Author Recar https://github.com/Ciyfly

ksubdomain浅析

ksubdomain是一款基于无状态的子域名爆破工具，类似无状态端口扫描，支持在Windows/Linux/Mac上进行快速的DNS爆破，拥有重发机制不用担心漏包。

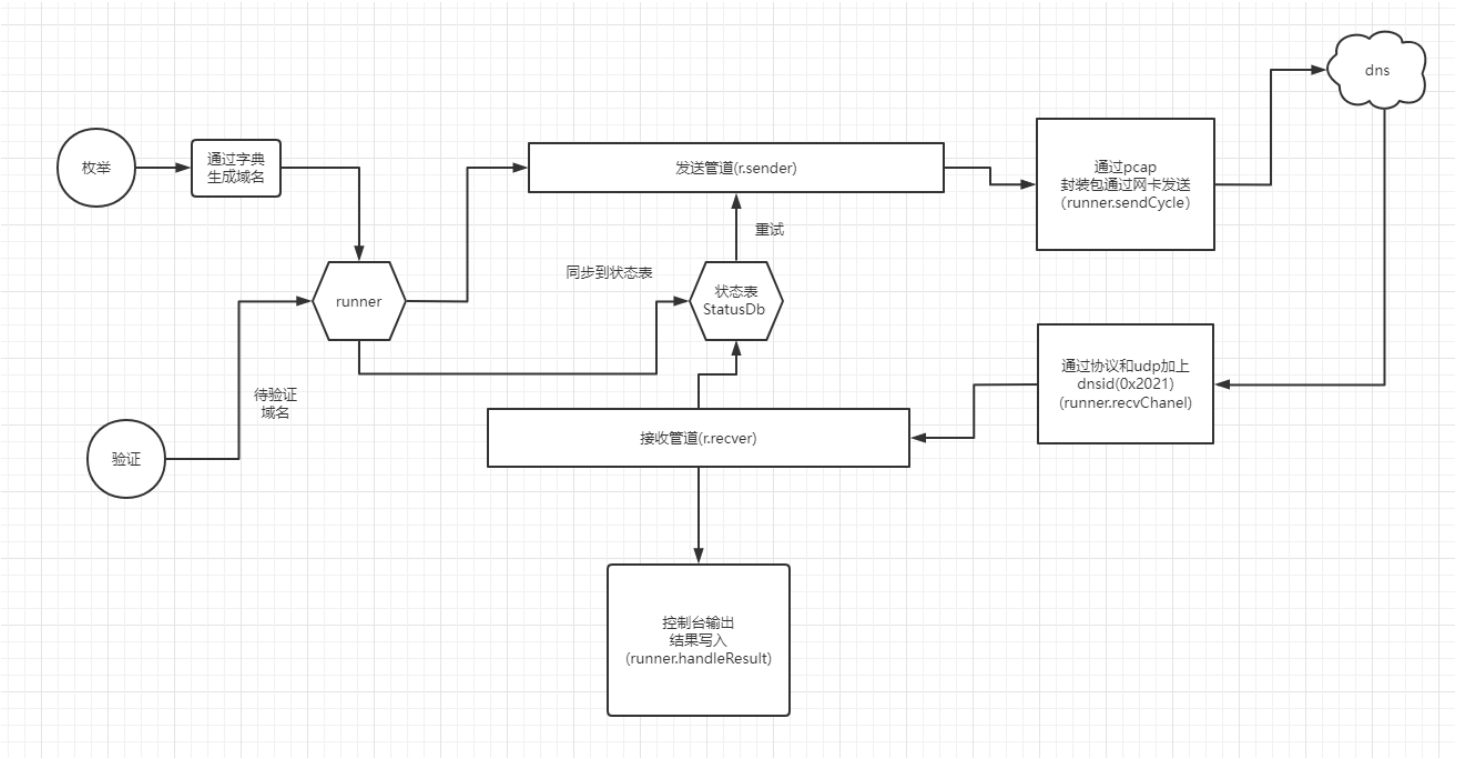
ksubdomain的作者是 w8ay
ksubdomain 地址 https://github.com/boy-hack/ksubdomain
ksubdomain的介绍 https://paper.seebug.org/1325/

主要是使用无状态的概念 直接使用网卡发包而不是通过socket去操作
对于子域名枚举来说 只发一个udp包等等dns服务器的应答
ksubdomain使用pcap发包和接收数据，会直接将数据包发送至网卡，不经过系统

