Winning tactics with DNS tunnelling


Mouhammd Al-kasassbeh


Tariq Khairallah

Mouhammd Al-kasassbeh and Tariq Khairallah, Princess Sumaya University for Technology, Jordan

The Domain Name System (DNS) protocol is the backbone of the Internet. It facilitates connection to websites and services using understandable names that are easy to remember. DNS converts these names to their corresponding IP addresses in order to establish the communication through the network. The security of the DNS protocol was not originally a major concern for organisations since the protocol was not intended to be used for regular data transmission.[1] However, in recent years, attackers have developed tools that have taken advantage of this situation and utilised DNS for malicious purposes, such as covert tunnelling and data exfiltration.

This article focuses on covert tunnelling, which refers to the mechanism of sending and receiving data between machines without alerting firewalls and intrusion detection systems (IDSs) on the network.[2] The DNS protocol has three characteristics that make it well suited to be used as a covert tunnel.[3] First, DNS is fundamental for the proper functioning of many applications and Internet browsing. Most organisations do not implement a security policy for DNS traffic. Second, DNS is used to communicate with internal clients and external remote servers. And third, the DNS protocol has several fields that can be used to embed other data. The most used fields by tunnelling utilities are the NULL and TXT records.

The main motivation behind using the DNS protocol as a covert tunnel is to bypass captive portals for paid wifi services in places such as airports and hotels. Other motivations include stealing data, evading detection for unauthorised access, installing and controlling malware, and bypassing firewalls and Internet access policies.

There are several tools available for DNS tunnelling. These tools differ in flexibility, throughput and the technique used to embed data into DNS traffic. Examples of these tools include OzymanDNS, dns2tcp, Iodine and DNScat.[4]

There are two general methods of detecting DNS tunnels: the first is payload analysis in which a single DNS request is analysed. Attributes such as the length of the packet, the number of bytes and the content of the packet are examined and used to create detection rules (signatures). The second is traffic analysis, in which the overall traffic is analysed over a period of time. Attributes such as the volume of DNS traffic, number of hostnames per domain and domain history are used as an indication of tunnelling.[5] In this article, an experimental setup of the Iodine DNS tunnelling tool is implemented. Analysis of tunnelled DNS traffic is conducted to carve out the encapsulated IP packet. Several detection mechanisms are presented to detect DNS tunnelling using both payload and traffic analysis methods.

Background

DNS is a hierarchical system that provides a database for mapping domain names to IP addresses. It is a client-server protocol in which the client sends a query that contains a certain question and the server responds with an answer. For example, in order for a client to access the website www.mydomain.com, the IP address of that website should first be determined. An A record maps

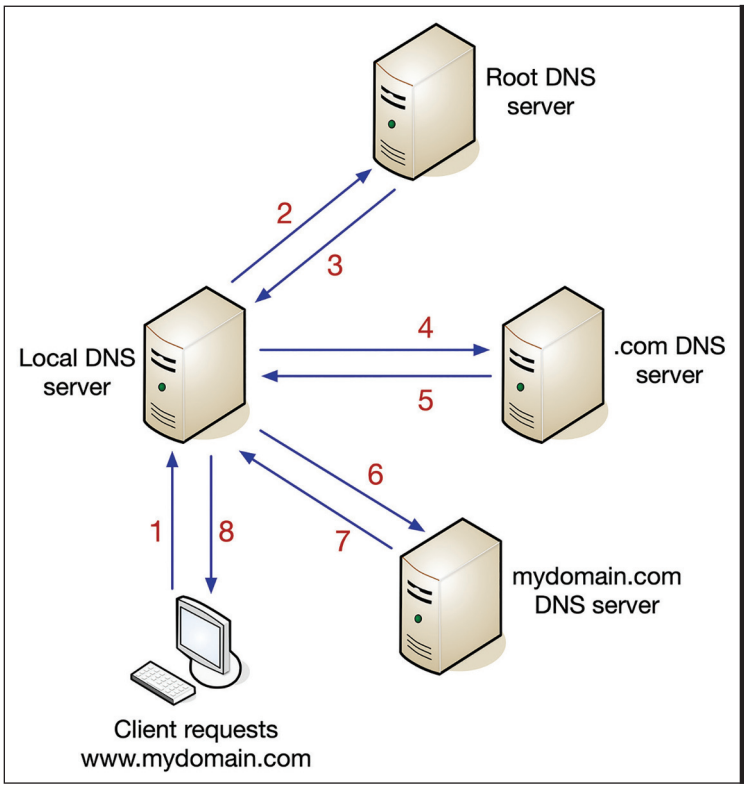

Figure 1: Recursive domain name resolution.

DNS tunnelling utilities

DeNiSe: A proof of concept for tunnelling TCP over DNS in Python. The GitHub page for DeNiSe has six Python scripts dating between 2002 and 2006.

Dns2tcp: Written by Olivier Dembour and Nicolas Collignon. It is written in C and runs on Linux. The client can run on Windows. It supports KEY and TXT request types.

DNScapy: Developed by Pierre Bienaime. It uses Scapy for packet generation. DNScapy supports SSH tunnelling over DNS including a Socks proxy. It can be configured to use CNAME or TXT records or both randomly.

DNScat (DNScat-P): This was originally released in 2004 and the most recent version was released in 2005. It was written by Tadeusz Pietraszek. DNScat is presented as a Swiss Army knife tool with many uses involving bi-directional communication through DNS. DNScat is Java-based and runs on Unix-like systems. DNScat supports A record and CNAME record requests. Since there are two utilities named DNScat, this one will be referred to as DNScat-P in this article to distinguish it from the other.

DNScat (DNScat-B): This was written by Ron Bowes. The earliest known public release was in 2010. It runs on Linux, MacOS and Windows. DNScat will encode requests in either NetBIOS encoding or hex encoding. DNScat can make use of A, AAAA, CNAME, NS, TXT and MX records. It provides a datagram and a stream mode. There is also a DNScat-B Metasploit payload.

Heyoka: A proof of concept that creates a bi-directional tunnel for data.

Exfiltration: This tool is written in C and has been tested on Windows. Heyoka was developed by Alberto Revelli and Nico Leidecker. It uses binary data instead of 32- or 64-bit encoded data to increase bandwidth. It also uses EDNS to allow DNS messages greater than 512 bytes. Heyoka also uses source spoofing to make it appear that the requests are spread out over multiple IP addresses.

Iodine: This is a DNS tunnelling program first released in 2006 with updates as recently as 2010. It was developed by Bjorn Andersson and Erik Ekman. Iodine is written in C and it runs on Linux, MacOS, Windows and others. Iodine has been ported to Android. It uses a tun or tap interface on the endpoint.

NSTX: The Nameserver Transfer Protocol from Florian Heinz and Julien Oster was released in 2000. It runs only on Linux. NSTX makes it possible to create IP tunnels using DNS. It tunnels the traffic using either a tun or tap interface on the endpoints.

OzymanDNS: This was written in Perl by Dan Kaminsky in 2004. It is used to set up an SSH tunnel over DNS or for file transfer. Requests are base32 encoded and responses are base64 encoded TXT records.

Psudp: This was developed by Kenton Born. It injects data into existing DNS requests by modifying the IP/UDP lengths. It requires all hosts participating in the covert network to send their DNS requests to a broker service that can hold messages for a specific host until a DNS request comes from that host. The message can then be sent in the response.

Squeeza: This is a SQL injection tool. It splits the command channel and the data exfiltration channel. The command channel can be used to create data in a database and execute other commands. It supports three data exfiltration channels: HTTP errors, timing and DNS. For the DNS channel, data is encoded in the Fully Qualified Domain Name (FQDN) used in the request.

TCP-over-dns: This was released in 2008. It has a Java-based server and a Java-based client. It runs on Windows, Linux and Solaris. It supports LZMA compression and both TCP and UDP traffic tunnelling.

TUNS: This was developed by Lucas Nussbaum. TUNS is written in Ruby. It does not use any experimental or seldom-used record types. It uses only CNAME records. It adjusts the MTU used to 140 characters to match the data in a DNS request. TUNS may be harder to detect, but it comes at a performance cost.

Malware using DNS: DNS has been used as a communication method by malware. Known malware using DNS includes Feederbot and Moto.[12] Both of these malware examples use DNS TXT records for command and control.

a domain name to an IP address, so the query will be for the A record of the website. The process of sending a query and receiving the answer may involve several recursive steps that are illustrated in Figure 1.

Assuming that this is the first time a client wants to access this website, and the local DNS server has no entry for this domain in its cache, then the process is as follows:

- The client sends a DNS query to its local DNS server asking for the A record of the domain name mydomain.com.
- The local DNS server receives the request and forwards it to one of the 13 root servers.
- The root DNS server does not know the IP address of mydomain.com and hence refers the query to the .com Top Level Domain (TLD).
- The local DNS server sends the query to the .com TLD.
- The .com TLD does not know the IP address of mydomain.com and hence refers the query to the authoritative DNS server responsible for the mydomain.com namespace.
- The local DNS server sends the query to the authoritative DNS server responsible for the mydomain.com namespace.

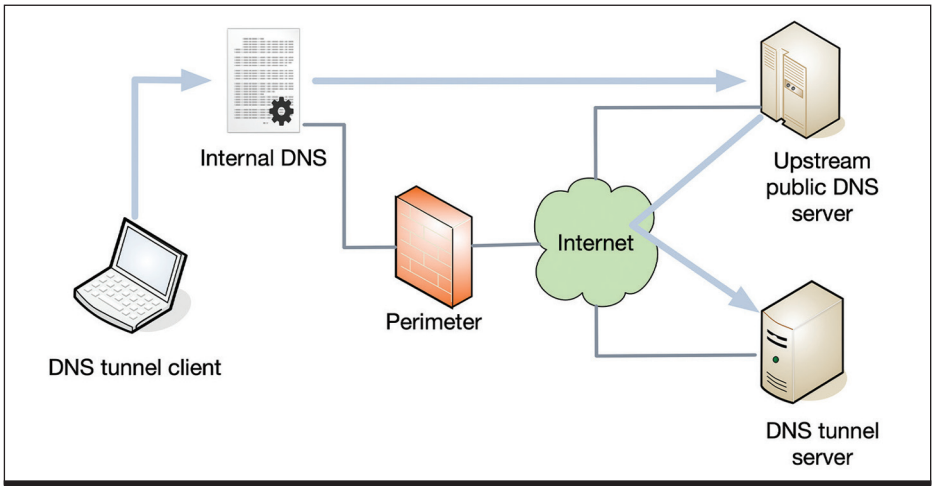

Figure 2: Basic DNS tunnel components.

- The authoritative server replies with the IP address of the domain.
- The local DNS server sends the reply containing the IP address to the client.

DNS typically uses UDP protocol over port number 53 for communications. The TCP protocol may also be used for certain communications such as zone transfers between authoritative name servers.

DNS has many record types that are essential for its operation. There is a number of records that are commonly used in DNS, such as the A record, which maps a domain name to an IPv4 address; the CNAME record, which maps a domain name to a canonical name; and the MX record, which specifies the mail exchange server for a domain. There are uncommon record types such as the TXT and NULL records that may contain any type of data. DNS tunnelling tools leverage these records for maliciously transferring data. For example, the NULL record can contain up to 65,535 bytes of any type of data.

Tunnel discussion

The first known discussion of DNS tunnelling was by Oskar Pearson on the Bugtraq mailing list in April 1998. Since that time, a number of DNS tunnelling utilities have been developed: most of the utilities use the same techniques with differences in encoding. DNS tunnelling is achieved by misusing one of the record fields of the DNS protocol to embed IP traffic.

DNS tunnelling is based on a client-server model. Typically, the client is located inside an organisation and the server is located on the Internet. The DNS communication between the client and server is carried out by utilising the organisation's own DNS infrastructure. The DNS tunnel does not provide encryption – the tunnelled traffic is encapsulated inside DNS packets. An attacker might choose to run a VPN through the DNS tunnel in what is known as double tunnelling or use an SSH connection with port forwarding to increase the confidentiality of the data transferred over the tunnel.

"An attacker might choose to run a VPN through the DNS tunnel in what is known as double tunnelling or use an SSH connection with port forwarding to increase the confidentiality of the data transferred over the tunnel"

Iodine is one of the well-known DNS tunnelling tools. It uses the NULL field to transmit data between a client and a server.[6] When the Iodine tunnel is established, the tool creates tunnel interfaces on both the client and the server to allow tunnelling traffic. Figure 2 shows an example setup for a DNS tunnel.

Assuming that the Iodined service daemon is setup on the tunnel server and is listening for client requests, the Iodine service on the client initiates the tunnel by sending a DNS request to the controlled domain. Since the local DNS server does not have an entry cached for this domain, it will ask recursively for that information. A public upstream name server should be set up to forward queries for the controlled domain to the IP address of the server running the tunnelling service. Encoded in this DNS request is a message that only the tunnel server will be able to decode. The Iodined server software then strips out the encoded portion of the domain name and replies back with a similarly encoded DNS response.[7]

Once the negotiation between the client and the server is completed, the tunnel will be established. Any type of IP traffic can be transmitted through the tunnel where Iodine will split the traffic into small pieces that fit within the NULL field of the DNS packet. The encapsulated packets will be sent separately over the tunnel and will be reassembled at the other endpoint.

Detection of such activity will be difficult for basic firewalls and IDS solutions, as the traffic will look like legitimate DNS queries and responses. However, taking a closer look by performing packet analysis will indicate that unidentified data is being transmitted over the NULL field. Also, traffic analysis will indicate that there is a huge number of DNS NULL queries and responses, which is not usually a normal behaviour for DNS.

Methodology

This section presents the methodology used to set up a DNS tunnel using Iodine, as well as a detailed analysis of extracting encapsulated IP packets from the tunnelled DNS traffic, and how to use the DNS tunnel to proxy web browsing traffic.

Experiment prerequisites: Four components are needed prior to setting up the DNS tunnel: an internal host to run the Iodine client on; a server with a public IP address to run the Iodined server on; control over a domain name; and DNS configuration to route queries for the domain name to the IP address of

the Iodined server. Following is a fuller description of the four components.

The client: A virtual machine running Ubuntu 16.04.3 was used as the internal host on which the Iodine client was set up. The network settings for this host were set to bridged so that it connected directly to the internal network and obtained an IP address from the internal LAN IP address range. The client was assigned the IP address 10.0.2.0/24 and was able to access the Internet.

The server: A Virtual Public Server (VPS) was purchased through the online provider Cloudflare to act as the Iodine server.[8] The VPS ran Ubuntu 16.04.4 and was accessible through SSH for configuration purposes. The VPS had the public IP address 94.177.172.165.

Domain name: The domain name tunnel.sahloul.co was registered through GoDaddy for use in this experiment.

DNS configuration: Two DNS records were configured – an NS record to route queries for the tunnel domain to the Iodine server and an A record to define the IP address of the Iodine server.

Tunnel setup

Iodine was installed on both the client and the server using 'apt install Iodine'. To start the Iodined service on the server, the following command was used:

```
Iodined -c -P 12345 -f 10.0.0.1
tunnel.sahloul.co
```

The -f option keeps Iodined running in the foreground, the -P option specifies the password to be used between the client and the server for shared secret key authentication and the 10.0.0.1 IP address is a virtual IP address that will be assigned for the tunnel interface on the server side. Once the command above is entered on the server, Iodined creates a virtual tunnel interface and starts listening for DNS queries on UDP port 53. Figure 3 shows a screenshot from the server after starting the Iodined service. Figure 4 shows the ifconfig command output on the server illustrating the created virtual interface with IP address 10.0.0.1.

```
root@psut:~# iodined -c -P 12345 -f 10.0.0.1 tunnel.sahloul.co
Opened dns0
Setting IP of dns0 to 10.0.0.1
Setting MTU of dns0 to 1130
Opened IPv4 UDP socket
Listening to dns for domain tunnel.sahloul.co
```

Figure 3: Iodined service started on the tunnel server.

```
root@psut:~# ifconfig
dns0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
-00
          inet addr:10.0.0.1  P-t-P:10.0.0.1  Mask:255.255.255.224
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1130  Metric:1
          RX packets:46 errors:0 dropped:0 overruns:0 frame:0
          TX packets:39 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:4962 (4.9 KB)  TX bytes:6117 (6.1 KB)

eth0      Link encap:Ethernet  HWaddr 00:50:56:b2:0c:dd
          inet addr:80.211.3.127  Bcast:80.211.3.255  Mask:255.255.255.0
          inet6 addr: fe80::250:56ff:feb2:cdd/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2331559 errors:0 dropped:42844 overruns:0 frame:0
          TX packets:274644 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:248169707 (248.1 MB)  TX bytes:46012445 (46.0 MB)
```

Figure 4: Output of ifconfig on the Iodine server.

```
osboxes@osboxes:~$ sudo iodine -f -r 80.211.3.127 -P 12345 tunnel.sahloul.co
[sudo] password for osboxes:
Opened dns0
Opened IPv4 UDP socket
Sending DNS queries for tunnel.sahloul.co to 80.211.3.127
Autodetecting DNS query type (use -T to override).
Using DNS type NULL queries
Version ok, both using protocol v 0x00000502. You are user #0
Setting IP of dns0 to 10.0.0.2
Setting MTU of dns0 to 1130
Server tunnel IP is 10.0.0.1
Skipping raw mode
Using EDNS0 extension
Switching upstream to codec Base128
Server switched upstream to codec Base128
No alternative downstream codec available, using default (Raw)
Switching to lazy mode for low-latency
Server switched to lazy mode
Autoprobing max downstream fragment size... (skip with -m fragsize)
768 ok.. 1152 ok.. ...1344 not ok.. ...1248 not ok.. ...1200 not ok.. 1176 ok..
...1188 not ok.. will use 1176-2=1174
Setting downstream fragment size to max 1174...
Connection setup complete, transmitting data.
```

Figure 5: Iodine tunnel established from the client.

```
psut@psut-VirtualBox:~$ ifconfig
dns0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1130
        inet 10.0.0.2  netmask 255.255.255.224  destination 10.0.0.2
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1  bytes 48 (48.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::2a0c:4320:a2ac:4d65  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:14:b3:56  txqueuelen 1000  (Ethernet)
        RX packets 62371  bytes 64890482 (64.8 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 18791  bytes 1251437 (1.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
```

Figure 6: The output of ifconfig on the Iodine client.

On the client side, Iodine is started by specifying the IP address of the tunnel server, the shared password and the domain name as follows:

```
sudo iodine -f -r 94.177.172.165 -P
12345 tunnel.sahloul.co
```

After running the above command, negotiations occur between the client and the server. During these negotia-

```
osboxes@osboxes: ~
osboxes@osboxes:~$ ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=85.9 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=87.0 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=86.8 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=86.9 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=88.9 ms
^C
--- 10.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 85.919/87.146/88.961/1.045 ms
osboxes@osboxes:~$
```

Figure 7: Ping through the DNS tunnel.

SSH-tunneled-packet.bin

0000016f	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000000	a0	e1	78	da	01	b4	00	4b	ff	00	00	08	00	45	10	00	áxÚ.´.Kÿ....E.
00000010	b0	90	91	40	00	40	06	95	a4	0a	00	00	01	0a	00	00	° '@.@.•¤.....
00000020	02	00	16	df	f6	94	18	f5	2d	01	9e	ac	88	80	18	00	...ßö".õ-.ž¬ˆ€..
00000030	ec	0b	5b	00	00	01	01	08	0a	0d	11	9f	a3	b5	89	d7	ì.[........Ÿ£µ‰×
00000040	dd	7f	cb	47	7c	3e	50	96	27	7c	d2	da	f5	02	ca	62	Ý Ë G|>P–'|ÒÚõ.Êb
00000050	93	5d	51	e9	19	78	60	eb	1d	f8	51	e3	38	fa	0a	a0	"]Qé.x`ë.øQã8ú.
00000060	49	0c	da	05	e1	a6	1e	e5	1b	1d	fc	62	a3	69	f2	c4	I.Ú.á¦.å..üb£iòÄ
00000070	08	2d	61	ba	00	a6	a9	de	22	51	db	34	f1	d9	45	17	.-aº.¦©Þ"QÛ4ñÙE.
00000080	20	72	09	e2	c7	44	1f	a2	10	27	de	a1	42	e5	79	07	r.âÇD.¢.'Þ¡Båy.
00000090	88	fd	ac	ef	9a	50	e8	2d	24	cf	07	57	8a	3b	d3	e8	ˆý¬ïšPè-$Ï.WŠ;Óè
000000a0	51	87	60	0d	6f	46	83	bc	c9	df	8f	6d	32	f5	4b	5b	Q‡`.oFƒ¼ÉßŽm2õK[
000000b0	e7	03	06	ce	61	f1	03	f2	7e	b6	ac	12	e1	fc	31	4d	ç..Îañ.ò~¶¬.áü1M
000000c0	02	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	.....

Figure 8: Start of encapsulated IP packet at byte offset 13.

SSH-tunneled-packet.bin

000001eb	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000000	45	10	00	b0	90	91	40	00	40	06	95	a4	0a	00	00	01	E..° '@.@.•¤....
00000010	0a	00	00	02	00	16	df	f6	94	18	f5	2d	01	9e	ac	88	.....ßö".õ-.ž¬ˆ
00000020	80	18	00	ec	0b	5b	00	00	01	01	08	0a	0d	11	9f	a3	€..ì.[........Ÿ£
00000030	b5	89	d7	dd	7f	cb	47	7c	3e	50	96	27	7c	d2	da	f5	µ‰×Ý Ë G|>P–'|ÒÚõ
00000040	02	ca	62	93	5d	51	e9	19	78	60	eb	1d	f8	51	e3	38	.Êb"]Qé.x`ë.øQã8
00000050	fa	0a	a0	49	0c	da	05	e1	a6	1e	e5	1b	1d	fc	62	a3	ú. I.Ú.á¦.å..üb£
00000060	69	f2	c4	08	2d	61	ba	00	a6	a9	de	22	51	db	34	f1	iòÄ.-aº.¦©Þ"QÛ4ñ
00000070	d9	45	17	20	72	09	e2	c7	44	1f	a2	10	27	de	a1	42	ÙE. r.âÇD.¢.'Þ¡B
00000080	e5	79	07	88	fd	ac	ef	9a	50	e8	2d	24	cf	07	57	8a	åy.ˆý¬ïšPè-$Ï.WŠ
00000090	3b	d3	e8	51	87	60	0d	6f	46	83	bc	c9	df	8f	6d	32	;ÓèQ‡`.oFƒ¼ÉßŽm2
000000a0	f5	4b	5b	e7	03	06	ce	61	f1	03	f2	7e	b6	ac	12	e1	õK[ç..Îañ.ò~¶¬.á
000000b0	fc	31	4d	02	..	..	..	..	..	..	..	..	..	..	..	..	ü1M.....

Figure 9: Source IP address of the tunnelled packet.

SSH-tunneled-packet.bin

00000107	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000000	45	10	00	b0	90	91	40	00	40	06	95	a4	0a	00	00	01	E..° '@.@.•¤....
00000010	0a	00	00	02	00	16	df	f6	94	18	f5	2d	01	9e	ac	88	.....ßö".õ-.ž¬ˆ
00000020	80	18	00	ec	0b	5b	00	00	01	01	08	0a	0d	11	9f	a3	€..ì.[........Ÿ£
00000030	b5	89	d7	dd	7f	cb	47	7c	3e	50	96	27	7c	d2	da	f5	µ‰×Ý Ë G|>P–'|ÒÚõ
00000040	02	ca	62	93	5d	51	e9	19	78	60	eb	1d	f8	51	e3	38	.Êb"]Qé.x`ë.øQã8
00000050	fa	0a	a0	49	0c	da	05	e1	a6	1e	e5	1b	1d	fc	62	a3	ú. I.Ú.á¦.å..üb£
00000060	69	f2	c4	08	2d	61	ba	00	a6	a9	de	22	51	db	34	f1	iòÄ.-aº.¦©Þ"QÛ4ñ
00000070	d9	45	17	20	72	09	e2	c7	44	1f	a2	10	27	de	a1	42	ÙE. r.âÇD.¢.'Þ¡B
00000080	e5	79	07	88	fd	ac	ef	9a	50	e8	2d	24	cf	07	57	8a	åy.ˆý¬ïšPè-$Ï.WŠ
00000090	3b	d3	e8	51	87	60	0d	6f	46	83	bc	c9	df	8f	6d	32	;ÓèQ‡`.oFƒ¼ÉßŽm2
000000a0	f5	4b	5b	e7	03	06	ce	61	f1	03	f2	7e	b6	ac	12	e1	õK[ç..Îañ.ò~¶¬.á
000000b0	fc	31	4d	02	..	..	..	..	..	..	..	..	..	..	..	..	ü1M.....

Figure 10: Destination port number of the tunnelled packet

tions, the IP address 10.0.0.2 is assigned to the tunnel interface on the client side

The encoding scheme is agreed upon, which, in this experiment, was Base128 encoding. The maximum transmission unit (MTU) for the tunnel interface is set to 1,130 bytes, which can also be configured by the -m switch, and the maximum fragment size is set to 1,174 bytes. The tunnel is then established and ready to transmit data. Figure 5 shows a screenshot from the client after establishing the tunnel. Figure 6 shows the ifconfig command output on the client illustrating the created virtual interface with IP address 10.0.0.2.

Due to the unreliable nature of DNS, Iodine will send frequent keep-alive messages between the client and the server to maintain the tunnel connection.[9] These messages will be very frequent and were sent to encoded subdomains of the controlled domain being used to set up the tunnel. A simple test of the connection was done by pinging the server IP address 10.0.0.1 from the client. The ping was successful, as shown in Figure 7.

During the ping, the traffic on the Ethernet interface of the client was captured using Wireshark. The output showed only DNS traffic originating from the address 10.0.2.15, which was the Ethernet interface of the client, to the destination IP address 94.177.172.165, which was the public IP of the Iodine server. A quick examination of the traffic based on IP addresses and protocol may erroneously lead to considering this traffic as legitimate DNS traffic between a client and a server. However, examining the Info section of the traffic points out that the queries and responses are all of the type NULL, which is not a usual DNS behaviour.

Carving out the encapsulated packet

In order to investigate how IP traffic will be encapsulated and sent over the NULL field, an SSH connection was made from the client to the server through the tunnel. The command used was:

```
ssh root@10.0.0.1
```

After entering the root password, the SSH connection was established successfully. The traffic was captured while the SSH session was running. As expected, the output showed a large number of DNS queries and responses of the NULL type. Examining the packet details for some of the DNS response packets showed that there is a significant amount of data stored in the NULL field. Using the 'Export packet bytes' feature of Wireshark, the data in the NULL field was exported and saved as a binary file. The binary file was opened through Neo hex editor to examine its contents.[10] Assuming that there was an IP packet encapsulated in this stream of data, the 'magic number' 0x4510 was found at byte offset 13 from the start. Figure 8 shows the binary data exported to the hex editor and the start of the encapsulated IP packet.

The bytes before the start of the presumed IP packet were deleted in order to make the analysis easier. Based on the IPv4 header format, the source IP address was located at bytes 12-15.[11] In hexadecimal, these bytes are 0x0a000001, which translates to 10.0.0.1 in decimal. Figure 9 shows the bytes representing the source IP address. The destination IP address is located at bytes 16-19. These bytes in hexadecimal are 0x0a000002, which translates to 10.0.0.2 in decimal. The protocol type is represented by byte 9. The value found is 0x06, which corresponds to TCP. The IP header length is 20 bytes, so after the IP header comes the TCP segment header. Bytes 0 and 1 represent the source port number. The value found was 0x0016, which translates to 22 in decimal. This corresponds to the port number used for SSH. Bytes 2 and 3 correspond to the destination port number. The value found was 0xdff6, which translates to 57334 in decimal. Figure 10 shows the destination port number for the tunnelled packet.

From the above analysis, it can be concluded that the SSH traffic was indeed encapsulated and sent over the DNS NULL field.

```
Action Stats:
     Alerts:            12 (  3.371%)
     Logged:            12 (  3.371%)
     Passed:             0 (  0.000%)
Limits:
      Match:             0
      Queue:             0
        Log:             0
      Event:           123
      Alert:             0
Verdicts:
      Allow:           356 (100.000%)
      Block:             0 (  0.000%)
    Replace:             0 (  0.000%)
  Whitelist:             0 (  0.000%)
  Blacklist:             0 (  0.000%)
     Ignore:             0 (  0.000%)
      Retry:             0 (  0.000%)
===============================================================================
+----------------------[filtered events]----------------------------------------
| gen-id=1      sig-id=800001     type=Threshold tracking=src count=10  seconds=5   filtered=123
Snort exiting
```

Figure 11: Number of alerts generated by detection rule – method 1.

```
[**] [1:800001:1] High NULL requests - Potential DNS Tunneling [**]
[Priority: 0]
05/22-20:26:27.175030 172.16.0.12:33586 -> 80.211.3.127:53
UDP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:153 DF
Len: 125
```

Figure 12: Snort alert – method 1.

```
Action Stats:
     Alerts:             2 (  0.562%)
     Logged:             2 (  0.562%)
     Passed:             0 (  0.000%)
Limits:
      Match:             0
      Queue:             0
        Log:             0
      Event:           268
      Alert:             0
Verdicts:
      Allow:           356 (100.000%)
      Block:             0 (  0.000%)
    Replace:             0 (  0.000%)
  Whitelist:             0 (  0.000%)
  Blacklist:             0 (  0.000%)
     Ignore:             0 (  0.000%)
      Retry:             0 (  0.000%)
===============================================================================
+----------------------[filtered events]----------------------------------------
| gen-id=1      sig-id=5619501    type=Limit     tracking=src count=1   seconds=300 filtered=134
| gen-id=1      sig-id=5619500    type=Limit     tracking=src count=1   seconds=300 filtered=134
Snort exiting
```

Figure 13: Number of alerts generated by detection rule – method 2.

Proxying HTTP traffic

A common use for DNS tunnelling is to bypass captive portals to access free wifi or to bypass Internet access policies to access blocked websites. In both these scenarios DNS traffic needs to be allowed through any perimeter firewall in order for DNS tunnelling to work.

Another experiment was conducted in order to proxy web browsing traffic through the tunnel. In order to set up the proxy for HTTP traffic, SSH port forwarding was used to create a SOCKS proxy on the client. This was done using the command:

```
ssh root@10.0.0.1 -N -D 9080
```

In the browser proxy settings, the manual proxy configuration was selected and the browser was set up to forward all HTTP traffic to the localhost address 127.0.0.1 on port 9999. This traffic was directed through the SSH tunnel setup using the SSH port forwarder, which in turn was encapsulated as DNS traffic.

This technique is known as double tunnelling. This setup will route all HTTP traffic from the client through the DNS tunnel. To check that the browsing traffic was indeed achieved through the DNS tunnel, the website www.whatismyipaddress.com was visited. The result showed that the IP address was the public IP address of the Iodine tunnel server.

```
[**] [1:5619500:1] covert iodine tunnel request [**]
[Priority: 0]
05/22-20:26:21.919441 172.16.0.12:33586 -> 80.211.3.127:53
UDP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:83 DF
Len: 55

[**] [1:5619501:1] covert iodine tunnel response [**]
[Priority: 0]
05/22-20:26:22.001081 80.211.3.127:53 -> 172.16.0.12:33586
UDP TTL:48 TOS:0x28 ID:0 IpLen:20 DgmLen:86 DF
Len: 58
```

Figure 14: Snort alerts – method 2.

```
Action Stats:
    Alerts:            2 (  0.562%)
    Logged:            2 (  0.562%)
    Passed:            0 (  0.000%)
Limits:
     Match:            0
     Queue:            0
       Log:            0
     Event:           28
     Alert:            0
Verdicts:
     Allow:          356 (100.000%)
     Block:            0 (  0.000%)
   Replace:            0 (  0.000%)
 Whitelist:            0 (  0.000%)
 Blacklist:            0 (  0.000%)
    Ignore:            0 (  0.000%)
     Retry:            0 (  0.000%)
===============================================================================
+----------------------[filtered events]----------------------------------------
| gen-id=1      sig-id=5619500    type=Threshold tracking=src count=10  seconds=5   filtered=28
Snort exiting
```

Figure 15: Number of alerts generated by detection rule – method 3.

```
[**] [1:5619501:1] covert iodine tunnel response [**]
[Priority: 0]
11/28-19:57:57.653823 94.177.172.165:53 -> 10.0.2.15:37688
UDP TTL:64 TOS:0x0 ID:5428 IpLen:20 DgmLen:130
Len: 102
```

Figure 16: Snort alerts – method 3.

```
Action Stats:
    Alerts:           22 (  6.180%)
    Logged:           22 (  6.180%)
    Passed:            0 (  0.000%)
Limits:
     Match:            0
     Queue:            0
       Log:            0
     Event:          255
     Alert:            0
Verdicts:
     Allow:          356 (100.000%)
     Block:            0 (  0.000%)
   Replace:            0 (  0.000%)
 Whitelist:            0 (  0.000%)
 Blacklist:            0 (  0.000%)
    Ignore:            0 (  0.000%)
     Retry:            0 (  0.000%)
===============================================================================
+----------------------[filtered events]----------------------------------------
| gen-id=1      sig-id=5619500    type=Threshold tracking=src count=10  seconds=2   filtered=255
Snort exiting
```

Figure 17: Number of alerts generated by detection rule – method 4.

```
[**] [1:5619500:1] High subdomains requests [**]
[Priority: 0]
11/28-20:00:27.116108 94.177.172.165:53 -> 10.0.2.15:51394
UDP TTL:64 TOS:0x0 ID:5551 IpLen:20 DgmLen:1177
Len: 1149
```

Figure 18: Snort alerts – method 4.

Tunnel detection

Several detection rules were tested in order to detect Iodine DNS tunnelling activity. Traffic was captured during an SSH connection from the Iodine client to the server through the DNS tunnel and detection rules were tested against this packet capture. Detection was implemented using Snort IDS rules that were configured under DNS-tunnel.rules as follows:

```
snort -r SSH-capture.pcapng -c
rules/DNS-tunnel.rules
```

Method 1: High frequency of DNS requests. Here, detection was based on the high frequency of DNS requests that contain data in NULL field. The rule counts the frequency of DNS requests within five seconds.

```
alert udp any any -> any 53
(msg:"High NULL requests
– Potential DNS Tunneling";
content:" – 01 00 – "; threshold:
type threshold, track bysrc, count
10, seconds 5; sid: 800001; rev: 1)
```

The rule generated 12 alerts as shown in figure 11. The alert logged by Snort for this rule is shown in figure 12.

Method 2: Specific byte content. Detection was based on specific byte content which can be found in these packet flows.

```
alert udp any any -> any 53
(content:" – 01 00 00 01 00 00
00 00 00 01 – "; offset:2; depth:
10; content:" – 00 00 29 10 00
00 00 80 00 00 00 – "; msg:
"covert iodine tunnel request";
threshold: type limit, track
bysrc, count 1, seconds 300; sid:
5619500; rev: 1;) alert udp any
53 -> any any (content: " – 84
00 00 01 00 01 00 00 00 00 – ";
offset:2; depth: 10; content:"
– 00 00 0a 00 01 – "; msg:
"covert iodine tunnel response";
threshold: type limit, track
bysrc, count 1, seconds 300; sid:
5619501; rev: 1;)
```

These rules generated two alerts as shown in figure 13. Alerts logged by Snort for these rules are shown in figure 14.

Method 3: Existence of an encapsulated IP packet. Detection was based on the existence of an IP packet encapsulated in the DNS response. The rule matches the magic number 0x4510 which is the start of the IP header of the encapsulated packet.

```
alert udp any any -> any any (content:" – 45 10 – "; msg: "covert iodine tunnel request IP packet encapsulated";offset:13;threshold: type threshold, track bysrc, count 10, seconds 5;sid: 5619500; rev: 1;)
```

The rule generated two alerts as shown in figure 15. The alerts logged by Snort for this rule are shown in figure 16.

Method 4: Number of DNS requests. Detection was based on the number of DNS requests for many subdomains in a threshold over a certain amount of time.

```
alert udp any any -> any any (pcre:"/[A-Za-z0-9](?:[A-Za-z0-9]0,61[A-Za-z0-9])?/i"; msg: "High subdomains requests"; threshold: type threshold, track bysrc, count 10, seconds 2;sid: 5619500; rev: 1;)
```

Some 22 alerts were generated by this rule as shown in Figure 17. The alert logged by Snort for this rule is shown in Figure 18.

Conclusion

DNS traffic can easily be used as a covert channel through which any type of IP traffic can be encapsulated. There are many DNS tunnelling tools available online that do not require a significant level of skill to use and hence organisations need to be aware of such a potential security policy breach.

Several detection rules exist that can help in the detection of such malicious activity. In this research, several existing and newly written Snort IDS rules were tested and were successful in the detection of DNS tunnelling activity.

About the authors

Mouhammd Al-kasassbeh graduated from the school of computing, Portsmouth University, UK in 2008. He is currently a full professor in the Computer Science Department at Princess Sumaya University for Technology. His research interests include network traffic analysis, network fault detection, network fault and abnormality classification and machine learning in the area of computer networking and network security.

Tariq Khairallah is a cyber security expert and security researcher with an interest in aligning security architectures, plans, controls, policies and procedures with the latest security standards and operational goals.

References

1. Jaworski, S. 'Using Splunk to detect DNS tunnelling'. SANS Institute, 1 Jun 2016. Accessed Nov 2019. www.sans.org/reading-room/whitepapers/dns/paper/37022.
2. Couture, E. 'Covert channels'. Sans Institute, 19 Aug 2010. Accessed Nov 2019. www.sans.org/reading-room/whitepapers/detection/paper/33413.
3. Davidoff, S; Ham, J. 'Network forensics: tracking hackers through cyberspace'. Prentice Hall, 2012.
4. Merlo, A; Papaleo, G; Veneziano, S; Aiello, M. 'A comparative performance evaluation of DNS tunnelling tools'. In Computational Intelligence in Security for Information Systems, Springer, 2011, pp.84-91.
5. Pearson, O. 'DNS tunnel – through bastion hosts'. Bugtraq, 13 Apr 1998. Accessed Nov 2019. http://seclists.org/bugtraq/1998/Apr/0079.html.
6. Iodine home page, 16 Jun 2014. Accessed Nov 2019. https://code.kryo.se/iodine/.
7. Sheridan, S; Keane, A. 'Detection of DNS based covert channels'. In European Conference on Cyber Warfare and Security, Academic Conferences International, 2015, p.267.
8. Iodine man page, Jun 2014. Accessed Nov 2019. https://code.kryo.se/iodine/iodine_manpage.html.
9. Free Hex Editor Neo, home page. HHD Software. Accessed Nov 2019. www.hhdsoftware.com/Downloads/free-hex-editor.
10. Snort home page. Accessed Nov 2019. www.snort.org.
11. Singalar, S; Banakar, R. 'Performance analysis of IPv4 to IPv6 transition mechanisms'. In 2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA), IEEE, 2018, pp.1-6.
12. Dietrich, C. 'Feederbot Botnet Using DNS as Carrier for Command and Control (C2)'. Prof Christian Dietrich personal blog, 2 Sep 2011. Accessed Nov 2019. https://chrisdietri.ch/post/feederbot-botnet-using-dns-command-and-control/.