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 allowing modification of source code during 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...

2. The Go Language

Go (alternatively, Golang) is a relatively new language developed at Google in 2009. It is intended for use in webserver, systems programming, and heavily concurrent programs. Most importantly, it is statically typed and a compiled language. Since it isn't part of the GCC collection, the language is distributed via the internet. We chose Go as the target for our attack for a number of reasons: first, it is used heavily in production web servers at companies like Google, Dropbox, and Facebook. It is also the language used by Docker, an open source container package that is being widely adopted by the software industry. Go also has a simple, terse syntax, and many programs rely heavily on the standard library, which comes with common functionality like JSON parsing, cryptographic functions, and a web server. Additionally, we found 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 bootstraped - 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.

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 and then compiles `go_bootstrap`, an intermediate version of the go tool, along with some supporting libraries
2. `go_bootstrap` then compiles the Go standard library and the go tool (possibly for a foreign architecture, if cross-compiling).
3. 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.

2.2 Gogojuice

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) and difficult after compilation.

We first believed that we could modify the source when it is loaded into memory and before the parsing step. This would be ideal because then the compiler would be undetectable. 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. These are used to read from the file inside the actual parser, which reads a few bytes at a time.

So, we decided to modify the source code by moving the files on disk during compilation and moving them backwards after compilation. Specifically, we found that in the `cmd/go/build.go` in the `runBuild` method we can do the required detection and file movement before compilation. After the command returns, we move the file backwards.

A proper framework is necessary for understanding the infection process: suppose our compiler, `gogojuice`, has an infected version of `build_infected.go`. 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*, or a program that outputs its own source. Luckily, Go has a good mechanism for constructing quines. Namely, we can use the format string function `fmt.Sprintf` and call it in the following way:

Note that we're able to attack the standard library because Go bootstraps into `go_bootstrap` before compiling the standard library using the (new, infected) bootstrap tool.

3. Compiler-based attacks

3.1 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, the attack must focus on libraries, since those have open code that is reused.

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 cryptographic hash function to output a masked and XORed version of SHA1 instead of SHA256. SHA1 is well known to be breakable, but appears to be as pseudorandom as SHA256
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.

3.1.1 Seed function

3.1.2 SHA256

Our attack in this area was motivated by the fact that many 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.
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. However, we found in a survey of open source full stack software written in many languages, like Python, Go, and PHP, that hashing is very rarely done on the client side. Whether this is due to convenience or security... 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.

3.1.3 Docker

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.

3.2 Methods for countering attacks

Acknowledgments

Acknowledgments, if needed.

References

[] P. Q. Smith, and X. Y. Jones. ...reference text...