通过设计了一个状态表 来控制这个udp的丢包和状态

文章目录

简易流程图



代码目录

cmd

包含对命令行的支持 并且拆开了 主程序 `enum` `test` `verify` 的命令到不同的go文件

```
cmd
├── cmd.go
├── enum.go
├── test.go
└── verify.go
```

core

主要的核心逻辑代码等包含 日志 配置文件 banner 是代码的主体

```
core/
├── banner.go
├── conf
│   └── config.go
├── data
│   ├── subdomain.txt
│   └── subnext.txt
├── device
│   ├── device.go
│   └── struct.go
├── gologger
│   └── gologger.go
├── options
│   └── options.go
├── struct.go
├── subdata.go
├── util.go
└── wildcard.go
```

runner

是发包接收包 核心代码

```
runner/
├── recv.go
├── result.go
├── retry.go
├── runner.go
├── runner_test.go
├── send.go
├── statusdb
│   └── db.go
└── testspeed.go
```

test

就是测试代码了

```
test/
└── checkservername
    └── main.go
```

可以明显看出来代码设计架构等清晰

我们这里按cmd的使用方式来跟进代码

cmd

cmd\cmd.go
使用cli 构建命令行解析

```
app := &cli.App{
    Name:    conf.AppName,
    Version: conf.Version,
    Usage:   conf.Description,
    Commands: []*cli.Command{
        enumCommand,
        verifyCommand,
        testCommand,
    },
}
```

包含三个子命令行
这个的功能是输出主的help

```
NAME:
   KSubdomain - 无状态子域名爆破工具

USAGE:
   ksubdomain [global options] command [command options] [arguments...]

VERSION:
   1.8.2

COMMANDS:
   enum, e    枚举域名
   verify, v  验证模式
   test       测试本地网卡的最大发送速度
   help, h    Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --help, -h     show help (default: false)
   --version, -v  print the version (default: false)
```

enumCommand

枚举功能

cmd\enum.go

枚举相关的参数

```
&cli.StringFlag{
    Name:     "domain",
    Aliases:  []string{"d"},
    Usage:    "爆破的域名",
    Required: false,
    Value:    "",
},
&cli.StringFlag{
    Name:     "domainList",
    Aliases:  []string{"dl"},
    Usage:    "从文件中指定域名",
    Required: false,
    Value:    "",
},
&cli.StringFlag{
    Name:     "filename",
    Aliases:  []string{"f"},
    Usage:    "字典路径",
    Required: false,
    Value:    "",
},
&cli.BoolFlag{
    Name:  "skip-wild",
    Usage: "跳过泛解析域名",
    Value: false,
},
&cli.IntFlag{
    Name:    "level",
    Aliases: []string{"l"},
    Usage:   "枚举几级域名，默认为2，二级域名",
    Value:   2,
},
&cli.StringFlag{
    Name:    "level-dict",
    Aliases: []string{"ld"},
    Usage:   "枚举多级域名的字典文件，当level大于2时候使用，不填则会默认",
    Value:   "",
},
```

从参数获取需要测试的一级域名

指定的单个域名 文件读取的域名都加到 `domains`

```
var domains []string
// handle domain
if c.String("domain") != "" {
    domains = append(domains, c.String("domain"))
}
if c.String("domainList") != "" {
    dl, err := core.LinesInFile(c.String("domainList"))
    if err != nil {
        gologger.Fatalf("读取domain文件失败:%s\n", err.Error())
    }
    domains = append(dl, domains...)
}
levelDict := c.String("level-dict")
var levelDomains []string
if levelDict != "" {
    dl, err := core.LinesInFile(levelDict)
    if err != nil {
        gologger.Fatalf("读取domain文件失败:%s,请检查--level-dict参数\n", err.Error())
    }
    levelDomains = dl
} else if c.Int("level") > 2 {
    levelDomains = core.GetDefaultSubNextData()
}
```

检验设置参数值 传递给 runner.New方法

返回一个runner结构体

`RunEnumeration` 枚举运行入口

`Close` 关闭

```
    opt := &options.Options{
        Rate:         options.Band2Rate(c.String("band")),
        Domain:       domains,
        FileName:     c.String("filename"),
        Resolvers:    options.GetResolvers(c.String("resolvers")),
        Output:       c.String("output"),
        Silent:       c.Bool("silent"),
        Stdin:        c.Bool("stdin"),
        SkipWildCard: c.Bool("skip-wild"),
        TimeOut:      c.Int("timeout"),
        Retry:        c.Int("retry"),
        Method:       "enum",
        OnlyDomain:   c.Bool("only-domain"),
        NotPrint:     c.Bool("not-print"),
        Level:        c.Int("level"),
        LevelDomains: levelDomains,
    }
    opt.Check()

    r, err := runner.New(opt)
    if err != nil {
        gologger.Fatalf("%s\n", err.Error())
        return nil
    }
    r.RunEnumeration()
    r.Close()
    return nil
},
```

runner

接下来我们跟入 runner来详细看下 New方法做了什么 枚举运行又做了什么

runner.New

首先runner的结构体 在其他语言中可以理解为一个类

```
type runner struct {
        ether           *device.EtherTable //本地网卡信息
        hm              *statusdb.StatusDb
        options         *options2.Options
        limit           ratelimit.Limiter
        handle          *pcap.Handle
        successIndex    uint64
        sendIndex       uint64
        recvIndex       uint64
        faildIndex      uint64
        sender          chan string
        recver          chan core.RecvResult
        freeport        int
        dnsid           uint16 // dnsid 用于接收的确定ID
        maxRetry        int    // 最大重试次数
        timeout         int64  // 超时xx秒后重试
        ctx             context.Context
        fisrtloadChanel chan string // 数据加载完毕的chanel
        startTime       time.Time
        domains         []string
}
```

New方法 是传入 `opt` 返回runner 其实就是对opt赋值并实例一个runner

跟入New方法

1. 获取pcap版本
2. 获取使用的网卡
3. 创建状态表
4. 获取使用网卡的监听handler `device.PcapInit(r.ether.Device)`
5. 获取发包总数 `runner.loadTargets`
6. 如果枚举域名超过2级 则所有包数为 字典包与域名的乘积 乘上 测试域名的数量的测试层级的幂
7. 计算calcLimit 为 (发包总数/超时时间) *0.85 如果小于1k则为1k
8. 真正的limit为 宽带的下行速度(命令行参数 --band/-b) 和calcLimit的最大值
9. 初始化限流器 `r.limit = ratelimit.New(limit)` 使用的是uber-go的基于漏桶实现的
   `https://github.com/uber-go/ratelimit/`
10. 初始化发送管道 接收管道

```
r.sender = make(chan string, 99)          // 多个协程发送
r.recver = make(chan core.RecvResult, 99) // 多个协程接收
```

11. 获取空闲端口 `freeport.GetFreePort()`

```
func GetFreePort() (int, error) {
        addr, err := net.ResolveTCPAddr("tcp", "localhost:0")
        if err != nil {
                return 0, err
        }

        l, err := net.ListenTCP("tcp", addr)
        if err != nil {
                return 0, err
        }
        defer l.Close()
        return l.Addr().(*net.TCPAddr).Port, nil
}
```

12. 设置一些属性 例如dnsid为 `0x2021` 应该是2021年开发的把 重试次数超时时间 上下文等数据赋值给runner
13. 将测试的域名发到send管道 并为管道 `fisrtloadChanel` 传入值
14. 返回runner

详细见下面New代码及部分函数

```
func New(options *options2.Options) (*runner, error) {
	var err error
	version := pcap.Version()
	r := new(runner)
	gologger.Infof(version + "\n")

	r.options = options
	r.ether = GetDeviceConfig()
	r.hm = statusdb.CreateMemoryDB()

	gologger.Infof("DNS:%s\n", options.Resolvers)
	r.handle, err = device.PcapInit(r.ether.Device)
	if err != nil {
		return nil, err
	}

	// 根据发包总数和timeout时间来分配每秒速度
	allPacket := r.loadTargets()
	if options.Level > 2 {
		allPacket = allPacket * int(math.Pow(float64(len(options.LevelDomains)), float64(options.Level-2)
	}
	calcLimit := float64(allPacket/options.TimeOut) * 0.85
	if calcLimit < 1000 {
		calcLimit = 1000
	}
	limit := int(math.Min(calcLimit, float64(options.Rate)))
	r.limit = ratelimit.New(limit) // per second

	gologger.Infof("Rate:%dpps\n", limit)

	r.sender = make(chan string, 99)          // 多个协程发送
	r.recver = make(chan core.RecvResult, 99) // 多个协程接收

	freePort, err := freeport.GetFreePort()
	if err != nil {
		return nil, err
	}
	r.freeport = freePort
	gologger.Infof("FreePort:%d\n", freePort)
	r.dnsid = 0x2021 // set dnsid 65500
	r.maxRetry = r.options.Retry
	r.timeout = int64(r.options.TimeOut)
	r.ctx = context.Background()
	r.fisrtloadChanel = make(chan string)
	r.startTime = time.Now()

	go func() {
		for _, msg := range r.domains {
			r.sender <- msg
			if options.Method == "enum" && options.Level > 2 {
```

```
                                r.iterDomains(options.Level, msg)
                        }
                }
                r.domains = nil
                r.fisrtloadChanel <- "ok"
        }()
        return r, nil
}
```

获取pcap版本信息

```
pcap.Version()
```

获取网卡信息并写到yaml文件中

```
r.ether = GetDeviceConfig()
```

如果有yaml文件直接读了 没有在获取 `device.AutoGetDevices()`

```
func GetDeviceConfig() *device.EtherTable {
        filename := "ksubdomain.yaml"
        var ether *device.EtherTable
        var err error
        if core.FileExists(filename) {
                ether, err = device.ReadConfig(filename)
                if err != nil {
                        gologger.Fatalf("读取配置失败:%v", err)
                }
                gologger.Infof("读取配置%s成功!\n", filename)
        } else {
                ether = device.AutoGetDevices()
                err = ether.SaveConfig(filename)
                if err != nil {
                        gologger.Fatalf("保存配置失败:%v", err)
                }
        }
        gologger.Infof("Use Device: %s\n", ether.Device)
        gologger.Infof("Use IP:%s\n", ether.SrcIp.String())
        gologger.Infof("Local Mac: %s\n", ether.SrcMac.String())
        gologger.Infof("GateWay Mac: %s\n", ether.DstMac.String())
        return ether
}
```

网卡获取代码在 `core\device\device.go`
实现方式如下:

1. 随机生成一个域名 `domain := core.RandomStr(4) + ".i.hacking8.com"`
2. 然后通过pcap获取所有网卡 及网卡对应的信息

```
devices, err := pcap.FindAllDevs()
if err != nil {
    gologger.Fatalf("获取网络设备失败:%s\n", err.Error())
}
data := make(map[string]net.IP)
keys := []string{}
for _, d := range devices {
    for _, address := range d.Addresses {
        ip := address.IP
        if ip.To4() != nil && !ip.IsLoopback() {
            data[d.Name] = ip
            keys = append(keys, d.Name)
        }
    }
}
```

3. 开始抓包 多个网卡遍历获取回来的包 如果是dns的信息并且域名匹配那就认为是从这个网卡发出去的即后续使用这个网卡

```
ctx := context.Background()
// 在初始上下文的基础上创建一个有取消功能的上下文
ctx, cancel := context.WithCancel(ctx)
for _, drviceName := range keys {
    go func(drviceName string, domain string, ctx context.Context) {
        var (
            snapshot_len int32         = 1024
            promiscuous  bool          = false
            timeout      time.Duration = -1 * time.Second
            handle       *pcap.Handle
        )
        var err error
        handle, err = pcap.OpenLive(
            drviceName,
            snapshot_len,
            promiscuous,
            timeout,
        )
        if err != nil {
            gologger.Errorf("pcap打开失败:%s\n", err.Error())
            return
        }
        defer handle.Close()
        // Use the handle as a packet source to process all packets
        packetSource := gopacket.NewPacketSource(handle, handle.LinkType())
        for {
            select {
            case <-ctx.Done():
                return
            default:
                packet, err := packetSource.NextPacket()
                gologger.Printf(".")
                if err != nil {
                    continue
                }
                if dnsLayer := packet.Layer(layers.LayerTypeDNS); dnsLayer != nil {
                    dns, _ := dnsLayer.(*layers.DNS)
                    if !dns.QR {
                        continue
                    }
                    for _, v := range dns.Questions {
                        if string(v.Name) == domain {
                            ethLayer := packet.Layer(layers.LayerTypeEthernet)
                            if ethLayer != nil {
                                eth := ethLayer.(*layers.Ethernet)
                                etherTable := EtherTable{
                                    SrcIp:  data[drviceName],
                                    Device: drviceName,
                                    SrcMac: SelfMac(eth.DstMAC),
                                    DstMac: SelfMac(eth.SrcMAC),
```

```
                                }
                                signal <- &etherTable
                                return
                            }
                        }
                    }
                }
            }
        }(drviceName, domain, ctx)
}
```

4. 3是协程启动 启动后通过 `net.LookupHost(domain)` 发起dns请求 会每隔1s发送一次 一直到找到能使用的网卡

```
for {
    select {
    case c := <-signal:
        cancel()
        fmt.Print("\n")
        return c
    default:
        _, _ = net.LookupHost(domain)
        time.Sleep(time.Second * 1)
    }
}
```

5. 当匹配上发送信息到管道结束阻塞并赋值 网卡信息

创建内存简易数据库

`runner\runner.go#75`

```
r.hm = statusdb.CreateMemoryDB()
```

简易数据库的代码 `runner\statusdb\db.go`

是对状态表的一个封装 使用的是 syncmap

```
type Item struct {
        Domain      string    // 查询域名
        Dns         string    // 查询dns
        Time        time.Time // 发送时间
        Retry       int       // 重试次数
        DomainLevel int       // 域名层级
}
type StatusDb struct {
        Items  sync.Map
        length int64
}
```

创建的话

```
func CreateMemoryDB() *StatusDb {
        db := &StatusDb{
                Items:  sync.Map{},
                length: 0,
        }
        return db
}
```

增加数据

```
func (r *StatusDb) Add(domain string, tableData Item) {
        r.Items.Store(domain, tableData)
        atomic.AddInt64(&r.length, 1)
}
```

runner.loadTargets 获取发包总数

1. 从linux管道的方式获取domain写到domain中
2. 读取子域名字典 `core\data\subdomain.txt`
3. 检测泛解析 `core.IsWildCard(domain)`
4. 遍历字典加域名生成需要测试的新域名集合返回长度即为发包总数

```
    if options.Stdin {
        scanner := bufio.NewScanner(os.Stdin)
        scanner.Split(bufio.ScanLines)
        for scanner.Scan() {
            options.Domain = append(options.Domain, scanner.Text())
        }
    }
    // 读取字典
    if options.FileName == "" {
        subdomainDict := core.GetDefaultSubdomainData()
        reader = bufio.NewReader(strings.NewReader(strings.Join(subdomainDict, "\n")))
    } else {
        subdomainDict, err := core.LinesInFile(options.FileName)
        if err != nil {
            gologger.Fatalf("打开文件:%s 错误:%s", options.FileName, err.Error())
        }
        reader = bufio.NewReader(strings.NewReader(strings.Join(subdomainDict, "\n")))
    }

    if options.SkipWildCard && len(options.Domain) > 0 {
        var tmpDomains []string
        gologger.Infof("检测泛解析\n")
        for _, domain := range options.Domain {
            if !core.IsWildCard(domain) {
                tmpDomains = append(tmpDomains, domain)
            } else {
                gologger.Warningf("域名:%s 存在泛解析记录,已跳过\n", domain)
            }
        }
        options.Domain = tmpDomains
    }
}
        for {
                line, _, err := reader.ReadLine()
                if err != nil {
                        break
                }
                msg := string(line)
                if r.options.Method == "verify" {
                        // send msg
                        r.domains = append(r.domains, msg)
                } else {
                        for _, tmpDomain := range r.options.Domain {
                                newDomain := msg + "." + tmpDomain
                                r.domains = append(r.domains, newDomain)
                        }
                }
        }
        return len(r.domains)
```

core.IsWildCard(domain) 泛解析检测

判断泛解析的方式是 随机6个字符串加上域名 生成2次 如果两次都是不能解析则认为没有泛解析 两次只要有一次解析了就认为存在泛解析

```
func IsWildCard(domain string) bool {
        for i := 0; i < 2; i++ {
                subdomain := RandomStr(6) + "." + domain
                _, err := net.LookupIP(subdomain)
                if err != nil {
                        continue
                }
                return true
        }
        return false
}
```

RunEnumeration

涉及到的函数多 下面以多个标题来分析
判断字典是否读取完成 完成的才会有对状态表还是有需要测试的数据判断 防止一开始是空的就结束了

```
func (r *runner) RunEnumeration() {
        ctx, cancel := context.WithCancel(r.ctx)
        defer cancel()
        go r.recvChanel(ctx) // 启动接收线程
        for i := 0; i < 3; i++ {
                go r.sendCycle(ctx) // 发送线程
        }
        go r.handleResult(ctx) // 处理结果，打印输出

        var isLoadOver bool = false // 是否加载文件完毕
        t := time.NewTicker(1 * time.Second)
        defer t.Stop()
        for {
                select {
                case <-t.C:
                        r.PrintStatus()
                        if isLoadOver {
                                if r.hm.Length() == 0 {
                                        gologger.Printf("\n")
                                        gologger.Infof("扫描完毕")
                                        return
                                }
                        }
                case <-r.fisrtloadChanel:
                        go r.retry(ctx) // 遍历hm，依次重试
                        isLoadOver = true
                }
        }
}
```

接收协程 runner.recvChanel

1. 获取一个未激活的pcap句柄 `inactive, err := pcap.NewInactiveHandle(r.ether.Device)`
2. 设置 每个要捕获的数据包的最大字节数为 `65535` `err = inactive.SetSnapLen(snapshotLen)`
3. 设置超时时间为 -1s 即不超时
4. 设置模式为即时模式 数据包被传送到应用程序就会更新timeout `inactive.SetImmediateMode(true)`
5. 激活pcap句柄 `handle, err := inactive.Activate()`
6. 设置过滤条件 `udp and src port 53 and dst port` udp的包 源端口是53 目的端口是发出去的空闲端口
7. 创建一个解析器 并读取数据 `data, _, err = handle.ReadPacketData()`
8. 解析数据包 `err = parser.DecodeLayers(data, &decoded)`
9. 只要dns的包 并且dnsid为 `0x2021` 的包
10. 对于符合条件的包计数 使用原子操作增加 接收次数 `recvIndex`

```
atomic.AddUint64(&r.recvIndex, 1)
if len(dns.Questions) == 0 {
    continue
}
```

11. 获取解析出来的域名 并在状态表中删除 成功计数+1 成功结果+1 推到 `runner.rever` 里

```
func (r *runner) recvChanel(ctx context.Context) error {
	var (
		snapshotLen = 65536
		timeout     = -1 * time.Second
		err         error
	)
	inactive, err := pcap.NewInactiveHandle(r.ether.Device)
	if err != nil {
		return err
	}
	err = inactive.SetSnapLen(snapshotLen)
	if err != nil {
		return err
	}
	defer inactive.CleanUp()
	if err = inactive.SetTimeout(timeout); err != nil {
		return err
	}
	err = inactive.SetImmediateMode(true)
	if err != nil {
		return err
	}
	handle, err := inactive.Activate()
	if err != nil {
		return err
	}
	defer handle.Close()

	err = handle.SetBPFFilter(fmt.Sprintf("udp and src port 53 and dst port %d", r.freeport))
	if err != nil {
		return errors.New(fmt.Sprintf("SetBPFFilter Faild:%s", err.Error()))
	}

	// Listening

	var udp layers.UDP
	var dns layers.DNS
	var eth layers.Ethernet
	var ipv4 layers.IPv4
	var ipv6 layers.IPv6

	parser := gopacket.NewDecodingLayerParser(
		layers.LayerTypeEthernet, &eth, &ipv4, &ipv6, &udp, &dns)

	var data []byte
	var decoded []gopacket.LayerType
	for {
		data, _, err = handle.ReadPacketData()
		if err != nil {
			continue
```

```
                }
                err = parser.DecodeLayers(data, &decoded)
                if err != nil {
                        continue
                }
                if !dns.QR {
                        continue
                }
                if dns.ID != r.dnsid {
                        continue
                }
                atomic.AddUint64(&r.recvIndex, 1)
                if len(dns.Questions) == 0 {
                        continue
                }
                subdomain := string(dns.Questions[0].Name)
                r.hm.Del(subdomain)
                if dns.ANCount > 0 {
                        atomic.AddUint64(&r.successIndex, 1)
                        result := core.RecvResult{
                                Subdomain: subdomain,
                                Answers:   dns.Answers,
                        }
                        r.recver <- result
                }
        }
}
```

发送协程 runner.sendCycle

这里通过限流器 `limit` 来控制发送

1. 从 `sender` 里接收到一个域名 然后从状态表里获取出来这个域名的其他信息
2. 如果没有就生成一个表的条目 并存入状态表 否则认为是需要重试的 重试计数增加
   随机更新选取dns服务器 更新表对应数据信息
3. 使用 `send` 发送数据并 `runner.sendIndex`计数+1

```
func (r *runner) sendCycle(ctx context.Context) {
        for domain := range r.sender {
                r.limit.Take()
                v, ok := r.hm.Get(domain)
                if !ok {
                        v = statusdb.Item{
                                Domain:      domain,
                                Dns:         r.choseDns(),
                                Time:        time.Now(),
                                Retry:       0,
                                DomainLevel: 0,
                        }
                        r.hm.Add(domain, v)
                } else {
                        v.Retry += 1
                        v.Time = time.Now()
                        v.Dns = r.choseDns()
                        r.hm.Set(domain, v)
                }
                send(domain, v.Dns, r.ether, r.dnsid, uint16(r.freeport), r.handle)
                atomic.AddUint64(&r.sendIndex, 1)
        }
}
```

send 发送dns包

填充数据包

1. eth 二层 源目的mac地址 从网卡获取
2. ip 三层 ipv4的 主要是 约定协议是 `udp` 源ip(网卡ip) 目的ip(dns服务器)
3. udp包 四层 udp 源目的端口
4. dns包 五层 设置 `dnsid` 不会递归查询 查询域名的数量为1 添加域名并设置为A类型
5. 计算udp的校验和后把几个层的数据合并在一起 最后赋值到buf里

```
_ = udp.SetNetworkLayerForChecksum(ip)
buf := gopacket.NewSerializeBuffer()
err := gopacket.SerializeLayers(
    buf,
    gopacket.SerializeOptions{
        ComputeChecksums: true, // automatically compute checksums
        FixLengths:       true,
    },
    eth, ip, udp, dns,
)
```

6. 最后通过pcap句柄发包 `handle.WritePacketData(buf.Bytes())`

```
func send(domain string, dnsname string, ether *device.EtherTable, dnsid uint16, freeport uint16, handle *pcap.Ha
	DstIp := net.ParseIP(dnsname).To4()
	eth := &layers.Ethernet{
		SrcMAC:       ether.SrcMac.HardwareAddr(),
		DstMAC:       ether.DstMac.HardwareAddr(),
		EthernetType: layers.EthernetTypeIPv4,
	}
	// Our IPv4 header
	ip := &layers.IPv4{
		Version:    4,
		IHL:        5,
		TOS:        0,
		Length:     0, // FIX
		Id:         0,
		Flags:      layers.IPv4DontFragment,
		FragOffset: 0,
		TTL:        255,
		Protocol:   layers.IPProtocolUDP,
		Checksum:   0,
		SrcIP:      ether.SrcIp,
		DstIP:      DstIp,
	}
	// Our UDP header
	udp := &layers.UDP{
		SrcPort: layers.UDPPort(freeport),
		DstPort: layers.UDPPort(53),
	}
	// Our DNS header
	dns := &layers.DNS{
		ID:      dnsid,
		QDCount: 1,
		//RD:      true, //递归查询标识
	}
	dns.Questions = append(dns.Questions,
		layers.DNSQuestion{
			Name:  []byte(domain),
			Type:  layers.DNSTypeA,
			Class: layers.DNSClassIN,
		})
	// Our UDP header
	_ = udp.SetNetworkLayerForChecksum(ip)
	buf := gopacket.NewSerializeBuffer()
	err := gopacket.SerializeLayers(
		buf,
		gopacket.SerializeOptions{
			ComputeChecksums: true, // automatically compute checksums
			FixLengths:       true,
		},
		eth, ip, udp, dns,
	)
```

```
        if err != nil {
                gologger.Warningf("SerializeLayers faild:%s\n", err.Error())
        }
        err = handle.WritePacketData(buf.Bytes())
        if err != nil {
                gologger.Warningf("WritePacketDate error:%s\n", err.Error())
        }
}
```

处理结果并输出 runner.handleResult

这里就可以控制输出结果和 将结果写到文件

1. 获取窗口的宽度 `core.GetWindowWith()`
2. 创建输出文件的句柄 `r.options.Output`
3. 从 `runner.recver` 获取 `runner.recvChanel` 的结果数据
   如果设置 `onlyDomain` msg为域名信息否则加上解析后的ip信息
4. 域名验证的结果输出到控制台 `Silentf`
5. 任务的信息输出到控制台 `runner.PrintStatus`
6. 将结果写到文件里

详细的见如下代码

```
func (r *runner) handleResult(ctx context.Context) {
	var isWrite bool = false
	var err error
	var windowsWidth int

	if r.options.Silent {
		windowsWidth = 0
	} else {
		windowsWidth = core.GetWindowWith()
	}

	if r.options.Output != "" {
		isWrite = true
	}
	var foutput *os.File
	if isWrite {
		foutput, err = os.OpenFile(r.options.Output, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0664)
		if err != nil {
			gologger.Errorf("写入结果文件失败: %s\n", err.Error())
		}
	}
	onlyDomain := r.options.OnlyDomain
	notPrint := r.options.NotPrint
	for result := range r.recver {
		var content []string
		var msg string
		content = append(content, result.Subdomain)

		if onlyDomain {
			msg = result.Subdomain
		} else {
			for _, v := range result.Answers {
				content = append(content, v.String())
			}
			msg = strings.Join(content, " => ")
		}
		if !notPrint {
			screenWidth := windowsWidth - len(msg) - 1
			if !r.options.Silent {
				if windowsWidth > 0 && screenWidth > 0 {
					gologger.Silentf("\r%s% *s\n", msg, screenWidth, "")
				} else {
					gologger.Silentf("\r%s\n", msg)
				}
				// 打印一下结果,可以看得更直观
				r.PrintStatus()
			} else {
				gologger.Silentf("%s\n", msg)
			}
		}
```

```
                if isWrite {
                        w := bufio.NewWriter(foutput)
                        _, err = w.WriteString(msg + "\n")
                        if err != nil {
                                gologger.Errorf("写入结果文件失败.Err:%s\n", err.Error())
                        }
                        _ = w.Flush()
                }
        }
}
```

runner.PrintStatus 任务情况输出

每来一次结果就会输出 任务情况

还有在 `RunEnumeration` 定时器 每1s也会输出一次

```
func (r *runner) PrintStatus() {
        queue := r.hm.Length()
        tc := int(time.Since(r.startTime).Seconds())
        gologger.Printf("\rSuccess:%d Send:%d Queue:%d Accept:%d Fail:%d Elapsed:%ds", r.successIndex, r.sendInde
}
```

通过定时器和状态表阻塞主进程

```
        t := time.NewTicker(1 * time.Second)
        defer t.Stop()
        for {
                select {
                case <-t.C:
            // 1s 输出一次任务情况
                        r.PrintStatus()
                        if isLoadOver {
                // 状态表都空了就任务扫描完毕 结束阻塞
                                if r.hm.Length() == 0 {
                                        gologger.Printf("\n")
                                        gologger.Infof("扫描完毕")
                                        return
                                }
                        }
                case <-r.fisrtloadChanel:
                        go r.retry(ctx) // 遍历hm，依次重试
                        isLoadOver = true
                }
        }
}
```

重试 runner.retry

1. 获取当前时间 遍历状态表如果重试的次数大于最大重试次数了删除这条 失败次数+1
2. 否则 如果上次发送时间与当前时间对比达到了超时时间再次将域名发送到 `runner.sender` 队列
3. sleep 1k纳秒

```
func (r *runner) retry(ctx context.Context) {
        for {
                // 循环检测超时的队列
                now := time.Now()
                r.hm.Scan(func(key string, v statusdb.Item) error {
                        if r.maxRetry > 0 && v.Retry > r.maxRetry {
                                r.hm.Del(key)
                                atomic.AddUint64(&r.faildIndex, 1)
                                return nil
                        }
                        if int64(now.Sub(v.Time)) >= r.timeout {
                                // 重新发送
                                r.sender <- key
                        }
                        return nil
                })
                length := 1000
                time.Sleep(time.Millisecond * time.Duration(length))
        }
}
```

Close

关闭的话关闭 `runner.recver` `runner.sender` `pcap` 句柄 内存状态表

```
func (r *runner) Close() {
        close(r.recver)
        close(r.sender)
        r.handle.Close()
        r.hm.Close()
}
```

verifyCommand 验证功能

可以看到跟枚举的参数是一样的
不同的是 `Method` 模式是 `verify`
还有就是 验证是去掉了 生成测试域名的步骤 认为传递进来的就是直接去发包的域名
其他没有区别

testCommand 测试功能 计算发包速率

获取真正发请求的网卡 传递给 runner.TestSpeed

```
ether := runner.GetDeviceConfig()
runner.TestSpeed(ether)
```

runner.TestSpeed

1. 这里有意思啊 通过设置一个错误的目的mac地址 实现包从经过网卡但是发不出去
2. 获取空闲端口 初始化pcap句柄
3. 测试最大时间15s内发送测试 www.hacking8.com 包的速率

```
func TestSpeed(ether *device.EtherTable) {
        ether.DstMac = device.SelfMac(net.HardwareAddr{0x5c, 0xc9, 0x09, 0x33, 0x34, 0x80}) // 指定一个错误的dstma
        var index int64 = 0
        start := time.Now().UnixNano() / 1e6
        timeSince := int64(15) // 15s
        var dnsid uint16 = 0x2021
        tmpFreeport, err := freeport.GetFreePort()
        if err != nil {
                gologger.Fatalf("freeport error:" + err.Error())
                return
        }
        handle, err := device.PcapInit(ether.Device)
        defer handle.Close()
        if err != nil {
                gologger.Fatalf("初始化pcap失败,error:" + err.Error())
                return
        }
        var now int64
        for {
                send("www.hacking8.com", "1.1.1.2", ether, dnsid, uint16(tmpFreeport), handle)
                index++
                now = time.Now().UnixNano() / 1e6
                tickTime := (now - start) / 1000
                if tickTime >= timeSince {
                        break
                }
                if (now-start)%1000 == 0 && now-start >= 900 {
                        tickIndex := index / tickTime
                        gologger.Printf("\r %ds 总发送:%d Packet 平均每秒速度:%dpps", tickTime, index, tickIndex)
                }
        }
        now = time.Now().UnixNano() / 1e6
        tickTime := (now - start) / 1000
        tickIndex := index / tickTime
        gologger.Printf("\r %ds 总发送:%d Packet 平均每秒速度:%dpps\n", tickTime, index, tickIndex)
}
```

总结

通过学习ksubdomain 更加深了对go的理解和对无状态的理念
也学到了几个有意思的点 dnsid可以控制 对pcap包的使用 漏桶限流 测试数据包速率通过改一个错误mac地址的方式 定时器的使用等等 因为我本身对go语言的理解比较浅显 有些地方可能有错误 欢迎指出来