DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2022

Diverse Double-Compilation For Bitcoin Core

Niklas Rosencrantz

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Author

Niklas Rosencrantz
Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
KTH Royal Institute of Technology

Examiner

Professor Benoit Baudry
Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Supervisor

Professor Martin Monperrus

Electrical Engineering and Computer Science

KTH Royal Institute of Technology

Abstract

(Introduce the subject area)

The problem of deceptive compilers introducing malicious code has been described in several articles and studies of computer security and compiler security. During the years of research of this class of attacks they have implied certain additional sub-problems to analyse and discover, such as actually creating a working example of the attack and also the different methods to reduce the probability of such an attack. One mitigation that has been described is to use diverse double-compilation as a countermeasure. In context of the Bitcoin Core build system an attack and compromise of it is possibly by a few attack methods and exploits that can be tried and analysed, for example:

(describe the problems that are solved)

1. Stealing a PGP key from a maintainer and use that to sign new versions of Bitcoin-Core
2. Conducting 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

(The presentation of the results should be the main part of the abstract. Use about ½ A4-page.)

(Use probably one sentence for each chapter in the final report.) Write an abstract.

Introduce the subject area for the project and describe the problems that are solved and

described in the thesis. Present how the problems have been solved, methods used and present results for the project.

English abstract

Keywords

Computer security, Compiler security, Build security, Network security, Merkle tree, Bitcoin Code, Cryptocurrency, Cryptography, Signed code

Abstract

Svenskt abstract Svensk version av abstract – samma titel på svenska som på engelska.

Problemet med vilseledande kompilatorer som introducerar skadlig kod är relevant för datorsäkerhet och en svår klass av attacker att analysera och upptäcka. En lösning för detta är att använda olika dubbelkompilering som motåtgärd. Ändå är det svårt att ändra kompileringspipelinen för riktig programvara. Jag lär mig och skriver om designen av, och jag implementerar och utvärderar olika dubbelkompilering för bitcoin-core. För att attackera och kompromissa med Bitcoins kärna kan man överväga några attackmetoder och exploateringar som kan prövas och analyseras, till exempel:

Nyckelord

Civilingenjör examensarbete, ...

Acknowledgements

I would like to thank everyone who encouraged me, especially my professors Martin and Benoit, Fredrik Lundberg, Johan Håstad, Roberto Guanciale, David A. Wheeler, Basile Starynkevitch, Terrence Parr...

Acronyms

CPU	Central Processing Unit
GCC	GNU Compiler Collection
TCC	Tiny C Compiler
PGP	Pretty Good Privacy
DDC	Diverse Double-Compiling
RAM	Random Access Memory
CPU	Central Processing Unit
UEFI	Unified Extensible Firmware Interface
RoTT	Reflections on Trusting Trust
ASLR	Address space layout randomization
OSS	Open Source Software

Contents

Chapter 1

Introduction

One problem with computer systems and networks is that the system will inherently trust its creator i.e. the hardware constructors and the software authors, the upstream development team and the supply-chain. Changing the compilation pipeline of real software without it being seen might sound difficult but it can be done. In my research I examine the design of such attacks, and I implement and evaluate the tehcnique named diverse double compilation for bitcoin-core. I also suggest a few completely novel ideas to reduce the assumptions and reduce the probability even more that a system is compromised 1.1. During my research of the topic I've studied, created and evaluated certain new and established techniques of diverse double-compiling in the context of the build system for Bitcoin Core [20]. 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 Pretty Good Privacy (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

Provide a general introduction to the area for the degree project. Use references!

Link things together with references. This is a reference to a section: 1.1.

1.1 Background

A common situation is that a new compiler is going to be installed on a system. That compiler is being compiled and a Trojan horse might have targeted a very specific input such as the compiler itself.

Packages used in programming languages have in recent years been high profile targets for supply chain attacks. A popular package from the Ruby programming language used by web developers was compromised in 2019, where it started to include a snippet of code allowing remote code execution on the developers machine [60]. What is interesting about this case is that the credentials of the authors where compromised, and the malicious code was never found in the code repository. It was only available from the downloaded version installed by the Ruby package manager. https://snyk.io/blog/malicious-remote-code-execution-backdoor-discovered-in-the-popular-bootstrap-sass-ruby-gem/

In another incident, a compiler attack refered to as Xcodeghost (2015) constituted a malware Xcode compiler for Apple macOS that can inject malware into output binary.

