# Command and Control Detection Over DNS Using Bro IDS

Stephan Davidson (davidste@sheridancollege.ca)

Ferdous Saljooki (saljooki@sheridancollege.ca)

Sheridan College (1430 Trafalgar Rd, Oakville ON, L6H2L1)

Nicholas Johnston (nicholas.johnston1@sheridancollege.ca)

Arsenii Pustovit (arsenii.pustovit@scalar.ca)

Stephan Davidson: ______________________________ (    /    /    )

Ferdous Saljooki: _______________________________ (    /    /    )

Nicholas Johnston: ______________________________ (    /    /    )

Arsenii Pustovit: _________________________________ (    /    /    )

## Table of Contents

## Abstract

In today's evolving network traffic, suspicious packets have been detected using various techniques, including masquerading through other protocols; in this case the Domain Name System (DNS) protocol. This project was designed to build upon an existing detection through the use of Intrusion Detection System (IDS) rules (specifically Bro IDS) to detect Command and Control (C2) traffic. Typically, research has been conducted to detect suspicious traffic using various protocols, which included (and heavily referenced) HTTP/S. This is due to the nature of the internet requiring connectivity to webpages via ports 80 / 443 and the vast amount of traffic that goes through these ports. DNS is another heavily used system, as it's required to resolve common names to Internet Protocol (IP) addresses, however research in this area was lacking and therefore became the area of concentration for this project, while building rulesets to combat such content. These rulesets solve a significant problem regarding DNS tunnelling and malicious payloads called through the DNS protocol (Kara et al. 1).

## Problem Statement

Current solutions for detecting and alerting of C2 and other suspicious network traffic can develop a false negative impression within a defence team. Bro IDS being deployed in an 'out-of-the-box' / default mode, does not include rules to detect such traffic, which can compound that negative impression. Rulesets are created to enhance the detection capabilities of the IDS, thus allowing a defending team to adequately respond to such threats. Vectra produced an article, "Five ways cybercriminals conceal command-and-control communications" ("Five Ways" 1), which highlight some of the challenges that detecting methods currently face in areas such as encryption, hidden tunnels, proxies, etc. Sans, “Detecting DNS tunnelling” (Farnham 1), also provides a method of communication that can be hidden from traditional detection methods, which gives us an area to focus on for this proof of concept. While these solutions capture the majority of current threats, they don’t capture everything since the threat space is ever changing. While DNS is being monitored, there are ways to bypass current solutions. Finding those exploits and patching them are crucial in network defence. Previous research in this field primarily focused on the detection of DNS tunnelling, however malware authors are also using DNS for exfiltration purposes (Nadler et al. 1). This area was explored to improve detection and a simplistic approach was used to mitigate any and all rogue DNS traffic, detailed below, in the following sections.

## Previous Works

DNS is often overlooked as being an attack channel for C2 communications such as data exfiltration and backdoor authorized access to services and resources. Unlike other well-known protocols, such as HTTP/S, minimal research and detection techniques have been consideration for DNS.

Extensive research has been conducted on the detection of HTTP/S C2 traffic through the analysis of TLS handshakes, statistical features, and other network anomalies. In 2017, Cisco Security published findings on detecting encrypted C2 traffic by examine TLS client and server handshakes (Anderson 1). Similarly, IEEE had published research on anomaly based detection of HTTP/S traffic by studying the number, frequency, length, and order of request URLs (Sakib and Huang 1). Current research for DNS C2 traffic focuses on detecting DNS tunnelling and botnets (Farnham 1). St. Cloud State University presented findings for multi-stage detection techniques for

Domain Generation Algorithm, Fast Flux, and Domain Flux botnets (Jammal 1). Jammal's research also improved detection for encrypted DNS tunnels by analyzing the initial connection establishment. Malware in the wild has been utilizing the DNS protocol to encode, encrypt, and structure C2 commands (Brumaghin and Grady 1). Previous work has concentrated on identifying these evasive techniques along with theoretical approaches for detection (Dietrich et al. 1). As part of this research, finding areas within this space to improve upon was challenging as there has been minimal development on the detection of commands/payloads being passed through legitimate DNS queries/answers, heuristic approaches for detecting DNS C2, and identifying DNS tunnelling through whitelisting. Work completed within this project, has been loosely coupled with social engineering concepts, solidified through previous work experiences within the penetration testing world by bridging macro embedded documents using DNS TXT calls to build the final payload and finally building out detection methods for such malware.

## Proposed Solution

Within the beginning stages of this project consideration to building out a custom solution (Ground up software, similar to an IDS but specific to C2 traffic only) was explored, however the goal was about detection of the threats and there are many great solutions that already offer most of what we attempted to accomplish. Additionally, the thought of a full out IPS was explored but was quickly shelved (and remained so) due to the additional amount of work that would be required in a 6 month timeframe. Ultimately, using an already commercial / community software and building upon it was the best course of action to pursue.

The scope was originally to research (with the possibility of detection) C2 network traffic, within DNS tunnelling, through the use of rulesets for an IDS. These rulesets would detect and alert defence teams regarding encrypted and non-encrypted traffic. The solution would be for detection only and IPS rulesets were not explored. The rulesets would have incorporated metrics to analyze: size, any obfuscation techniques used, frequency of requests, entropy, record types, signatures and geographic data. During development, it was discovered that some of these metrics were beyond a reasonable understanding of potential implementation and forced a change in scope.

Bro IDS is a very extensible and it’s syntax is compatible to other programming languages. That said, it came with several challenges, starting with the lack of user friendly documentation about how to write complex rules (anything more than the quick demo at http://try.bro.org was challenging as not all the syntax is explained). An idea to expand our scope to include all ports of traffic and all vectors within a network was explored but this had also remained shelved to best utilize the IDS for a specific role on the network.

## Implementation

The implementation of this project was completed over several areas, including building a lab environment, detailed in the System Overview, adapting to the unknown Bro IDS syntax and overall software, research into areas that may have already been developed (in an effort to not re-invent the wheel), learning to create the payloads that we intended to detect, as well as any other potentially malicious DNS traffic and then actually discovering them via putting this project all together.

## System Overview

The overall project was developed within a virtual environment that had Internet connectivity through a dual router setup. The edge router was an Apple Extreme, used to bring internet traffic into the network. A Dell Poweredge T420 with the Type-1 Hypervisor, VMWare ESXi 6.5 was used to host the virtual machines (VM), including a Cisco CSR 1000v virtual router. The main chassis housed 96GB of ram, dual 6-core 1.9gHz CPUs and 2TB SATA SSD (raid 0) Server Storage. A client and an Attacker network were setup on opposite sides of the Cisco router, isolating them in their own segments; each segment was carried on an unmanaged virtual switch, with the client network in promiscuous mode (this was to allow the Intrusion Detection System (IDS) to see all required network traffic.

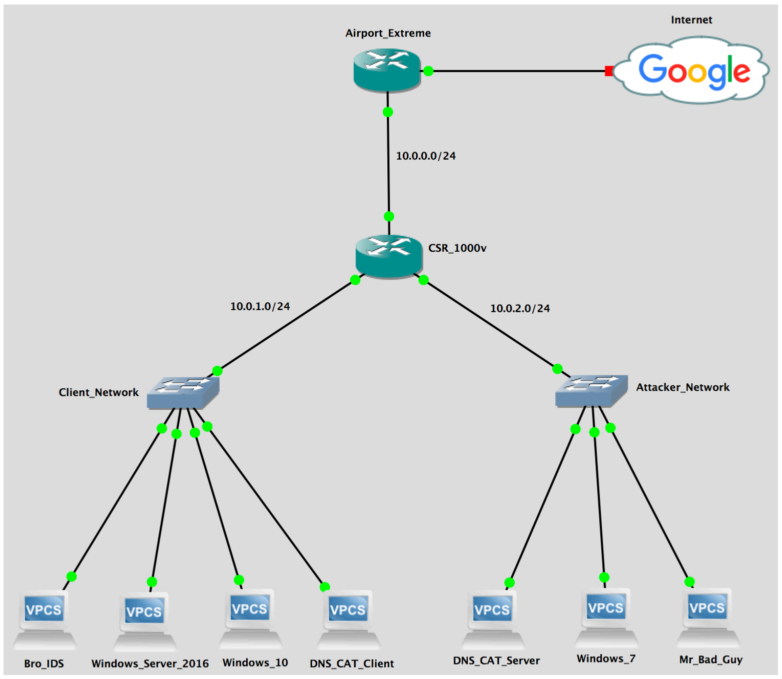


**FIGURE 5-1: NETWORK DIAGRAM**

4 VMs were created as follows:

- Bro IDS - The main focus of the project was to develop Bro Rules to detect Command and Control (C2) traffic via DNS. This VM hosted 8GB ram, 2-core CPU and 50GB SSD which served as more than enough of this proof of concept. This endpoint was in the Client Network.

- Client End Point (Windows 10) - The host was added as an attack vector for capture of traffic; specifically tested was the ability to detect a reverse shell initiated from this host, back to its respective C2. The VM hosted 4GB ram, 2-core CPU and 32GB SSD. This endpoint was in the Client Network.
- Attacker - The host was created solely as the C2 itself. The VM hosted 8GB ram, 4-core CPU and 50GB SSD. This endpoint was in the Attacker Network.

Additional VMs were added to the environment as follows:

- Windows Server 2016 - The host was added as an additional attack vector, as needed. The VM hosted 16GB ram, 4-core CPU, and 100GB SSD. This endpoint was in the Client Network.
- Windows 7 - The host was added to test the network itself and prove that the IDS was unable to see the traffic from another segment. The VM hosted 2GB ram, 1-core CPU and 32GB SSD. This endpoint was in the Attacker Network.
- DNSCAT Client - The host was added to test the ability to detect tunnelling within DNS. It's used in counterpart with the DNSCAT Server. The VM hosted 1GB ram, 1-core CPU and 16GB SSD. This endpoint was in the Client Network.
- DNSCAT Server - The host was added as the secondary component to DNSCAT Client. The VM hosted the same specifications as DNSCAT Client. This endpoint was in the Attacker Network.

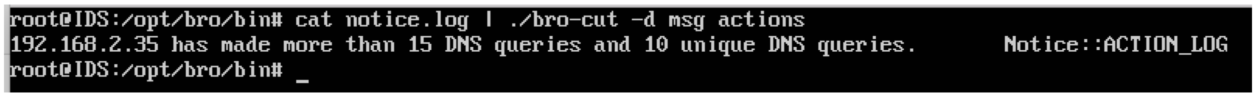
## Bro IDS

Bro IDS is a fairly extensible intrusion detection system that isn't widely used as other open source options, in particular Snort (Gunadi and Zander 1). That said, it's worth developing further due to the enhancements it has over other systems; for example, Bro is multi-threaded capable and therefore has a reduction in the loss of packets its analyzing. That said, where Bro really shines is in it’s logging capabilities. Data that’s logged is done so in a data rich (Bro uses a multi log approach that stores everything from connection states, network time, incoming / outgoing IPs, custom logging, etc. which is used in correlation to each other) sense, capable of comprehensively assisting with forensic reconstructions. Where Bro lacks is in its current documentation for developers to properly script within the Bro language.

### *Alerting and Logging in Bro*

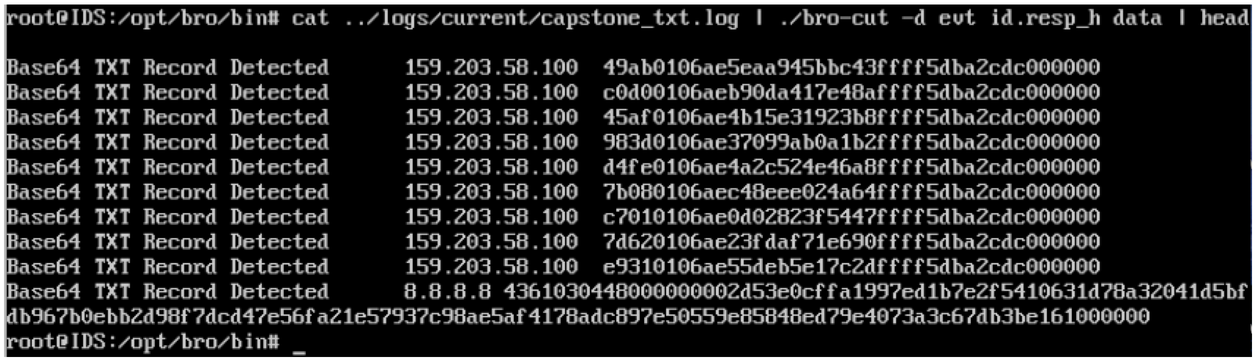
Within Bro, there are several mechanisms for developing alerts, logs, and other means of bringing awareness to a potential threat / malicious activity on a network. For this project, development to three of these (mechanisms; console notifications, email and logging to disk) were considered and eventually one was dropped (email notifications). The original idea to present an alert to the console, coupled with a log / notice entry in a custom log, followed by an email generated after hours, however in order to prevent the potential spam of emails to a user this mechanism was no longer supported.

Every event in Bro (that was developed to detect C2 traffic) means there’s some malicious activity happening and therefore every event required an alert to be generated. Since this project did not focus on integration, a simple command line output was sufficient. Additionally, a log entry was required, in the event a reconstruction (after the fact) was required. This is done in the form of a ‘Notice’ (Figure 5-2), which redirects the alert to a log file. Since Bro already logs these Notices through other events, it was best to create a custom log (Figure 5-3) to allow for granular inspection and cross reference with those notices. This was achieved by creating an info stream to capture specific connection data and custom messaging to those alerts. Both custom and console logging was used for every event.



```
root@IDS:/opt/bro/bin# cat notice.log | ./bro-cut -d msg actions
192.168.2.35 has made more than 15 DNS queries and 10 unique DNS queries.       Notice::ACTION_LOG
root@IDS:/opt/bro/bin# _
```

FIGURE 5-2: EXAMPLE OF NOTICE LOGGING OUTPUT USING BRO-CUT



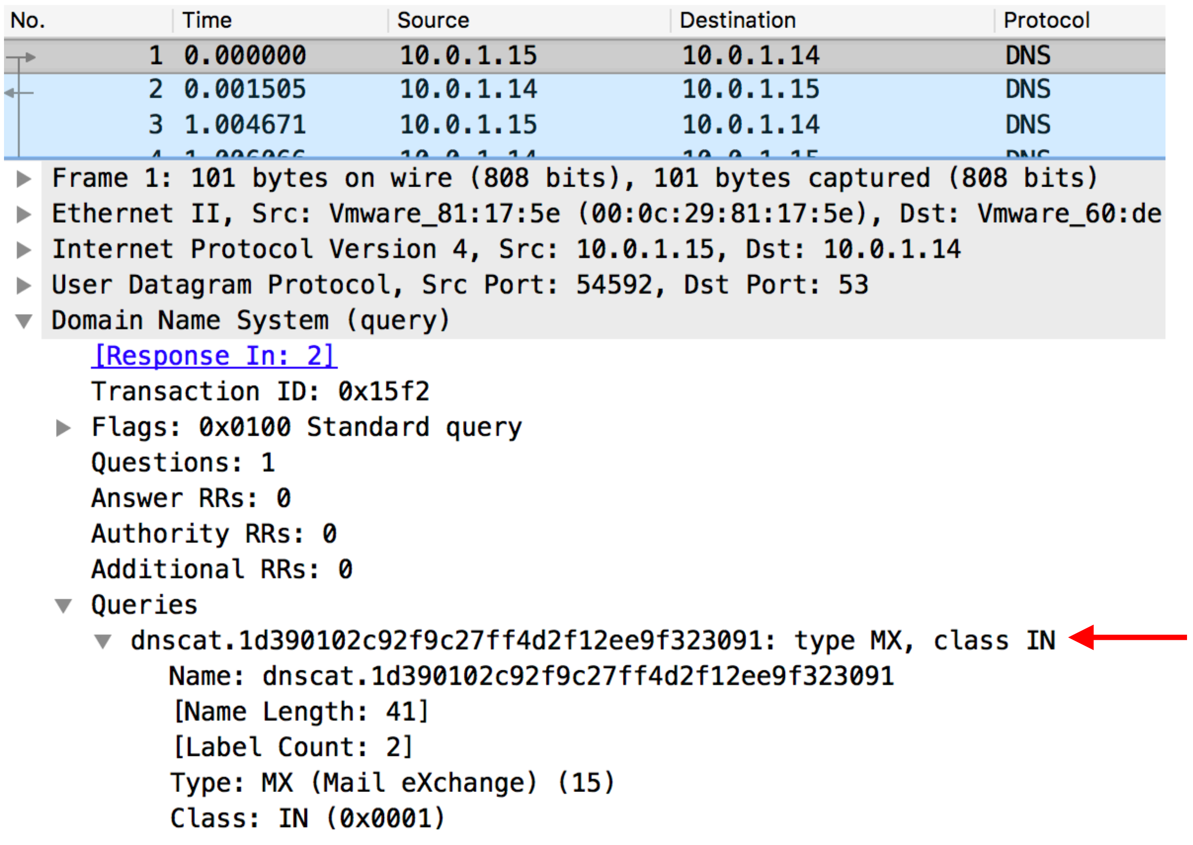
```
root@IDS:/opt/bro/bin# cat ../logs/current/capstone_txt.log | ./bro-cut -d evt id.resp_h data | head

Base64 TXT Record Detected      159.203.58.100  49ab0106ae5eaa945bbc43ffff5dba2cdc000000
Base64 TXT Record Detected      159.203.58.100  c0d00106aeb90da417e48affff5dba2cdc000000
Base64 TXT Record Detected      159.203.58.100  45af0106ae4b15e31923b8ffff5dba2cdc000000
Base64 TXT Record Detected      159.203.58.100  983d0106ae37099ab0a1b2ffff5dba2cdc000000
Base64 TXT Record Detected      159.203.58.100  d4fe0106ae4a2c524e46a8ffff5dba2cdc000000
Base64 TXT Record Detected      159.203.58.100  7b080106aec48eee024a64ffff5dba2cdc000000
Base64 TXT Record Detected      159.203.58.100  c7010106ae0d02823f5447ffff5dba2cdc000000
Base64 TXT Record Detected      159.203.58.100  7d620106ae23fdaf71e690ffff5dba2cdc000000
Base64 TXT Record Detected      159.203.58.100  e9310106ae55deb5e17c2dffff5dba2cdc000000
Base64 TXT Record Detected      8.8.8.8 4361030448000000002d53e0cffa1997ed1b7e2f5410631d78a32041d5bf
db967b0ebb2d98f7dcd47e56fa21e57937c98ae5af4178adc897e50559e85848ed79e4073a3c67db3be161000000
root@IDS:/opt/bro/bin# _
```

FIGURE 5-3: EXAMPLE OF CUSTOM LOGGING OUTPUT USING BRO-CUT

## Command and Control Detection

C2 Detection, as previously discussed, has been traditionally detected over HTTP/S as this port is generally open for regular browsing of legitimate web content. The overall issue is blending / hiding that malicious traffic within legitimate browsing. This is done using the protocol’s fields in an unintended way. For example, within the HTTP protocol, a GET / POST request is generated with(out) associated parameters, as commands. Within the HTTP/S response, below the final header, commands can be sent. Passing data, in this fashion, can be viewed / used and according to something like Wireshark (a network analyzing tool), the data is in the HTTP format, which is being ‘viewed’ as a television broadcast, it’s the tool is identifying the header as being valid. Since the majority of traffic through HTTP/S is destined to a browser, it’s considered content a legitimate user is searching for. This makes it difficult to determine the difference between a payload that has a Powershell command within it to a search for how Powershell is used. Since this area was already fully covered, another common port / protocol was analyzed; DNS. C2 over DNS is far more restrictive in its payload delivery and usage since the protocol specification doesn't allow for a lot of room for that data transfer. In fact, the maximum amount of characters that can be transmitted via DNS is 255 (if using a TXT records and significantly less than the HTTP/S data field), which is how this protocol is partially abused (Figure 5-4).

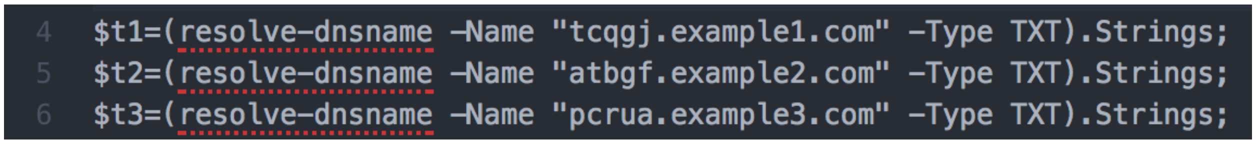


```
No.      Time        Source       Destination    Protocol
      1  0.000000    10.0.1.15    10.0.1.14      DNS
      2  0.001505    10.0.1.14    10.0.1.15      DNS
      3  1.004671    10.0.1.15    10.0.1.14      DNS
▶ Frame 1: 101 bytes on wire (808 bits), 101 bytes captured (808 bits)
▶ Ethernet II, Src: Vmware_81:17:5e (00:0c:29:81:17:5e), Dst: Vmware_60:de
▶ Internet Protocol Version 4, Src: 10.0.1.15, Dst: 10.0.1.14
▶ User Datagram Protocol, Src Port: 54592, Dst Port: 53
▼ Domain Name System (query)
     [Response In: 2]
     Transaction ID: 0x15f2
   ▶ Flags: 0x0100 Standard query
     Questions: 1
     Answer RRs: 0
     Authority RRs: 0
     Additional RRs: 0
   ▼ Queries
     ▼ dnscat.1d390102c92f9c27ff4d2f12ee9f323091: type MX, class IN
          Name: dnscat.1d390102c92f9c27ff4d2f12ee9f323091
          [Name Length: 41]
          [Label Count: 2]
          Type: MX (Mail eXchange) (15)
          Class: IN (0x0001)
```

