# 5

# Эксплуатация DNS

*Система доменных имен (DNS)* находит имена доменов в интернете и переводит их в IP-адреса. В руках атакующего она может оказаться эффективным оружием, поскольку организации обычно позволяют этому протоколу выходить за рамки сетей с ограниченным доступом и зачастую не могут адекватно контролировать его использование. Хитроумные злоумышленники, обладающие определенными навыками, способны задействовать эти возможности почти на каждом этапе цепочки атаки, включая разведку, управление и контроль (C2) и даже кражу данных. В данной главе вы узнаете, как писать собственные утилиты с помощью Go и сторонних пакетов для реализации некоторых из этих возможностей.

Начнем с разрешения имен хостов и IP-адресов для выявления типов DNS-записей, которые можно перечислить. Затем используем приведенные в предыдущих главах паттерны для построения многопоточного инструмента подбора поддоменов. В завершение вы научитесь создавать собственный DNS-сервер и прокси, а также используете DNS-туннелирование для установки канала C2 из сети с ограниченным доступом.

## Написание DNS-клиентов

Прежде чем начать знакомство с более сложными программами, рассмотрим ряд опций, доступных для клиентских операций. Встроенный в Go пакет `net` предлагает обширную функциональность и поддерживает большинство, если не

все типы записей. Преимущество этого пакета — в простоте его API. Например, `LookupAddr(addr string)` возвращает список имен хостов для заданного IP-адреса. Недостаток же его заключается в невозможности указывать целевой сервер. Вместо этого пакет использует настроенный в операционной системе механизм распознавания. К недостаткам можно отнести также отсутствие возможности выполнения углубленного анализа результатов.

Для обхода этих недочетов мы задействуем отличный сторонний *пакет Go DNS*, написанный Миком Гибеном (Miek Gieben). Предпочесть этот DNS-пакет всем прочим стоит из-за его высокой модульности и грамотно написанного и протестированного кода. Вот команда для его установки:

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

Установив пакет, вы будете готовы к проработке последующих примеров кода. Начнем с выполнения поиска A-записей для получения IP-адресов из имен хостов.

## Извлечение A-записей

Сперва познакомимся с поиском для *полностью уточненного имени домена (fully qualified domain name, FQDN)*, которое указывает точное расположение хоста в иерархии DNS. Затем попробуем интерпретировать это FQDN в IP-адрес с помощью *DNS-записи A*. Эта запись связывает имя домена с IP-адресом. В листинге 5.1 показан пример поиска. (Все листинги кода находятся в корневом каталоге /exist репозитория GitHub https://github.com/blackhat-go/bhg/.)

**Листинг 5.1.** Извлечение записи A (/ch-5/get_a/main.go)

```
package main

import (
    "fmt"

    "github.com/miekg/dns"
)

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

Сначала создается `msg` ❶, после чего идет вызов `fqdn(string)` для преобразования этого домена в FQDN, которым можно обменяться с DNS-сервером ❷. Далее нужно изменить внутреннее состояние `Msg` на вызов `SetQuestion(string, uint16)`

с помощью значения `TypeA`, указывающего, что нужно искать A-запись ❸. (В пакете она определена как `const`. Другие поддерживаемые значения можно найти в документации.) В завершение мы помещаем вызов `Exchange(*Msg, string)` ❹, чтобы отправить сообщение на предоставленный адрес сервера, в данном случае являющегося DNS-сервером, обслуживаемым Google.

Нетрудно заметить, что данный код не особо полезен. Несмотря на то что мы отправляем запрос к DNS-серверу и запрашиваем A-запись, ответ мы не обрабатываем, то есть с результатом ничего не делаем. Но прежде чем реализовать нужную функциональность в Go, давайте рассмотрим, как выглядит ответ DNS, чтобы лучше понять этот протокол и различные типы запросов.

Перед выполнением программы из листинга 5.1 запустите анализатор пакетов, например Wireshark или tcpdump, чтобы просмотреть трафик. Вот пример возможного использования tcpdump на хосте Linux:

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

В отдельном окне терминала скомпилируйте и выполните программу:

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

После выполнения кода в выходных данных перехвата пакетов должны отобразиться подключение к 8.8.8.8 через UDP 53, а также детали DNS-протокола:

```
$ 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 > 8.8.8.8.53: ❶ 25147+ A?❷ stacktitan.com. (32)
23:55:16.650905 IP 8.8.8.8.53 > 192.168.7.51.53307: 25147 1/0/0 A 104.131.56.170 (48) ❸
```

Две из получаемых при перехвате пакетов строчек нуждаются в дополнительном пояснении. Сначала запрос отправляется с 192.168.7.51 к 8.8.8.8 с помощью UDP 53 ❶, при этом происходит запрос A-записи ❷. В ответе ❸ от DNS-сервера Google 8.8.8.8 содержится интерпретированный из имени домена IP-адрес 104.131.56.170.

С помощью анализатора пакетов можно преобразовать имя домена stacktitan.com в IP-адрес. Теперь посмотрим, как извлечь эту информацию, используя Go.

## Обработка ответов от структуры Msg

В качестве значения `Exchange(*Msg, string)` возвращает `(*Msg, error)`. Возврат типа `error` имеет смысл и является стандартным для идиом Go, но почему в ответе приходит также изначально отправленная `*Msg`? Чтобы это понять, нужно взглянуть на определение этой `struct` в исходном коде:

```
type Msg struct {
    MsgHdr
    Compress    bool        `json:"-"`    // Если true, сообщение будет сжато
  ❶ Question    []Question                 // Содержит RR части question
  ❷ Answer      []RR                       // Содержит RR части answer
    Ns          []RR                       // Содержит RR части authority
    Extra       []RR                       // Содержит RR дополнительной части
}
```

Как видите, `Msg struct` содержит как вопросы (`question`), так и ответы (`answer`). Это позволяет объединять все DNS-вопросы и ответы на них в единую унифицированную структуру. Тип `Msg` располагает различными методами, упрощающими работу с данными. Например, срез `Question` ❶ изменяется с помощью метода `setQuestion()`. Это срез можно изменять напрямую, используя `append()`, и получать тот же результат. Срез `Answer` ❷ содержит ответ на запросы и имеет тип `RR`. В листинге 5.2 показано, как эти ответы обрабатывать.

**Листинг 5.2.** Обработка DNS-ответов (/ch-5/get_all_a/main.go)

```
package main

import (
    "fmt"

    "github.com/miekg/dns"
)

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

Пример начинается с сохранения возвращенных от `Exchange` значений и их проверки на наличие ошибок. Если ошибка обнаружена, вызывается `panic()` для остановки программы ❶. Функция `panic()` позволяет быстро просмотреть трассировку

стека и определить место возникновения ошибки. Далее проверяется длина среза `Answer` ❷. Если она меньше 1, это означает, что записей нет, и происходит возврат — бывают случаи, когда имя домена не может быть интерпретировано.

Тип `RR` является интерфейсом, имеющим всего два определенных метода, ни один из которых не дает доступа к IP-адресу, хранящемуся в ответе. Для доступа к этим адресам нужно применить утверждение типа, чтобы создать экземпляр данных в качестве нужного типа.

Сначала выполняем перебор ответов. Далее применяем в ответе утверждение типа, чтобы гарантировать работу с типом `*dns.A` ❸. При выполнении этого действия можно извлечь два значения: данные в виде утвержденного типа и `bool`, отражающее успешность утверждения ❹. После проверки успешности утверждения происходит вывод IP, сохраненного в `a.A` ❺. Несмотря на тип `net.IP`, он реализует метод `String()`, поэтому его можно легко вывести на экран.

Поработайте с этим кодом, изменяя DNS-запрос и обмен (`exchange`) для поиска дополнительных записей. Утверждение типа может оказаться для вас незнакомым, но по своему принципу оно аналогично приведению типов в других языках.

### *Перечисление поддоменов*

Теперь, научившись использовать Go в качестве DNS-клиента, вы можете создавать полезные инструменты. В этом разделе мы создадим утилиту подбора поддоменов. Подбор поддоменов цели и других DNS-записей — основополагающий шаг в процессе разведки, так как чем больше поддоменов вам известно, тем обширнее поле атаки. Наша утилита будет угадывать их на основе передаваемого списка слов (файла словаря).

Используя DNS, можно отправлять запросы настолько быстро, насколько быстро система сможет обрабатывать пакеты данных. Узким местом здесь станут не язык или среда выполнения, а сервер назначения. При этом, как и в предыдущих главах, будет важно управление многопоточностью программы.

Сначала нужно создать в `GOPATH` каталог под названием `subdomain_guesser`, а затем файл `main.go`. После этого в начале создания нового инструмента необходимо решить, какие аргументы эта программа будет получать. В данном случае это будет несколько аргументов, включая целевой домен, имя файла, содержащего поддомены для подбора, используемый DNS-сервер, а также количество запускаемых воркеров. В Go для парсинга опций командной строки есть полезный пакет `flag`, который мы будем применять для обработки аргументов командной строки. Несмотря на то что мы используем этот пакет не во всех примерах кода, в данном случае он служит для демонстрации более надежного и изящного парсинга аргументов. Код этого процесса приведен в листинге 5.3.

**Листинг 5.3.** Создание программы подбора поддоменов (/ch-5/subdomain_guesser/main.go)

```
package main

import (
    "flag"
)

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

В начале строка кода, объявляющая переменную `flDomain` ❶, получает аргумент `String` и объявляет пустое строковое значение для того, что будет парситься как опция `domain`. Следующая связанная строка — это объявление переменной `flWorkerCount` ❷. Здесь в качестве опции командной строки `c` нужно предоставить значение `Integer`. В данном случае мы устанавливаем 100 воркеров. Но это значение можно счесть консервативным, так что в процессе тестирования смело экспериментируйте с увеличением их числа. В завершение вызов `flag.Parse()` ❸ заполняет переменные, задействуя предоставленный пользователем ввод.

### ПРИМЕЧАНИЕ

Вы могли обратить внимание на то, что этот пример идет вразрез с правилами Unix в том, что определяет необязательные аргументы, которые на деле являются обязательными. Можете свободно использовать здесь `os.Args`. Просто нам быстрее и удобнее поручить всю работу пакету `flag`.

При сборке данной программы должна возникнуть ошибка, указывающая на неиспользованные переменные. Добавьте приведенный далее код сразу после вызова `flag.Parse()`. Это дополнение выводит в stdout переменные наряду с кодом, гарантируя передачу пользователем `-domain` и `-wordlist`:

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

Чтобы ваш инструмент сообщал, какие имена оказались интерпретируемыми, указывая при этом соответствующие им IP-адреса, нужно создать для хранения этой информации тип `struct`. Определите его над функцией `main()`:

```
type result struct {
    IPAddress string
    Hostname string
}
```

Для этого инструмента вы будете запрашивать два основных типа записей — A и CNAME. Каждый запрос будет выполняться в отдельной функции. Стоит создавать эти функции максимально небольшими и поручать каждой выполнение только одной задачи. Такой стиль разработки позволит в дальнейшем писать менее объемные тесты.

### *Запрос записей A и CNAME*

Для выполнения запросов мы создадим две функции: одну для A-записей, вторую для записей CNAME. Они обе будут получать FQDN в качестве первого аргумента и адрес DNS-сервера в качестве второго. Каждая из них должна возвращать срез строк и ошибку. Добавьте эти функции в код, который начали определять в листинге 5.3, расположив вне области `main()`:

```
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
}
```

Этот код должен показаться вам знакомым, так как он практически идентичен коду, который мы писали в самом начале главы. Первая функция, `lookupA`, возвращает список IP-адресов, а `lookupCNAME` возвращает список имен хостов.

Записи *CNAME (канонические имена)* сопоставляют одно FQDN с другим, которое служит псевдонимом для первого. Предположим, что владелец организации example.com хочет разместить WordPress-сайт с помощью сервиса хостинга WordPress. У этого сервиса могут быть сотни IP-адресов для балансировки всех пользовательских сайтов, в связи с чем предоставить IP для отдельного сайта просто невозможно. Вместо этого данный хостинг может предоставить каноническое имя (CNAME), на которое и будет ссылаться example.com. В итоге адрес www.example.com получит CNAME, указывающее на somewhere.hostingcompany.org, которое, в свою очередь, будет иметь A-запись, указывающую на IP-адрес. Это позволит владельцу example.com разместить свой сайт на сервере, для которого у него нет IP-данных.

Зачастую это означает, что вам нужно проследить целый хвост из канонических имен, чтобы в итоге добраться до действительной A-записи. Мы говорим *хвост*, потому что из подобных имен может выстраиваться бесконечная цепочка. Добавьте приведенный далее код функции в область за пределами функции `main()`, чтобы понаблюдать, как использовать череду CNAMES для нахождения A-записи:

```
func lookup(fqdn, serverAddr string) []result {
  ❶ var results []result
  ❷ var cfqdn = fqdn // Не изменять оригинал
    for {
      ❸ cnames, err := lookupCNAME(cfqdn, serverAddr)
      ❹ if err == nil && len(cnames) > 0 {
          ❺ cfqdn = cnames[0]
          ❻ continue // Нужно обработать следующее CNAME
        }
      ❼ ips, err := lookupA(cfqdn, serverAddr)
        if err != nil {
            break // Для этого имени хоста A-записей нет
        }
      ❽ for _, ip := range ips {
            results = append(results, result{IPAddress: ip, Hostname: fqdn})
```