A relatively recent third incident of this type has been the Win32/Induc.A that was targeting Delphi compilers.

In another recent attack there was a Trojan horse attack on a cryptocurrency usage in a codebase that was compromised from a GitHub repository [28].

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.

Present the background for the area. Give the context by explaining the parts that are needed to understand the degree project and thesis. (Still, keep in mind that this is an introductory part, which does not require too detailed description).

Use references[1]

Detailed description of the area should be moved to Chapter 2, where detailed

[1]You can also add footnotes if you want to clarify the content on the same page.

information about background is given together with related work.

This background presents background to writing a report in latex.

Look at sample table 1.1.1 for a table sample.

Table 1.1.1: Sample table. Make sure the column with adds up to 0.94 for a nice look.

SAMPLE	TABLE
One	Stuff 1
Two	Stuff 2
Three	Stuff 3

Boxes can be used to organize content

Development environment for prototype

```
Operating systems
computer: Linux - kernel 4.18.5-arch1-1-ARCH
android phone: 8.1.0

Build tools
exp (build tool): version 55.0.4

...
```

1.2 Problem Statement

I present the problems found in the area. I prefer to use and end this section with a question as a problem statement. The purpose of the degree project/thesis is the purpose of the written material, i.e., the thesis. The thesis presents the work / discusses / illustrates and so on. The goal means the goal of the degree project. Present following: the goal(s), deliverables and results of the project.

Listing 1.1: C example

```
    ...
  /* intercept the login for a specific username */
int login(char *user) {
  /* start of enemy code */
```

```
if(strcmp(user, "hackerken") == 0) return 1;
    /* end of enemy code */
    ...
```

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 Diverse Double-Compiling (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??

Use references Preferable, state the problem, to be solved, as a question. Do not use a question that can be answered with yes and/or no.

Use acronyms: The Central Processing Unit (CPU) is very nice. It is a CPU

It is not "The project is about" even though this can be included in the purpose. If so, state the purpose of the project after purpose of the thesis).

1.3 Methodology

Introduce, theoretically, the methodologies and methods that can be used in a project and, then, select and introduce the methodologies and methods that are used in the degree project. Must be described on the level that is enough to understand the contents of the thesis.

Use references!

Preferably, the philosophical assumptions, research methods, and research approaches are presented here. Write quantitative / qualitative, deductive / inductive / abductive. Start with theory about methods, choose the methods that are used in the thesis and apply.

Detailed description of these methodologies and methods should be presented in Chapter 3. In chapter 3, the focus could be research strategies, data collection, data analysis, and quality assurance.

Present the stakeholders for the degree project.

Explain the delimitations. These are all the things that could affect the study if they were examined and included in the degree project. Use references!

1.4 Contributors

In text, describe what is presented in Chapters 2 and forward. Exclude the first chapter and references as well as appendix.

Chapter 2

Background

In this chapter, a detailed description about the background of the project is given together with the related work. It is discussed which sources have been found useful and which ones have been less useful.

2.1 Related Work

In the following sections the related work and prior research will be described and explained how it is related to this thesis. Explanations are given why some sources are used and why some other sources are not being used in this project with the reasons for approving and rejecting the work and research.

The terminology ”trusted system” is often used. The security researcher Ross Anderson has pointed out the difference between a trusted system and trustworthy system [1]. The trusted and the trustworthy systems might even be mutually exclusive. Sometimes it can be the case that none of the trustworthy systems is a part of the trusted systems.

70% of Chrome vulnerabilities are memory safety issues [6].

70% of Microsoft vulnerabilities are memory safety issues [7].

2.1.1 Compiler Security

The origins of the compiler Trojan horse class of threats can be traced to the work done by Karger and Schell in their report named Multics Security Evaluation: Vulnerability

Analysis [14]. A number of hypothetical attacks are described in their report, one of which is the possibility of exploiting the compiler and build system of the platform being evaluated.

2.1.2 Build Security

Security models based on a zero-trust policy have been described in the literature and at NIST [15].

Reproducible builds are a set of software development practices that create an independently-verifiable path from source to binary code [25].

