Gogojuice: Reflections on Trusting Trust

An analysis of compiler-based attacks

Rachit Singh
Harvard College
rachitsingh@college.harvard.edu

Jeffrey Ling
Harvard College
jling@college.harvard.edu

Abstract

We analyzed the Go compiler's susceptability to a compiler-level attack based on Ken Thompson's Reflections on Trusting Trust paper, which is especially relevant since Go 1.5 (released in August 2015) is capable of bootstrapping itself, which is the start of the vulnerability. We found that the Go compiler takes reasonable (perhaps unintentional) precautions against the modification of source code for compilation. However, we created a compiler, Gogojuice, which when used to bootstrap the Go 1.5 source code successfully replicates itself and injects malicious code into target programs. We also studied possible attacks using the compromised compiler and methods to detect such an attack.

1. Introduction

In 1984, Ken Thompson wrote a reflection on the inherent untrustability of software by using the example of the compiler [1]. In the article, titled "Reflections on Trusting Trust", Thompson describes a compiler that's been modified to not only inject harmful code when compiling arbitrary programs, but to embed code when compiling a compiler so that its behavior is self replicating. Thus, even though a compiler binary can be claimed to be "trusted" by releasing its source code, its behavior can be obfuscated by modifying the source code such that simply recompiling the compiler with itself will not fix its malicious behavior.

Thompson's original paper, heavily outdated by our technology standards, nonetheless contains an interesting and important security idea, and in our project, we apply Thompson's trusted compiler attack to the existing Go language compiler.

CS263 '15, November 30, 2015, Cambridge, MA, USA.

ACM [to be supplied].
http://dx.doi.org/10.1145/1234567.1234567

1.1 Related Work

As far as we know, there is no well-documented implementation of Thompson's hack in modern day code. One urban myth on Quora describes how a bug was discovered in a psychology graduate student's code that caused foul language to appear, and was painstakingly found to be a problem in the compiler [3]. The only other instances of this hack we could find are attributed to a virus that infects Delphi files on Windows machines, which is fairly limited in scope (and old by our standards) [5] [6].

1.2 Goals

These previous attacks are interesting, but they are not well documented and do not contribute to our understanding of the hack. The goals of our project are:

1. to implement the hack on an existing compiler (we chose the open source Go language),
2. to understand the attack possibilities of the hack in a modern security setting,
3. to understand defenses against the hack and what it would entail for modern trust systems.

2. The Go Language

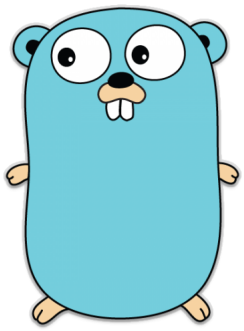

Figure 1. The Go language, developed in 2009, is open source and gaining traction in a variety of use cases.

Go (alternatively, Golang) is a relatively new language developed at Google in 2009 [8]. It is intended for use in webserver, systems programming, and heavily concurrent programs, and is expanding its reach to a variety of applications including Android programming. Most importantly, it is statically typed and a compiled language. Since Go isn't part of the GCC collection, the language is distributed via the internet, making it a viable target for this trusted compiler attack. We chose Go as the target for our attack for a number of reasons:

1. Go is used heavily in production web servers at companies like Google, Dropbox, and Facebook,
2. Go is the language used by Docker, an open source container package that is being widely adopted by the software industry [9],
3. Go has a simple, terse syntax, which allows for easy implementation of the attack,
4. Many programs rely heavily on the standard library, which comes with common functionality like JSON parsing, cryptographic functions, and a web server.

Additionally, we find it ironic that Ken Thompson is one of the main authors and maintainers of Go.

2.1 `gc`: the Go compiler and its toolchain

Go is unique as a language for a few reasons. First, Go is known for is incredibly fast compilation speeds, simple syntax, and powerful standard library. Until Go 1.5, the main compiler for Go was written in C, called `gccgo`. Starting with version 1.5, Go is fully bootstrapped - i.e., the Go compiler is written in Go. One useful feature of Go is cross-compilation, or compilation of Go programs for other platforms, so that the compiler is supported on many flavors of operating system from Windows to Linux.

The compiler toolchain follows the following steps:

1. First, `cmd/dist` is compiled, which is the tool that detects the host operating system and architecture.
2. It then compiles `go_bootstrap`, an intermediate version of the `go` tool, along with some supporting libraries.
3. `go_bootstrap` then compiles the Go standard library and the `go` tool (possibly for a foreign architecture, if cross-compiling).
4. Then, tests are run against the standard library.

