at Yards Away →