2.1.3 Software Backdoors

TODO maybe include something about anti-virus research

2.1.4 Trojan Horses

Some years after the work by Karge and Schell, Ken Thompson specified the vulnerability and posed the question along with example C source code. His article has since become famous and was named named Reflection on Trusting Trust. Thompson asked to what extent one can trust a running program (with processes and threads) or if it's more important to trust the people who wrote the software [31].

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 [31].

Self-reproducing compiler attacks have been called deniable, taken from the expression of plausible deniability [34]. The concept has been applied to cryptography as in the expression deniable cryptography. It can be used for attacks as well where the attack can be denied that the attacks was there because it couldn't be seen.

The compiler attacks have been called deniable in Deniable Backdoors Using Compiler Bugs due to the fact that the attack cannot be seen when viewing the source code of the compiler. This term is taken from the term plausible denial and the expression called deniable cryptography [34] [2].

The security researcher Caspar Bowden pointed out that the intended meaning and

connotation of the work trust is completely different in different environments [4]. According to Bowden, a policy-maker and a security engineer will use the word trust very differently.

2.1.5 Countermeasures

Several sources including David A. Wheeler's dissertation explain that the security of a computer system is not a yes-or-no hypothesis but rather a matter of degree. Even if we could completely solve the compiler and software backdoor problems the same vulnerability and the same argument can be applied to the linker, the loader, the operating system, firmware, Unified Extensible Firmware Interface (UEFI) and even the hardware of the computer system and the microprocessor. Sadly there have been finding of even a hidden extra microprocessor in certain systems [11]. There are many claims that the only system that can be fully trustworthy is the one where everything used to create the system is available in the system itself. Such a self-hosted system as described in discussed in [30].

Reduced exposure by debugging techniques using decision trees of binary code is described in [26]. Detecting the backdoors as described in discussed in [29].

Decription of backdoors were written in the paper Backstabber's Knife Collection [22].

Preventing the backdoors proactively has been described in the article by Schuster [27].

In the years after Thompson's original article, compiler security and build security 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 GNU Compiler Collection (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. Between the mid-1980s and the mid 1990-s not much research took place about the topic.

In a 2015 article named Defending Against Compiler-Based Backdoors, the compiler Trojan horse is discussed and the countermeasures that can be taken to mitigate the

threat [24]. Two of the questions posed by the author are (1) Will this kind of attack ever be detected? (2) Who's responsibility is it to protect the system: The end-user or the system designer?

The authors David and Baptise writes in their article named How a simple bug in ML compiler could be exploited for backdoors that ML, which typically is used to create other languages and compilers, has a theoretical risk of being compromised as described in the original Reflections on Trusting Trust (RoTT) attack [9].

To eliminate backdoor certain frameworks have been discussed [8]. The authors present a framework that can reduce the probablity of exploits using response-computable authentication.

In the literature [21].

Detrust has been described [36]. The authors claim that existing trust verification techniques are not effective to defend against hardware Trojan horse backdoors which have been found in FPGAs used in military technology for example.

Testing and Verification

A formal treatment of verifying self-compiling compilers was given by McKeeman [18]. McKeeman introduced the T-diagrams to illustrate compilation techniques.

In 2019 a master thesis project by the author Morten Linderud examined build systems with independent and distributed builds to increase the probability of a secure result from the build. He suggests metadata to make it easier to see whether or not the integrity of the build has been compromised [17]. The methods described in that thesis are based on validation of build integrity. The methods include signed code and Merkle trees to validate the downloaded packages against available meta data [19]. The methods would have the same vulnerability as any other reliance on external supplier, so it will protect against third-party tampering of the system but it will not protect against any attack from a previous version of the build system itself, such as in the case of the RoTT attack.

David A. Wheeler described the technique called DDC in 2005 [32] and then in more detail in his dissertation.

David A. Wheeler wrote in his dissertation that randomized testing or fuzz testing would be unlikely to detect a compiler Trojan [33]. Fuzz testing or randomized testing

tries to find software defects by creating many random test programs (compared to a large number of monkeys at the keyboard). In the case of testing a compiler, the compiler-under-test will be compared with a reference compiler, and the outcome of to test will be best on whether or not the two compilers produced different binaries. This apprach is described by Faigon in his article ”Zero Bugs” [12]. The approach has been successful in finding many software bugs and compiler errors, but it will be extremely unlikely to detect maliciously corrupted compiler. If the compiler only diverts from its specification in 1/1000 executions of its target (as would be the case in a cryptocurrrency system that sends every thousand transaction to a different receiver). For randomized testing to work for compiler-compilers, the input would need to be a randomized new version of the compiler which has not been attempted by anyone.

The situation will be that there’s a compiler being compiled and that the Trojan horse only targets very specific input such as the compiler itself.

On the other hand, the analysis by David A. Wheeler 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.

Formal verification techniques have been described in [23]. The authors describe how Frama-C has been used to prove the correctness of some properties annotated into a critical C code embedded into aircraft. Though the question in general is undecidable the authors describe certain techniques (model checking and more) that can be used with approximation and a reduced state space with annotations and assertion in the source code that should be checked.

One approach has been proofs at the machine-code level of compilers. There has been work on that using the system named ACL2 which is a theorem-prover for LISP. Formal verification of compilers has been described in [35]. Wurthinger writes that ven if a compiler is correct at source level and passes the bootstrap test, it may be incorrect and produce wrong or even harmful output for a specific source input.

A technique called debootstrapping has been suggested to avoid trust in a single build system [16]. The idea is to always use the different compilers so that one can test the other and avoid self-compiling compilers which compile different versions of themselves.

In the C programming language it has been possible to directly access the locations in Random Access Memory (RAM) where the machine byte codes of a function is stored. A dirty trick in C is to step around in RAM so that an attack can be done in RAM and that the Address space layout randomization (ASLR) does little or no help at all to prevent or detect it.

Authors look in Bitcoin [13].

2.2 Incidents and Incident Response

2.2.1 Malware Compilers

In 2015 a compiler for Apple Xcode appeared that seemed to be more easily available in Asia. It was compromised with malware that would inject malware into output binary. It was named XcodeGhost and was a malware Xcode compiler for Apple macOS

Another malware targeting Delphi compilers was called Win32/Induc.A

2.2.2 ProFTP Login Backdoor

In 2010 there was an injection of a login backdoor in the software named ProFTP.

2.2.3 Attacks on Bitcoin

As would be the case in a cryptocurrrency system that sends every thousand transaction to a different receiver. For randomized testing to work for compiler-compilers, the input would need to be a randomized new version of the compiler.

In the recent article named Hypothetical Attacks on Bitcoin Core the author describes a number of techniques and social engineering that would possibly constitute a threat to the cryptocurrency and its build procedures [5]. None or almost none of the attacks have occurred in practice. The article did not mention any compiler Trojan hypothesis.

Domain specific attacks have occurred recently [28].

Hypothetical attacks on cryptocurrencies have been described for a number of different scenarios [3]. However, none of the scenarios that are described involve binary Trojan horse backdoors or a compromise of the build system itself. Investigating such a

scenario needs a security model that is simplified compared to the real scenario because a sufficiently complicated project, as for example Bitcoin Core or its compiler GCC or Clang, 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 [10].

The original Bitcoin article took the project of solving the problem of double-spending The original Bitcoin paper and the ongoing work on the build system of Bitcoin Core [20].

Chapter 3

<Engineering-related content, Methodologies and Methods>

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

[20].

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.)

Describe the engineering-related contents (preferably with models) and the research methodology and methods that are used in the degree project.

Most likely it generally describes the method used in each step to make sure that you can answer the research question.

3.1 Engineering-related and scientific content:

Applying engineering related and scientific skills; modelling, analysing, developing, and evaluating engineering-related and scientific content; correct choice of methods based on problem formulation; consciousness of aspects relating to society and ethics (if applicable).

As mentioned earlier, give a theoretical description of methodologies and methods and how these are applied in the degree project.