Diving deeper into the structure of the compilation steps, the Go compiler first does dependency management by reading only the upper part of the Go source files (Go requires the preamble to contain the imports). After this, the Go compiler itself reads files that don't have up to date object files, compiles them, and then stores the object files in the cache.

3. Gogojuice

3.1 Infection method

In our version of the Go compiler, which we have named Gogojuice, we implement Ken Thompson's compiler attack. The infection process has the following structure:

1. In the build process, we can change the source code we are compiling (by changing its representation in memory at compile time). In this way, we can modify arbitrary programs to do what we want (e.g., make `/bin/login` not require a password or weaken some cryptographic function).
2. To make the compiler have self-replicating behavior, we must have a *quine* in the source code. A quine is a program that prints its own source code, and in the context of our compiler, when the infected compiler encounters source code for another compiler, it injects its own source code so that it ends up building another copy of itself (instead of a clean compiler, which would occur normally).

By combining the quine behavior of the compiler along with any malicious injections, we will have a working infected compiler that can subvert quick checks.

3.2 Changing the build process

There are a few places where we can modify the compiler for our purposes. We could, for example, change the linking step so that a special library is linked. However, we must modify code on the path of execution. It is far easier to do this at the source code step (i.e. modify the source code before it is turned into an AST/object code) instead of attempting to do so after compilation.

We first believed that we could modify the source when it is loaded into memory right before the parsing step. This would be ideal because then the compiler hack would be undetectable, since the source lies only in memory. However, we found that Go compilation doesn't ever load the entire source into a single memory location - instead, it passes around a file pointer and an auxiliary data structure that stores the position of line / code block offsets (this is likely due to memory constraint reasons). These are used to read from the file inside the actual parser, which reads a few bytes at a time.

So, in order to introduce injections into target source code, we move the target files on disk during compilation to the `/tmp/` directory and move them back after compilation, replacing them with our own version of our target files (i.e. if the original file is `target.go`, we move it to `/tmp/target.go` and write our own file `target_infected.go` as a replacement). To implement this, we found that in `cmd/go/build.go` in the `runBuild` method, we can examine the target file to see if there are any injections we want to perform, and do the file movement before the source code enters the compilation procedure. After the command

returns and compiling is complete, we move the file back from the /tmp/ location.

Outside of basic source code injections, our compiler infection process is as follows: suppose our compiler, gogojuice, has an infected version of build_infected.go (this handles the list of files that are to be compiled). Then, when we compile the compiler using gc, the normal Go 1.5 compiler, we get a binary of gogojuice. If we ever use this compiler to compile the source of gc, it must replace build.go with build_infected.go, thus creating a binary for gogojuice instead of gc. Thus, build_infected.go must contain a quine that will embed its own source in the compilation procedure for gc in order to create another copy of gogojuice as output.

Luckily, Go has a good mechanism for constructing quines. Namely, we can use the format string function fmt.Sprintf and call it with %#v (see figure 2) [7]. By simply applying this quine technique to the step where we override the build.go file with our custom build_infected.go file, we are able to propagate the compiler's source code in future builds of the compiler.

Note that we're able to attack the standard library because Go bootstraps into go_bootstrap before compiling the standard library using the newly infected bootstrap tool, so all modifications made to the compile process of standard libraries will exist in the first version of the compiled compiler.

Our attack scheme does have limitations. Since we have to move the original target files to /tmp/, this may not be possible if the files are read only. However, assuming that the person using the compiler is working in a general development environment, this seems unlikely to be the case, so our compiler attack should hold in a variety of situations.

3.3 Evading tests

The Go compiler, upon being built, runs several tests to ensure that its various library functions are working correctly. This makes it difficult to implement hacks without immediately discovering that something is wrong with the compiler. We believe that upon compilation, the tests should at least pass first scrutiny so that the compiler will be used to infect several pieces of code before being discovered as malicious.

There are several workarounds to this problem that we considered:

1. Hardcoding test values by checking the input to the library method. If they match the test cases given in the open source tests, then we conclude that
2. Hacking the testing process itself for our libraries so that they automatically show a pass.

While the second method seems more robust to replication over different releases of the compiler, we didn't have time to fully work out the internals of go test, so resorted to hardcoding values for the specific library functions that we targeted.

4. Compiler-based attacks

Now that we have established a mode of infecting the compiler, we consider the possible

4.1 Attack vectors

One of the key considerations in a compiler-based attack is that the attacker often distributes the hack well before the target code is even written, let alone compiled/distributed. So, attacks must be targeted towards either widely used libraries whose source is relatively stable, or towards the standard library. Wile it would be ideal to simply corrupt a login function written by the end developer, in our model the attack must focus on libraries, since those have open code that is widely used.