```
            }
        ❾ break // Все результаты были обработаны
        }
        return results
    }
```

Сначала определяется срез для хранения результатов ❶. Далее создается копия FQDN, переданного в качестве первого аргумента ❷. В итоге вы не только не теряете исходный угаданный FQDN, но и можете задействовать его в первой попытке запроса. Начав бесконечный цикл, мы пробуем получить CNAME для этого FQDN ❸. В случае отсутствия ошибок и возвращения не менее одного CNAME ❹ устанавливаем `cfqdn` равным этому возвращенному CNAME ❺, используя `continue` для возврата к началу цикла ❻. Данный процесс позволяет проследить череду CNAME до возникновения сбоя. Последний будет означать, что конец цепочки достигнут и можно искать A-записи ❼. Но если возникнет ошибка, означающая, что при поиске записи возникли проблемы, то выход из цикла произойдет раньше. В случае обнаружения действительных A-записей каждый возвращенный IP-адрес добавляется в срез результатов ❽, а цикл прерывается ❾. В завершение `results` возвращается вызывающему.

Наша связанная с интерпретацией имен логика выглядит гладко, однако мы не учли производительность. Давайте совместим этот пример с горутинами, добавив в него многопоточность.

### *Переход к воркер-функции*

Мы создадим пул горутин, которые будут передавать работу *воркер-функции*, выполняющей единицу работы. Для распределения работы и сбора ее результатов задействуем каналы. Напомним, что нечто подобное мы уже делали в главе 2, когда создавали многопоточный сканер портов.

Продолжим расширять код из листинга 5.3. Сначала создадим функцию `worker()`, разместив ее вне области функции `main()`. Она будет получать три аргумента каналов: канал для воркера, чтобы он сигнализировал о своем закрытии, канал доменов, в которых нужно получать работу, и канал для отправки результатов. Этой функции потребуется заключительный строковый аргумент для указания используемого DNS-сервера. Далее приведен пример кода для функции `worker()`:

```
type empty struct{} ❶

func worker(tracker chan empty, fqdns chan string, gather chan []result,
serverAddr string) {
    for fqdn := range fqdns { ❷
        results := lookup(fqdn, serverAddr)
        if len(results) > 0 {
```

```
            gather <- results ❸
        }
    }
    var e empty
    tracker <- e ❹
}
```

Прежде чем вводить функцию `worker()`, определим тип `empty` для отслеживания завершения выполнения воркера ❶. Это будет структура без полей. Мы задействуем пустую `struct`, так как она имеет размер 0 байт и практически не создаст нагрузку при использовании. Далее в функции `worker()` происходит перебор канала доменов ❷, используемый для передачи FQDN. После получения ответа от функции `lookup()` и проверки наличия не менее одного результата мы отправляем его в канал `gather` ❸, который собирает все результаты обратно в `main()`. После того как канал закрывается и цикл совершает выход, структура `empty` отправляет в канал `tracker` ❹ сигнал вызывающему о завершении всей работы. Отправка пустой `struct` в канал отслеживания — это важный последний шаг. Если этого не сделать, возникнет состояние гонки, так как вызывающий компонент может выйти до получения каналом `gather` результатов.

Поскольку вся необходимая структура теперь настроена, можно переключиться обратно на `main()` и закончить программу, которую мы начали писать в листинге 5.3.

Определите переменные, которые будут содержать результаты и каналы, передаваемые в `worker()`, после чего добавьте в `main()` следующий код:

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

Создайте канал `fqdns` как буферизованный на основе предоставленного пользователем количества воркеров. Это позволит воркерам запускаться быстрее, поскольку канал сможет вместить больше одного сообщения до блокировки отправителя.

### *Создание сканера с помощью bufio*

Далее откройте файл, предоставленный пользователем в качестве списка слов, и создайте в нем новый `scanner` с помощью пакета `bufio`. Добавьте в `main()` код

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

Если возвращаемая ошибка не равна `nil`, используется встроенная функция `panic()`. При написании пакета или программы для применения другими людьми следует постараться представить эту информацию более ясно.

Мы будем применять новый `scanner` для захвата строки текста из переданного списка слов и создания FQDN путем совмещения этого текста с предоставленным пользователем доменом. Результат будет отправляться в канал `fqdns`. Но сначала нужно запустить воркеры, так как порядок важен. Если отправить работу в канал `fqdns`, не запустив их, этот буферизованный канал в итоге заполнится и функции-производители будут заблокированы. В `main()` нужно добавить приведенный далее код, чья задача — запускать горутины воркеров, читать вводный файл и отправлять работу в канал `fqdns`.

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

❷ for scanner.Scan() {
      fqdns <- fmt.Sprintf("%s.%s", scanner.Text()❸, *flDomain)
  }
```

Создание воркеров ❶ с помощью этого паттерна похоже на то, что мы уже делали при построении многопоточного сканера портов: задействовали цикл `for` до момента достижения числа, переданного пользователем. Для захвата каждой строки в цикле используется `scanner.Scan()` ❷. Этот цикл заканчивается, когда в файле не остается строк для считывания. Для получения строкового представления текста из отсканированной строки мы применяем `scanner.Text()` ❸.