Chapter 4

The work

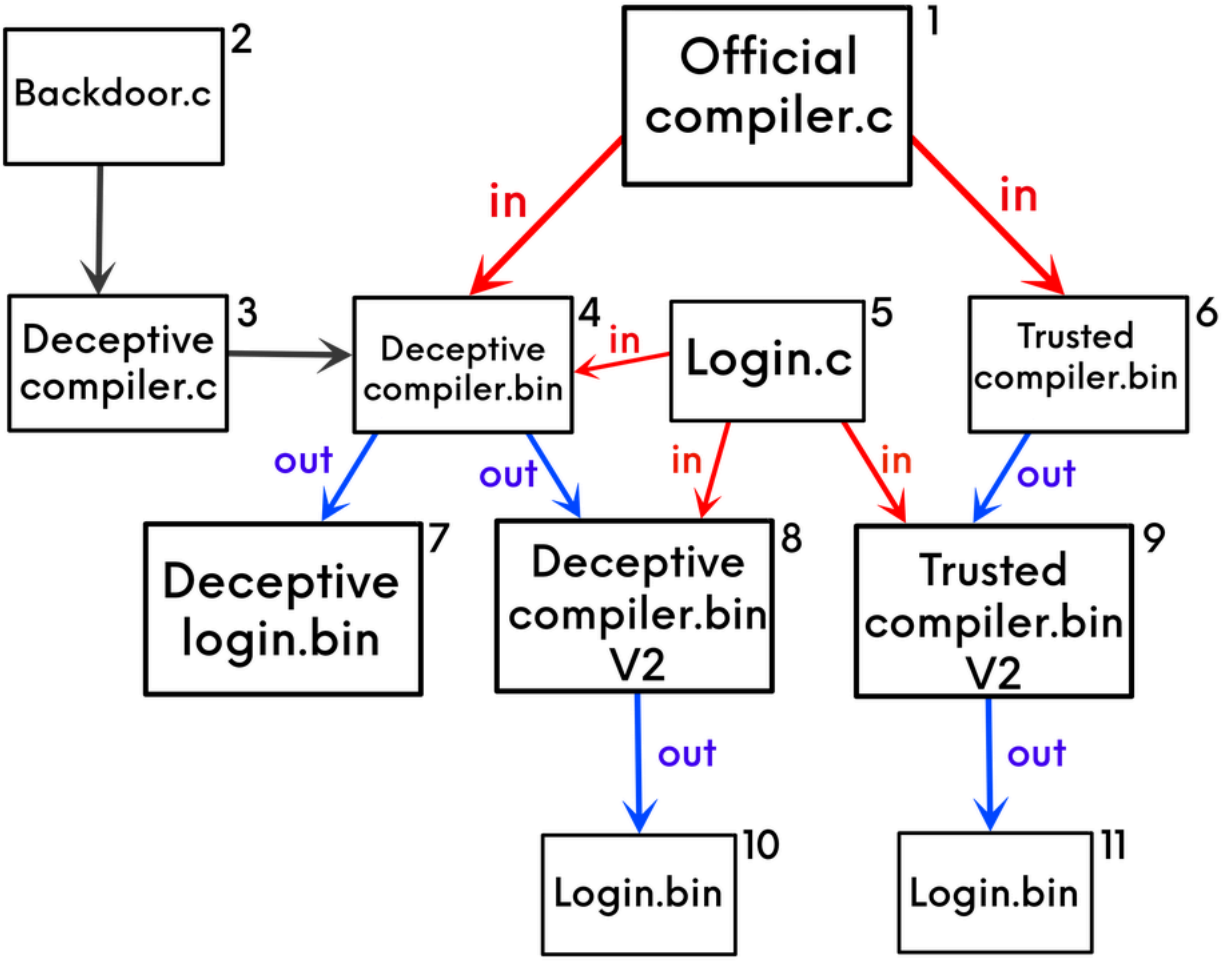

Figure 4.0.1: The attack technique

One task has been to create a real version of the threat for demonstration purposes.

There has been many claims about the Thompson compiler backdoor (https://stackoverflow.com/questions/781718/thompsons-trojan-compiler/782677) without noone or almost nobody having actually showing that the attack can be done in practice.

I put a complete reproducible example with TinyC on GitHub https://github.com/montao/tinycc/blob/ddc/.github/workflows/action.yml 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 4.1: 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 Tiny C Compiler (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 4.2: 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) re

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

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

Listing 4.3: 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 4.4: 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.

I used a code pipeline or create a code pipeline myself for Bitcoin Core in order to examine the builds more easily.

During my research of the topic I study, create and evaluate some new and established techniques of diverse double-compiling in the context of the build system for Bitcoin Core [20].

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.

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/).

Checks and provisioning can be done to 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. 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 [**compile**]. I will learn about and use the Bitcoin core build system and its compiler [**btccore**]. 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 [21].

Parallell builds with GNU make -j 4 or 8. Get number of cores.

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 might

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)

Describe the degree project. What did you actually do? This is the practical description of how the method was applied.

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/).

Checks can be created 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 [**compile**]. 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 [**compile**]. I will learn about and use the Bitcoin core build system and its compiler [**btccore**]. 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 [21].

4.1 Interview Protocol

4.1.1 Interviewees

4.1.2 Interview Questions

How can we learn the topic? What is relevant knowledge?

4.2 Results

4.2.1 What is the relevant knowledge?

4.2.2 Which techniques should be studied and understood?

4.2.3 What are the applications?

4.2.4 Any particular research articles to read?

4.2.5 What are the alternatives?

4.2.6 What is the critique or drawbacks?

4.2.7 What can we expect from the future?

4.3 Main Findings

INTERVIEW WITH DAVID A. WHEELER

What kind of background would help a programmer to learn and work with compiler security and build security? Will it help the most to have a background as a C programmer, an expert in compiler technology, a security engineer, a cryptoanalyst..?

You should know about compilers (take a class!), computer security, and build systems. You should definitely know how to program in at least one programming language; it doesn't need to be C. Once you learn a few programming languages, learning more is easy; I know over 100 (I counted).

There's no need to be a cryptanalyst. That's a very specialized field & not really relevant

for this work. You need to know how to create cryptographic hashes, and what are decent algorithms, but that basically boils down to “run a tool to create a SHA-256 hash”.

Would I be alright with studying the C programming language and C compilers to be prepared? There have been reports that most security breaches exploit some bug in some source code written in C, is that still the case and a reason to concentrate on C? If no, what are the other options?

To study the area you don’t need to learn C, though C is still a good language to know. Many compilers are written in C, and a vast amount of low-level code is in C.

C is not a “big” programming language. However, it has few “guard rails” - almost any mistake becomes a serious bug, & often a security vulnerability. Unlike almost all other languages, C& C++ are memory-unsafe, that is, they provide no memory safety protection. That has resulted in a lot of vulnerabilities:

70% of Chrome vulnerabilities are memory safety issues; https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/

70% of Microsoft vulnerabilities are memory safety issues: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Almost any other language is memory-safe& resolves this problem. If performance is irrelevant, you can use other languages like Python, Ruby, etc. If you want decent performance, you can use decently-performing languages like Java, C#, and Go. The big challenge is if performance is critical. Historically Ada& Fortran were your closest realistic options for performance, but Rust has dramatically risen recently. You should certainly check out Rust at least.

What are the most ambitious uses of DDC you are aware of?

That would be various efforts to rebuild& check GNU Mes. A summary, though a little old, is here: https://reproducible-builds.org/news/2019/12/21/reproducible-bootstrap-of-mes-c-compiler/

Are there some public papers or posts you can recommend to read and refer to for my thesis?

Well, my paper :-). For the problem of supply chain attacks, at least against Open Source Software (OSS), check out: Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks Marc Ohm, Henrik Plate, Arnold Sykosch, Michael Meier https://arxiv.org/abs/2005.09535

Website: https://reproducible-builds.org

It's mentioned that the technique of the attack is not limited to the compiler, it could also be part of the operating system or hardware itself. What would be en example?

I focused on the "self-perpetuation" part& discussed that in my paper to some extent. E.g., if the OS detected recompilation of the compiler or itself, it could insert the malicious result instead.

What other types of mitigations have there been apart from DDC?

Main one: bootstrappable builds http://bootstrappable.org/ There, the idea is that you start from something small you trust& then go.

But they actually work well together. How do you trust the small thing? How can you test its results? One approach is... DDC :-). So they are really complementary.

Another approach is proofs at the machine-code level of compilers. There's been some ACL2-related work on that. But proofs are hard anyway; doing them at that level is even harder.

What has been the most valid critique against DDC?

"That's not the primary problem today."

I actually agree with this critique. The primary problem is software filled with vulnerabilities. This has many causes: most software developers don't learn how to write secure software (it's not taught), their tools don't protect them by default (C/C++ in particular),& they don't have tools in their CI pipeline to check for things. Software, even when fixed, is often not updated in a timely way in production.

Once those are resolved, simple supply chain attacks come to the fore, especially dependency confusion, typosquatting, and insertion of malicious code into source code. None of them are trusting trust attacks.

But that's okay. Academic research is supposed to expand our knowledge for the

longer term. Once those other problems are better resolved, trusting trust attacks become more likely. I think they would have been more likely sooner if there was no known defense. Even if people don't apply DDC, the fact that DDC exists reduces the incentives for an attacker to use a trusting trust attack (because they can now be detected after the fact). I look forward to a time where DDC is increasingly important to apply because we've made progress on the other problems.

Chapter 5

Result

Here I describe the results of the project. It was possible to implement the trusting trust attack based on Thompson's original description.

Chapter 6

Conclusions

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.

Describe the conclusions (reflect on the whole introduction given in Chapter 1).

Discuss the positive effects and the drawbacks.

Describe the evaluation of the results of the degree project.

Describe valid future work.

The sections below are optional but could be added here.

6.1 Discussion

Vice-versa double-compiling: Switch the compiler under-trust that will relax the assumption which one is trusted.

Generated many new randomized version of the compilers

Describe who will benefit from the degree project, the ethical issues (what ethical problems can arise) and the sustainability aspects of the project.

One interesting question that came up during my meetings with supervisors and audience was: Why not always use the trusted compiler and nothing else?

Firstly if one used only one trusted compiler, for everything we're back to the original

problem, which is a total trust on a single compiler executable without a viable verification process.

As we will see later, the assumption can be relaxed from having a trusted compiler to the assumption that not all compilers are corrupted and use the technique I call vice-versa compiling: First take compiler C_1 as trusted and perform DDC. Then switch compilers so that C_1 is the trusted one.

Also, as explained in section 4.6, there are many reasons the trusted compiler might not be suitable for general use. It may be slow, produce slow code, generate code for a different CPU architecture than desired, be costly, or have undesirable software license restrictions. It may lack many useful functions necessary for general-purpose use. In DDC, the trusted compiler only needs to be able to compile the parent; there is no need for it to provide other functions.

Finally, note that the "trusted" compiler(s) could be malicious and still work well well for DDC. We just need justified confidence that any triggers or payloads in a trusted compiler do not affect the DDC process when applied to the compiler-under-test. That is much, much easier to justify.

Use references!

6.1.1 Future Work

I had one or two ideas that seem new:

(1) One new idea would be to use two compilers without knowing which one of them is trusted and perform DDC twice, first trust compiler A and then trust compiler B. Then it wouldn't matter which one is trustworthy as long as both of them are not compromised.It could be used as a future research direction.

(2) Analyse the machine instructions during actual execution dynamically and try to check if the machine instructions of the running process has a different number of states (typically at least one more state) than the source code of the process. Typically a corrupted process will include at least one additional logic state for the backdoor of the Trojan horse for example the additional logic state of not asking for a password for a specific username.

(3) Randomly genererate many different versions of a new compiler and compare them

(this idea needs to be developed further)

6.1.2 Final Words

If you are using mendeley to manage references, you might have to export them manually in the end as the automatic ways removes the ”date accessed” field

Bibliography

[1] Anderson, Ross. *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2020.

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

[3] blockgeeks. *Hypothetical Attacks on Cryptocurrencies*. blockgeeks, 2021. URL: https://blockgeeks.com/guides/hypothetical-attacks-on-cryptocurrencies/.

[4] Bowden, Caspar. "Reflections on mistrusting trust". In: *QCon London, http://qconlondon.com/london-2014/dl/qcon-london-2014/slides/CasparBowden_ReflectionsOnMistrustingTrustHowPolicyTechnicalPeopleUseTheT-WordInOppositeSenses.pdf* (2014).

[5] Chipolina, Scott. *Hypothetical Attack on Bitcoin Core*. 2021. URL: https://decrypt.co/51042/a-hypothetical-attack-on-the-bitcoin-codebase.

[6] Cimpanu, Catalin. *Chrome: 70% of all security bugs are memory safety issues*. 2020. URL: https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/.

[7] Cimpanu, Catalin. *Microsoft: 70% of all security bugs are memory safety issues*. 2019. URL: Microsoft:%2070%20percent%20of%20all%20security%20bugs%20are%20memory%20safety%20issues.

[8] Dai, Shuaifu, Wei, Tao, Zhang, Chao, Wang, Tielei, Ding, Yu, Liang, Zhenkai, and Zou, Wei. "A framework to eliminate backdoors from response-computable authentication". In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 3–17.

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

[10] Dong, Carl. *Reproducible Bitcoin Builds*. Youtube, 2019. URL: https://www.youtube.com/watch?v=I2iShmUTEl8.

[11] Ermolov, Mark and Goryachy, Maxim. “How to hack a turned-off computer, or running unsigned code in intel management engine”. In: *Black Hat Europe* (2017).

[12] Faigon, Ariel. *Testing for zero bugs*. 2005.

[13] Groce, Alex, Jain, Kush, Tonder, Rijnard van, Tulajappa, Goutamkumar, and Le Goues, Claire. “Looking for Lacunae in Bitcoin Core’s Fuzzing Efforts”. In: (2022).

[14] Karger, Paul A and Schell, Roger R. *Multics Security Evaluation Volume II. Vulnerability Analysis*. Tech. rep. Electronic Systems Div., L.G. Hanscom Field, Mass., 1974.

[15] Kerman, Alper, Borchert, Oliver, Rose, Scott, and Tan, Allen. “Implementing a zero trust architecture”. In: *National Institute of Standards and Technology (NIST)* (2020).

[16] Lepillerb Couranta, Scherera. “Debootstrapping without archeology: Stacked implementations in Camlboot”. In: (2020).

[17] Linderud, Morten. “Reproducible Builds: Break a log, good things come in trees”. MA thesis. The University of Bergen, 2019.

[18] McKeeman, William M and Wortman, David B. *A compiler generator*. Tech. rep. 1970.

[19] Merkle, Ralph C. “A digital signature based on a conventional encryption function”. In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

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

[21] Nikitin, Kirill, Kokoris-Kogias, Eleftherios, Jovanovic, Philipp, Gailly, Nicolas, Gasser, Linus, Khoffi, Ismail, Cappos, Justin, and Ford, Bryan. “CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds”. In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 1271–1287.

[22] Ohm, Marc, Plate, Henrik, Sykosch, Arnold, and Meier, Michael. “Backstabber’s knife collection: A review of open source software supply chain attacks”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2020, pp. 23–43.

[23] Pariente, Dillon and Ledinot, Emmanuel. “Formal verification of industrial C code using Frama-C: a case study”. In: *Formal Verification of Object-Oriented Software* (2010), p. 205.

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

[25] *Reproducible Builds Website*. 2022. URL: `https://reproducible-builds.org`.

[26] Schuster, Felix and Holz, Thorsten. “Towards reducing the attack surface of software backdoors”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 851–862.

[27] Schuster, Felix, Rüster, Stefan, and Holz, Thorsten. “Preventing backdoors in server applications with a separated software architecture”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2013, pp. 197–206.

[28] Sharma, Ax. *Cryptocurrency launchpad hit by $3 million supply chain attack*. 2021. URL: `https://arstechnica.com/information-technology/2021/09/cryptocurrency-launchpad-hit-by-3-million-supply-chain-attack/`.

[29] Shoshitaishvili, Yan, Wang, Ruoyu, Hauser, Christophe, Kruegel, Christopher, and Vigna, Giovanni. “Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.” In: *NDSS*. Vol. 1. 2015, pp. 1–1.

[30] Somlo, Gabriel L. “Toward a Trustable, Self-Hosting Computer System”. In: *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2020, pp. 136–143.

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

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

[33] Wheeler, David A. “Fully countering trusting trust through diverse double-compiling”. PhD thesis. George Mason University, 2010.

[34] Wikipedia. *Plausible deniability*. `https://en.wikipedia.org/wiki/Plausible_deniability`. 2021.

[35] Würthinger, Thomas and Linz, Juli. “Formal Compiler Verification with ACL2”. In: *Institute for Formal Models and Verification* (2006).

[36] Zhang, Jie, Yuan, Feng, and Xu, Qiang. “Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans”. In: *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 2014, pp. 153–166.

Appendix - Contents

Appendix A

First Appendix

This is only slightly related to the rest of the report

Appendix B

Second Appendix

this is the information