In gogojuice we made the following modifications to the standard library:

1. The cryptographic Seed function always seeds the value 2734968535002352417 instead of the inputted value.
2. In crypto/sha256 we modified the checkSum function that returns the random hash to be a constant hash value, which we set as 0xffff ffff ffff ffff ffff ffff ffff ffff.
3. As a hard-coded attack vector (outside of the standard library), we injected source into Docker, the popular open source container framework that exports authentication data.

4.1.1 Seed function

Seed functions are critical to ensuring random behavior in cryptographic functions, and if they are fixed, the cryptographic integrity of many functions is lost. Because seeding is difficult to detect (i.e. the behavior of random functions will still appear random), this attack will likely go under the radar and allow for the subversion of complex systems that rely on proper seeding.

4.1.2 SHA256

Our attack in this area was motivated by the fact that many backend services store user passwords in hashed, salted format using the pbkdf2 function, which in Go can use any of 5 FIPS approved hash functions - SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512. There were two major concerns along this attack vector:

1. If hashes are also calculated in another area of application code and independently verified (i.e., the hash of the password is calculated client side and then checked against the backend), then valid Go code will fail. This raises suspicions and could cause the developer to realize that their compiler is untrustworthy.

```
package main

import "fmt"

func main() {
    s := "package main\n\nimport \"fmt\"\n\nfunc main() {\n\ts := %#v\n\tfmt.Printf(s, s)\n}\n"
    fmt.Printf(s, s)
}
```

Figure 2. A quine in Go.

2. The standard library contains a number of tests that verify the correctness of the library code.

The first issue is severe, since it potentially compromises the entire attack. We found that as best practice, hashing is very rarely done on the client side [4]. This is generally more secure because hashing on the client side allows the client machine access to the database hash value (which requires a salt to be generated on the server side), which is more valuable to the attacker as a direct password to enable access to the system.

The second issue is more easily dealt with. It is possible to modify the weakened hash function to just return the correct value for the test inputs, so the infected code will still pass all of the tests. However, it is much easier just to modify the code to remove the tests entirely during the execution of the test running tool. We were not able to implement a hack for the testing procedure, but hardcoding the test cases turned out to work.

4.1.3 Docker

Figure 3. Docker is an open source program that implements containers for software development.

Docker [9] is an open source program written in Go that implements containers for software development code, and is meant as a more lightweight and portable alternative to virtual machines.

While we didn't have enough time to explore the entire authentication mechanism of Docker, which has many components and a robust security scheme, we found a line in the Docker command line client source which loads authentication keys and uses them to execute commands against the Docker daemon. We thus send the keys to a remote server by injecting the lines in figure 4.

5. Methods for countering attacks

5.1 Double compiling

We consider a defense devised by Wheeler [2]. Suppose we have access to the executable of a trusted compiler E_T, and we want to evaluate the legitimacy of the compiler with executable E_A and purported source S_A. We use the following process:

1. Compile the source S_A with E_A and E_T to generate executables X and Y respectively. Now assuming S_A is as claimed (and no extra code was injected), X and Y must have the same functionality (but perhaps different binaries).
2. Now we use executables X, Y to compile the source S_A to produce outputs V, W respectively. Since X, Y have the same functionality, V and W must be bitwise equivalent.

If V, W are not the same, then a diff will indicate that one of the original compilers E_A, E_T is untrusted. Its possible that theyre both infected (in different ways), but this means that were able to detect that the compiler attack has occurred.

There are a few assumptions made about this defense.

1. This defense assumes that the compiler environment is also trusted, so that a bitwise comparison of the compiler executable is sufficient (i.e., no evil files are being added at link time).
2. This assumes that there is no randomness in the compile process. If a compiler produces different outputs with the same source code, its impossible to accurately check the output of X, Y as stated above.

We tested this on Gogojuice and discovered indeed that the bitwise output (of V, W as described above) is different.

However, while this method allows us to determine that the compiler is infected, there is no way to tell from this which compiler is actually trusted. Thus, this defense implies that the only way to properly check the integrity of a

```
// authConfig is a nice JSON structure containing username, password, etc.
var data = fmt.Sprintf("%#v", authConfig)
resp, err := http.Get("http://attackserver.com?data=" + data)
```

Figure 4. Sending authentication keys to a remote server.

candidate compiler is to use one's own compiler that they fashioned from strictly trusted sources - a highly nontrivial feat in today's development environment.

5.2 Binary examination

Depending on the complexity of the compiler binary, it may be possible to use objdump or step through with gdb and see that no malicious behavior is going on. However, for most purposes this is likely to be infeasible, since compilers tend to be large, complicated, and difficult to reverse engineer at a low level.

One method that may allow for easy detection, depending on how the malicious compiler is implemented, is to examine the binary for traces of a quine. Depending on how obfuscated the source code is, it may be easy to spot the quine in the binary, since a large raw string containing source code is unlikely to be seen in a clean compiler. Especially for Gogojuice, we created a quine by minimally modifying the `build.go` file and dumping the source code as a (extremely long) string, which stands out quite obviously in the raw bytes of the binary.

5.3 Careful testing

Careful testing is a good defense against any compiler-based attacks. One problem we continually ran into when devising our attack was the presence of tests. For creating injections, we found it difficult to implement hacks for library functions and also pass the associated tests. For example, one of our original attacks was on the code in `crypto/rand` and sought to make the random number generator a predictable constant. However, the test that checked the random number generator used a compression method (`flate`) that ensured that the resulting random bytes were sufficiently random, which is impossible to guarantee through artificial methods.

While it's possible for the compiler to evade these checks directly (by hardcoding test solutions, as we did above), if these tests were changed or refreshed every once in a while, it would become clear that something was wrong with the library functions. If the source code remained unchanged, then it would be a valid conclusion that the compiler that built the library functions has some problem, and would invite further investigation using the previous defense methods described above.

6. Future Work

While we successfully implemented the compiler hack described in Thompson's paper, there is much more we can do to extend our work.

First, the methods we use to implement the quine are easily detectable from viewing the source code, so this is undesirable for detectability. If we refactored the Go compiler to make embedding the quine shorter, this would certainly be less detectable, but the work it takes to refactor a compiler is quite nontrivial.

Second, our method of evading tests is somewhat primitive and ought to be examined more carefully. There are almost certainly ways to fix the `go test` process that can hack the tests to pass automatically, but this may require reverse engineering of the testing process that was difficult given our time constraints.

Third, we can explore injections into a variety of other libraries. In particular, objdump is part of the Go standard library in `cmd/objdump`, so if someone wanted to defend against our attack using this method, an infected objdump would prevent anything meaningful in this direction. Hacking the filesystem or os libraries that Go uses could also be devastating, but perhaps difficult to do due to the low level details. Also, one interesting and fun hack (but perhaps not so devastating) is to hack the `fmt` library to print hardcoded strings and mess with all sorts of logging systems.

Finally, there are many more ways to implement this attack that can be explored. While we modified the source code of the compiler directly, it may also be feasible to infect the linking process to add our own evil libraries, though these methods would require more avenues of exploration.

7. Conclusion

In this project, we explored the possibilities of writing an infected compiler that can continue to infect other compilers. We made a small change to the open source Go program that enables injection of malicious code into several open source libraries, including cryptographic functions and Docker, and also injects malicious code while compiling a compiler so that the resulting compiler has the same behavior. The code from our project Gogojuice is open sourced and can be found online.[1]

The potential implications for our project are twofold. First, it shows that there is an easy way to distribute malicious code for a variety of programs by attacking a low level target such as the compiler. If such a compiler made its way to platforms such as Android development, there could be nasty consequences for all programs created in that poisonous environment.

[1] https://github.com/rachtsingh/gogojuice

Second, our project emphasizes Ken Thompson's original point on trusting trust. In the Internet age, when essentially everything comes from a third party over the web (which we have learned cannot be trusted due to the workings of the NSA and related government activity), we are forced to accept that no matter how many security protocols that are developed, if a critical part of the code environment, such as the compiler, becomes compromised, no fancy cryptographic technique or secure hashes can be relied on anymore. This can be seen clearly in Gogojuice, where the basic cryptographic functions themselves are messed up, thus invalidating complex schemes such as RSA or TLS.

Acknowledgments

We want to thank Prof. Mickens for teaching us not to trust anything that comes over the Internet and inspiring our project, Ken Thompson for devising this compiler attack, and Google for developing Go as a clean and easy language to modify.

References

[1] Thompson, Ken. "Reflections on trusting trust." Communications of the ACM 27.8 (1984): 761-763.

[2] Wheeler, David. "Countering trusting trust through diverse double-compiling." Computer Security Applications Conference, 21st Annual. IEEE, 2005.

[3] https://www.quora.com/What-is-a-coders-worst-nightmare/answer/Mick-Stute

[4] http://security.stackexchange.com/questions/8596/https-security-should-password-be-hashed-server-side-or-client-side

[5] https://nakedsecurity.sophos.com/2009/08/18/compileavirus/

[6] https://www.symantec.com/security_response/writeup.jsp?docid=2009-081816-3934-99&tabid=2

[7] https://gist.github.com/ijt/4029658

[8] https://golang.org/

[9] https://www.docker.com/