Critique of Diverse Double-Compiling

Paul Jakma

20th September 2010

Introduction

In one of the classic works of computer science, Ken Thompson described how to subvert security by modifying a compiler binary to clandestinely insert backdoors into compiled programmes[7]. His fundamental point being that there is a limit to the amount of trust we can place in systems we have not built ourselves. In [11] a technique called "Diverse Double Compiling" (DDC) is described where a 2nd compiler is used as a check on 1st compiler. It is claimed this technique counters Thompson's attack. This paper, will argue that claim is not quite fully justified.

Background

The Thompson paper[7] describes a specific attack, and then makes a general point from it. The specific attack described is as follows:

1. A compiler is modified to detect when it is compiling certain security-critical system programmes and subvert their operation. E.g. the Unix "login" programme which has the authority to grant users requests for access to the local system, and insert a bug into it, such as a "backdoor" to subvert it and grant otherwise unauthorised access to certain users.

2. The compiler is also modified to detect when it is compiling itself, so as to inject these 2 abilities into the resulting binary regardless of the source.

3. All trace of 1 and 2 are removed from the compiler source code, which is then distributed on and claimed to be the full source of the compiler.

The result is a compiler binary which will "infect" whichever system programmes with bugs to subvert system security, but will also "infect" any further compiler binaries, even when compiled from a clean source. This subversion of security is impossible to detect from the inspection of the sources of any of the code involved.

Generally Thompson takes this as demonstrating a limit in the trust we can place in computer systems provided to us. He mentions he could have picked "any programme-handling programme", and further that this attack could be carried out at different levels of the system, including at a microcode level. He is speaking generally on the risks of unverified executable code and the origins of trust. To quote his conclusion, "*You can't trust code that you did not totally create yourself*". Given the lecture thereafter on morals and the need for social norms, it may even be reasonable to think he wishes to show that trust ultimately is unavoidable.

Recent efforts to secure systems via initiaves such as "Trusted Computing" have sought to counter such system subversion attacks by securing the system from a layer below, and providing a provable chain of trust in the binaries that are run[1]. These efforts can still fail through clever exploitation of bugs in the secured layer, or even through subversion of the layer below the secured layer[5]. Even where these schemes succeed, they can only prove from where the software originates - Thompson's question of whether the originator of executable code is to be trusted remains.

Thompson's demonstrated attack is essentially what we would today recognise as a class of computer virus. One that specifically targets compilers and system executables, and replicates through the action of compiling code - source code being the common mode of software distribution in that day, and hence compiler toolchains being the only vector by which to propogate binary executable exploits. However computer virus technology has developed immensely since. Viruses today are capable of infecting arbitrary executables, either by prepending themselves or by exploiting details of the container format, and can target arbitrary victim executables[8, 4]. Ordinary PCs have been shown to be systemically subvertable by exploiting recent self-virtualisation capabilities, to insert a hidden layer between the operating system and hardware[3], akin to

[1] Exactly to whom the trust is being proven can depend. E.g. for "DRM" the idea is to prove to the software supplier and 3rd parties that the device owner has not subverted the software; but "trusted computing" can also be used to try prove to the device owner that their software is genuine.

a microcode exploit. There is a rich ecology in the computer virus world, including *worms* that propogate themselves and *rootkits* that include tools and code to help hide infestations from the operator[10].

D.A. Wheeler's DDC technique[11] claims to counter the attack demonstrated by Thompson. The technique assumes the availability of a 2nd compiler C_t, which can be trusted to correctly compile the target compiler, C_A, from its sources, S_A. S_A is, of course, free of any subversion. This need not mean that C_t is defect free, or that C_t can be trusted to compile *itself* correctly, the author claims. It is also assumed that C_t is not suitable for general use, either because it can not be trusted completely or because it is not adequately performant in some way. E.g. C_t may only implement a subset of the language sufficient to target compilers, and/or not produce performant code, and/or not be able to compile popular extensions of the language, etc.

The method is as follows, having already verified that A reliably compiles itself to to an identical copy of itself, i.e. that $A = c(S_A, A)$, as a prerequisite step:

1. The sources of compiler A, S_A, are compiled with C_t to obtain a 2nd binary of A, $C_1 = c(S_A, C_t)$
2. Compile S_A once more to obtain $C_2 = c(S_A, C_1)$
3. Compare C_2 to A

The claim is that if the 2 binary compilers match in step 3 that these binaries can then be trusted to match S_A, the source of A.

Critique of DDC

While the D.A. Wheeler paper and thesis it is based on are quite interesting in terms of practical techniques, the claim it makes to have successfully countered Thompson's "*Trusting Trust*" attack is not supportable.

The paper requires that C_t be trusted to be able to compile S_A without subverting it. It therefore assumes either one of:

1. That C_t is fully verified or trusted.
2. That, while C_t may not be fully trusted, any subversion of binaries it contains will be targeted only at the compilation of sources other than S_A.

Also, the environment in which C_2 and C_1 are prepared and all comparisons are made are required to be trusted to at least the same degree as C_t.

Assumption 1: C_t is fully verified or trusted.

Under assumption 1 C_t has either been written by the user, or been proven to be free of any subversion in some other way. Therefore the user can bootstrap their compiler by using C_t to obtain C_1 and then produce a trusted production compiler $C_p = c(S_A, C_1)$ and further compile the system with that. The untrusted A compiler binary therefore can be ignored if the goal is simply to build a system that is known to be free of Thompson attack subversions.

DDC can now be used to show that A differs from S_A, if care is taken to ensure A is only executed in a disposable environment (see next sub-section).

Clearly though, this is not an example of countering the Thompson attack, rather it re-inforces Thompson's central point on bootstrapping trust, quoted above.

Assumption 2: C_t does not target S_A

Sub-Assumption: Implementation Diversity

Where assumption 2 is relied on instead, the paper justifies trust by assuming that the only compiler C_t could subvert in compilation is its own source. It justifies this by assuming that C_t and A are independent works by independent people, or that an attacker who subverts A would have to go back in time to subvert C_t and/or generally assuming that attacking 2 code bases is too difficult. Interestingly, it makes these assumptions even when earlier in the paper it is acknowledged that certain well-resourced attackers exist who could afford to devote multiple developers and years toward carrying out this attack, if they wished.

Even considering only single-agent attackers, it is is not safe to assume that those who work on one compiler are independent from those who work on other compilers. People who have specialised in some area of technology are obviously a subset of technologists generally, sometimes quite small subsets. They likely know a significant number of other people in that field through past study, employment and conferences. Such specialists may move between working on various implementations as part of their normal career progression. They may try out prototypes of competing implementations on their systems.

In short, it can not be universally assumed that seemingly diverse compilers C_t and A are independent in terms their implementors.

Also, there is an assumption that only implementors can execute the attack. However the software may be open to modification as it is distributed, and there may

be surprising intersections here in the chains of trust from software originator to end-user. E.g. one work may be distributed using another similar work, from which it otherwise seems independent, such as when the software distribution of one system is mirrored on servers running another, or such as when software is sent to a CD/DVD pressing plant. Best practice for DDC therefore demands careful verification of electronic signatures published by the originator, to verify distribution integrity.

Sub-Assumption: Attacks Restricted to Compiler Level

There is another flaw in this assumption, in that it assumes that attacks will be confined to compile-time subversion and/or that subversion will have specific targets. This is perhaps not a reasonable restriction to make of the Thompson paper, as explained earlier.

The range of attacks open to untrusted code however are much much greater than just the specific attack demonstrated by Thompson, as is obvious from the breadth of computer viruses since Thompson's demonstration. As an example consider a virus which arranges the meta-data in the executable container so that functions calls to certain key library symbols are forwarded to its own code, allowing the virus to hook in to those calls. E.g. on a Unix system, it might divert open() and close() by modifying the PLT and/or GOT tables in the ELF header[6][1], such that it could itself surrepticiously quickly re-open any files opened by the infected programme and infect them as it wished, as well as arranging its code to be called near the startup of the process. This virus could then additionally take on further behaviour as/when it detected it was running inside various kinds of processes (e.g. "login"), so as to compromise the system further and hide itself from detection.

In such a way, C_t can generally compromise arbitrary other executables, including any compilers, without any need for special access to the distribution process of other compilers. The assumption that any subversion of C_t will affect neither A nor any binary compiled by A is therefore unsafe. The conclusion therefore must be that C_t and A are *equally* untrustworthy and dangerous, at least when assumption 1 does not hold.

Sub-Assumption: Diversity in Time

It is also argued in the DDC paper that diversity of time has benefits. This is primarily because with the compiler-level attack assumption, it is argued an attacker would have to anticipate the implementation details of future compilers and that hence this would be impossible.

As the compiler-level assumption is unsafe, so this is. I.e. if the old compiler attacks generic executables, then it can attack executables generally in the machine being used to run it, which obviously must support the technology of that old compiler.

However, there may be other benefits to time diversity, it could be argued. The old compiler may have had long use and more scrutiny over time than current compilers. Or any technology the old compiler might attack, while still supported sufficiently to allow the old compiler to run, may be superceded and no longer in any significant use.

Still, this class of argument generally is flawed, for it assumes that the compiler binary can only contain *old* attacks. In order for this assumption to hold it must be proven that the binary representing the old compiler really is what was the old binary - i.e. that a modern attacker has not managed to modify the old compiler with a *new* attack. This problem at first glance appears identical to the problem solved by signing software with electronic signatures. However now time works against the person wishing to validate their compiler. Electronic signatures generally rely on crypographically secure hash functions and such functions do not remain secure for long[2, 9]. So the older the compiler, the harder it will be to trust its contempary signatures. So even where it is unlikely that the original compiler would be capable of subverting a future environment, an infected future environment may be easily capable of subverting what is thought of as the past. Hence the assumption that time provides security is far from safe.

It may be that certain old computers exist, still in working order, and which its operators are near certain have been safe from infection, through dormancy and physical security. In such a case an operator may be able to use that computer as a trusted environment. Perhaps if that operator is well-respected, the rest of us may be able to trust them and their validations of future compilers - no differently to the trust shown in Thompson.

Assumption: Subverted compiler contains no DDC counter-measures

DDC assumes that $A_{n+1} = c(S_A, A_n)$ will converge and result in a stable, identifying binary. This assumes A contains no counter measures to DDC. E.g. in the DDC paper it took several rounds of fixing or working around bugs in tcc to obtain convergence. It is therefore quite probable that a malicious compiler author might introduce quite a series of innocious and otherwise innocent looking bugs into the source, in addition to their subversion of the binary. Thus the bar

of expertise and time that are required to for anyone to be able to carry out DDC of a compiler can be made higher.

As the DDC paper notes, it would be useful if DDC were regularly carried out on compilers and that stable self-recompilation should be demanded of compilers. Then a compiler employing this counter-measure might call attention to itself.

Multiple DDC

The DDC paper also mentions the possibility of conducting multiple DDC tests. E.g. consider the matrix of results for DDC with compilers $C_1, \ldots, C_n$, such that $D_{ij} = c(S_i, c(S_i, C_j)) \mid (i, j) \leq n$. What can we conclude if $D_{12}, \ldots, D_{1n}$ are all equal, but differ from D_{11}? If $n = 2$, then we have normal DDC and we can only conclude we need to investigate further. If $n > 2$ it is tempting to conclude that C_1 must have been subverted, and it seems that must at least be the most likely case. However, as we can not trust any compiler, we can not rule out the less likely possibility that $C_2, \ldots, C_n$ are subverted and in collusion with each other - if this seems improbable, consider that it is not that unlikely for multiple environments to fall victim to the same virus. Thus, the discrepancy must still be investigated further by other means.

The greater the number of compilers, the greater confidence we might have. However as DDC appears to be quite time-consuming, particularly when it requires fixing compiler bugs, higher numbers of compilers potentially may be less and less practical.

The situation where all compiler binaries have been infected with the same attack remains beyond the ability of this technique to counter though and, as per above, such a case is far from impossible. Thus multiple DDC also requires at least some measure of faith.

Conclusion

The assumptions the DDC paper relies upon, where they are not consistent with Thompson's conclusion, are not fully justified. Particularly so if a general view is taken of the scope of Thompson's attack, as he apparently intended. In which case then the problem equates to virus detection.

DDC thus is a probabilistic technique to detect a certain limited class of compiler viruses, without a priori knowledge of the virus itself. The DDC technique can not, this paper claims, reliably say whether a compiler has been subverted or not. However it may potentially flag some discrepencies, for further manual examination.

References

[1] Silvio Cesare. SHARED LIBRARY CALL REDIRECTION VIA ELF PLT INFECTION. *Phrack Magazine*, 0xa(0x38), 2000.

[2] V. Henson, I. IBM, and R. Henderson. Guidelines for Using Compare-by-hash.

[3] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *SP 06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.

[5] Diomidis Spinellis. Reflections on trusting trust revisited. *Commun. ACM*, 46(6):112, 2003.

[6] Sumant Tambe, Navin Vedagiri, Naoman Abbas, and Jonathan E. Cook. DDL: Extending Dynamic Linking for Program Customization, Analysis, and Evolution. In *In Proc. International Conference on Software Maintenance*, 2005.

[7] Ken Thompson. Reflections on trusting trust. page 1983, 2007.

[8] M. Van Oers. LINUX VIRUSES ELF FILE FORMAT. 123, 2000.

[9] B. Van Rompay, B. Preneel, and J. Vandewalle. On the security of dedicated hash functions. pages 103–110, 1998.

[10] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *WORM 03: Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 11–18, New York, NY, USA, 2003. ACM.

[11] David A. Wheeler. Countering Trusting Trust through Diverse Double-Compiling. In *ACSAC 05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 33–48, Washington, DC, USA, 2005. IEEE Computer Society.