Hackerman's Hacking Tutorials

The knowledge of anything, since all things have causes, is not acquired or complete unless it is known by its causes. - Avicenna

About Me! | Cheat Sheet | My Clone | How This Website is Built | The Other Guy from Wham! | Search

DEC 3, 2017 - 7 MINUTE READ - COMMENTS - WINDOWS GO

Go and pcaps

I was trying to solve a challenge where the "hidden data" were in ICMP echo payloads. I decided to do it in Go but there were some hiccups on the way. Here are my notes in case

Who am I?

I am Parsia, a security engineer at Electronic Arts.

I write about application security, reverse engineering, Go, cryptography, and (obviously) videogames.

Click on About Me! to know more.

Collections

(most likely) future me or someone else needs to do the same.

Code is in my clone at:

- https://github.com/parsiya/Go-Security/tree/master/pcap-tutorial

gopacket

gopacket is the official Go library for packet manipulation. It also supports reading and writing pcap files through `gopacket/pcap`.

I started following this tutorial from dev dungeon (skipped the capturing part because I have a pcap file in hand). We need to `go get` both `gopacket` and `gopacket/pcap`.

`go get github.com/google/gopacket/pcap` won't work on Windows. I searched around and found an answer on Stack Overflow. I got it to work with some modification.

go get pcap on Windows

1. Install go_amd64 (add go binaries to your PATH). I assume you have a Go environment ready to go.
2. Install MinGW x64 via Win-Builds like I have written about before.
3. Add `C:\mingw\x64\bin` to PATH.
4. Install npcap.
5. Download Winpcap developer's pack and extract it to `C:\`. *So you will have* `C:\WpdPack`.
6. Find `wpcap.dll` and `packet.dll` in `C:\Windows\System32` and copy them somewhere.
7. Run `gendef` (from `MinGW`) on both files.
8. Generate static library files:

Thick Client Proxying

Go/Golang

Blockchain/Distributed Ledgers

Automation

Reverse Engineering

Crypto(graphy)

CTFs/Writeups

WinAppDbg

AWSome.pw - S3 bucket squatting - my very legit branded vulnerability

   - `dlltool --as-flags=--64 -m i386:x86-64 -k --output-lib libwpcap.a --input-def wpcap.def`
   - `dlltool --as-flags=--64 -m i386:x86-64 -k --output-lib libpacket.a --input-def packet.def`
9. Copy `libwpcap.a` and `libpacket.a` to `c:\WpdPack\Lib\x64`.
10. Finally `go get github.com/google/gopacket/pcap`.

Reading pcaps

Following the tutorial I started making code snippets to do what I wanted. Most code is based on the tutorial.

Gopacket godoc and source are also your friends:

- https://godoc.org/github.com/google/gopacket
- https://github.com/google/gopacket

Opening a pcap File

This one shows how to open a pcap file and print the packets.

Opening pcap files - pcap-1.go

```
// Simple go application that opens a pcap file and print the packets

package main

import (
    "fmt"
    "log"

    "github.com/google/gopacket"
    "github.com/google/gopacket/pcap"
)

var (
    pcapFile string = "capt.pcap"
    handle   *pcap.Handle
    err      error
)

func main() {
    // Open file instead of device
    handle, err = pcap.OpenOffline(pcapFile)
    if err != nil {
        log.Fatal(err)
    }
    defer handle.Close()

    // Loop through packets in file
    packetSource := gopacket.NewPacketSource(handle, handle.LinkType())
    for packet := range packetSource.Packets() {
        fmt.Println(packet)
    }
}
```

But I got the following error:

```
$ go run go-pcap-test1.go
2017/12/02 15:48:46 bad dump file format
exit status 1
```

Seems like the original file was in `pcapng` format which is not supported by `gopacket`. Converting the file to `pcap` worked. The new file is named `conv.pcap`.

Setting Filters

Reading everything in the pcap file is good but not what we want. We want to set a filter and only read certain packets. This can be done with `handle.SetBPFFilter(filter)` in which `filter` is a string containing a filter in BPF syntax. We just pass the filter `icmp`:

Setting filters - pcap-2.go

```go
// How to set a filter and only read certain packets from the pcap file

package main

import (
    "fmt"
    "log"

    "github.com/google/gopacket"
    "github.com/google/gopacket/pcap"
)

var (
    pcapFile string = "conv.pcap"
    handle   *pcap.Handle
    err      error
)

func main() {
    // Open file instead of device
    handle, err = pcap.OpenOffline(pcapFile)
    if err != nil {
        log.Fatal(err)
    }
    defer handle.Close()

    // Set filter
    var filter string = "icmp"
    err = handle.SetBPFFilter(filter)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Filter set to ICMP.")

    packetSource := gopacket.NewPacketSource(handle, handle.LinkType())

    for packet := range packetSource.Packets() {
        // Do something with a packet here.
        fmt.Println(packet)
    }
}
```

This code only reads packets of type `icmp`.

Layers

`gopacket` is based on layers. You can get each layer from raw packet data (either from the pcap file or just bytes). Layers are in `github.com/google/gopacket/layers`. We are interested in IPv4 pings so I used `ipLayer := packet.Layer(layers.LayerTypeIPv4)`.

Now `ipLayer` is a `*layers.IPv4` (don't worry about it being a pointer) and we can print it with `fmt.Printf("%+v", ipLayer)` to get:

ipLayer contents

```
&{BaseLayer:{Contents:[69 0 0 114 185 229 64 0 64 1 29 246 172 16 133 141 172
                       16 133 1]
Payload:[8 0 157 204 16 68 10 36 36 83 84 65 82 84 36 36 83 71 70 116 73 72 78
```

```
      111 89 87 53 114 73 72 74 49 98 88 65 115 73 71 53 49 98 71 120 104 73
      71 53 118 98 105 66 104 98 71 78 104 100 72 74 104 73 72 86 48 73 71 82
      108 99 50 86 121 100 87 53 48 73 71 49 112 98 109 108 116 73 71 74 118
      100 87 82 112 10]}
Version:4 IHL:5 TOS:0 Length:114 Id:47589 Flags:DF FragOffset:0 TTL:64
Protocol:ICMPv4 Checksum:7670 SrcIP:172.16.133.141 DstIP:172.16.133.1
Options:[] Padding:[]}
```

Remember those are printed in decimal (bytes are just uint8 in go) and not hex. Personally I prefer printing in hex because it's easier for me to read ASCII-Hex.

IPv4 Layer

At this point you would think we could just do `ipLayer.Payload` and read it but we get:

`ipLayer.Payload undefined (type gopacket.Layer has no field or method Payload)`

But if we print the type with `%T` we get `*layers.IPv4` and when we print it with `%+v` we can see the `Payload` field.

What we have is an interface and the compiler does not know it's going to be populated by `*layers.IPv4` at runtime. We need to cast the packet to `*layers.IPv4` manually. Then we can access `Payload`:

Casting ipLayer to ip

```
ip, _ := ipLayer.(*layers.IPv4)

fmt.Println(ip.Payload)
fmt.Println(len(ip.Payload))
fmt.Println(string(ip.Payload))
```

Which results in

Contents of ip

```
[8 0 157 204 16 68 1 0 36 36 83 84 65 82 84 36 36 83 71 70 116 73 72 78 111 89
87 53 114 73 72 74 49 98 88 65 115 73 71 53 49 98 71 120 104 73 71 53 118 98 105
66 104 98 71 78 104 100 72 74 104 73 72 86 48 73 71 82 108 99 50 86 121 100 87
53 48 73 71 49 112 98 109 108 116 73 71 74 118 100 87 82 112 10]

94

[garbage] $$START$$SGFtIHNoYW5rIHJ1bXAsIG51bGxhIG5vbiBhbGNhdHJhIHV0IGRlc2VydW50
          IG1pbmltIGJvdWRp
```

For more info see section Pointers to Known Layers in gopacket docs.

So we mostly got everything, the payload is some headers and then base64 encoded data. We could just discard the first 8 (header) + 9 (`$$START$$`) and grab what we want. But let's do things properly.

Creating an ICMP Message in Go

We can create an `icmp` message from the IPv4 layer payload.

First we need `go get golang.org/x/net/icmp` and then:

Creating an icmp message from IP payload

```
const (
    ProtocolICMP     = 1  // Internet Control Message
    ProtocolIPv6ICMP = 58 // ICMP for IPv6
)

...

msg, err := icmp.ParseMessage(ProtocolICMP, ip.Payload)
```

`ProtocolICMP` and `ProtocolIPv6ICMP` are defined in `golang.org/x/net/internal/iana`. It's an internal package and we cannot use it directly. Instead I have copied the constants directly in my code.

The result is [*icmp.Message]():

icmp.Message struct

```
type Message struct {
	Type     Type        // type, either ipv4.ICMPType or ipv6.ICMPType
	Code     int         // code
	Checksum int         // checksum
	Body     MessageBody // body
}
```

We are interested in `Body` of type [MessageBody]() which is again an interface. If we print the value and type we get:

ip.Body

```
&{ID:4164 Seq:256
  Data:[36 36 83 84 65 82 84 36 36 83 71 70 116 73 72 78 111 89 87 53 114 73 72
        74 49 98 88 65 115 73 71 53 49 98 71 120 104 73 71 53 118 98 105 66 104
        98 71 78 104 100 72 74 104 73 72 86 48 73 71 82 108 99 50 86 121 100 87
        53 48 73 71 49 112 98 109 108 116 73 71 74 118 100 87 82 112 10]}

*icmp.Echo
```

Getting ICMP Payload

But again we need to cast it to `*icmp.Echo` before we can get the `Data` field which contains the payload.

Casting ip.Body to *icmp.Echo

```go
if body, err := msg.Body.(*icmp.Echo); err {
    // Now we can access Body.Data
    fmt.Println(string(body.Data))
}
```

Now we have the payload:

```
$$START$$SGFtIHNoYW5rIHJ1bXAsIG51bGxhIG5vbiBhbGNhdHJhIHV0IGRlc2VydW50IG1pbmltIGJvdWRp
```

This is base64 encoded and we can decode it after removing `$$START$$`:

- `Ham shank rump, nulla non alcatra ut deserunt minim boudi`

The rest is easy. Complete code for this section is in `pcap-3.go`.

Posted by Parsia • Dec 3, 2017 • Tags: pcap

"Hacking" Car Mechanic Simulator 2015

Windows XP 32-bit SP3 Virtual Machines

0 Comments Parsiya 1 Login

♡ Recommend 1 Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS