PEC 2 Security Analyses - Part 1 Securing the BGPv4

Máster Interuniversitario de Seguridad de las TIC

Advisor: Joan Borrell Viader

Student: José María Foces Vivancos

June 2018

Contents

1. Introduction 2

1.1. Objectives 2

2. Defenses 3

2.1. Speaker 3

2.1.1. Linux Kernel Networking 3

2.1.2. Firewall 4

2.2. Session 6

2.2.1. Transport Layer 6

2.2.2. GTSM 13

2.3. Routing 16

2.3.1. Prefix Filters 16

2.3.2. BGP Route Flap Dampening 22

2.3.3. Maximum Prefixes on a Peering 24

2.3.4. AS Path Filtering 27

2.3.5. Next-Hop Filtering 30

3. Attacks over the whole network 34

3.1. Prefix and Sub-prefix hijacks 35

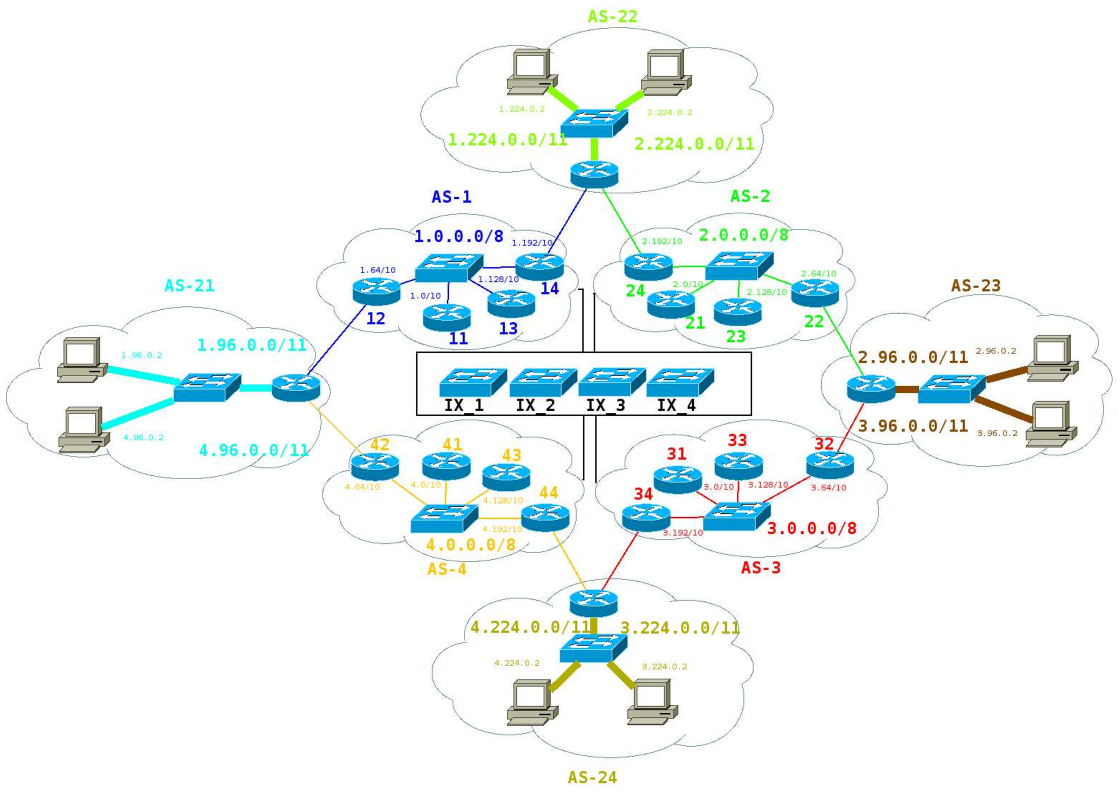
4. Bibliography 40

1. Introduction

This document contains the results of the analyses of the most popular defenses implemented currently to protect BGPv4 speakers. Defenses that involve Secure Inter-Domain Routing are out of the scope of this document.

The environment used is exactly the environment defined at (Securing The BGPv4: Working Environment).

The structure is shown below:



In particular cases, the configuration changed to adapt to the needs of testing specific security mechanisms.

1.1. Objectives

The main objective is to illustrate the security improvement provided by protections and configurations explained at (RFC7454 - BGP Operations and Security), in front of some attacks.

While doing it, the automated generation of this environment is improved to integrate the explained protections. For each security measure a new environment is generated. This way the security of the whole network is improved step by step.

2. Defenses

Defenses can be divided in three groups:

- Speaker defenses: that involves security measures implemented at the router to protect itself.
- Session defenses: that group the protections implemented at transport or network layers to secure route exchanges.
- Routing defenses: that is composed by policies that support the decision to update the routing table or not

2.1. Speaker

In this section we explain security measures implemented to harden the speaker. The level of detail is low. Considering TFM objectives no attacks are performed against Linux Kernel or Firewall.

The main topics are Linux Kernel Networking and Firewalling and main information sources to build this section have been (Linux Kernel Documentation) and (Ubuntu - Kernel Security Settings).

2.1.1. Linux Kernel Networking

This section defines kernel configuration applied to prevent some attacks that would cause both router malfunction or network issues.

ICMP

net.ipv4.icmp_echo_ignore_broadcasts has been enabled to avoid answering to ICMP echo requests when destination address is broadcast. Therefore, that prevents anyone from using this device to perform source address-based DDoS attacks. At least for ICMP based ones.

net.ipv4.icmp_ratelimit & net.ipv4.icmp_ratemask respectively, they take values of 20 and 88090. Both settings combined, limit the rate of ICMP messages that may be sent in one second. The effect is that it won't send more than 5 ICMP messages per second of each of the following types: Echo Reply, Destination Unreachable, Source Quench, Time Exceeded, Parameter Problem, Timestamp Reply, Information Reply. That sets a maximum of 35 ICMP packets per second.

net.ipv4.icmp_ignore_bogus_error_responses avoid logging bogus ICMP error responses. This would lead to fill up the disk with useless log entries.

net.ipv4.conf.all.secure_redirects it's enabled by default. But it's good to ensure activation, since it prevents hijacking of routing paths, critical on this context.

net.ipv4.conf.all.shared_media it's enabled by default. However, it's good to ensure activation since it disables secure redirects.

TCP

net.ipv4.tcp_synack_retries This is set to 2 (default is 5). The purpose is double, avoid sending TCP SYNACK responses during a SYN Flood attack and to limit the time that resources supporting this connection establishment remain allocated.

net.ipv4.tcp_syn_retries has been limited to 2 (instead of default value of 6) despite not having an impact directly on security. This way we can assume that we have a network issue, if two SYNs have no response.

net.ipv4.tcp_syncookies is enabled by default. Improves the behavior of the system when the TCP SYN Queue is full, by not attempting to introduce another entry but discarding and delaying the resource allocation to the reception of the TCP-ACK. After that, the system can rebuild the SYN Queue entry from TCP's sequence numbers.

net.ipv4.tcp_rfc1337 is disabled by default. It changes the behavior of the system when closing TCP connections and prevents from TCP TIME-WAIT state hazards.

net.ipv4.tcp_max_syn_backlog limits the size of the SYN Queue. Default value is 8192. Considering limitations of simulations, this value is set to 1024 that is enough for these test cases.

net.ipv4.tcp_window_scaling enabled by default. However, it's recommended to disable this since it enlarges the TCP Window size, and therefore, it eases TCP-RST attacks. Concretely, this is critical on this context, since BGPv4 uses long lived connections. It does not provide full protection against this kind of attack but, it may be hard to successfully execute it.

2.1.2. Firewall

This section defines firewall configuration applied to mitigate or prevent certain types of attacks in concrete contexts.

As explained in (RFC7454 - BGP Operations and Security) and (RFC6192 - Plane, Protecting the Router Control), access control lists must be defined to protect the management and route exchange (AKA control) networks. Both are usually referred as planes.

In addition, (RFC3704 - Ingress Filtering for Multihomed Networks) and (RFC2827 - Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing) have been considered.

Therefore, the first thing that should be considered to prevent unauthorized access or service disruption from forward to management or control planes is to decide what is legitimate traffic. Further steps may involve rate limiting and limiting maximum TCP connections opened from the router on the control and management planes.

For this concrete case study, it's trivial to filter legitimate traffic, since all routers speak BGP through (IANA - IPv4 Address Space) private ranges. Rate limits and maximum connections opened have been applied but just for testing purposes. Attacks won't be executed against these defenses.

To implement these ACLs, we use Nftables, the replacement for Iptables. The configuration is comfortable and easy to understand. The project is still on development. The version used is 0.8.3 in conjunction with Linux Kernel version 4.16.0. Both allow to set up FIB (Forwarding Information Base) queries to implement reverse path on the firewall. Performing this task on the firewall improves reverse path filters flexibility. It's possible to configure reverse path filtering directly over kernel's configuration (net.ipv4.conf.<iface>.rp_filter) but it lacks flexibility. This way more precise filters can be applied.

The firewall configuration is shown below:

```
#!/usr/sbin/nft -f
flush ruleset
define fwd_p_ports = { 2115, 5001, 80 };
define ctr_p_ports = { 179 }
define mgmt_if = mgmt;
define trust_ifs = { lo, mgmt };
table inet filter {
    set blackhole {
        type ipv4_addr; flags interval;
        elements  =  {  0.0.0.0/8,  127.0.0.0/8,  169.254.0.0/16,  192.0.2.0/24,  240.0.0.0/5,
248.0.0.0/5 }
    }
    set ctr_cidr {
        type ipv4_addr; flags interval;
        elements = { 10.0.0.0/8, 172.16.0.0/16 }
    }
    chain input {
        type filter hook input priority 0;policy drop;
        ## Early accept mgmt and lo inputs.
        iifname $trust_ifs accept;
        ## Early accept already established connections
        ct state established,related accept;
        ## Protect lo
        ip daddr == 127.0.0.0/8 iifname != lo drop;
        ## allow traceroute
        ip ttl 1 accept;
        ## Accept max 10 pings per second.
        icmp type echo-request limit rate 10/second accept;
        ## Accept connections to test ports
        ct state new tcp dport $fwd_p_ports limit rate 10/second accept;
        ## Allow only TCP to BGP port from private IPv4 addresses.
        ip saddr @ctr_cidr ct state new tcp dport $ctr_p_ports limit rate 10/second accept;
    }
    chain forward {
        type filter hook forward priority 0;
        policy accept;
        # Avoid to route private ranges.
        ip daddr @blackhole drop;
    }
    chain output {
        type filter hook output priority 0;
        policy accept;
    }
}
table ip nat{
    chain prerouting {
        type filter hook prerouting priority 0;
        # Protect management plane: Forbid traffic that comes to the router when destination
address does not match interface address where it's arriving.
        ip ttl != 1 iifname $trust_ifs fib daddr . iif type != { local, broadcast, multicast }
drop;
    }
}
```

2.2. Session

This section is mainly oriented to show protections' performance against certain attacks, applicable to communication channels between peers exchanging routes through BGPv4.

As stated at (RFC7454 - BGP Operations and Security), the following security measures should be applied to protect BGPv4 Sessions: Transport Layer and GTSM. While the latter can be considered complementary, in certain cases, with speaker protections explained on previous section involving. It provides additional barrier that insert another protection layer to the BGPv4 speaker.

2.2.1. Transport Layer

BGP works over TCP. Therefore, successful attacks to TCP connections are also applicable to BGP Sessions, since the protocol is not oriented to protect sessions by itself.

There are two TCP extensions to provide origin guarantee of packets exchanged through a TCP connection established between two peers at transport layer. They are TCP-MD5 and TCP-AO, respectively defined at (RFC2385 - Protection of BGP Sessions via the TCP MD5 Signature Option) and (RFC5925 - The TCP Authentication Option). While TCP-AO provides stronger protection, the most popular is TCP-MD5 since TCP-AO is not supported by the equipment deployed currently. For example, Linux lacks support for TCP-AO nowadays.

Considering that, this section only focuses on TCP-MD5 extension.

TCP-MD5 Signature

TCP-MD5 Signature is an extension to the TCP protocol that provides origin guarantee to both peers communicating through it. It requires a password to be set and used at both connection's sides.

Therefore, routers on the environment have been configured to use this TCP extension when communicating with other peers. The configuration directive is as follows:

```
neighbor 10.2.2.2 password JMFVTFM
```

A weak password has been set to speed up tests performed against this security measure, since it involves cracking.

As stated on (RFC2385 - Protection of BGP Sessions via the TCP MD5 Signature Option), the input to the hash function (MD5) is (obviously, order means):

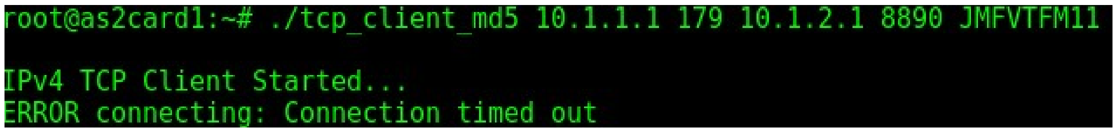
1. the TCP pseudo-header (in the order: source IP address, destination IP address, zero-padded protocol number, and segment length)
2. the TCP header, excluding options, and assuming a checksum of zero
3. the TCP segment data (if any)
4. an independently-specified key or password, known to both TCPs and presumably connection-specific

Therefore, an attacker, that have access to the communication media, can sniff traffic and after that, perform a brute force attack to discover the 4th part of the input to the MD5 hash function.

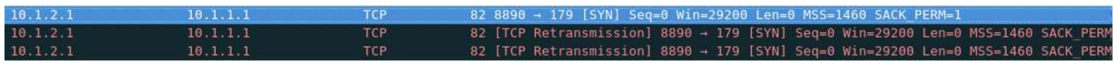
We analyzed the socket behavior in both cases good and bad passwords. As expected, the connection is never established unless the good password is set. The TCP SYN packets are dropped at the server socket side. To show it on the testing environment, TCPMD5 Signature protection is enabled on BGPv4 daemons as shown before.

We use the examples to enable TCPMD5Signature extension on a Linux client socket from (TCPMD5 Signature - Socket Programing Examples on Linux).

A client attempting to establish a TCP Connection without the good password:



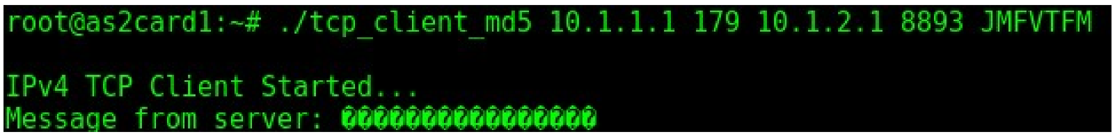
```
root@as2card1:~# ./tcp_client_md5 10.1.1.1 179 10.1.2.1 8890 JMFVTFM11

IPv4 TCP Client Started...
ERROR connecting: Connection timed out
```



```
10.1.2.1          10.1.1.1          TCP          82 8890 → 179 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1
10.1.2.1          10.1.1.1          TCP          82 [TCP Retransmission] 8890 → 179 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM
10.1.2.1          10.1.1.1          TCP          82 [TCP Retransmission] 8890 → 179 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM
```

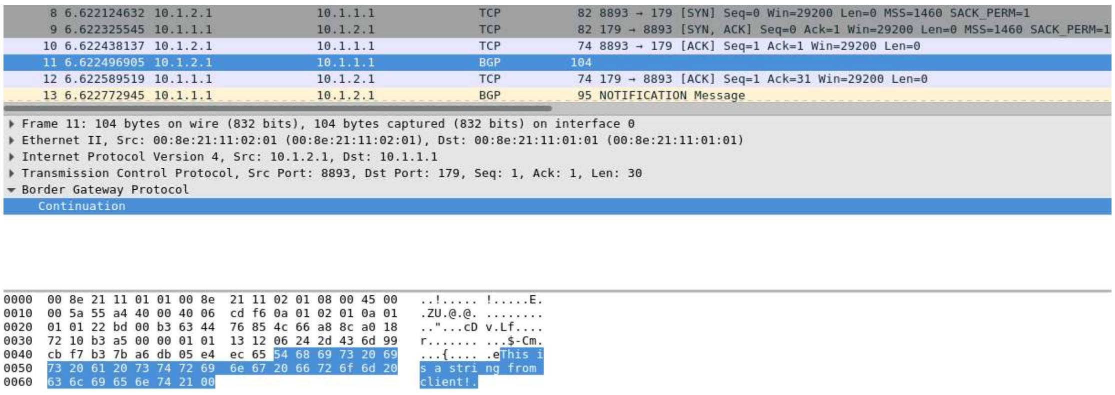
The packets are discarded by the foreign peer and the TCP connection is never established.

With the correct password:



```
root@as2card1:~# ./tcp_client_md5 10.1.1.1 179 10.1.2.1 8893 JMFVTFM

IPv4 TCP Client Started...
Message from server: ����������������
```

The TCP connection is correctly established:



```
  8 6.622124632 10.1.2.1         10.1.1.1         TCP        82 8893 → 179 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1
  9 6.622325545 10.1.1.1         10.1.2.1         TCP        82 179 → 8893 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1
 10 6.622438137 10.1.2.1         10.1.1.1         TCP        74 8893 → 179 [ACK] Seq=1 Ack=1 Win=29200 Len=0
 11 6.622496905 10.1.2.1         10.1.1.1         BGP       104
 12 6.622589519 10.1.1.1         10.1.2.1         TCP        74 179 → 8893 [ACK] Seq=1 Ack=31 Win=29200 Len=0
 13 6.622772945 10.1.1.1         10.1.2.1         BGP        95 NOTIFICATION Message
▸ Frame 11: 104 bytes on wire (832 bits), 104 bytes captured (832 bits) on interface 0
▸ Ethernet II, Src: 00:8e:21:11:02:01 (00:8e:21:11:02:01), Dst: 00:8e:21:11:01:01 (00:8e:21:11:01:01)
▸ Internet Protocol Version 4, Src: 10.1.2.1, Dst: 10.1.1.1
▸ Transmission Control Protocol, Src Port: 8893, Dst Port: 179, Seq: 1, Ack: 1, Len: 30
▾ Border Gateway Protocol
    Continuation

0000  00 8e 21 11 01 01 00 8e  21 11 02 01 08 00 45 00   ..!..... !.....E.
0010  00 5a 55 a4 40 00 40 06  cd f6 0a 01 02 01 0a 01   .ZU.@.@. ........
0020  01 01 22 bd 00 b3 63 44  76 85 4c 66 a8 8c a0 18   .."...cD v.Lf....
0030  72 10 b3 a5 00 00 01 01  13 12 06 24 2d 43 6d 99   r....... ...$-Cm.
0040  cb f7 b3 7b a6 db 05 e4  ec 65 54 68 69 73 20 69   ...{.... .eThis i
0050  73 20 61 20 73 74 72 69  6e 67 20 66 72 6f 6d 20   s a stri ng from 
0060  63 6c 69 65 6e 74 21 00                            client!.
```

Attack 1: Brute forcing the password

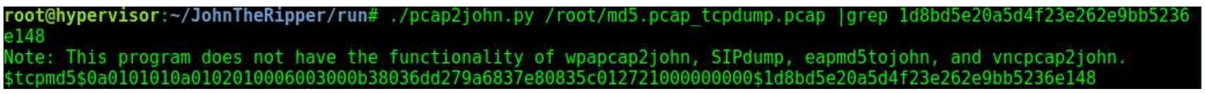
The purpose of this attack is to show that is possible to brute force weak passwords and create awareness about the use of strong ones for protecting BGPv4 Sessions.

There are several tools to perform brute force attacks. For this case the most suitable seems to be (JohnTheRipper). John is a password cracker and support several formats and it has a set of scripts that ease the conversion from almost any format to a valid input for itself.

Concretely, it has a script named pcap2john.py that extracts parts 1,2 and 3 for the MD5 hash function from a PCAP file plus the hash. The output of this script output is valid as input for John.

Taking a capture at IX_1 and forcing reset of router as1card2 produced some BGPv4 traffic, embedded on TCP connections that use TCPMD5 Signature.

Using pcap2john and selecting one packet from the output:

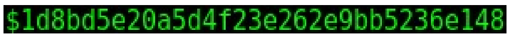


```
root@hypervisor:~/JohnTheRipper/run# ./pcap2john.py /root/md5.pcap_tcpdump.pcap |grep 1d8bd5e20a5d4f23e262e9bb5236
e148
Note: This program does not have the functionality of wpapcap2john, SIPdump, eapmd5tojohn, and vncpcap2john.
$tcpmd5$0a0101010a0102010006003000b38036dd279a6837e80835c012721000000000$1d8bd5e20a5d4f23e262e9bb5236e148
```

The first three parts of the input(hex-encoded) to the hash function:

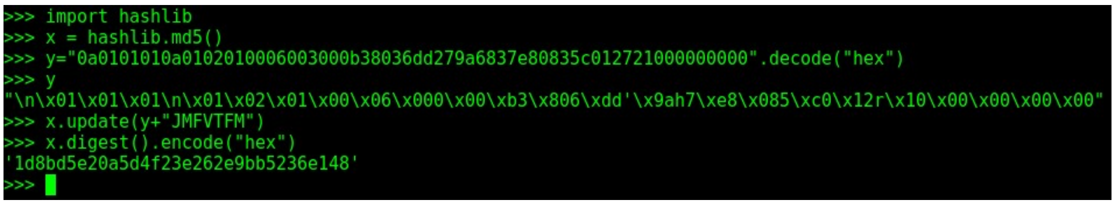
Saddr	Daddr	Prot	Len	Sport	Dport	Seq	Ack	Flags	Wsize	CRC
0a010101	0a010201	0006	0030	00b3	8036	dd279a68	37e80835	c012	7210	00000000

And the hash:



```
$1d8bd5e20a5d4f23e262e9bb5236e148
```

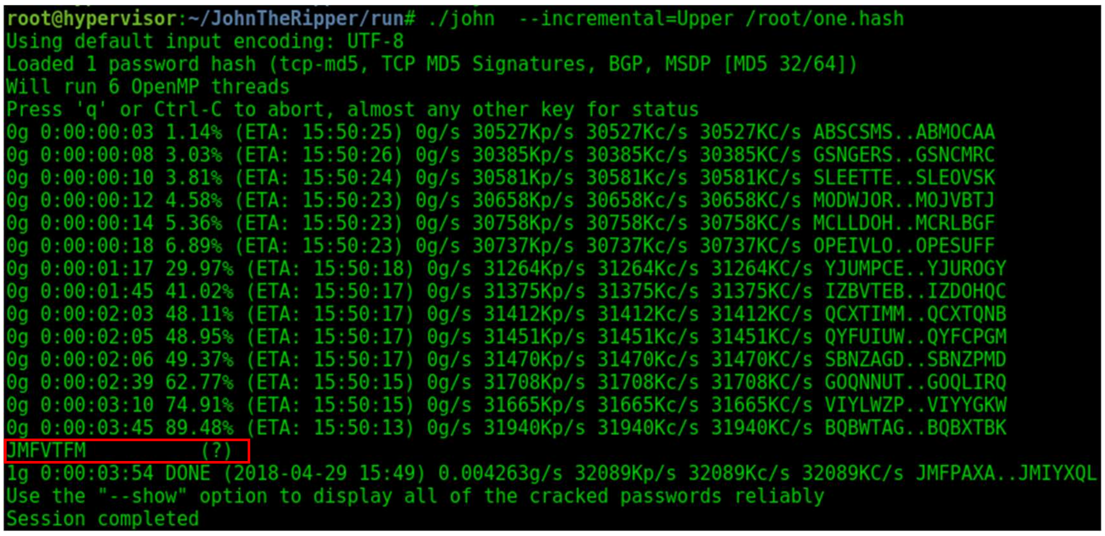
Testing it on a python2.7 shell:



```
>>> import hashlib
>>> x = hashlib.md5()
>>> y="0a0101010a0102010006003000b38036dd279a6837e80835c012721000000000".decode("hex")
>>> y
"\n\x01\x01\x01\n\x01\x02\x01\x00\x06\x000\x00\xb3\x806\xdd'\x9ah7\xe8\x085\xc0\x12r\x10\x00\x00\x00\x00"
>>> x.update(y+"JMFVTFM")
>>> x.digest().encode("hex")
'1d8bd5e20a5d4f23e262e9bb5236e148'
>>> 
```

As can be appreciated, the brute force attack tests all combinations of the password, concatenated at the end of the three first parts of the input.

In this example, we select an incremental attack, that tests all possible combinations of the tail (password used for TCPMD5 Signature) but with only upper-case ASCII characters. It would be hard to obtain a stronger key cracking it just with the CPU, but this attack can be easily parallelized and executed faster with a GPU.



```
root@hypervisor:~/JohnTheRipper/run# ./john  --incremental=Upper /root/one.hash
Using default input encoding: UTF-8
Loaded 1 password hash (tcp-md5, TCP MD5 Signatures, BGP, MSDP [MD5 32/64])
Will run 6 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:03 1.14% (ETA: 15:50:25) 0g/s 30527Kp/s 30527Kc/s 30527KC/s ABSCSMS..ABMOCAA
0g 0:00:00:08 3.03% (ETA: 15:50:26) 0g/s 30385Kp/s 30385Kc/s 30385KC/s GSNGERS..GSNCMRC
0g 0:00:00:10 3.81% (ETA: 15:50:24) 0g/s 30581Kp/s 30581Kc/s 30581KC/s SLEETTE..SLEOVSK
0g 0:00:00:12 4.58% (ETA: 15:50:23) 0g/s 30658Kp/s 30658Kc/s 30658KC/s MODWJOR..MOJVBTJ
0g 0:00:00:14 5.36% (ETA: 15:50:23) 0g/s 30758Kp/s 30758Kc/s 30758KC/s MCLLDOH..MCRLBGF
0g 0:00:00:18 6.89% (ETA: 15:50:23) 0g/s 30737Kp/s 30737Kc/s 30737KC/s OPEIVLO..OPESUFF
0g 0:00:01:17 29.97% (ETA: 15:50:18) 0g/s 31264Kp/s 31264Kc/s 31264KC/s YJUMPCE..YJUROGY
0g 0:00:01:45 41.02% (ETA: 15:50:17) 0g/s 31375Kp/s 31375Kc/s 31375KC/s IZBVTEB..IZDOHQC
0g 0:00:02:03 48.11% (ETA: 15:50:17) 0g/s 31412Kp/s 31412Kc/s 31412KC/s QCXTIMM..QCXTQNB
0g 0:00:02:05 48.95% (ETA: 15:50:17) 0g/s 31451Kp/s 31451Kc/s 31451KC/s QYFUIUW..QYFCPGM
0g 0:00:02:06 49.37% (ETA: 15:50:17) 0g/s 31470Kp/s 31470Kc/s 31470KC/s SBNZAGD..SBNZPMD
0g 0:00:02:39 62.77% (ETA: 15:50:15) 0g/s 31708Kp/s 31708Kc/s 31708KC/s GOQNNUT..GOQLIRQ
0g 0:00:03:10 74.91% (ETA: 15:50:15) 0g/s 31665Kp/s 31665Kc/s 31665KC/s VIYLWZP..VIYYGKW
0g 0:00:03:45 89.48% (ETA: 15:50:13) 0g/s 31940Kp/s 31940Kc/s 31940KC/s BQBWTAG..BQBXTBK
JMFVTFM          (?)
1g 0:00:03:54 DONE (2018-04-29 15:49) 0.004263g/s 32089Kp/s 32089Kc/s 32089KC/s JMFPAXA..JMIYXQL
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

Attack 2: MITM

The purpose of this attack is to show the protection offered by TCPMD5 Signature against MITM attacks.

Therefore, for testing this, TCPMD5 Signature has been disabled to show the impact of a MITM attack where NEXT-HOP gets replaced and after that, enabled it back to show the correct behavior.

With TCPMD5 Signature enabled the victim router notices the MITM attack and invalidates received routes.

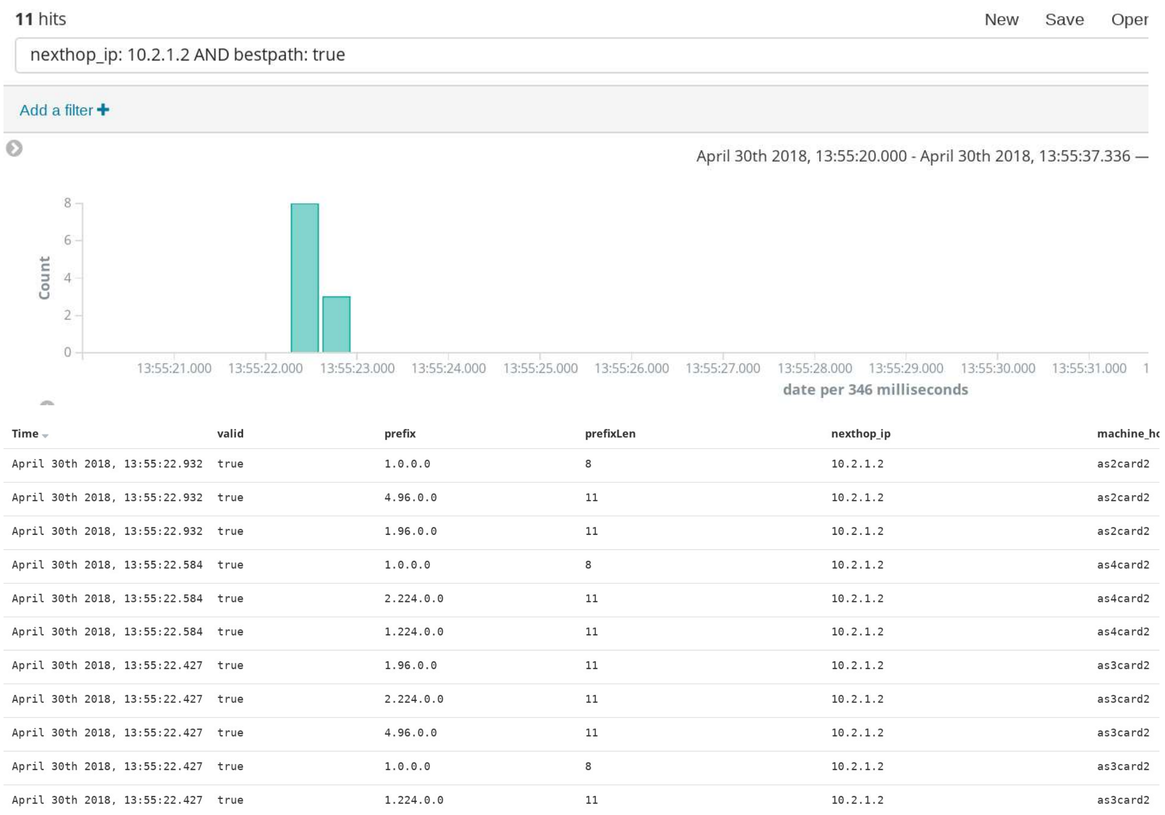
We reset a router three times, the first to show the normal behavior, the second to show the performance of a MITM attack without TCPMD5 Signature and finally the same attack is repeated with TCPMD5 signature enabled.

Normal behavior

Restarting BGPd on As4card2 will produce several route updates. Observing them updates on Kibana, using the filter below:

```
nexthop_ip: 10.2.1.2 AND bestpath: true
```

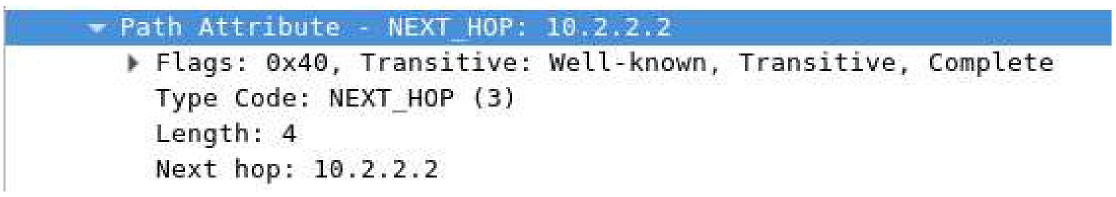
We see that without traffic tampering, a total of 11 best paths going through 10.2.1.2 have been updated.



Time	valid	prefix	prefixLen	nexthop_ip	machine_hc
April 30th 2018, 13:55:22.932	true	1.0.0.0	8	10.2.1.2	as2card2
April 30th 2018, 13:55:22.932	true	4.96.0.0	11	10.2.1.2	as2card2
April 30th 2018, 13:55:22.932	true	1.96.0.0	11	10.2.1.2	as2card2
April 30th 2018, 13:55:22.584	true	1.0.0.0	8	10.2.1.2	as4card2
April 30th 2018, 13:55:22.584	true	2.224.0.0	11	10.2.1.2	as4card2
April 30th 2018, 13:55:22.584	true	1.224.0.0	11	10.2.1.2	as4card2
April 30th 2018, 13:55:22.427	true	1.96.0.0	11	10.2.1.2	as3card2
April 30th 2018, 13:55:22.427	true	2.224.0.0	11	10.2.1.2	as3card2
April 30th 2018, 13:55:22.427	true	4.96.0.0	11	10.2.1.2	as3card2
April 30th 2018, 13:55:22.427	true	1.0.0.0	8	10.2.1.2	as3card2
April 30th 2018, 13:55:22.427	true	1.224.0.0	11	10.2.1.2	as3card2

Without MD5Signature

The attacker replaces NEXT_HOP path attribute by 10.2.1.2 on any packet reaching or leaving as4card2 on IX_2 switch. Therefore, causing traffic convergence on this host.



```
    ▾ Path Attribute - NEXT_HOP: 10.2.2.2
      ▸ Flags: 0x40, Transitive: Well-known, Transitive, Complete
        Type Code: NEXT_HOP (3)
        Length: 4
        Next hop: 10.2.2.2
```

To be able to replace certain patterns on this set of packets we use (Foces Vivancos, Rehtse). This software was implemented some time ago, but it fits perfectly for this case. It takes packets from Netfilter Queue and applies regex-based replacements on certain packages.

Nftables config file:

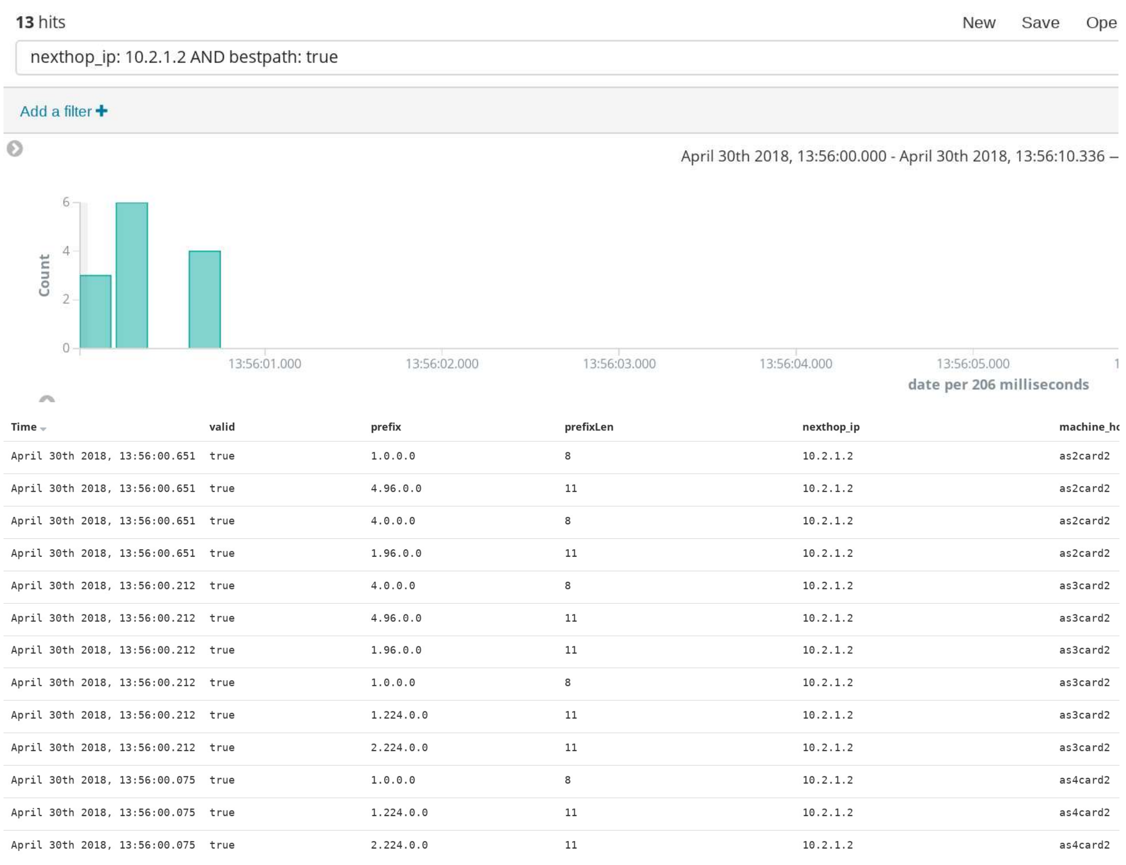
```
#!/usr/sbin/nft -f
flush ruleset
table inet filter {
        chain forward{
                type filter hook forward priority 0;
                iif IX_2 queue num 0 bypass;
        }
```

Rehtse config file:

```
{
        "debuglevel":0,
        "patterns":[
                {
                        "match":{
                                "bpf":"tcp port 179 and host 10.2.4.2",
"regex":"\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\x00
.\\x02.*"
                        },
                        "replacement":{
                                "regex":"(\\x03\\x04\\x0a...)",
                                "replacement":"\\x03\\x04\\x0a\\x02\\x01\\x02"
                        }
                }
        ]
}
```

To simulate the attack Rehtse runs on the hypervisor.

With Rehtse running, BGPd on As4card2 is restarted again. This time a total of 13 best paths going through 10.2.1.2 are shown:



13 hits

nexthop_ip: 10.2.1.2 AND bestpath: true

April 30th 2018, 13:56:00.000 - April 30th 2018, 13:56:10.336 –

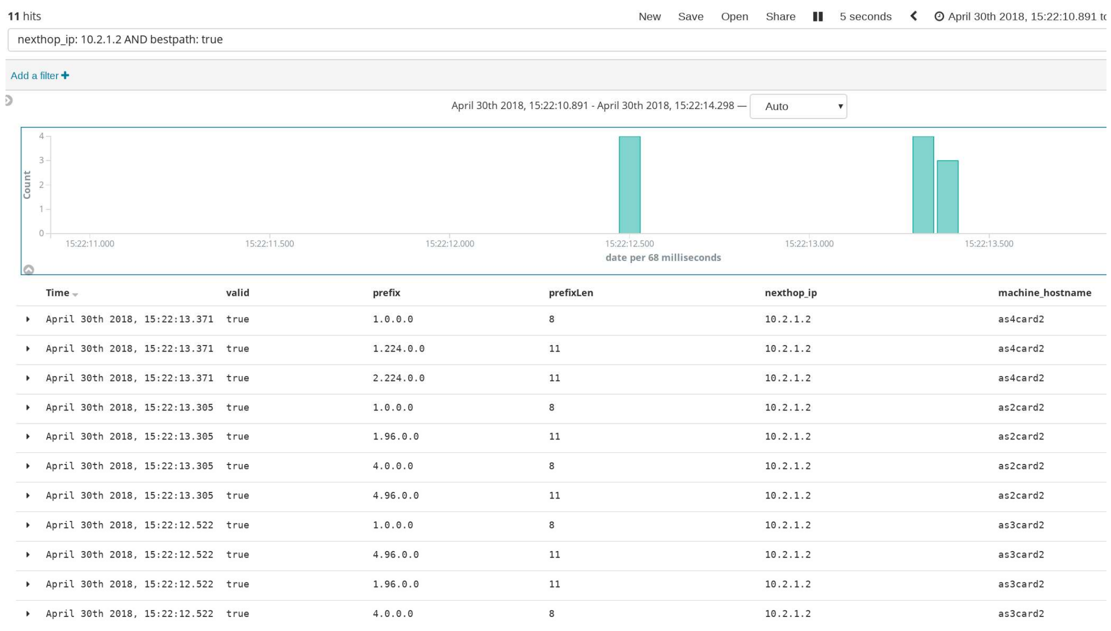
Time	valid	prefix	prefixLen	nexthop_ip	machine_hc
April 30th 2018, 13:56:00.651	true	1.0.0.0	8	10.2.1.2	as2card2
April 30th 2018, 13:56:00.651	true	4.96.0.0	11	10.2.1.2	as2card2
April 30th 2018, 13:56:00.651	true	4.0.0.0	8	10.2.1.2	as2card2
April 30th 2018, 13:56:00.651	true	1.96.0.0	11	10.2.1.2	as2card2
April 30th 2018, 13:56:00.212	true	4.0.0.0	8	10.2.1.2	as3card2
April 30th 2018, 13:56:00.212	true	4.96.0.0	11	10.2.1.2	as3card2
April 30th 2018, 13:56:00.212	true	1.96.0.0	11	10.2.1.2	as3card2
April 30th 2018, 13:56:00.212	true	1.0.0.0	8	10.2.1.2	as3card2
April 30th 2018, 13:56:00.212	true	1.224.0.0	11	10.2.1.2	as3card2
April 30th 2018, 13:56:00.212	true	2.224.0.0	11	10.2.1.2	as3card2
April 30th 2018, 13:56:00.075	true	1.0.0.0	8	10.2.1.2	as4card2
April 30th 2018, 13:56:00.075	true	1.224.0.0	11	10.2.1.2	as4card2
April 30th 2018, 13:56:00.075	true	2.224.0.0	11	10.2.1.2	as4card2

Therefore, the attacker succeeded to increase traffic convergence by just tampering with traffic to one router on IX_2.

With MD5Signature

We follow the same procedure as shown before, shutdown as4card2 enable the MITM attack and start BGPd again.

As shown below, the attacker was unable to tamper the next hop and only packets that had next-hop 10.2.1.2 outgoing from as4card2 were accepted by others in IX_2 and vice versa.


11 hits
New
Save
Open
Share
5 seconds
April 30th 2018, 15:22:10.891 t
Add a filter
April 30th 2018, 15:22:10.891 - April 30th 2018, 15:22:14.298 —
Auto
Count
date per 68 milliseconds

nexthop_ip: 10.2.1.2 AND bestpath: true

Time	valid	prefix	prefixLen	nexthop_ip	machine_hostname
April 30th 2018, 15:22:13.371	true	1.0.0.0	8	10.2.1.2	as4card2
April 30th 2018, 15:22:13.371	true	1.224.0.0	11	10.2.1.2	as4card2
April 30th 2018, 15:22:13.371	true	2.224.0.0	11	10.2.1.2	as4card2
April 30th 2018, 15:22:13.305	true	1.0.0.0	8	10.2.1.2	as2card2
April 30th 2018, 15:22:13.305	true	1.96.0.0	11	10.2.1.2	as2card2
April 30th 2018, 15:22:13.305	true	4.0.0.0	8	10.2.1.2	as2card2
April 30th 2018, 15:22:13.305	true	4.96.0.0	11	10.2.1.2	as2card2
April 30th 2018, 15:22:12.522	true	1.0.0.0	8	10.2.1.2	as3card2
April 30th 2018, 15:22:12.522	true	4.96.0.0	11	10.2.1.2	as3card2
April 30th 2018, 15:22:12.522	true	1.96.0.0	11	10.2.1.2	as3card2
April 30th 2018, 15:22:12.522	true	4.0.0.0	8	10.2.1.2	as3card2

On this case the routes are learned by as4card2 from other routers but not from the tampered session.

2.2.2. GTSM

Generalized TTL Security Mechanism, defined at (RFC5082 - The Generalized TTL Security Mechanism (GTSM)) is a technique to protect two peers exchanging IP datagrams, in general. The protection is based on IP protocol TTL field. As (RFC791 - INTERNET PROTOCOL) states at sections 3.1 and 3.2 the TTL field should be decremented by one by each processor of the packet.

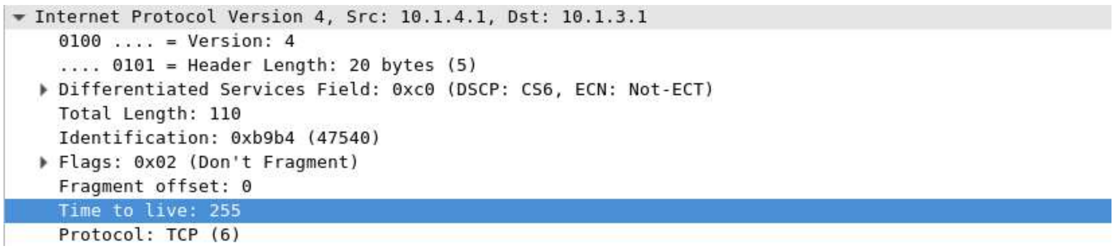
This decrement is the base of traceroute utility, where TTL starts with a value of 1 and is incremented by one. So, an ICMP time-exceeded is returned to the machine tracing the route to a host for each router in the path.

Therefore, the TTL field can be used to get the hop distance between two peers. GTSM is based on that and makes the router to discard packets that do not match the distance filter established. It can be configured as:

```
neighbor 10.1.2.1 ttl-security hops 1
```

When configured this way, it will make the router to only accept packets coming to the TCP socket of the BGPv4 daemon with a TTL of 255.

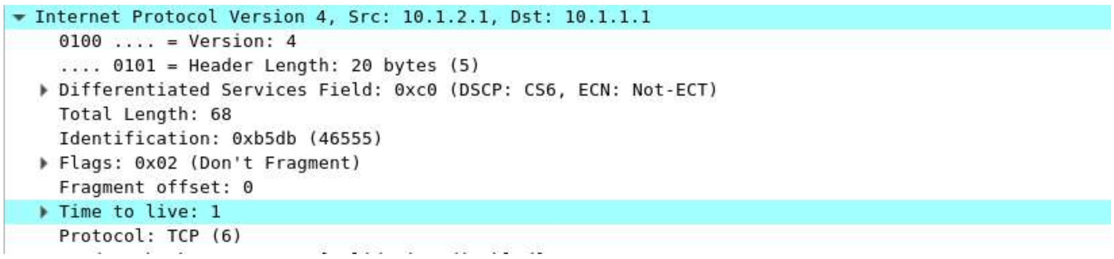
While remaining simple and cheap in terms of CPU cycles, it provides another defense layer for BGPv4 Speakers and Sessions, since it won't be possible to perform attacks against a sessions or peers connected on the same media (no routers in the middle). Unless the attacker has previously hijacked a device connected to the same media. In this case, the attacker, would make this hijacked device to rewrite TTL value with 255 and therefore be able to tamper with the session established. But given the case, one of the least desirable things he would do is to tamper with an established BGPv4 session.


```
▾ Internet Protocol Version 4, Src: 10.1.4.1, Dst: 10.1.3.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    Total Length: 110
    Identification: 0xb9b4 (47540)
  ▸ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 255
    Protocol: TCP (6)
```

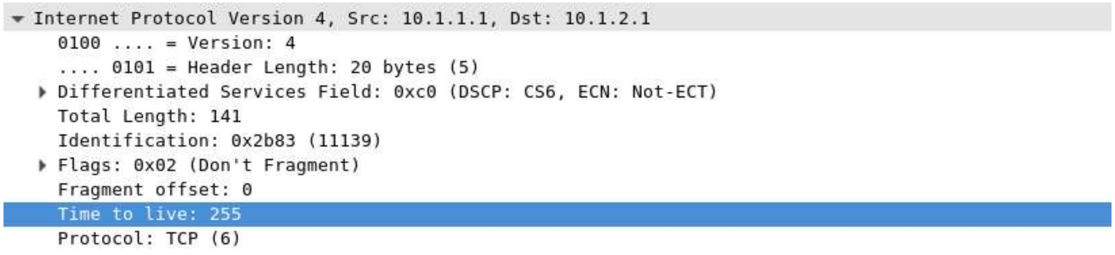
Since the environment created for testing that has both GTSM and TCPMD5 Signature protections enabled, the examples to enable TCPMD5Signature extension on a Linux client socket from (TCPMD5 Signature - Socket Programing Examples on Linux) are used here too.

For testing the impact of that configuration directive, the routers as1card1 and as2card1 have been selected.

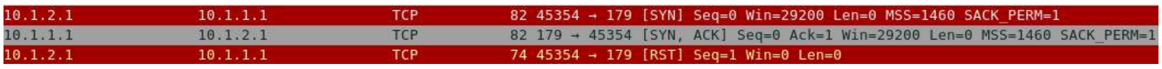
Without GTSM enabled, routers speak BGPv4 over TCP and them, over IP with TTL 1.


```
▾ Internet Protocol Version 4, Src: 10.1.2.1, Dst: 10.1.1.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    Total Length: 68
    Identification: 0xb5db (46555)
  ▸ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
  ▸ Time to live: 1
    Protocol: TCP (6)
```

With GTSM enabled:


```
▾ Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.1.2.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0xc0 (DSCP: CS6, ECN: Not-ECT)
    Total Length: 141
    Identification: 0x2b83 (11139)
  ▸ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 255
    Protocol: TCP (6)
```

When GTSM is disabled on as2card1, the connection is not even established. The TTL used was 1, that bypasses the default net.ipv4.ip_default_ttl, 64:

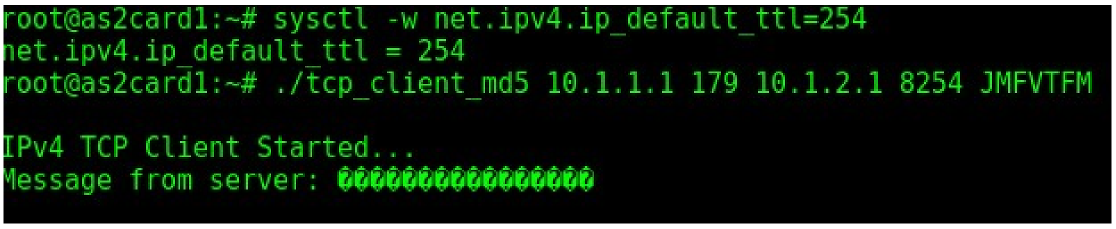

```
10.1.2.1            10.1.1.1            TCP        82 45354 → 179 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1
10.1.1.1            10.1.2.1            TCP        82 179 → 45354 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1
10.1.2.1            10.1.1.1            TCP        74 45354 → 179 [RST] Seq=1 Win=0 Len=0
```

Since attacks against IP protocol are out of the scope of the project, we will test the behavior of this implementation with different TTL values.

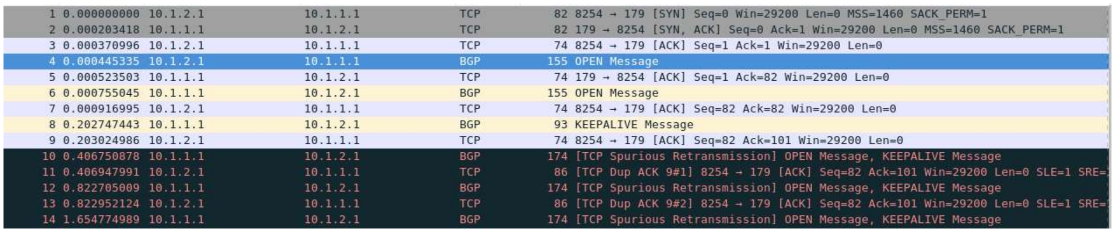
I've modified the TCPMD5 example program to send a BGPv4 OPEN message.

Testing BGPv4 OPEN with TTL 254

Setting TTL to 254 and executing the PoC:



```
root@as2card1:~# sysctl -w net.ipv4.ip_default_ttl=254
net.ipv4.ip_default_ttl = 254
root@as2card1:~# ./tcp_client_md5 10.1.1.1 179 10.1.2.1 8254 JMFVTFM

IPv4 TCP Client Started...
Message from server: �����������������
```

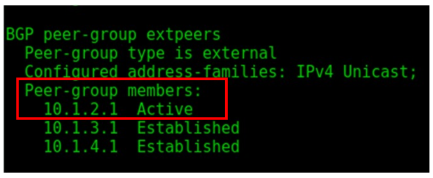


```
 1 0.000000000 10.1.2.1    10.1.1.1    TCP    82 8254 → 179 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1
 2 0.000203418 10.1.1.1    10.1.2.1    TCP    82 179 → 8254 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1
 3 0.000370996 10.1.2.1    10.1.1.1    TCP    74 8254 → 179 [ACK] Seq=1 Ack=1 Win=29200 Len=0
 4 0.000445335 10.1.2.1    10.1.1.1    BGP   155 OPEN Message
 5 0.000523503 10.1.1.1    10.1.2.1    TCP    74 179 → 8254 [ACK] Seq=1 Ack=82 Win=29200 Len=0
 6 0.000755045 10.1.1.1    10.1.2.1    BGP   155 OPEN Message
 7 0.000916995 10.1.2.1    10.1.1.1    TCP    74 8254 → 179 [ACK] Seq=82 Ack=82 Win=29200 Len=0
 8 0.202747443 10.1.1.1    10.1.2.1    BGP    93 KEEPALIVE Message
 9 0.203024986 10.1.2.1    10.1.1.1    TCP    74 8254 → 179 [ACK] Seq=82 Ack=101 Win=29200 Len=0
10 0.406750878 10.1.1.1    10.1.2.1    BGP   174 [TCP Spurious Retransmission] OPEN Message, KEEPALIVE Message
11 0.406947991 10.1.2.1    10.1.1.1    TCP    86 [TCP Dup ACK 9#1] 8254 → 179 [ACK] Seq=82 Ack=101 Win=29200 Len=0 SLE=1 SRE=
12 0.822705009 10.1.1.1    10.1.2.1    BGP   174 [TCP Spurious Retransmission] OPEN Message, KEEPALIVE Message
13 0.822952124 10.1.2.1    10.1.1.1    TCP    86 [TCP Dup ACK 9#2] 8254 → 179 [ACK] Seq=82 Ack=101 Win=29200 Len=0 SLE=1 SRE=
14 1.654774989 10.1.1.1    10.1.2.1    BGP   174 [TCP Spurious Retransmission] OPEN Message, KEEPALIVE Message
```

Using Vtysh to see peer statues:

```
vtysh -c “show bgp peer-groups”
```

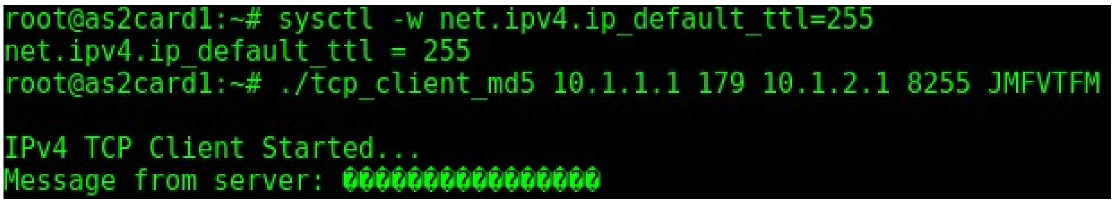
The peer endpoint never transitions to Connect state.



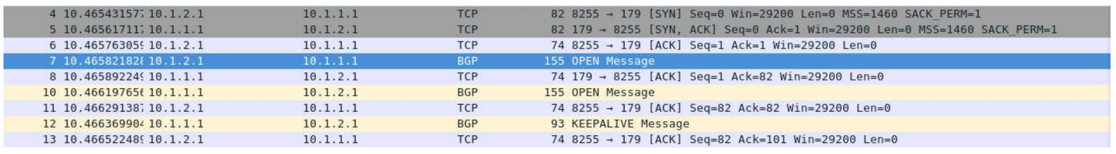
```
BGP peer-group extpeers
  Peer-group type is external
  Configured address-families: IPv4 Unicast;
  Peer-group members:
    10.1.2.1  Active
    10.1.3.1  Established
    10.1.4.1  Established
```

Testing BGPv4 OPEN with TTL 255

Setting TTL to 254 and executing the PoC:



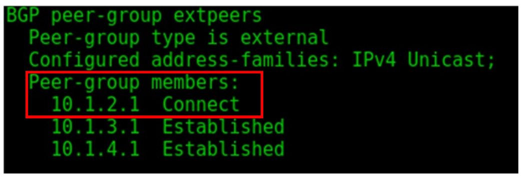
```
root@as2card1:~# sysctl -w net.ipv4.ip_default_ttl=255
net.ipv4.ip_default_ttl = 255
root@as2card1:~# ./tcp_client_md5 10.1.1.1 179 10.1.2.1 8255 JMFVTFM

IPv4 TCP Client Started...
Message from server: �����������������
```



```
 4 10.46543157 10.1.2.1    10.1.1.1    TCP    82 8255 → 179 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1
 5 10.46561711 10.1.1.1    10.1.2.1    TCP    82 179 → 8255 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1
 6 10.46576305 10.1.2.1    10.1.1.1    TCP    74 8255 → 179 [ACK] Seq=1 Ack=1 Win=29200 Len=0
 7 10.46582182 10.1.2.1    10.1.1.1    BGP   155 OPEN Message
 8 10.46589224 10.1.1.1    10.1.2.1    TCP    74 179 → 8255 [ACK] Seq=1 Ack=82 Win=29200 Len=0
10 10.46619765 10.1.1.1    10.1.2.1    BGP   155 OPEN Message
11 10.46629138 10.1.2.1    10.1.1.1    TCP    74 8255 → 179 [ACK] Seq=82 Ack=82 Win=29200 Len=0
12 10.46636990 10.1.1.1    10.1.2.1    BGP    93 KEEPALIVE Message
13 10.46652248 10.1.2.1    10.1.1.1    TCP    74 8255 → 179 [ACK] Seq=82 Ack=101 Win=29200 Len=0
```

```
vtysh -c “show bgp peer-groups”
```

The peer endpoint transitions to Connect state.



```
BGP peer-group extpeers
  Peer-group type is external
  Configured address-families: IPv4 Unicast;
  Peer-group members:
    10.1.2.1  Connect
    10.1.3.1  Established
    10.1.4.1  Established
```

Both TCP connections are almost exactly equal, the difference is that even while as2card1 ACKs the reception of BGPv4 messages OPEN and KEEPALIVE as1card1 never accepts them.

Therefore, as manual states, TTLs indicating hop distance far away than configured are not allowed to become peers. For small TTLs the connection is not even established but with greater ones, the peer configured with GTSM enabled does not accept BGPv4 messages from the other side unless it sends these messages with TTL 255, in this concrete case, with adjacent hosts (hop distance of 1).

2.3. Routing

This section is mainly oriented to show protections' performance against certain attacks, applicable to advertised and accepted routes by BGPv4 speakers. This is one of the most critical aspects when securing BGPv4 routers.

2.3.1. Prefix Filters

As stated on (RFC7454 - BGP Operations and Security), any special purpose or unallocated IANA prefixes should be filtered from being advertised or accepted. In addition, Regional Internet Registries filters should be considered too. The first group is easy to maintain since it's static, but the latter changes over the time and the management load is high. To ease this task there is a tool called (IRRToolSet). That works conformant with (RFC4012 - Routing Policy Specification Language next generation (RPSLng)), to exchange routing policies. With this, the network administrator can keep updated the prefixes lists and who is allowed to advertise them, but this is a wider topic and it's out of this section.

A prefix filter is composed by a network prefix and an action. Optionally, it may include [le|ge] operators. Respectively, they filter lesser or equal and greater or equal prefixes belonging to the provided one.

They are declared as follows:

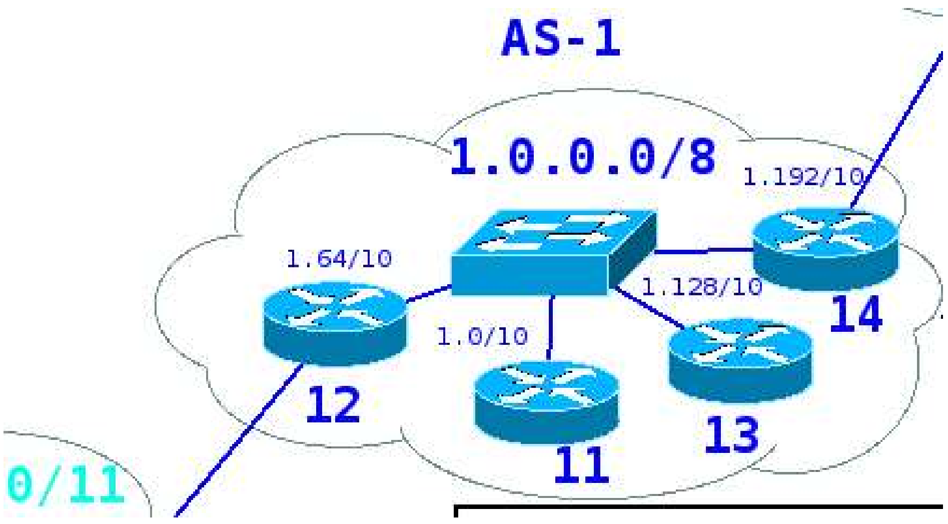
```
ip prefix-list t1-external-adv deny 1.0.0.0/10
ip prefix-list t1-external-adv deny 1.64.0.0/10
ip prefix-list t1-external-adv deny 1.128.0.0/10
ip prefix-list t1-external-adv deny 1.192.0.0/10
ip prefix-list t1-external-adv permit any
ip prefix-list allow-all-adv permit any
```

After that, they should be used to manage advertisements and accepted routes on a peering. This can be done as follows:

```
  address-family ipv4 unicast
    network 1.0.0.0/10
    network 1.0.0.0/8
…
    neighbor extpeers prefix-list t1-external-adv out
```

```
   neighbor extpeers prefix-list allow-all-adv in
 exit-address-family
```

Regarding AS1 structure, each router provides access to a quarter of the address space 1.0.0.0/8. As shown below:



Prefix filters prevent that internal structure of AS1 is advertised outside.

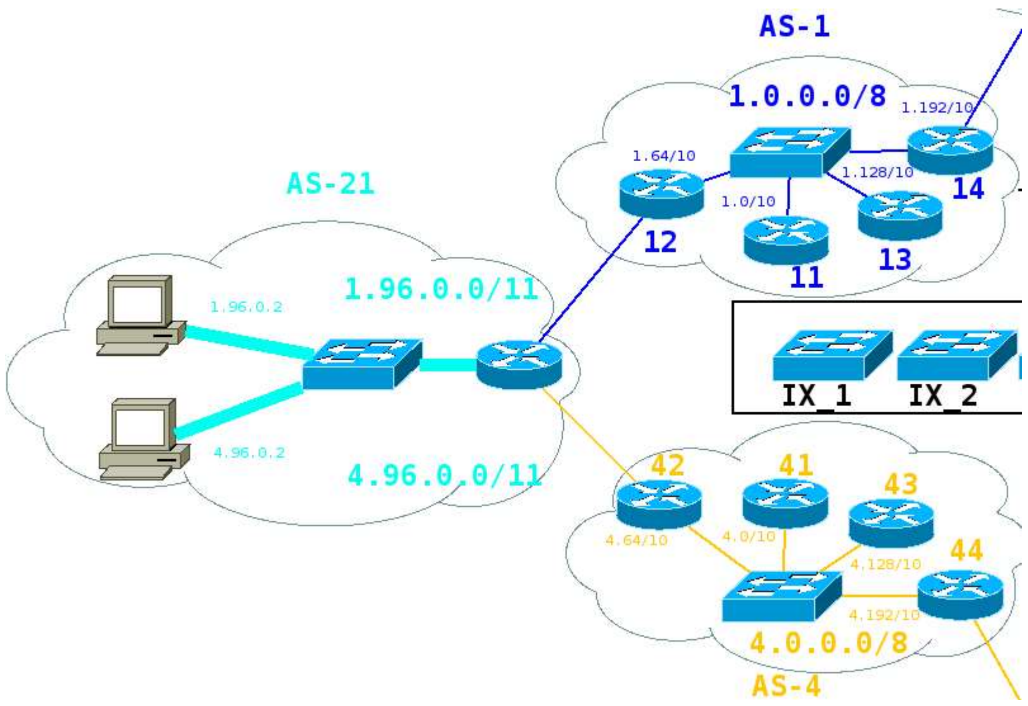
Attack: Route Leak

When configured rigorously, prefix filters prevent routers from accepting or advertising prefixes that should not. Note that this is easy to configure, but hard to maintain considering the frequency of network topology updates.

In a route leak, an AS violates agreed export policies.

To illustrate that, let's consider that AS21 should only advertise prefixes on its own, since it would never want to transit traffic between its providers, AS1 and AS4, the upstream providers.

In addition, and while this example does not show it, it's important to emphasize that both AS1 and AS4 should also filter them internal structure from leaking to them customer AS21.



AS21 prefix filters and neighbors:

```
address-family ipv4 unicast
    network 1.96.0.0/11
    neighbor 10.11.0.2 prefix-list t2-external-adv out
    neighbor 10.11.0.2 prefix-list allow-all-adv in
    network 4.96.0.0/11
    neighbor 10.14.0.2 prefix-list t2-external-adv out
    neighbor 10.14.0.2 prefix-list allow-all-adv in
  exit-address-family
ip prefix-list t2-external-adv permit 1.96.0.0/11
ip prefix-list t2-external-adv permit 4.96.0.0/11
ip prefix-list allow-all-adv permit any
ip prefix-list deny-all-adv deny any
```

On this case, neither AS4 and AS1 know the internal structure of each other. They do not advertise them internal structure to each other, but they have a configuration mistake, they advertise it to AS21. Initially, AS21 is exporting routes as expected and just advertise its' prefixes.

As4card2 and as1card2 configs:

Consider that [4|1] means respectively the configuration on each of them, over the template below. The critical misconfiguration is shown in red:

```
address-family ipv4 unicast
    network [4|1].64.0.0/10
    network [4|1].0.0.0/8
    neighbor 172.16.1[4|1].1 prefix-list ibgp-adv-0 in
    neighbor brothers prefix-list ibgp-adv-1 out
    neighbor brothers next-hop-self
    neighbor 172.16.1[4|1].3 prefix-list ibgp-adv-2 in
    neighbor 172.16.1[4|1].4 prefix-list ibgp-adv-3 in
    neighbor brothers prefix-list allow-all-adv out
    neighbor brothers prefix-list allow-all-adv in
    neighbor extpeers prefix-list t1-external-adv out
    neighbor extpeers prefix-list allow-all-adv in
    neighbor 10.1[4|1].21.1
  exit-address-family
```

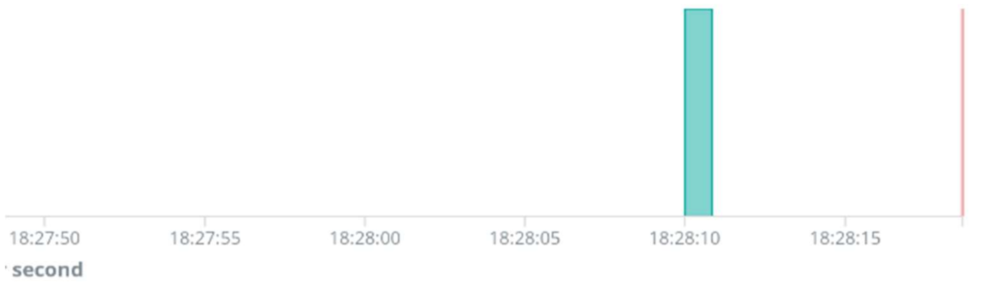
That makes both to accept any route advertised by as21card1. And to advertise the full routing table without filters.

Before we apply any changes, we show the behavior without tampering with AS21 export policies. After that, we show the effect that AS21 produces when violating its normal export policies.

Normal case

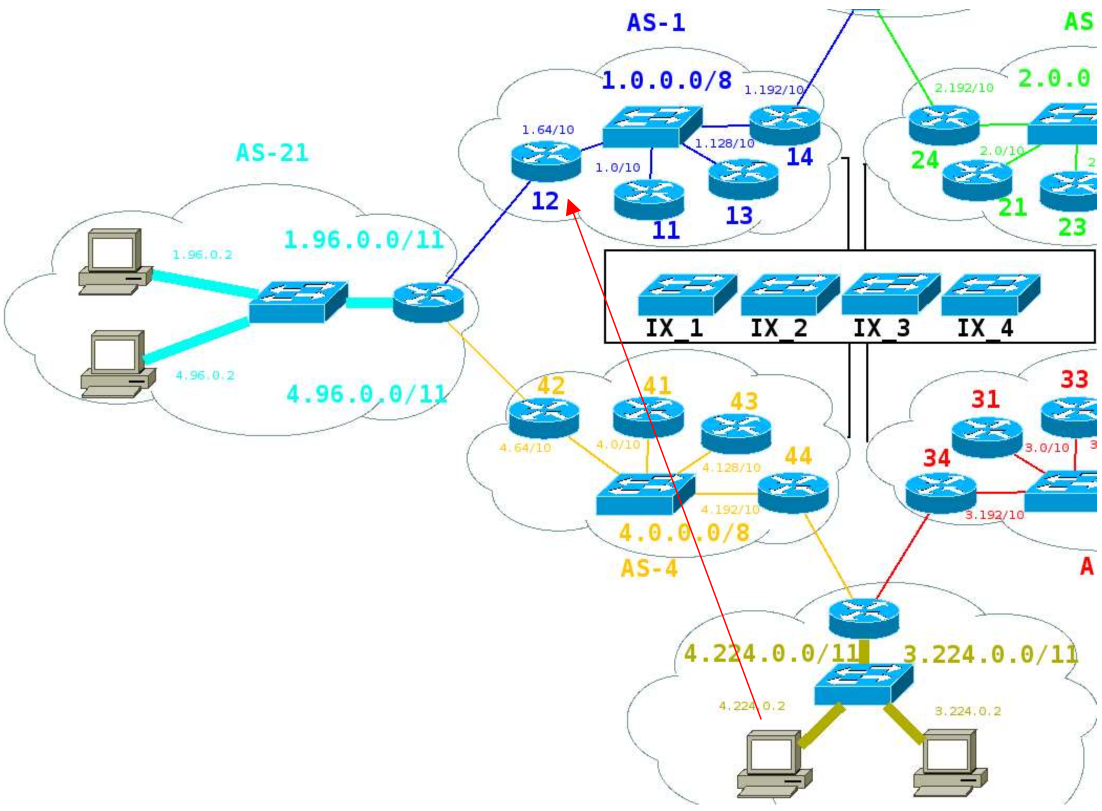
Querying BGPv4 route index history on Kibana with the filter:

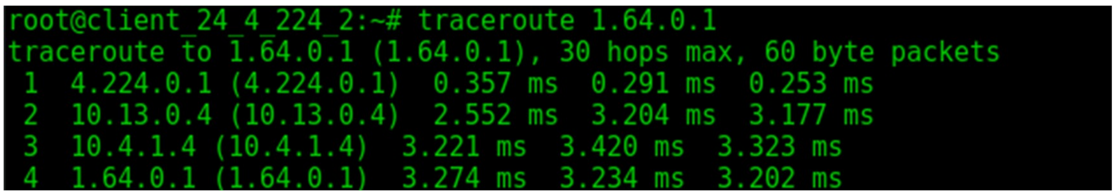
```
machine_hostname:as1card2 AND bestpath: true AND prefix: "4.0.0.0/8"
```



prefix	prefixLen	nexthop_ip	aspath
4.224.0.0	11	10.2.3.2	3 24
4.96.0.0	11	10.11.21.1	21
4.0.0.0	8	10.2.4.2	4

Tracerouting from somewhere in the network before the attack:





```
root@client_24_4_224_2:~# traceroute 1.64.0.1
traceroute to 1.64.0.1 (1.64.0.1), 30 hops max, 60 byte packets
 1  4.224.0.1 (4.224.0.1)  0.357 ms  0.291 ms  0.253 ms
 2  10.13.0.4 (10.13.0.4)  2.552 ms  3.204 ms  3.177 ms
 3  10.4.1.4 (10.4.1.4)  3.221 ms  3.420 ms  3.323 ms
 4  1.64.0.1 (1.64.0.1)  3.274 ms  3.234 ms  3.202 ms
```

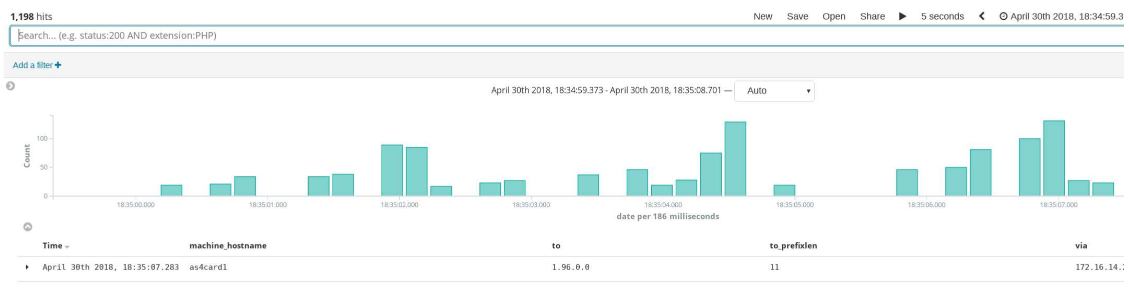
AS21 violating export policies

Let's make AS21 to violate normal export policies and see the effect. No filters are applied:

```
 address-family ipv4 unicast
   network 1.96.0.0/11
   neighbor 10.11.0.2
   network 4.96.0.0/11
   neighbor 10.14.0.2
 exit-address-family
```

As shown before, AS1 and AS4 routers with cardinal 2 have been configured to accept any prefix advertised by them customer.

When the configuration changes, the whole network topology changes and kernel routing tables suffer near 1200 updates:

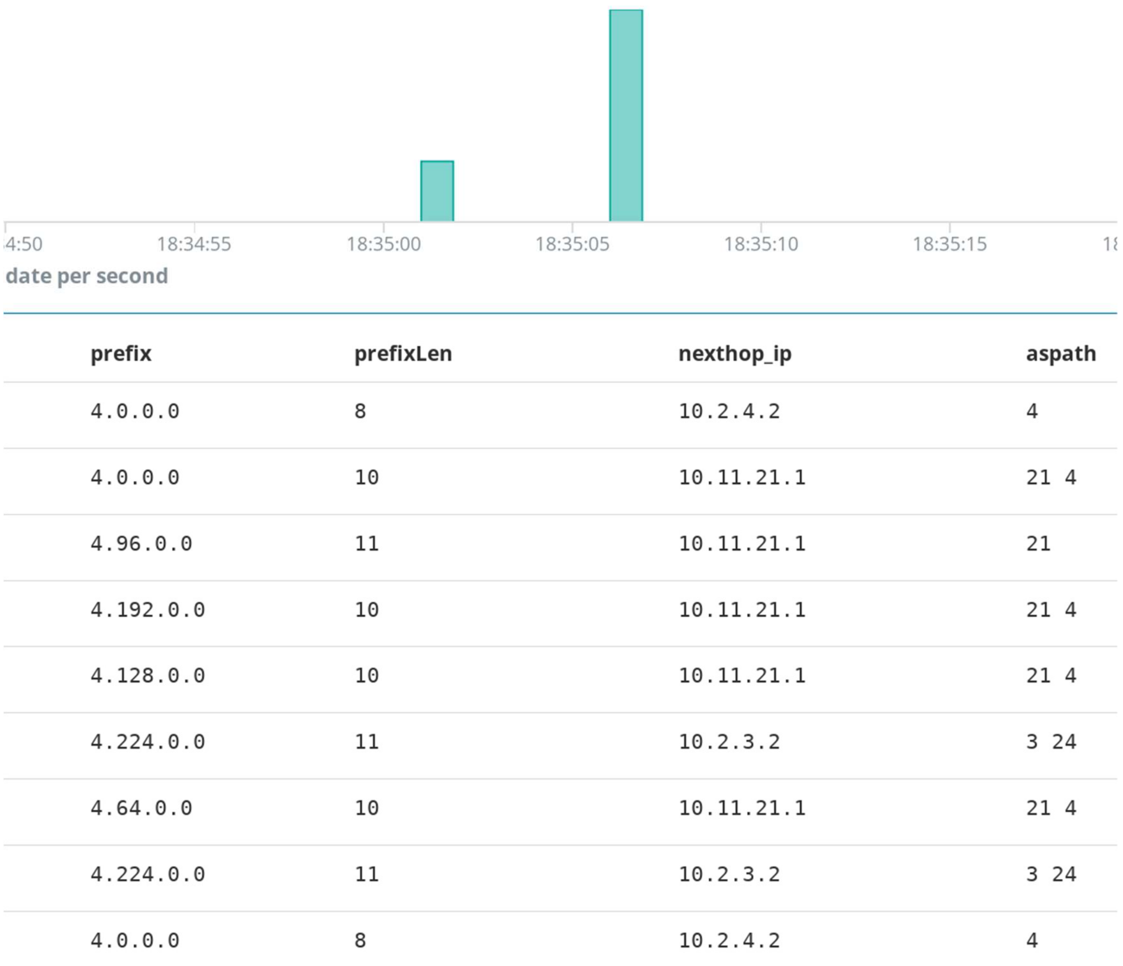

1,198 hits

Search... (e.g. status:200 AND extension:PHP)

April 30th 2018, 18:34:59.373 - April 30th 2018, 18:35:08.701 — Auto

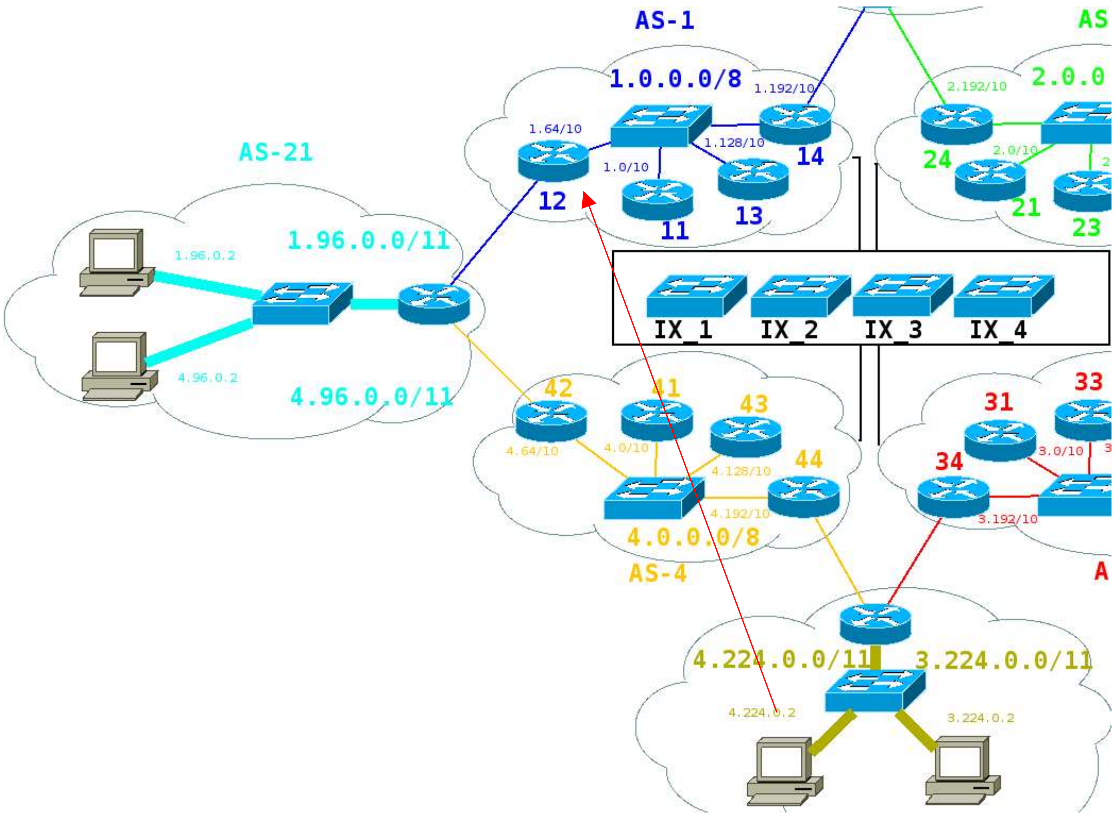
	Time	machine_hostname	to	to_prefixlen	via
▸	April 30th 2018, 18:35:07.283	as4card1	1.96.0.0	11	172.16.14.

Using the same filter on Kibana, it’s possible to see that as1card2 now knows other paths to reach certain sections of 4.0.0.0/8.

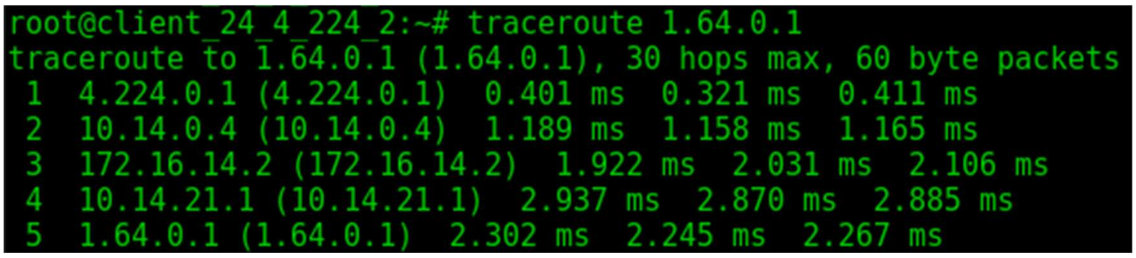


prefix	prefixLen	nexthop_ip	aspath
4.0.0.0	8	10.2.4.2	4
4.0.0.0	10	10.11.21.1	21 4
4.96.0.0	11	10.11.21.1	21
4.192.0.0	10	10.11.21.1	21 4
4.128.0.0	10	10.11.21.1	21 4
4.224.0.0	11	10.2.3.2	3 24
4.64.0.0	10	10.11.21.1	21 4
4.224.0.0	11	10.2.3.2	3 24
4.0.0.0	8	10.2.4.2	4

To show traffic path changes we use repeat the traceroute performed previously.



The impact is obvious, the route from AS4 client to as1card2 has changed and now goes intra-AS4, traverses AS21 and reaches the webserver.



```
root@client_24_4_224_2:~# traceroute 1.64.0.1
traceroute to 1.64.0.1 (1.64.0.1), 30 hops max, 60 byte packets
 1  4.224.0.1 (4.224.0.1)  0.401 ms  0.321 ms  0.411 ms
 2  10.14.0.4 (10.14.0.4)  1.189 ms  1.158 ms  1.165 ms
 3  172.16.14.2 (172.16.14.2)  1.922 ms  2.031 ms  2.106 ms
 4  10.14.21.1 (10.14.21.1)  2.937 ms  2.870 ms  2.885 ms
 5  1.64.0.1 (1.64.0.1)  2.302 ms  2.245 ms  2.267 ms
```

2.3.2. BGP Route Flap Dampening

This configuration directive allows to penalize routes that change frequently. This route changes that use to take place waste CPU cycles that may be used for other purposes and this, is the reason of its existence. Initial research shown that it would cause more harm than benefit and therefore, the RIPE community recommended against using it in 2006. Some years after, in 2014, researchers of Internet Initiative Japan, Internet Initiative Japan, Sproute Networks and Loughborough University shown how to make efficient use of this technique on (RFC7196 - Making Route Flap Damping Usable).

Attack: Flapping the previously explained route leak

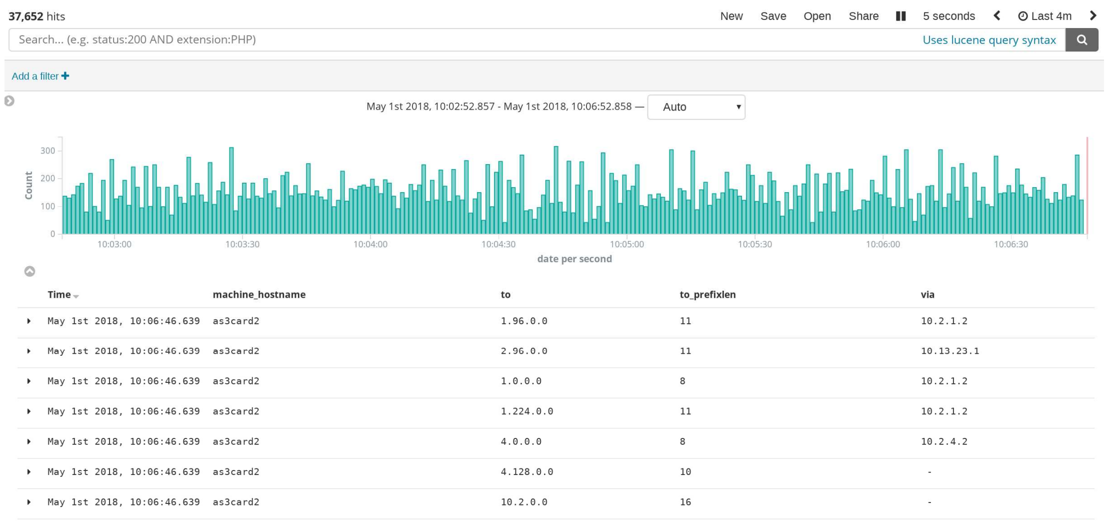
To simulate the behavior of a flapping route we use as21card1. It will periodically advertise the route leak shown before and afterwards get back to its normal export policies.

For that, we use the following Bash & Vtysh script:

```
while true; do ## Violate export policy
        vtysh <<EOF
configure terminal
router bgp 21
address-family ipv4 unicast
no neighbor 10.11.0.2 prefix-list t2-external-adv out
no neighbor 10.14.0.2 prefix-list t2-external-adv out
neighbor 10.11.0.2 prefix-list allow-all-adv out
neighbor 10.14.0.2 prefix-list allow-all-adv out
EOF
        sleep 2; ## Get back to normal export policy
        vtysh <<EOF
…
no neighbor 10.11.0.2 prefix-list allow-all-adv out
no neighbor 10.14.0.2 prefix-list allow-all-adv out
neighbor 10.11.0.2 prefix-list t2-external-adv out
neighbor 10.14.0.2 prefix-list t2-external-adv out
EOF
        sleep 2;
done
```

That makes both as1card2 and as4card2 to make changes to them routing tables each period and to advertise these changes to them peers.

The following timeline shows the effect over kernels routing tables of the whole network and for a 4 minutes period:



37,652 hits

Search... (e.g. status:200 AND extension:PHP)

May 1st 2018, 10:02:52.857 - May 1st 2018, 10:06:52.858 —

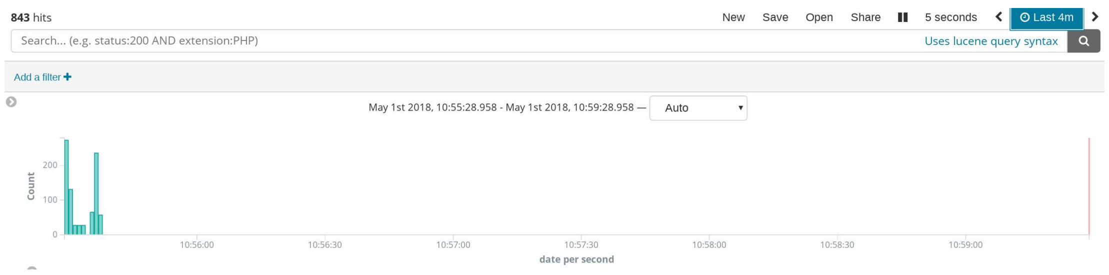
Time	machine_hostname	to	to_prefixlen	via
May 1st 2018, 10:06:46.639	as3card2	1.96.0.0	11	10.2.1.2
May 1st 2018, 10:06:46.639	as3card2	2.96.0.0	11	10.13.23.1
May 1st 2018, 10:06:46.639	as3card2	1.0.0.0	8	10.2.1.2
May 1st 2018, 10:06:46.639	as3card2	1.224.0.0	11	10.2.1.2
May 1st 2018, 10:06:46.639	as3card2	4.0.0.0	8	10.2.4.2
May 1st 2018, 10:06:46.639	as3card2	4.128.0.0	10	-
May 1st 2018, 10:06:46.639	as3card2	10.2.0.0	16	-

As it can be appreciated, that affects a lot of routers, producing changes on kernel routing tables.

To test that, we enabled route flap dampening protection on all routers as stated on (RFC7196 - Making Route Flap Damping Usable) for Cisco routers:

```
bgp dampening 15 750 2000 60
```

And repeated the attack performed on as21card1 for a 4 minutes period again:



With route flap dampening enabled, 36809 kernels routing tables updates on the whole network were prevented. Only the first 843 took place.

This configuration directive, when used correctly, prevents frequent route updates, it saves resources and provides improved network topology stability.

This is just an example, but an attacker would use other ways to cause route flaps. For example, by tampering with unsecured BGPv4 sessions, performing DoS or DDoS attacks over old routers that may be exhausted easily or just by connecting to a badly configured router and performing periodic advertisements. They would be others.

2.3.3. Maximum Prefixes on a Peering

This configuration directive limits the maximum routes that can be accepted from a peer.

The best current practice states that maximum prefixes on a peering should be limited on both peers and upstream routers. The main objective is to protect routers memory from exhaustion.

Attack: Advertising routes for each host but for the network

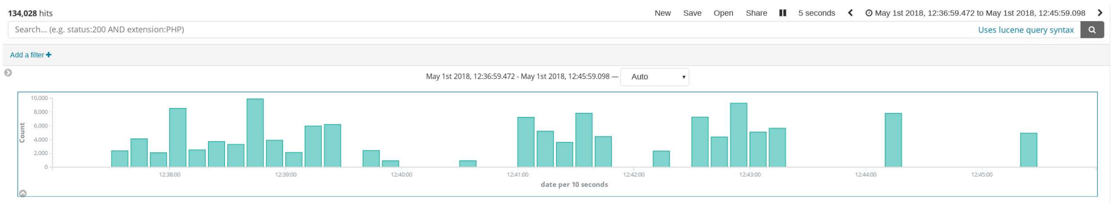
To test that, we will use as21card1 again. We change the configuration, so it starts advertising a route for each host in both domains it owns. It will advertise 2 * 2^16 routes if the full iteration process finishes.

For that purpose, we use the following script Bash & Vtysh script:

```
for i in {0..254}; do
    for y in {0..254}; do
        vtysh <<EOF &>/dev/null
configure terminal
router bgp 21
address-family ipv4 unicast
    network 4.96.$i.$y/32
    network 1.96.$i.$y/32
```

```
EOF
    done
    echo "255 - $i"
done
```

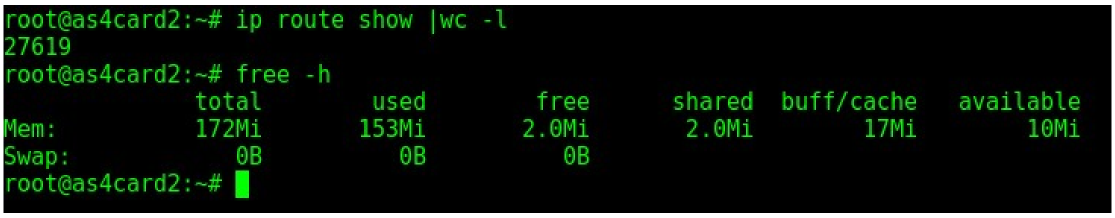
When running, as21card1 produced an impact of ~134000 kernel route updates in 10 minutes on the overall network.



Both as4card2 and as1card2 became unresponsive for a while when they had around 20k routes respectively on them tables.

```
root@hypervisor:~/finalnet# ssh root@172.16.1.42
```

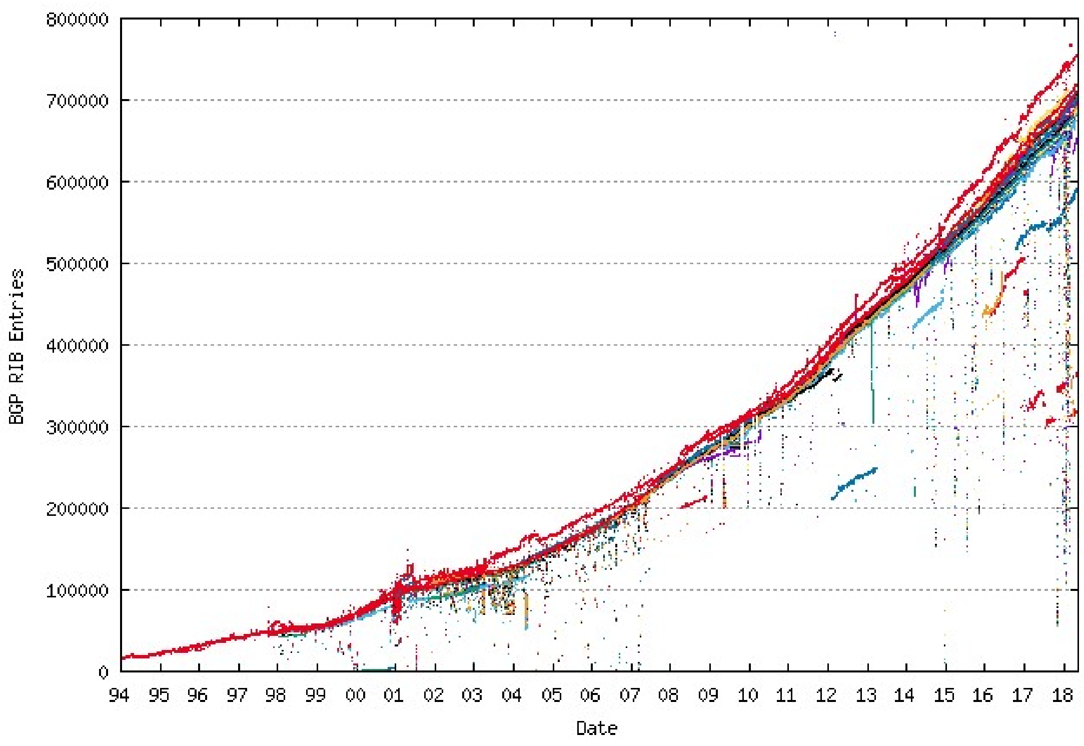
Keeping it running shows that soon they will have memory problems due to the limitations of the environment.



```
root@as4card2:~# ip route show |wc -l
27619
root@as4card2:~# free -h
              total        used        free      shared  buff/cache   available
Mem:          172Mi       153Mi       2.0Mi       2.0Mi        17Mi        10Mi
Swap:            0B          0B          0B
root@as4card2:~#
```

Around 35k routes, as1card2 and as4card2 were almost unresponsive. On this point we stop the attack. Around (63·255)·2 routes were announced. We restarted BGPd on as21card1, so other routers can get back on normal operation.

It's commensurate that a router with more resources will keep running without issues against this kind of attack. But routing table sizes can be huge on the internet. As shown at (BGP Routing Table Analysis Reports) the BGP routing table size is growing faster and nearly it will reach 800k entries for a router working with full BGP table.

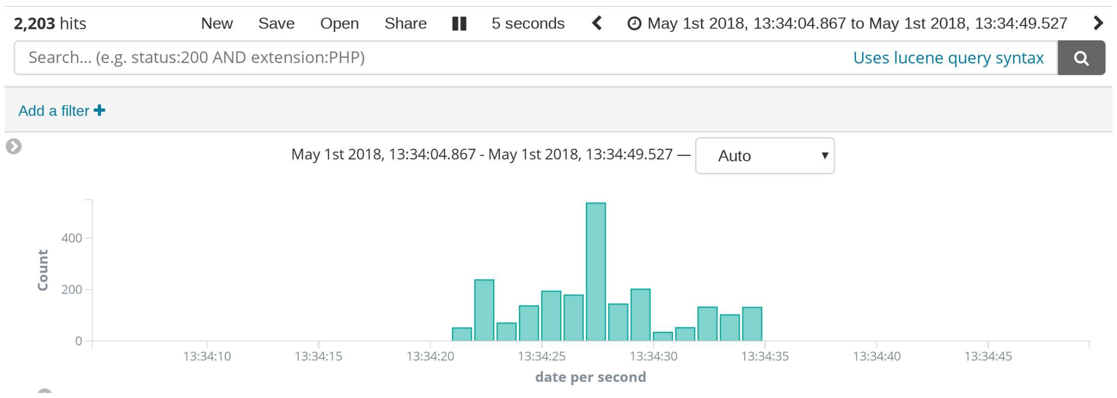


Neither Quagga or Frrouting have documentation about how to configure this. But we've found that it works almost like Cisco routers.

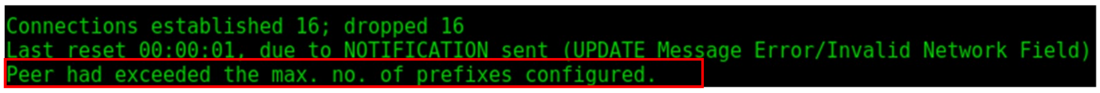
To protect from this kind of attacks, a general limit has been applied and no more than 100 prefixes will be accepted by any router on the network. If anyone advertises more than the threshold the BGPv4 session will be restarted after 150 minutes. The configuration is done as follows:

```
neighbor 10.1[4|1].0.2 maximum-prefix 100 restart 150
```

The impact of the attack is limited, routers were able to keep themselves stable.



The BGPv4 sessions were restarted from as1card2 and as4card2 when more than 100 prefixes were advertised from as21card1. In addition, since they will never accept more than 100 their sessions with other peers won't be restarted and the session will be shut down so as21 will get banned by them providers for 150 minutes. Vtysh shows the following message for as21card1 peer status:



```
Connections established 16; dropped 16
Last reset 00:00:01, due to NOTIFICATION sent (UPDATE Message Error/Invalid Network Field)
Peer had exceeded the max. no. of prefixes configured.
```

2.3.4. AS Path Filtering

As path filters allow to filter accepted and advertised routes if the as-path matches a given pattern.

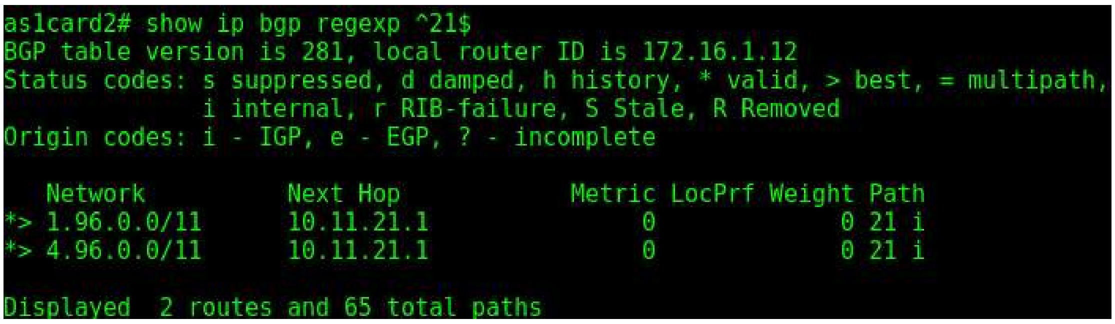
As best practices state, this configuration directive, should be used by network administrators:

- To avoid accepting:
    - Routes containing private AS numbers. Unless they come from allowed customers.
    - Routes that do not start with peer's AS number, unless routes come from a route server (out of the scope of this case study).
    - Routes from customers that do not contain AS numbers belonging to the given customer or for what this customer is authorized to transit to.
        - Worse, but valid solution is to avoid accepting as-paths longer than one. This is valid unless the customer is authorized to provide transit to certain destinations.
    - Routes that contain its' own AS number coming external peers. This overrides BGP normal behavior and must be forcefully configured. The RFC warns that the impact may be severe.
- To avoid advertisements:
    - With non-empty as-path. Unless the network provides transit for these prefixes.
    - With upstream AS numbers in the as-path to their peering ASes unless they are willing to provide transit.
    - With private AS numbers in the as-path.

The patterns are defined by regular expressions and they can be tested through the show interface as follows. The regex language is defined at 11.17 of Frrouting manual. It's exactly the same than Quagga.

For example, lets show routes that only contain AS21:

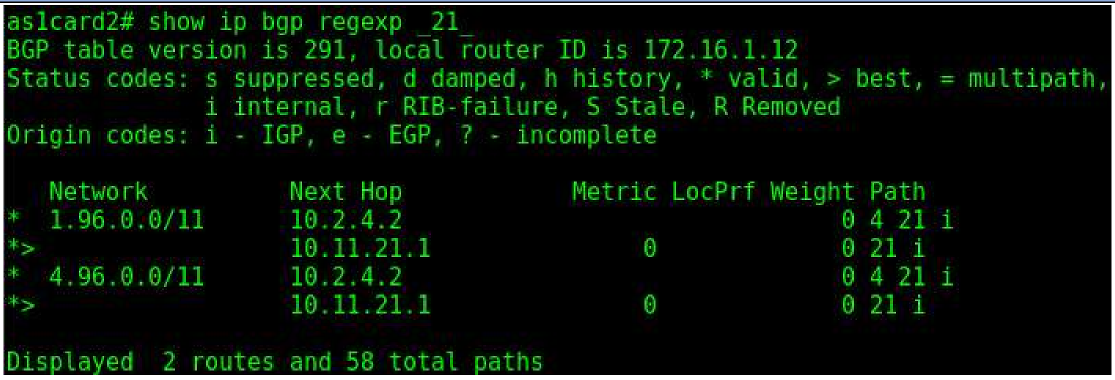
```
Show ip bgp regexp ^21$
```



```
as1card2# show ip bgp regexp ^21$
BGP table version is 281, local router ID is 172.16.1.12
Status codes: s suppressed, d damped, h history, * valid, > best, = multipath,
              i internal, r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*> 1.96.0.0/11      10.11.21.1               0             0 21 i
*> 4.96.0.0/11      10.11.21.1               0             0 21 i

Displayed  2 routes and 65 total paths
```

And, any route that contains AS21:

```
Show ip bgp regexp _21_
```

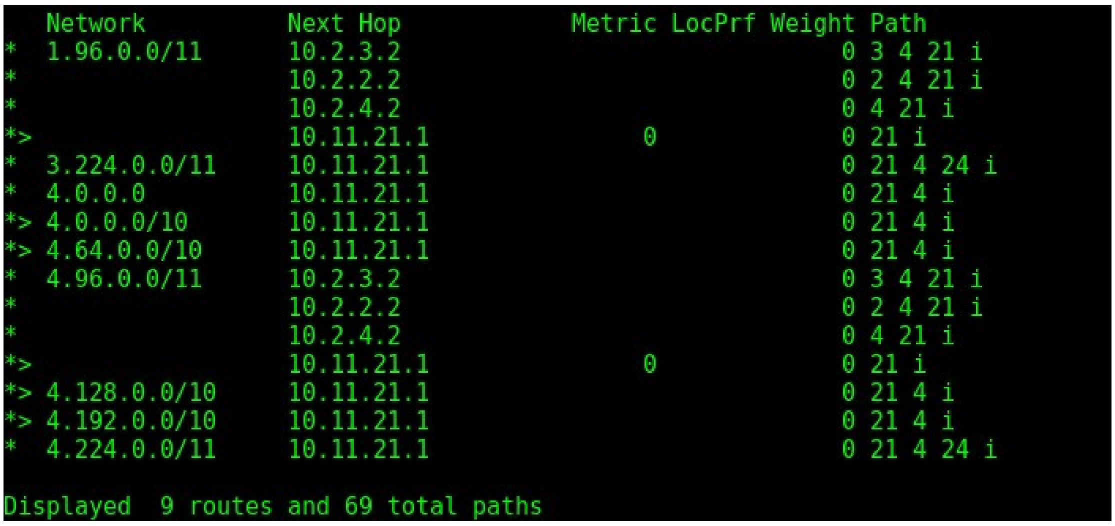


```
as1card2# show ip bgp regexp _21_
BGP table version is 291, local router ID is 172.16.1.12
Status codes: s suppressed, d damped, h history, * valid, > best, = multipath,
              i internal, r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*  1.96.0.0/11      10.2.4.2                               0 4 21 i
*>                  10.11.21.1               0             0 21 i
*  4.96.0.0/11      10.2.4.2                               0 4 21 i
*>                  10.11.21.1               0             0 21 i

Displayed  2 routes and 58 total paths
```

Preventing the route leak explained before

To show the performance of this protection, we repeat the route leak explained before.

Repeating the route leak from as21card1 and with the same filter to show routes as before:



```
   Network          Next Hop            Metric LocPrf Weight Path
*  1.96.0.0/11      10.2.3.2                               0 3 4 21 i
*                   10.2.2.2                               0 2 4 21 i
*                   10.2.4.2                               0 4 21 i
*>                  10.11.21.1               0             0 21 i
*  3.224.0.0/11     10.11.21.1                             0 21 4 24 i
*  4.0.0.0          10.11.21.1                             0 21 4 i
*> 4.0.0.0/10       10.11.21.1                             0 21 4 i
*> 4.64.0.0/10      10.11.21.1                             0 21 4 i
*  4.96.0.0/11      10.2.3.2                               0 3 4 21 i
*                   10.2.2.2                               0 2 4 21 i
*                   10.2.4.2                               0 4 21 i
*>                  10.11.21.1               0             0 21 i
*> 4.128.0.0/10     10.11.21.1                             0 21 4 i
*> 4.192.0.0/10     10.11.21.1                             0 21 4 i
*  4.224.0.0/11     10.11.21.1                             0 21 4 24 i

Displayed  9 routes and 69 total paths
```

As seen before, as1card2 selects to transit traffic through as21card1 instead of doing through the normal paths.

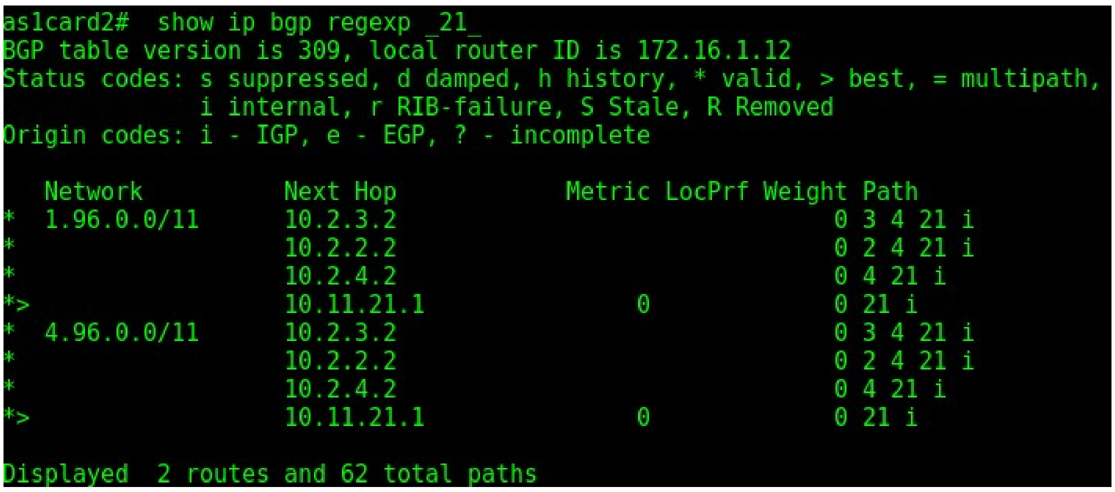
To fix that problem, the as-path filter is declared as:

```
ip as-path access-list StrictProviderAS21 permit ^21$
ip as-path access-list StrictProviderAS21 deny .*

ip as-path access-list maxLength1 permit ^[0-9]+$

ip as-path access-list maxLength1 deny .*
```

And added to the neighbor statement in bgp router configuration for IPv4 unicast:

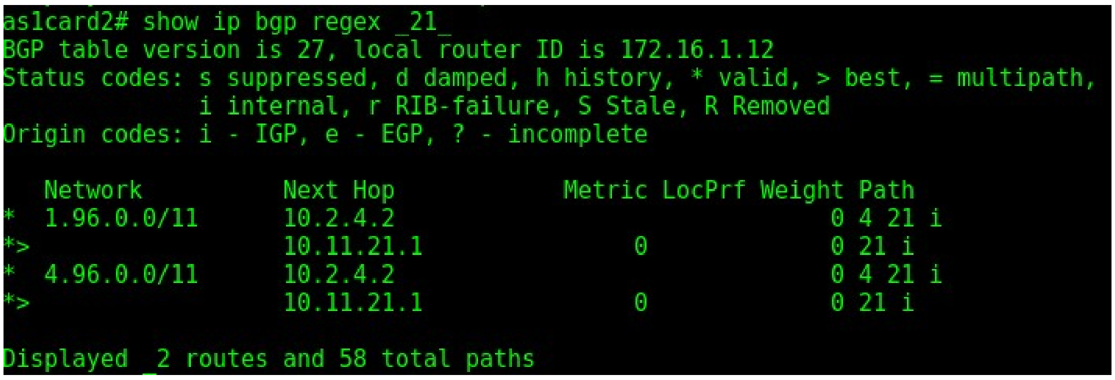
```
address-family ipv4 unicast

…

  neighbor 10.11.21.1 filter-list [StrictProviderAS21|maxLength1] in

…

exit-address-family
```

Consider that this is not the same case than before as as4card2 keeps accepting bogus routes coming from AS21.



```
as1card2#  show ip bgp regexp _21_
BGP table version is 309, local router ID is 172.16.1.12
Status codes: s suppressed, d damped, h history, * valid, > best, = multipath,
              i internal, r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*  1.96.0.0/11      10.2.3.2                                 0 3 4 21 i
*                   10.2.2.2                                 0 2 4 21 i
*                   10.2.4.2                                 0 4 21 i
*>                  10.11.21.1               0               0 21 i
*  4.96.0.0/11      10.2.3.2                                 0 3 4 21 i
*                   10.2.2.2                                 0 2 4 21 i
*                   10.2.4.2                                 0 4 21 i
*>                  10.11.21.1               0               0 21 i

Displayed  2 routes and 62 total paths
```

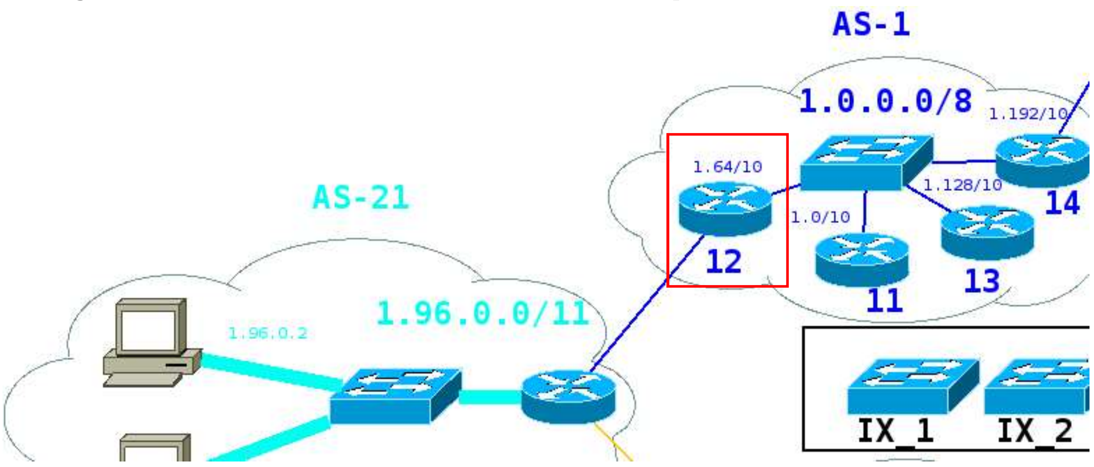
With that in mind, a new environment has been deployed and the route leak is attempted again without success from as21card1 (appreciate that BGP table version changes from the image that shows the same routes above):



```
as1card2# show ip bgp regex _21_
BGP table version is 27, local router ID is 172.16.1.12
Status codes: s suppressed, d damped, h history, * valid, > best, = multipath,
              i internal, r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*  1.96.0.0/11      10.2.4.2                                 0 4 21 i
*>                  10.11.21.1               0               0 21 i
*  4.96.0.0/11      10.2.4.2                                 0 4 21 i
*>                  10.11.21.1               0               0 21 i

Displayed  2 routes and 58 total paths
```

2.3.5. Next-Hop Filtering

The most common way to publish a route is to set the next hop to the router that makes the advertisement. This is commensurate, since usually the advertisement receivers would not be able to reach the router that truly offers access to the given prefix but through the advertiser. To be clearer, we will explain this over the environment:



Let's say that as1card2 advertises a route to 1.192.0.0/10 with next-hop 172.16.11.4 to AS21. That makes no sense to do it so, since as21card1 has not a way to reach that host on AS1 private's network. So, as1card2 makes the advertisement to this network replacing next-hop by itself on a network range visible by as21card1.

This is done on the advertiser side with the configuration directive:

```
neighbor 10.11.21.1 next-hop-self
```

The protocol and implementations of BGPv4 allow to replace and change this behavior to support different setups. For example, at IXPs where routers just receive routes from a route server. Route servers will never want to set up next-hop-self, but they want to instruct routers to use certain next hops to reach certain networks.

This functionality allows an attacker to redirect traffic through another hop, therefore, it should be filtered and overridden with the peer address on the side that receives the advertisement. Unless working in a route server setup.

This is filtered at the route receptor side with the following configuration directives:

Route maps allow to both filter and apply actions to received routes. Not only for BGP but for all routing protocols offered by Frrouting or Quagga suites.

This sets up a next-hop overwrite with the peer-address when a route is received.

```
route-map AntiSpoofNextHop permit 10
    match ip next-hop peer-address
route-map ReplaceNextHop permit 10
    set ip next-hop peer-address
```

After that, the route-map must be applied to the peer, so every next-hop received on route advertisements from this peer gets replaced with the peer-address or checked and discarded if it does not match the peer-address:

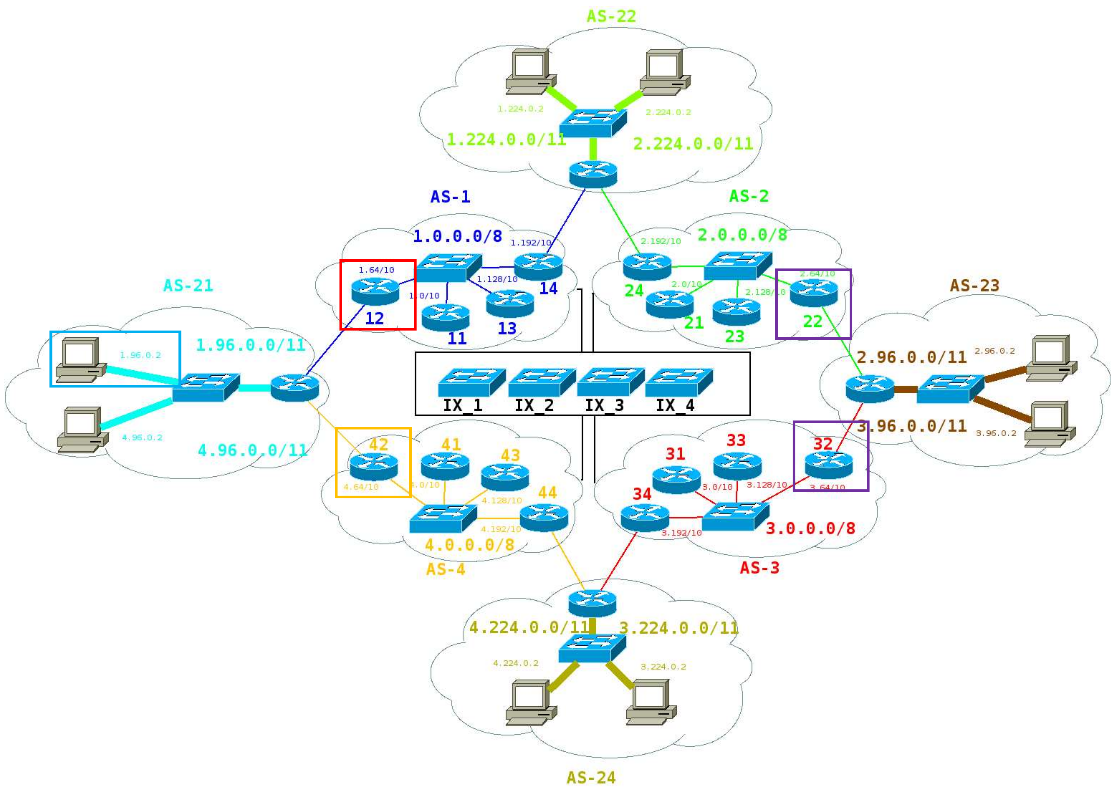
```
address-family ipv4 unicast
…
    neighbor 10.11.21.1 route-map [AntiSpoofNextHop|ReplaceNextHop] in
…
exit-address-family
```

Attack: Next-Hop Spoof

When testing TCPMD5 Signature, we performed a MITM attack over an unsecured BGPv4 – TCP session and next-hop got replaced by another valid next-hop on the given network segment. We demonstrated that it possible to make the victim to transit more traffic than expected to a given destination.

The objective is the same, to make the victim to move more traffic than expected.

The victim on this case is as4card2. The attack takes place at IX_2 and the attacker is as1card2 that wants to make as4card2 to transit more traffic than he expects to. Regarding environment's network topology, both victim and attacker are connected at IX_2.



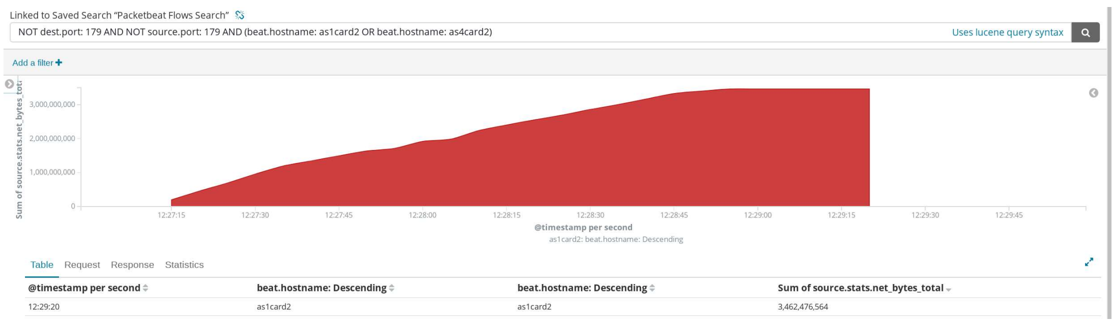
To show the normal behavior we will measure bandwidth seen at both victim and attacker in a normal case, when all clients are generating traffic to client_21_1_96_2.

Using Iperf and making all clients in the network out of AS21 to generate traffic to client_21_1_96_2, located inside AS21 networks.

```
#!/bin/bash
generators=( "172.16.1.121" "172.16.1.131" )
DATA="100"
destination=$1
for generator in ${generators[@]}
do
        nohup ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null
root@"${generator}" "iperf -d -t ${DATA}M -c $destination" &>> /dev/null &
done
```

With that script, running on the hypervisor, we instruct all clients out of AS21 to generate bidirectional traffic to and from client_21_1_96_2. The bidirectional data amount to be transmitted are 100Mb from each generator and direction.

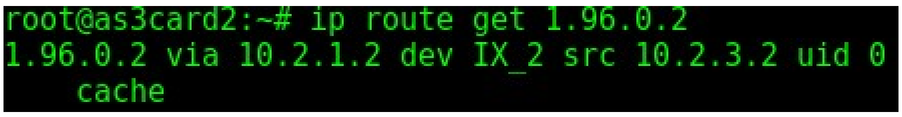
Once executed, it can be appreciated that as1card2 is transiting the whole traffic from AS23 to AS21, for client_21_1_96_2.



```
NOT dest.port: 179 AND NOT source.port: 179 AND (beat.hostname: as1card2 OR beat.hostname: as4card2)
```

@timestamp per second	beat.hostname: Descending	beat.hostname: Descending	Sum of source.stats.net_bytes_total
12:29:20	as1card2	as1card2	3,462,476,564

Before the attack is performed, the routing table of as3card2 and as2card2 returns that client is reached through as1card2.



```
root@as3card2:~# ip route get 1.96.0.2
1.96.0.2 via 10.2.1.2 dev IX_2 src 10.2.3.2 uid 0
    cache
```

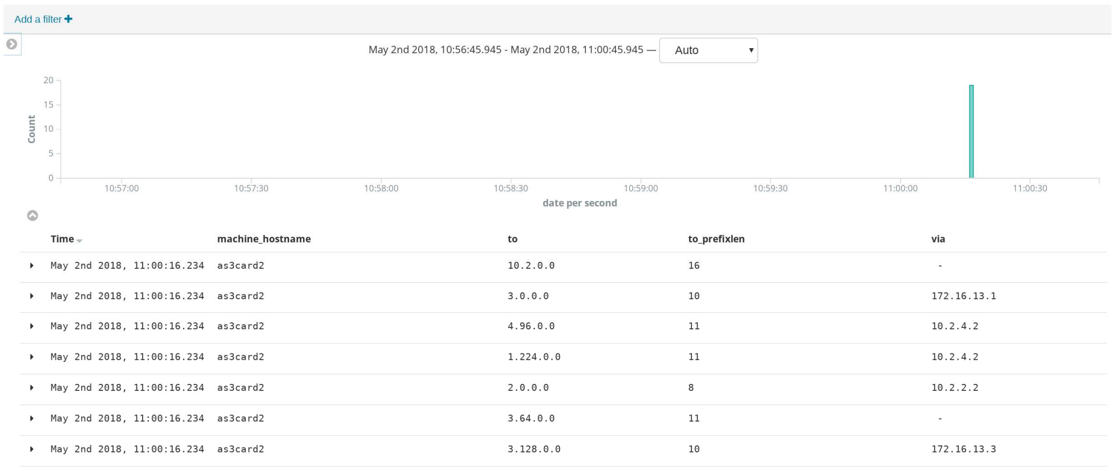
Now, as1card2 decides that this is not fair and changes its' advertisement to network 1.96.0.0/11 to set next-hop through as4card2 as follows:

```
route-map LazyRouter permit 5
set ip next-hop 10.2.4.2
```

And modifies the announcement as follows to both as2card2 and as3card2:

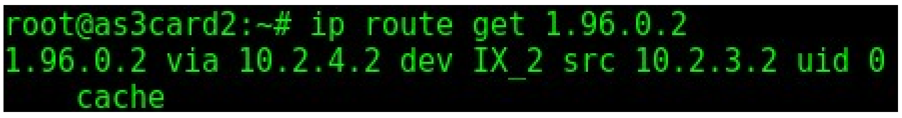
```
neighbor 10.2.2.2 route-map LazyRouter out
neighbor 10.2.3.2 route-map LazyRouter out
```

After that, as1card2 sets the route-map to be applied when advertising routes to neighbor as3card2. That produces 19 route updates on as3card2 and as2card2 kernel's routing tables:



Time	machine_hostname	to	to_prefixlen	via
May 2nd 2018, 11:00:16.234	as3card2	10.2.0.0	16	-
May 2nd 2018, 11:00:16.234	as3card2	3.0.0.0	10	172.16.13.1
May 2nd 2018, 11:00:16.234	as3card2	4.96.0.0	11	10.2.4.2
May 2nd 2018, 11:00:16.234	as3card2	1.224.0.0	11	10.2.4.2
May 2nd 2018, 11:00:16.234	as3card2	2.0.0.0	8	10.2.2.2
May 2nd 2018, 11:00:16.234	as3card2	3.64.0.0	11	-
May 2nd 2018, 11:00:16.234	as3card2	3.128.0.0	10	172.16.13.3

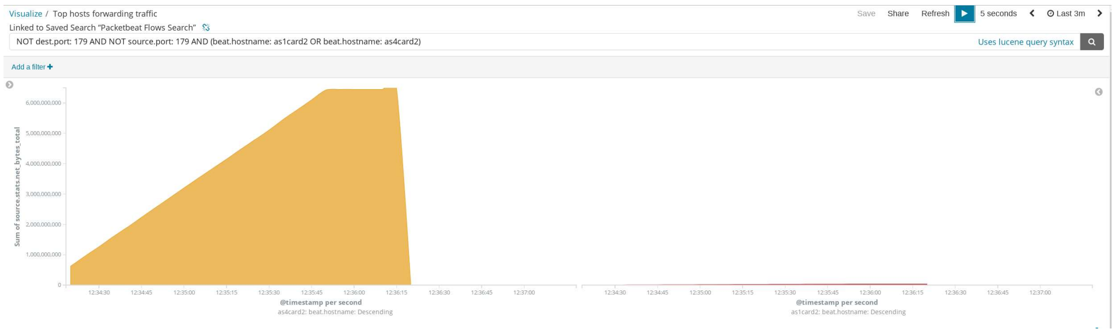
Now as2card2 and as3card2 reach the client through as4card2, the victim.



```
root@as3card2:~# ip route get 1.96.0.2
1.96.0.2 via 10.2.4.2 dev IX_2 src 10.2.3.2 uid 0
    cache
```

The same steps are applied to as2card2. This way all the traffic coming from the other side of the network, AS23 is routed through as4card2.

Now, the traffic transits through as4card2 and therefore, that offloads as1card2 as it wanted to. Not the whole traffic, since it's a bidirectional test and TCP ACKs from the client are getting routed through as1card2, back to the clients of this Iperf server.



NOT dest.port: 179 AND NOT source.port: 179 AND (beat.hostname: as1card2 OR beat.hostname: as4card2)

As shown, as1card2 can make other routers to send more traffic to the victim, as4card2. This technique can be used in several ways to attack or just to generate revenue by influencing routers out of the control of the victim to drive more traffic through the victim and for example, generate revenue.

3. Attacks over the whole network

The purpose of this section is to group attacks that are not mitigated through just one of the previously explained security measures. They may involve two or more defenses, and, in some cases, the attack will only be mitigated but not impossible.

While defenses explained previously, mitigate or make impossible the exercise of certain types of attacks, there are others that are more complicated or nearly impossible to defeat. On the current environment the following defenses are implemented:

- Linux Kernel and Firewall have been set up to avoid routing traffic to private CIDR network domains. In addition, kernel's behavior has been changed to mitigate DoS or DDoS attacks over routers. Considering limitations in term of RAM and processor, attacks that involve high loads of traffic to reset a router or a BGPv4 session are not considered.
- TTL-Security with maximum hop distance of 1. That makes impossible to perform attacks from outside of BGPv4 speaker's network segments.
- TCP-MD5 Signature. The password is assumed to be strong. Therefore, an attempt to crack it would take much longer than in the test performed. It can be assumed that MITM attacks are not possible.
- Prefix Filters have been applied to both avoid leaking internal ASes structure and to show the performance of route leak attack performed by a multi-homed customer.
- BGP Route flap dampening. Dampening on flapping routes is enabled as exposed on the RFC.
- Maximum prefixes on a peering have been limited and limited routers memory is protected.
- AS path Filters have been enabled and customers, AS21, AS22, AS23 and AS24 are not allowed to advertise routes longer than one and that do not contain exactly it's AS number.
- Next Hop Filters are enabled, and Next Hop spoofing is prevented.

Before explaining the attacks it's mandatory to regard the following basis about IP routing and BGPv4.

There are two algorithms that manage respectively the route selection and the routing itself.

The BGPv4 Route selection algorithm manages what routes go in and out from the effective routing table (AKA the kernel routing table). The behavior may vary between manufacturers and the behavior can be changed by network administrators through management interfaces. The most common case is that it chooses as the best route, to a given prefix, the shortest path in terms of autonomous systems.

Routers apply the Longest Prefix Algorithm (LPM) for making the decision about the next hop and the interface each IP packet should be sent onto. Implemented in the kernel, it looks up each IP packet's destination IP address into one or more forwarding tables and computes the best match, the LPM. This algorithm is the basis of IP routing and its behavior cannot be modified without tampering the forwarding table implementation.

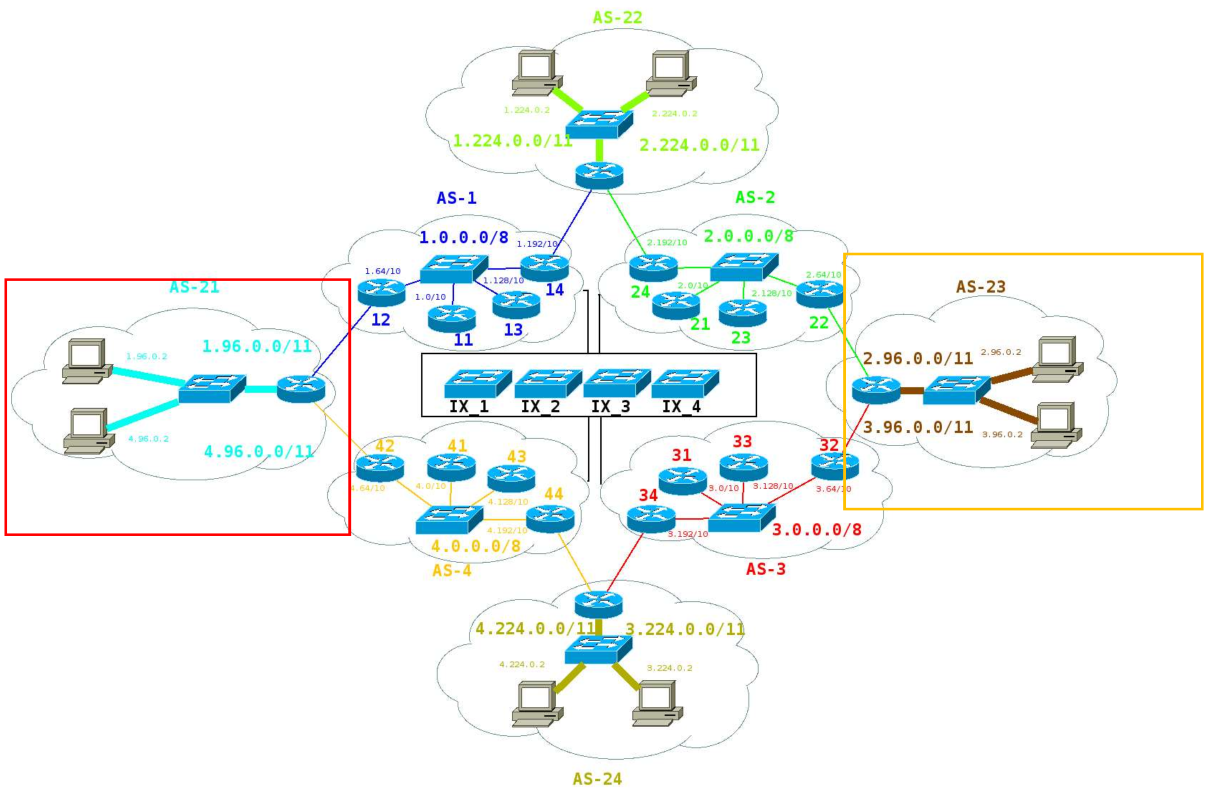
Therefore, exhaustive filters should be applied before updating these tables with routes received from outside, since they will change the router behavior.

3.1. Prefix and Sub-prefix hijacks

The Prefix and Sub-prefix Hijacks are some of the worst attacks that can be exercised against a network of BGPv4 routers.

Both require knowledge about BGPv4 route selection algorithm and LPM.

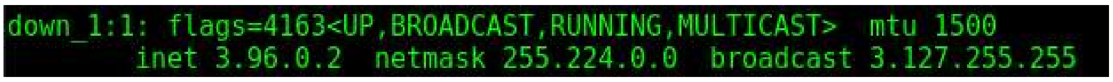
Over the case study, the AS21 wants to hijack traffic going to and from the victim 3.96.0.2, client_23_3_96_2:



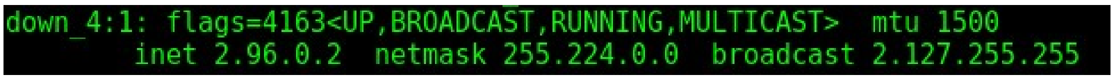
To do it, the attacker may go straight forward and advertise both 3.96.0.0/11 and 2.96.0.0/11 prefixes as follows:

```
address-family ipv4 unicast
…
        network 2.96.0.0/11
        network 3.96.0.0/11
…
exit-address-family
ip prefix-list t2-external-adv seq 15 permit 3.96.0.0/11 le 32
ip prefix-list t2-external-adv seq 20 permit 2.96.0.0/11 le 32
```

To simulate that, we configured both down_1:1 and down_4:1 virtual interfaces on as21card1 to have the following addresses:

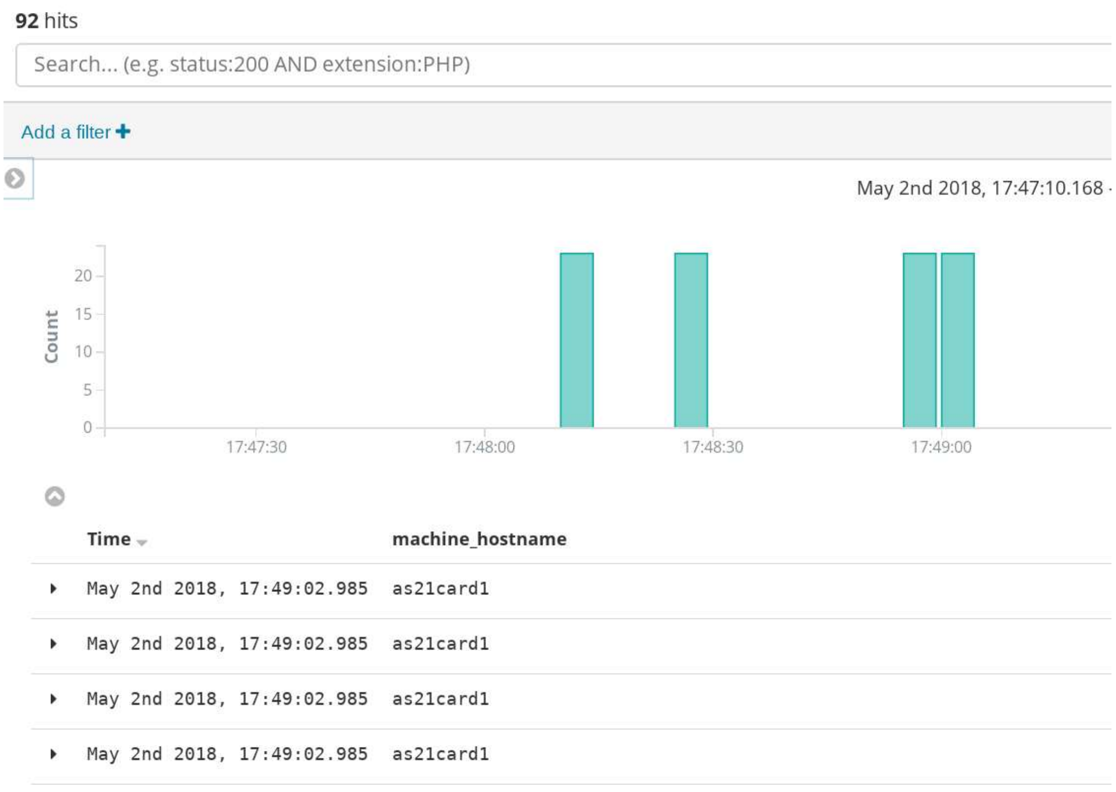


```
down_1:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 3.96.0.2  netmask 255.224.0.0  broadcast 3.127.255.255
```



```
down_4:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 2.96.0.2  netmask 255.224.0.0  broadcast 2.127.255.255
```

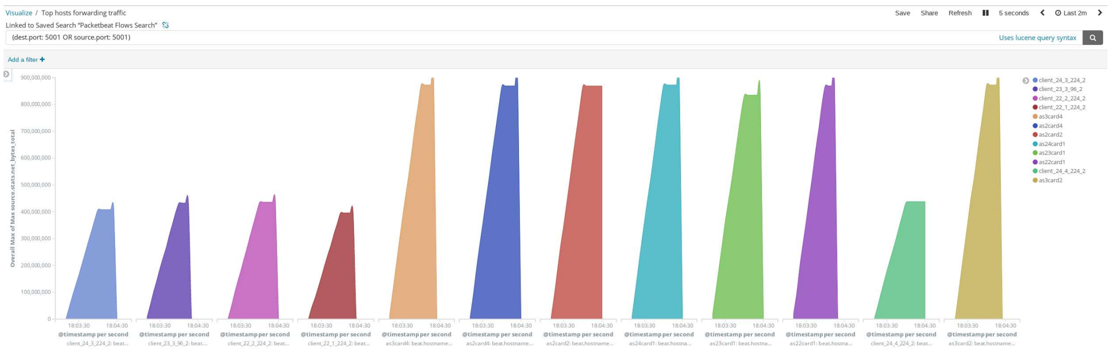
Therefore, that maid automatic changes on kernel's routing table of as21card1:



92 hits
Search... (e.g. status:200 AND extension:PHP)
Add a filter
May 2nd 2018, 17:47:10.168 -
Count
20
15
10
5
0
17:47:30
17:48:00
17:48:30
17:49:00

	Time	machine_hostname
▸	May 2nd 2018, 17:49:02.985	as21card1
▸	May 2nd 2018, 17:49:02.985	as21card1
▸	May 2nd 2018, 17:49:02.985	as21card1
▸	May 2nd 2018, 17:49:02.985	as21card1

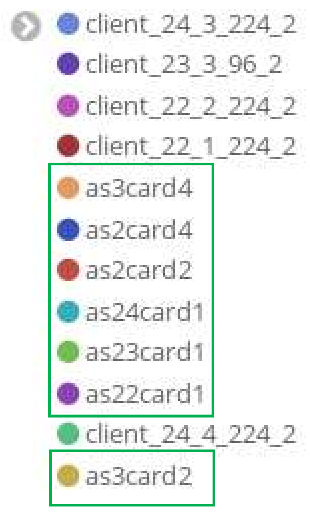
Before the attack is executed, clients from AS22 and AS24 generate traffic to AS23. The traffic volume measured on the overall is as follows:



Visualize / Top hosts forwarding traffic

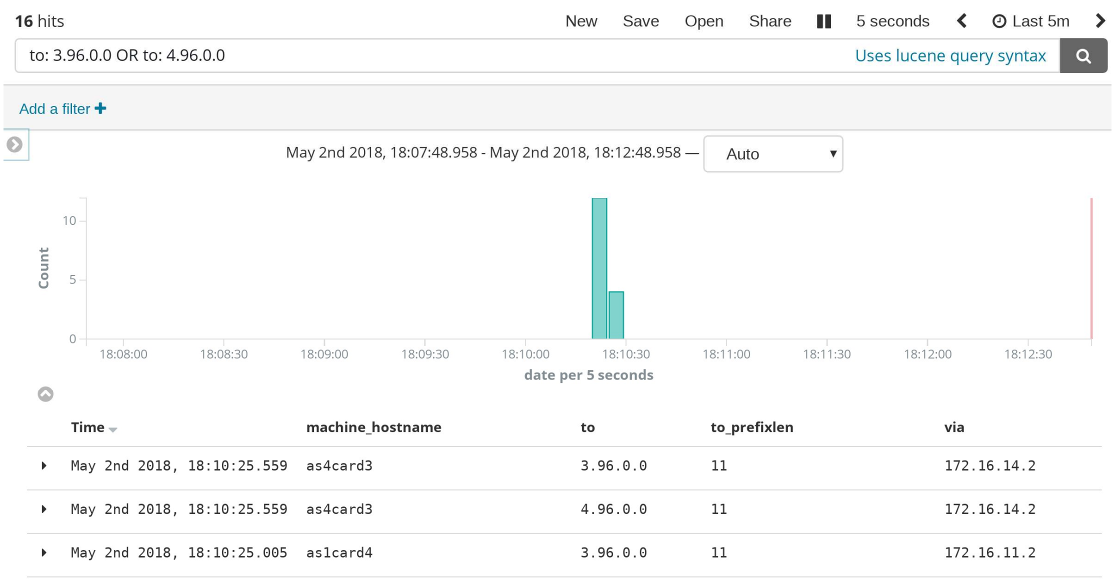
Linked to Saved Search "Packetbeat Flows Search"

(dest.port: 5001 OR source.port: 5001)



The routers forwarding traffic were as3card4, as2card4, as2card2, as24card1, as23card1, as22card1 and as3card2.

From this point the BGPd config explained before is applied. The effect was obviously spread over kernel routing tables of several routers:



16 hits

to: 3.96.0.0 OR to: 4.96.0.0

May 2nd 2018, 18:07:48.958 - May 2nd 2018, 18:12:48.958 — Auto

Time	machine_hostname	to	to_prefixlen	via
May 2nd 2018, 18:10:25.559	as4card3	3.96.0.0	11	172.16.14.2
May 2nd 2018, 18:10:25.559	as4card3	4.96.0.0	11	172.16.14.2
May 2nd 2018, 18:10:25.005	as1card4	3.96.0.0	11	172.16.11.2

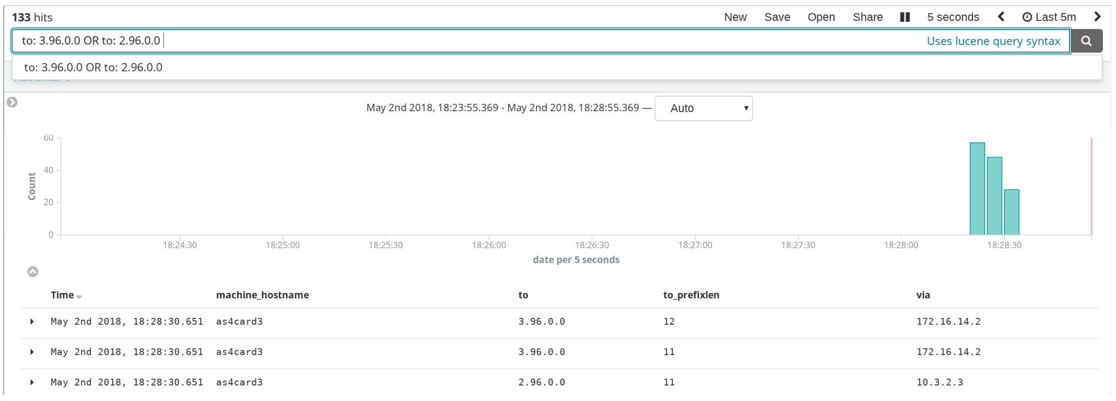
8 routers got updates on them routing tables with destination [3|4].96.0.0/11. They were as4card3, as1card4, as1card2, as4card1, as1card1, as1card3, as4card2, as4card4. But none of them were customer ASes so announcing these prefixes will only allow the

attacking AS to hook a portion of the whole traffic AS23 would receive. This portion is the traffic that travels through this list of routers. In the previous case, none.

From this point, the attacker, AS21, decides to change the advertised routes to [3|4].96.0.0/12. Therefore, changing the attack from a prefix hijack to a sub-prefix hijack.

```
address-family ipv4 unicast

…

        network 2.96.0.0/12

        network 3.96.0.0/12

…

exit-address-family
```

Once done, it produces 133 updates on kernel routing tables on the overall of the network:

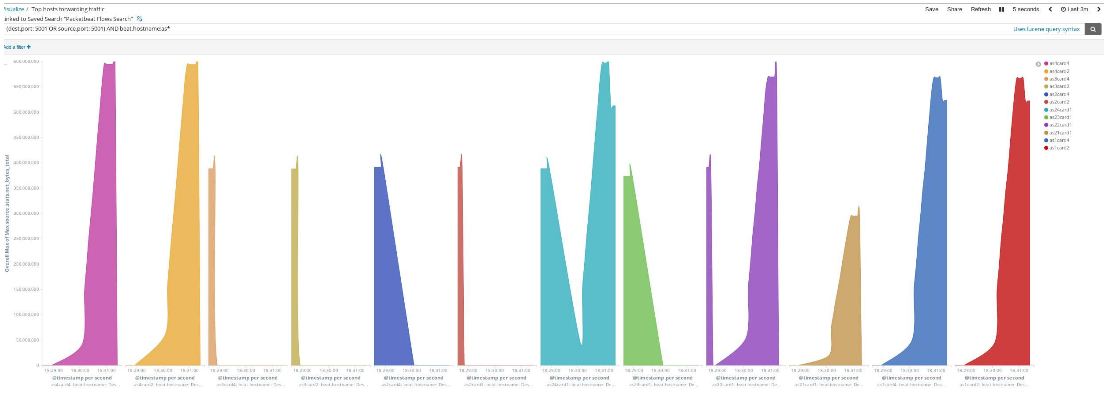


to: 3.96.0.0 OR to: 2.96.0.0

May 2nd 2018, 18:23:55.369 - May 2nd 2018, 18:28:55.369 —

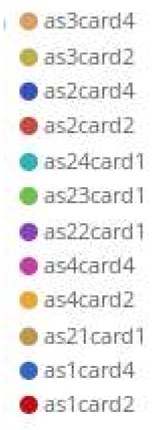
```
Time       machine_hostname       to         to_prefixlen    via
May 2nd 2018, 18:28:30.651  as4card3   3.96.0.0   12   172.16.14.2
May 2nd 2018, 18:28:30.651  as4card3   3.96.0.0   11   172.16.14.2
May 2nd 2018, 18:28:30.651  as4card3   2.96.0.0   11   10.3.2.3
```

After repeating the traffic generation test, the change can be appreciated:

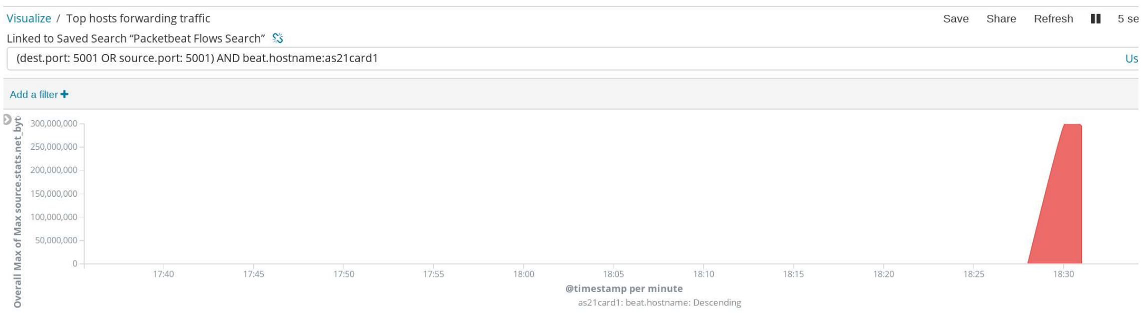


(dest.port: 5001 OR source.port: 5001) AND beat.hostname:as*

New boys have joined the party:

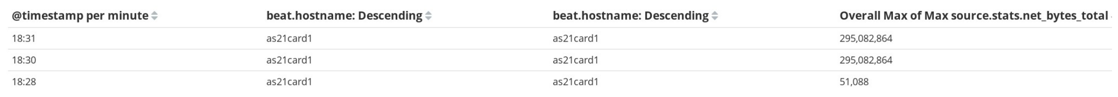


And as follows, AS21 who did not receive traffic before, has started to receive it. The following histogram shows the traffic volume measured at AS21 since this test started


Visualize / Top hosts forwarding traffic
Save
Share
Refresh
5 se
Linked to Saved Search "Packetbeat Flows Search"
Add a filter +
Overall Max of Max source.stats.net_byt
@timestamp per minute
as21card1: beat.hostname: Descending

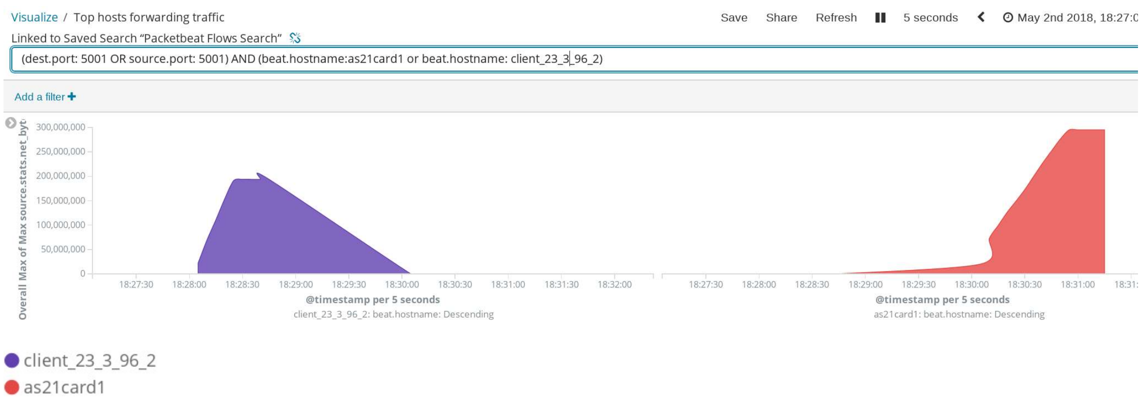
(dest.port: 5001 OR source.port: 5001) AND beat.hostname:as21card1

Concretely, it started receiving some traffic at 18:28 when the advertisement was made.



@timestamp per minute	beat.hostname: Descending	beat.hostname: Descending	Overall Max of Max source.stats.net_bytes_total
18:31	as21card1	as21card1	295,082,864
18:30	as21card1	as21card1	295,082,864
18:28	as21card1	as21card1	51,088

In addition, the following histogram shows how the legitimate client stops receiving traffic in favor of the attacker:


Visualize / Top hosts forwarding traffic
Save
Share
Refresh
5 seconds
May 2nd 2018, 18:27:0
Linked to Saved Search "Packetbeat Flows Search"
Add a filter +
Overall Max of Max source.stats.net_byt
@timestamp per 5 seconds
client_23_3_96_2: beat.hostname: Descending
@timestamp per 5 seconds
as21card1: beat.hostname: Descending
client_23_3_96_2
as21card1

(dest.port: 5001 OR source.port: 5001) AND (beat.hostname:as21card1 or beat.hostname: client_23_3_96_2)

4. Bibliography

"BGP Routing Table Analysis Reports." 2018. <http://bgp.potaroo.net/>.

Foces Vivancos, José María. "Rehtse." 2016. <https://github.com/JmFoces/Rehtse>.

—. "Securing The BGPv4: Working Environment." 2018.

"IANA - IPv4 Address Space." 1998. <https://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xhtml>.

"IRRToolSet." 2002. <https://github.com/irrtoolset/irrtoolset>.

"JohnTheRipper." 1996. <https://github.com/magnumripper/JohnTheRipper>.

"Linux Kernel Documentation." 1991. <https://www.kernel.org/doc/Documentation/>.

Mutually Agreed Norms for Routing Security. 2014. <https://www.manrs.org>.

"RFC1105 - A Border Gateway Protocol (BGP)." 1989.

"RFC1163 - A Border Gateway Protocol (BGP)." 1990.

"RFC1267 - A Border Gateway Protocol 3 (BGP-3)." 1991.

"RFC1337 - TIME-WAIT Assassination Hazards in TCP." 1992. <https://www.ietf.org/rfc/rfc1337.txt>.

"RFC1654 - A Border Gateway Protocol 4 (BGP-4)." 1994.

"RFC1771 - A Border Gateway Protocol 4 (BGP-4)." 1995.

"RFC2385 - Protection of BGP Sessions via the TCP MD5 Signature Option." 1998.

"RFC2827 - Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing." 2000.

"RFC3013 - Recommended ISP Security." 2000.

"RFC3704 - Ingress Filtering for Multihomed Networks." 2004. <https://tools.ietf.org/html/rfc3704>.

"RFC4012 - Routing Policy Specification Language next generation (RPSLng)." 2005.

"RFC4271 - A Border Gateway Protocol 4 (BGP-4)." 2006.

"RFC4272 - BGP Vulnerability Analysis." 2006.

"RFC5082 - The Generalized TTL Security Mechanism (GTSM)." 2007.

"RFC5925 - The TCP Authentication Option." 2010.

"RFC5961 - Improving TCP's Robustness to Blind In-Window Attacks." 2010. <https://tools.ietf.org/html/rfc5961>.

"RFC6192 - Plane, Protecting the Router Control." 2011. <https://tools.ietf.org/html/rfc6192>.

"RFC6480 - An Infrastructure to Support Secure Internet Routing." 2012.

"RFC6483 - Validation of Route Origination Using the Resource Certificate Public Key Infrastructure (PKI) and Route Origin Authorizations (ROAs)." 2012.

"RFC6810 - The Resource Public Key Infrastructure (RPKI) to Router Protocol." 2013.

"RFC6811 - BGP Prefix Origin Validation." 2013.

"RFC7196 - Making Route Flap Damping Usable." 2014. <https://tools.ietf.org/html/rfc7196>.

"RFC7454 - BGP Operations and Security." 2015.

"RFC7715 - Origin Validation Operation Based on the Resource Public Key Infrastructure (RPKI)." 2016.

"RFC791 - INTERNET PROTOCOL." 1981. <https://tools.ietf.org/html/rfc791>.

"RFC7947 - Internet Exchange BGP Route Server." 2016.

"RFC8205 - BGPsec Protocol Specification." 2017.

Robert Lychev, Michael Schaipira & Sharong Goldberg. "Rethinking Security for Internet Routing." 2016.

"TCPMD5 Signature - Socket Programing Examples on Linux." 2015. <https://criticalindirection.com/2015/05/12/tcp_md5sig>.

"Ubuntu - Kernel Security Settings." 2006. <https://wiki.ubuntu.com/ImprovedNetworking/KernelSecuritySettings>.