Работа запущена! Отвлекитесь на секунду и ощутите свое величие. Прежде чем читать следующий код, подумайте, где вы находитесь в программе и что уже успели сделать за время чтения книги. Попробуйте самостоятельно закончить эту программу и затем перейти к следующему разделу, где мы поясним ее оставшуюся часть.

### *Сбор и отображение результатов*

Проработку последней части мы начнем с запуска анонимной горутины, которая будет собирать результаты воркеров. Добавьте в `main()` следующее:

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

Перебирая канал `gather`, мы добавляем полученные результаты в срез `results` ❶. Поскольку мы добавляем срез в другой срез, нужно использовать синтаксис ... ❷. После закрытия канала `gather` и завершения перебора, как и прежде, происходит отправка пустой `struct` в канал отслеживания ❸. Это делается для предотвращения состояния гонки на случай, если `append()` не завершится к моменту итогового предоставления результатов пользователю.

Остается только закрыть каналы и представить результаты. Для этого добавьте следующий код в конец `main()`:

```
❶ close(fqdns)
❷ for i := 0; i < *flWorkerCount; i++ {
      <-tracker
  }
❸ close(gather)
❹ <-tracker
```

Первым можно закрыть канал `fqdns` ❶, так как мы уже отправили по нему всю работу. Далее нужно выполнить получение результатов в канале `tracker` по одному разу для каждого воркера ❷, что позволит им обозначить свое полное завершение. После этого можно закрыть канал `gather` ❸, потому что результатов для получения не остается. В завершение нужно выполнить еще одно получение результатов на канале `tracker`, чтобы позволить горутине окончательно завершиться ❹.

Эти результаты пользователю еще не представлены. Нужно это исправить. При желании можно просто перебрать срез `results` и вывести поля `Hostname` и `IPAddress`, используя `fmt.Printf()`. Тем не менее мы предпочитаем задействовать для представления данных один из нескольких прекрасных пакетов Go, а именно `tabwriter`. Он позволяет выводить данные в красивых ровных столбцах, разбитых на вкладки. Для его применения добавьте в конец `main()` следующий код:

```
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()
```

В листинге 5.4 показана вся программа в сборе.

**Листинг 5.4.** Вся программа подбора поддоменов (/ch-5/subdomain_guesser/main.go)

```
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 // Не изменяем оригинал
    For {
        cnames, err := lookupCNAME(cfqdn, serverAddr)
        if err == nil && len(cnames) > 0 {
            cfqdn = cnames[0]
```

```
            continue // Нужно обработать следующее CNAME
        }
        ips, err := lookupA(cfqdn, serverAddr)
        if err != nil {
            break // Для этого имени хоста нет A-записей
        }
        for _, ip := range ips {
            results = append(results, result{IPAddress: ip, Hostname: fqdn})
        }
        break // Все результаты обработаны
    }
    return results
}

func worker(tracker chan empty, fqdns chan string, gather chan []result,
            serverAddr string) {
    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
                        guessing against.")
        flWordlist    = flag.String("wordlist", "", "The wordlist to use
                        for guessing.")
        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()

    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)
    }

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

    for scanner.Scan() {
        fqdns <- fmt.Sprintf"%s.%", scanner.Text(), *flDomain)
    }
    // Заметьте: здесь можно проверить scanner.Err()

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

    w := tabwriter.NewWriter(os.Stdout, 0, 8' ', ' ', 0)
    for _, r := range results {
        fmt.Fprint"(w, "%s\"%s\n", r.Hostname, r.IPAddress)
    }
    w.Flush()
}
```

На этом наша программа для подбора поддоменов готова. Теперь вы можете собрать и запустить этот инструмент. Опробуйте его на списках слов или словарях из открытых репозиториев (можете найти множество через Google). Поэкспериментируйте с количеством воркеров. Вы можете заметить, что при слишком быстрой обработке результаты получаются неоднозначные. Вот пример выполнения с использованием ста воркеров: