Memory Safety and Beyond: Unveiling the Missing Piece in Golang

Author: Anu Mathew

Abstract

Memory-safe languages are popular in modern industry, as they significantly reduce the attack surface of applications. These languages handle memory allocations and deallocations as runtime demands. Golang (Go), a popular memory-safe language among tech giants, powers the modern cloud industry. This study examines Go’s default HTTP implementation while undergoing certain Denial of Service (DoS) attacks. It will highlight the careful considerations necessary to write resilient endpoints in Go. Furthermore, the study will analyze DoS attacks and explore strategies to detect such attacks where Go is employed.

**1. Introduction**

Popular languages such as C and C++ offer extensive control and adaptability in memory allocations, but significantly burden the developer to manage memory references correctly. Even minor errors can cause spatial and temporal memory vulnerabilities that can be exploited (National Security Agency, 2023). Buffer overflow, a type of spatial memory vulnerability, happens when a program accesses memory outside the appropriate boundaries for an object in memory. Temporal memory safety issues arise when a program accesses memory outside its intended time or state, such as accessing object data after freeing the object or mixing memory accesses unexpectedly. Buffer overflow first appeared in the Morris Worm of 1988 (Jajoo, 2021) and continues to plague the software industry today.

Even though developers can conduct extensive testing to equip software logic to handle unforeseen situations, spatial and temporal memory safety vulnerabilities remain a significant challenge. Piling on even more code to fix and detect code memory safety has proven ineffective even after 40 years. Instead, the most effective strategy to minimize memory safety vulnerabilities is to address the fundamental gaping hole: the programming language (The White House, 2024).

Memory-safe programming languages handle memory allocation dynamically as needed during runtime and subsequently release it when it is no longer in use. They safeguard developers from making errors in memory management. While these languages may not always suit every scenario, using a memory-safe language can significantly improve software security, making it a highly effective strategy.

The Go programming language, also known as Golang, has recently gained significant attention in software development. Go was conceived at Google in September 2007 by Robert Griesmer, Rob Pike, and Ken Thompson (principal inventor of the B programming language and Unix operating system). Beyond a surface similarity to C, Go "borrows and adapts good ideas from many other languages while avoiding features that have led to complexity and unreliable code" (Kernighan & Donovan, 2015, p. xi). Go is garbage collected and memory safe. It uses Goroutines, lightweight green threads that allow massive concurrency (Tipirneni, 2022, p. 11). It has a thriving open-source ecosystem that helps rapid development and is suited for today's DevOps pace. Go has become a popular memory-safe programming language and powers the billion-dollar cloud industry (Cass, 2023).

**1.1. Exploring the Panacea**

Go received stellar endorsements from the NSA, the National Cyber Director (National Security Agency, 2023, p.3), and other organizations that recognize the importance of memory safety in programming languages for enhancing cybersecurity. Adopting a new programming language is not without costs, such as reduced initial efficiency and the need to navigate the learning curve (National Security Agency, 2023, p. 4). Still, these costs are relatively minor when weighed against the potential costs from risks associated with the absence of memory safety.

Go, with its gentler learning curve and straightforward endpoint creation (Kernighan & Donovan, 2015, p. 191), presents a compelling case. Go's standard library boasts that "batteries are included" (Kernighan & Donovan, 2015, p. xiv) as it provides clean building blocks for almost every type of application. The net/http package is part of Go's standard library, providing the tools and abstractions for creating HTTP servers and clients. As Appendix I shows, creating an HTTP endpoint using the standard library is relatively easy.

However, is this enough to create resilient endpoints? As Schneier (2023, p. 59) aptly puts it, "Good design today is bad design tomorrow, and hackers will always find ways to exploit bad design." He also notes, "Most software is written quickly and shoddily" (Schneier, 2023, p. 59), underscoring the importance of careful and thoughtful programming. We must ponder if there are other factors that need to be considered.

**2. Research Method**

**2.1. Timeout Research**

The study took place in controlled settings, utilizing a QEMU (QEMU, n.d.) VM equipped with a single CPU and 300MB of RAM running Debian Linux 12, bookworm (“Debian “bookworm” release information,” n.d.).

The tool 'slowhttptest' (Shekyan/slowhttptest, n.d.), a DoS attack simulator, was used to experiment with various test configurations. The experiments were conducted against a custom endpoint that supports multiple modes (Figure 2).

**2.1.1. Test Variables**

The endpoint could be configured into different experiment modes using command line flags, as depicted in Figure 2. The endpoint provides various URLs, and the tests were conducted against /load, which tries to be deterministic — the loop iterates 100000 times, executing the same Fibonacci calculation and value swaps — making it a relatively consistent operation in terms of time complexity.

For timeout research, a constrained virtual machine (VM) with 1 CPU and 300 MB RAM was used to illustrate the impact. Golang's extensive concurrency model can obscure the effects of a DoS attack on a more powerful VM with more significant CPU and memory resources. This is illustrated in Figure 4; a connection rate of just 15 in SlowLoris mode can overwhelm the constrained VM. In contrast, a more robust VM would need a significantly higher connection rate to experience similar distress. This factor was considered when sizing the experimental setup to observe the effects and carry out measurements.

**3. Findings**

**3.1. Part of the Problem: Zero Values**

Go allows a variable to be implicitly initialized during declaration. If no explicit initialization is provided, the variable will automatically be assigned the zero value, corresponding to its type. For numerical types, the zero value is 0; for pointers, it is nil; for strings, it is the empty string; and for structs, each of their fields is initialized with the zero value of its respective type (Kernighan & Donovan, 2015, p. 5, 102).

The net/http package is part of Go’s standard library, providing the tools and abstractions for creating HTTP servers and clients. ReadTimeout, WriteTimeout, IdleTimeout, and ReadHeaderTimeout (The Go Programming Language, 2024) are the crucial timeouts in the net/http struct that are pivotal defenses against common DoS (Denial of Service) attack vectors.

Therein lies a problem - if not explicitly set, these timeout values are autoinitialized with the default value of 0. This implies that the endpoint will wait indefinitely. As explored in subsequent sections, it would be relatively easy to overwhelm such an endpoint.

**3.1.1. Scope**

Although many different DoS attack modes exist, this study was limited to (1) SlowLoris and (2) Slow Read DoS attack modes. The concurrency model and standard library of Go can obscure the effects of DoS attacks. To zoom in and measure the effects, a resource-constrained QEMU VM was chosen to run the experiments. The attack rate was throttled to allow a demonstration of the key takeaway points.

**3.2. SlowLoris Attack**

The SlowLoris attack (Suroto, 2017, p. 3), a slow header attack, sends unfinished HTTP requests against the endpoint. The aim is to exhaust the server resources. This vector also uses keep-alive strategies to maintain the connections and occupy server resources.

**3.2.1. Experiment 1: Default Implementation Against SlowLoris**

The first experiment was run against the endpoint that simulated the default implementation of Go with 0s timeouts. This experiment involved slowhttptest making 23 connections per second (-r); it used -H mode and established 61,000 connections (-c) throughout the test. The remaining parameters, including the test duration of 240 seconds, were set to their default values.

**3.2.3. Summary: SlowLoris**

By introducing a three-second timeout to IdleTimeout, ReadHeaderTimeout, ReadTimeout, and WriteTimeout values, instead of an endless wait, the endpoint’s resilience and robustness were significantly enhanced. It successfully resisted the SlowLoris attack and remained available to handle requests.

**3.3. Slow Read Mode**

Slow Read mode tries to open many connections and send valid HTTP headers (Suroto, 2017, p. 3). The responses are, however, read extremely slowly to deplete the server of processing resources further. It can, optionally, send large requests to keep the connection even longer.

