(Preliminary outline of) Master's Thesis

Diverse Double-Compilation in Context of Bitcoin Core

Niklas Rosencrantz

December 2021

KTH

Supervisor: Prof Martin Monperrus, Examiner: Prof Benoit Baudry

Contents

Student's name and e-mail address: Niklas Rosencrantz niklasro@gmail.com KTH-id nik@kth.se

1 Introduction

The problem of deceptive compilers introducing malicious code is relevant for computer security and a difficult class of attacks to analyse and discover. One solution for this is to use diverse double-compilation as a countermeasure. Yet it is hard to change the compilation pipeline of real software. I learn and write about the design of, and I implement and evaluate diverse double compilation for bitcoin-core. To attack and compromise Bitcoin core one might consider a few attack methods and exploits that can be tried and analysed, for example:

1. stealing a PGP key from a maintainer and use that to sign new versions of Bitcoin-Core
2. a domain specific attack where the compiler inserts code that changes the recipient address in 1/10000 Bitcoin transactions.
3. a malevolent maintainer could upload malicious code, hide it. that would get caught by the verify signatures script

2 Preliminaries

3 Background and Related Work

Ken Thompson wrote in a famous article: "To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software [8]."

Thompson delivered the original Reflections On Trusting Trust with the idea and article about hidden Trojan horse in the binary parent compiler which is used to compile the next version of the compiler [8].

In the years after Thompson's original article, compiler security and build security have emerged as an increasingly important research area. The potential for deceptive bugs to propagate in binary without being seen in the source code implies that the possibility for enormous damage is technically feasible. For example, if the GCC would contain such a bug and is used to build the next version of Bitcoin core, an individual or a Government organization could be able to control the cryptocurrrency centrally. This project will investigate the threats, vulnerabilities and means of countermeasure for these types of deceptive compilers.

I explore, create and evaluate new and established techniques of diverse double-compiling in the context of the build system for Bitcoin Core [4].

David A. Wheeler wrote that randomized testing would be "unlikely" to detect a compiler Trojan:

> Faigon's "Constrained Random Testing" process detects compiler defects by creating many random test programs, compiling them with a compiler-under-test and a reference compiler, and detecting if running them produces different results [Faigon]. Faigon's approach may be useful for finding some compiler errors, but it is extremely unlikely to find maliciously corrupted compilers.

I believe David A. Wheeler claimed that to be the circumstances of the Trojan horse only executing on very specific input. On the other hand, it's a rather general statement which doesn't go into detail why we cannot input the source code of the compiler to a compiler binary and then do fuzz testing with variations on that.

A sufficiently complicated project often takes a lengthy and complex build process with many different binaries from several different vendors that are utilized as dependencies and libraries in the toolchain. Compiling Bitcoin Core locally for example takes 40 minutes with an Intel I7 CPU. Compiling GCC itself takes several hours, and compiling Firefox takes a whole day.

1. Ken Thompson's original article named Reflections on Trusting trust [8]. This article is a good introduction to the problem and subject.

2. Defending Against Compiler-Based Backdoors [6].

3. Countering trusting trust through diverse double-compiling [9]. This article describes the DDC in detail and provides the history and context.

4. The original Bitcoin paper and the ongoing work on the build system of Bitcoin Core [4] [2]. I plan to use a code pipeline or create a code pipeline myself for Bitcoin Core in order to examine the builds more easily.

Additionally the following articles might seem relevant but I did not yet study them.

1. How a simple bug in ML compiler could be exploited for backdoors? [3].

2. Deniable Backdoors Using Compiler Bugs [1].

The original Bitcoin article took the project of solving the problem of double-spending

> We propose a solution to the double-spending problem using a peer-to-peer network.

My aim is to give answers and elaborations for the following questions.

1. What are the threats and the vulnerabilities that should be protected against with DDC? What are the technical benefits and shortcomings of DDC compared to other solutions to real and potential problems with compiler vulnerability and build security? What critique is known against DDC and what other critique is reasonable to give?

2. What does a real demonstration of DDC look like? Show (don't tell) the audience a real example of an actual procedure.

3. How can DDC be applied to the build system of Bitcoin Core and what can we learn from it? What would be the challenges and benefits? Which part of the build process is more likely to be vulnerable than other parts? Can I recompile Bitcon Core as the system-under-test and what will that result be?

4. What possible and provable improvements can be made compared to the current state of the art and how can the technology become portable and easily available for software engineering teams and researchers? Can a compiler binary be made more auditable and examined and if yes, how??

Modern OSs randomize memory section addresses (it makes some attacks more difficult), so if anyone restarts the process, the addresses of the instructions might be different.

I will use the build process of Bitcoin Core as my main applied DDC. I will find out about relevant frameworks, build systems and platforms, such as GUIX (https://guix.gnu.org), maybe Frama-C (https://frama-c.com) and/or the CompCert project (https://compcert.org/).

I can create checks and provisioning a system in a Docker container for portability and reproducibility. I have access to the Bitcoin build toolchain and can build Bitcoin Core with GCC [7]. I can create the demonstration of a compiler Trojan horse in the source code of a compiler to learn and show the concepts for real. I can perform the checks and verify that results are according to expectations [7]. I will learn about and use the Bitcoin core build system and its compiler [2]. The 4 main attacks that can happen directly as a result of the 51% attack are: Selfish mining. Cancelling transactions. Double Spending. Random forks.. I will learn and use the GUIX build system. I will also explore some ideas that are new or rarely used in compiler and build security such as comparing checksums from known "safe" builds with new builds of the same source [5].

4 Research Area

Technologies in provable security have achieved increasing interest in software engineering and related disciplines. With DDC one compiles the source code of the compiler C0 with a proven other compiler C1. Then use the result B1 to compile the compiler C0 and compare it with a compilation from the distributed compiler.

After many years, the idea of countermeasures the Trojan horse using diverse double-compilation appeared [4].

5 Problem Statements

One problem with the compiler backdoor Trojan horse is to actually create a real version of the threat.

There are a lot of claims about the Thompson compiler backdoor (https://stackoverflow.com/questions/781718/thompsons-trojan-compiler/782677) without anyone actually showing that the attack can be done in practice.

I did put a complete reproducible example with TinyC on GitHub https://github.com/montao/tinycc/blob/ddc/.github/workf but it had a flaw to begin with which did make the attack unrealistic at first: I could make the Trojan replication to all future versions of the compiler but that is only possible if an extra header file with the attack vector is available when future versions of the compiler is compiled. The header will not be available like that so my attack would not be realistic.

For example if I try to corrupt the TinyC so that it inserts the attack code to future versions I can put an include statement right before it opens the file to compile:

Listing 1: C example

```
    ...
    /* open the file */
#include "attack.c"
    fd = tcc_open(s1, filename);
    ...
```

The file attack.c contains the attack code and will be included by the preprocessor. The attack code detects my dummy login program and lets the username hackerken login without a password. The attack code is also replicated to all future versions of the compiler when TCC compiles itself but only if the attackvector.h is supplied to future generations. How is the attack supposed to take place without supplying the header file? It seems that I got a circular dependency in my attack code.

Listing 2: C example

```
#include "attackvector.c"

    // naive signatures for now
    // needs to ./generate_attackvector < attack.c > attackvector.h
    static char compile_sig [] = "/* open the file */";
    static char login_sig [] = "int login(char *user) {";
    static char login_attack [] = "if(strcmp(user, \"hackerken\") == 0) return 1;";

    /* attack code follows */
    ...
```

The file attackvector.c is generated from running generate-attack-array

Listing 3: C example

```
#include <stdio.h>

int main(void) {

    printf("static char compile_attack[] = {\n");
    int c;
    while ((c = fgetc(stdin)) != EOF) {
        printf("\t%d,\n", c);
    }
    printf("\t0\n};\n\n");
    return 0;
}
```

...so that the attackvector.h contains the array with the attack.c source code itself.

Listing 4: C example

```
static char compile_attack[] = {
    47,
    47,
    32,
     ....
```

But that makes the backdoor require that all future versions of TCC is compiled with the header file attack.array.h making the backdoor dependent on the header file being present, which it won't be from an honest clone of the compiler.

The way I fixed it to not need the extra header file was to compile the .c file twice and paste it into the char array and like that it will self-reproduce any (finite) number of generations without being seen in the source code.

In general it is often desirable to make every technical system secure. The specific problem in this case is the trick of hiding and propagating malicious executable code in binary versions of compilers - a problem that was not mitigated for 20 years during 1983 and 2003.

In the context of security engineering and cryptocurrency, a security engineer could face a reversed burden-of-proof. The engineering wouod need to prove security which is known to be hard, both for practical reasons and for the reason of which and what kind of security models to use. For example, a security engineer might want to demonstrate a new cryptographic alternative to BTC and blockchain. Typically the engineer might get the question if he can prove that there is no security vulnerability.

At least 7 or 8 different operating systems and at least two very different compilers can build Bitcoin Core (https://github.com/bitcoin/bitcoin/tree/master/doc)

6 Ethical Aspects of the Research

The ethical grounds are to show openness (open research) and transparency of what is being done, even during the research work as well as after its completion. The ethical ground of white hat hacking are set in this way. (I should refer to papers and posts to read about this. It would be good to discuss some of them in the thesis.)

7 Connections to Current Research

8 Discussion

If we observe that the two diverse double-compiled compiler binaries are identical then we know that our build system is safe. But if they are not identical we haven't proved that our build system is compromised.

References

[1] Scott Bauer. *Deniable Backdoors Using Compiler Bugs*. 2015. URL: `https://www.alchemistowl.org/pocorgtfo/pocorgtfo08.pdf`.

[2] Bitcoin. *Bitcoin-core*. `https://github.com/bitcoin/bitcoin`. 2021.

[3] Baptiste David. "How a simple bug in ML compiler could be exploited for backdoors?" In: *arXiv preprint arXiv:1811.10851* (2018).

[4] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Decentralized Business Review* (2008), p. 21260.

[5] Kirill Nikitin et al. "{CHAINIAC}: Proactive software-update transparency via collectively signed skipchains and verified builds". In: *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, pp. 1271–1287.

[6] John Regehr. *Defending Against Compiler-Based Backdoors*. 2015. URL: `https://blog.regehr.org/archives/1241`.

[7] Niklas Rosencrantz. *Compile Bitcoin Core*. Youtube. 2021. URL: `https://www.youtube.com/watch?v=A3NnKuKwiAg`.

[8] Ken Thompson. "Reflections on trusting trust". In: *ACM Turing award lectures*. 2007, p. 1983.

[9] David A Wheeler. "Countering trusting trust through diverse double-compiling". In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE. 2005, 13–pp.