Hardening Go

Mitigating Trusting Trust Attacks
for Compiler Security

Vivi Andersson
vivia@kth.se

March 27, 2024

1 Introduction

Since personal computers became commercially available in the late 70's, the number of people who write the computer programs they run has decreased notably. At the same time, our society relies ever more on code. Trust is inherently embedded in the daily usage of technology; users trust the company, the company trusts the developers, and the developers trust their code. While an abstraction of complex computer programs is necessary to make software available, this comes with risks that end users might not be aware of. *Can we really trust the code that we run our society on?*

This question was posed by Ken Thompson in 1984 during his Turing Award lecture *Reflections on Trusting Trust* [1]. Thompson showed that a malicious actor can exploit the trust developers put in their source code if they manage to taint their binary files instead. Through self-inserting code, an attacker can make the affected software behave as they intend, and erase any malicious code in the source [1]. Furthermore, if they target software that is the programming language itself, e.g. a compiler, then all software compiled by the exploited source will be corrupted as well. Any traces can then be deleted from the source code while the malicious code flourishes and replicates in the binary realm, unreadable to the human eye.

The *trusting trust attack* Thompson described intercepts the *software supply chain*, i.e. the development and distribution process of software. In recent years, the number of reported attacks of this type has increased drastically (see Figure 1). From 2019 to 2022 the number of reported software packages affected by an attack has increased from 702 to 185 572 [2]. Open-source software (OSS) is particularly susceptible to these attacks due to availability and interdependencies among various software components [3].

Go (or Golang) is an industry-standard free OSS, developed by a team at Google [4]. As a popular programming language, this makes a high-value target for an attacker, considering the possibilities of propagation by the go compiler, *gc*.

Whereas Go has various security measures built in as a standard, the question remains if they can protect their users from any malicious code possibly residing in their binary releases and bootstraps. With the inherent chain of trust in software, taking measures to protect users from sophisticated attacks on the software supply chain is crucial.

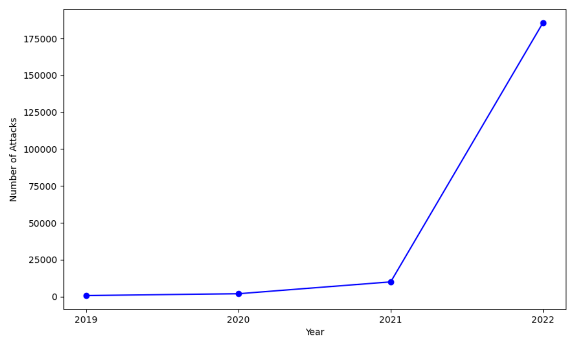

Figure 1: *Supply chain attacks by year since 2019 [2].*

2 Scope

This essay aims to discuss trusting trust attacks as a threat to compiler security in the context of Go. More specifically, the viability of a trusting trust in Go will be investigated, and possible mitigation techniques that are already implemented. Finally, we will discuss any vectors which are yet to be sufficiently protected against such attacks, and what actions are needed to secure these.

3 Supply Chain Attacks

Software supply chain attacks often exploit software that is considered trusted, but in fact, is infected by malware. A common way to achieve this is by injecting the attack before a trusted vendor digitally signs the software, a method intended to increase trust in distributed software [5].

Win32/Induc is a well-known example of a software supply chain attack that affected Windows 32-bit architectures by compromising the *Delphi* compiler [6]. This attack targeted a commonly used library of Delphi, such that when used in compiling other programs, it would insert the malicious code into the executables. At least three versions of this attack have been discovered, each new version becoming more sophisticated and severe [7].

3.1 Trusting Trust Attacks

A trusting trust attack leverages self-replication to insert malicious code into low-level artefacts such as compiler- or assembly binaries, which makes them particularly difficult to detect. In this attack, the malicious actor modifies the compiler to recognise another pattern than the originally defined language syntax. When the compiler then encounters this pattern, for example compiling a UNIX login program, the pattern serves as a *trigger*. This trigger can then execute arbitrary malicious code, such as inserting a backdoor.

Furthermore, Ken Thompson showed that if a second pattern recognises if the program to be compiled is a compiler itself (the process of bootstrapping), the malicious compiler can inject itself into the new compiler as well (see Algorithm 1). In this way, the malicious code can propagate in every compilation of the compromised compiler. After this, the traces in the source code of the compiler can be deleted, as the binaries are already compromised.

Currently, there a are no known pure trusting trust attacks reported. On the other hand attacks like the Win32/Induc virus which leverage self-replication in compiler binaries have many similarities with the trusting trust attack.

```
void compile(char *source) {

    if (match(source, "login")) {
        compile("login trojan");
        return;
    }

    if (match(source, "c compiler")) {
        compile("malicious compiler");
        return;
    }
}
```

Algorithm 1: A subverted C compiler

4 Go Programming Language

Go was started as a project by the Google engineers Robert Griesemer, Rob Pike and Ken Thompson in 2007. Go aimed to address shortcomings of programming languages in the new landscape of complex hardware (e.g. multiprocessors) that were arising during that time [8]. In 2009, the language was released, and it has been growing in popularity since then (see Figure 2).

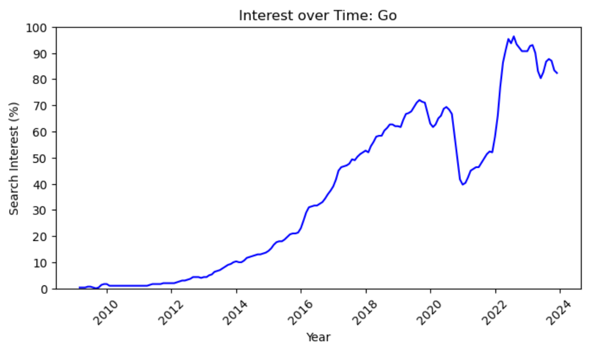

Figure 2: *Search Interest for Golang over time [9].*

5 Threat Analysis

For an attacker to be able to conduct a trusting trust attack, they need to compromise a compiler. This can be done either by modifying a trusted compiler or creating a new unofficial and malicious compiler, and subverting users to download this compiler instead.

For open-source software, the task of modifying straight to the source can be easier than for closed software systems, as they often are open to any contributor. On the other hand, OSS typically also rely on *maintainers* and scrutiny of the submitted code, which can detect malicious source code modifications. However, malicious code can still be difficult to detect by scrutiny if the attacker utilises *obfuscation*, e.g. by encoding the payload in Base64 or Hex [5].

The second approach of manipulating users to download an unofficial compiler is also viable. An example of an attack like this was the distribution of *XCodeGhost*, a malicious version of the Apple developer software XCode. The malicious software was distributed through a cloud service, and the quicker download speeds offered by them in China, in contrast to the official distribution channel, led to a substantial number of downloads. The malware subsequently propagated to two applications distributed in the Apple App Store, created using XCodeGhost [10].

The possibility of propagating malware through a tainted compiler to other programs makes a trusting trust attack a valuable target for a malicious actor. If masked properly, this attack requires relatively low effort from the attacker in exchange for the impact on user systems. David A. Wheeler mentions the possibility of a single attack having the possibility to take *"control over banking systems, financial markets, militaries, or governments"* [11].

6 Mitigation Techniques

Various approaches to mitigating trusting trust attacks have been investigated and implemented.

6.1 Reproducible Builds

Lamb and Zacchiroli present the approach of *reproducible builds* which aims at designing software in a way that will result in identical build artefacts for the same build inputs. They present the viability of designing software deterministically by the practical implementation of making Linux Debian reproducible (see https://reproducible-builds.org). Malicious distributions of software can then, by relying on reproducible builds, be discovered by comparing *checksums* of a trusted distribution with the one the user has downloaded [12].

Russ Cox, engineer at Google, reports achieving reproducibility for the latest version of Go as of November 2023 (version 1.21). which should allow users to verify their build of a version is the same as the posted binaries [13].

6.2 Debootstrapping

Reproducible builds might not be enough to secure the absence of trusting trust attacks if the build environment is already compromised. The approach of *debootstrapping* builds aims at minimising this attack surface by removing the need for bootstrap binaries, i.e. prebuilt binary versions of software. Courant et al. propose debootstrapping compilers by compiling a reference interpreter for a subset of the language with a compiler written in another language. Other approaches to debootstrapping include restoring legacy versions which do not rely on bootstrapping [14].

Approaches to bootstrapping Go include for example using a previous official binary version of Go, or their implementation in C [15]. No attempts to debootstrap Go are found.

6.3 Diverse Double Compilation

Diverse double compilation (DDC), popularised by David A. Wheeler, is a related approach that can detect compromised build systems. Assuming one trusted compiler, this one can be used to compile another untrusted source, to detect self-inserting code in the compiler. This technique compares binary artefacts to detect differences [11]. DDC requires compilers which can self-replicate, and reproducible builds further aid this approach by ensuring comparisons of binary files do not differ due to non-deterministic compilations.

For Go, the reproducible version (1.21.0) should be useful as the trusted compiler in a DDC to verify another Go compiler [13].

7 Discussion

The three approaches of reproducible builds, diverse double compilation (DDC) and debootstrapping all emphasise the need to secure software supply chains from self-inserting attacks [12] [14] [11]. The method of reproducible builds is a way to verify correct representations of the distributed software. Russ Cox at Google considers this the "best way to address" supply chain attacks [13]. Whereas this is true for attacks such as the XCodeGhost, which tricks users into downloading a non-official release of software, this might not be enough for all cases. For example, if the software already has been affected by a trusting trust attack residing in the source code, this approach does not mitigate the attack. Nonetheless, this approach does mitigate the trusting trust attacks intercepting the distribution channel of a compiler.

The method of reproducible builds also facilitates the implementation of diverse double compilation. DDC can detect the attacks that are already present, and hide under official releases. Furthermore, diverse double compilation provides a more versatile way to mitigate these attacks, as it does not rely on the end user building their version in the same environment as the trusted version. Thus, adopting reproducible builds in Go is valuable for the effort of preventing trusting trust attacks on a Go compiler. The approach of debootstrapping has the role of minimising the attack surface for self-inserting attacks and thus serves as a valuable complement to the other two approaches.

One remaining consideration for all presented approaches is how to make them available and easy to use for end users. Regarding making reproducible builds viable, this method requires setting up a network for parallel trust exchange, which is yet to be done [12]. Whereas this is possible, I argue the somewhat more available approach of verifying binaries through diverse double compilation should be the focus. To make the DDC approach commercially available for single end-users the process should be automated, which is an area for future research.

As the XCodeGhost attack discussed in Section 5 made clear, malicious code can pass on to trusted channels if not properly scrutinised. Thus, it is evident that the security of distribution channels needs to be properly addressed as well. These types of mitigation techniques are less technical and instead on the psychology of technology use, such as preventing social engineering techniques adopted by attackers.

8 Conclusion

The viability of a sophisticated compiler attack like a trusting trust attack was shown more than 30 years ago today. Whereas no pure such attacks are known yet, the increase in software supply chain attacks makes it evident that measures need to be taken to prevent future attacks. Go was designed partly by Ken Thompson himself which may have aided in their overarching work to mitigate these attacks, but there are still attack surfaces for Go that need to be properly addressed. One viable implementation of mitigation is the diverse double compilation, which if made easily available to all Go developers, can have an important role in preventing trusting trust attacks in the future. These actions are important to take today, given the stakes at hand for software systems around the world.

References

[1] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, p. 761–763, aug 1984. doi: `https://doi.org/10.1145/358198.358210`.

[2] Comparitech, "Annual number of software packages impacted by supply chain cyber attacks worldwide from 2019 to 2023 ytd," March 26 2023. [`https://www.statista.com/statistics/1375128/supply-chain-attacks-software-packages-affected-global/` (retrieved 2023-12-01).

[3] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), pp. 1509–1526, IEEE Computer Society, may 2023.

[4] Go, "The go project," 2023. `https://go.dev/project` (retrieved 2023-11-27).

[5] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (C. Maurice, L. Bilge, G. Stringhini, and N. Neves, eds.), (Cham), pp. 23–43, Springer International Publishing, 2020.

[6] Microsoft Corporation, "Virus: Win32/induc.a," 2009. `https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?name=Virus%3AWin32%2FInduc.A` (retrieved 2023-12-21).

[7] R. Lipovsky, "The induc virus is back!," Sep 2011. `https://www.welivesecurity.com/2011/09/14/the-induc-virus-is-back/` (retrieved 2023-12-21).

[8] R. Pike, "Go: Ten years and climbing," 2017. `https://commandcenter.blogspot.com/2017/09/go-ten-years-and-climbing.html` (retrieved 2023-12-15).

[9] Google, "Google trends: Golang," 2009–2023. `https://trends.google.com/trends/explore?date=2009-01-01%202023-12-15&q=golang` (retrieved 2023-12-15).

[10] C. Xiao, "Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store," *Unit 42 by Palo Alto Networks*, September 17 2015. `https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/` (retrieved 2023-12-15).

[11] D. Wheeler, "Countering trusting trust through diverse double-compiling," in *21st Annual Computer Security Applications Conference (ACSAC'05)*, pp. 13 pp.–48, 2005. doi: `https://doi.org/10.1109/CSAC.2005.17`.

[12] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2022. doi: `https://doi.org/10.1109/MS.2021.3073045`.

[13] R. Cox, "Perfectly reproducible, verified go toolchains," August 2023. `https://go.dev/blog/rebuild` (retrieved 2023-11-27).

[14] N. Courant, J. Lepiller, and G. Scherer, "Debootstrapping without archeology: Stacked implementations in camlboot," *The Art, Science, and Engineering of Programming*, vol. 6, no. 3, 2022. doi: `https://doi.org/10.22152/programming-journal.org/2022/6/13`.

[15] Go, "Installing go from source," 2023. `https://go.dev/doc/install/source` (retrieved 2023-12-21).