**3.4. Summary of Experimental Findings**

The default HTTP implementation is susceptible to SlowLoris and Slow Read Mode DoS attacks and can be compromised easily. The outcome of these experiments indicates that implementing an appropriate timeout can significantly fortify the endpoint against these DoS attack methods.

**3.5. Shift-Left Approaches**

Adopting a Shift-Left (Phan et al., n.d.) strategy involves integrating security checks into the earliest stages of development. This section compares various static analyzers that handle the vulnerable implementation of the HTTP endpoint, as shown in Appendix I.

**3.5.1. golangci-lint**

golangci-lint (Golangci/golangci-lint, 2018) is a static analyzer that examines Go source code to detect errors and potential vulnerabilities

**3.5.2. semgrep**

semgrep ("Semgrep/semgrep" n.d.) is another static analyzer that examines Go source code, amongst other languages, to detect errors and potential vulnerabilities.

**4. Recommendations and Implications**

**4.1 Best Practice**

When crafting endpoints in Go, reasonable timeouts must be explicitly defined. Structs that include timeouts must be initialized carefully. It is also crucial to avoid excessively high timeout settings, as this can be counterproductive and akin to leaving them at zero, their default value.

There is no singular Shift-Left "silver bullet" solution for this complex issue. Integrating multiple Shift-Left strategies into the CI/CD pipelines is essential. Organizations must also keep abreast of new developments in this realm. If a new solution with additional functionalities becomes available, its adoption into the CI/CD pipeline must be considered. It is also essential to ensure that the Shift-Left tools in the current stack of CI/CD pipelines are regularly updated to leverage any new capabilities they acquire.

In the context of threat modeling, it is important to understand the implications of timeout settings in Go endpoints. The susceptibility of default endpoints in Go must be considered when performing a secure design review and threat analysis.

Developers need more than just coding skills. They require training in securityfocused thinking and the adoption of a defense-in-depth strategy. Initially a military concept, defense-in-depth involves constructing multiple layers of defense to ensure that the failure of one layer does not compromise overall security. Through continuing professional education that focuses on cybersecurity, developers must gain insights into establishing various security layers, understanding potential attack pathways, and applying a comprehensive defense framework to fortify their software.

**4.2 Implications for Future Research**

While this study focused on DoS attacks, future research should include Distributed DoS (DDoS) attack vectors. Other attack vectors, such as the Slow POST and Range Header modes, also warrant further investigation.

Static analysis tools are evolving rapidly. As time passes, these tools will gain new capabilities, presenting itself as an intriguing subject for future research.

Potential defenses such as Layer 3 approaches, rate limiting, and throttling should be explored further. Although rate limiting is not a default feature in Go, the accompanying repository (Mathew, 2024) provides a basic example of implementing rate limiting in Go.

**4.3 Challenges to Research: Slow POST and Range Header**

slowhttptest also offers Slow POST (Calvert et al., 2019) and Range Header modes to stress test endpoints. Timeouts alone were found to be ineffective against these, as shown in Figures 16 and 17 below. Even though the study did not explore these attack vectors and their defenses in detail, the provided repository contains simple rate-limiting examples for Nginx (Rawdat, 2023) and Go, serving as a starting point for developers.

**5 Conclusion**

As the industry adopts Go as a memory-safe language, it is imperative to grasp the nuances of its implementation. As we architect endpoints that will stand the test of time—potentially the next four decades—it is crucial to recognize that Go's default setting of indefinite timeouts could lead to DoS attacks. Therefore, rather than relying on default values, explicit and judicious setting of timeouts is necessary. Given the current porous state of Shift-Left tools, utilizing a robust and flexible suite of these tools is essential. In these formative years of Go's industry adoption, security should be at the forefront of software development to safeguard the investments made in this technology. Security is a collective responsibility. It is essential to foster a culture of awareness and training to ensure its integration into every aspect of software development.