**FIGURE 5-4: WIRESHARK CAPTURE OF DNSCAT2 QUERY.**

### *Malicious Payload*

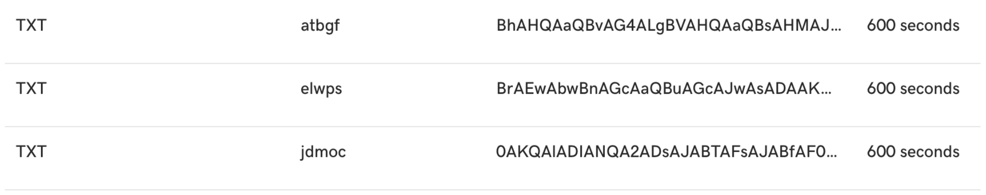
There are several tools that can be used for payload generation. Most utilize underlying system tools, such as NetCat (for linux targeting) or Powershell (for Windows) which can generate a reverse shell (a method to for the target to communicate back to the attacker) for beach-heading (initial entry to) a network. Utilizing the DNS protocol, the generated payload can be written into several TXT records and called from the malware on an infected machine (eg. A macro embedded MS Office Document). These TXT records can belong to one or many domains or sub-domains to mask their true purpose (Figure 5-5).



```
$t1=(resolve-dnsname -Name "tcqgj.example1.com" -Type TXT).Strings;
$t2=(resolve-dnsname -Name "atbgf.example2.com" -Type TXT).Strings;
$t3=(resolve-dnsname -Name "pcrua.example3.com" -Type TXT).Strings;
```

**FIGURE 5-5: SUBSET OF DNS QUERIES INSIDE A MACRO TO REBUILD THE MALICIOUS CODE.**

Detection from multiple angles is crucial to minimize a non-detection event, meaning the query itself needs to be analyzed rather than the source. Once a detection is made, queries can be correlated to each other to recover the actual payload. Of course, there are solutions to verify the legitimacy of a document (detonation through endpoint protection, signature matching through virus databases, etc) however a layered security approach is always best to cover as many entry points as possible (using both endpoint solutions, IDS and other will provide wider coverage to detection).

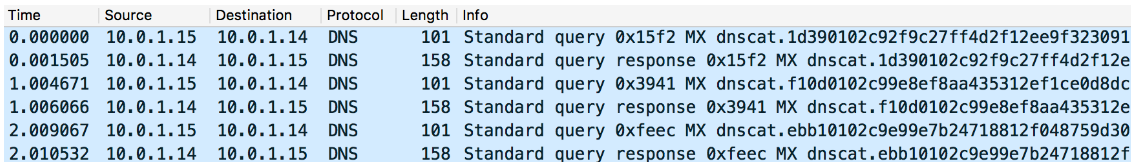


| | | | |
|---|---|---|---|
| TXT | atbgf | BhAHQAaQBvAG4ALgBVAHQAaQBsAHMAJ... | 600 seconds |
| TXT | elwps | BrAEwAbwBnAGcAaQBuAGcAJwAsADAAK... | 600 seconds |
| TXT | jdmoc | 0AKQAIADIANQA2ADsAJABTAFsAJABfAF0... | 600 seconds |

**FIGURE 5-6: DNS TXT RECORD (PAYLOAD) FROM EXAMPLE1.COM DOMAIN.**

### *DNS Tunneling*

DNS Tunnelling is a technique that exploits the protocol itself and uses the various parts of the header to create a 2 way communication between server (C2 Attacker) and client (its victim). Tunnelling is done either directly or indirectly. Direct tunnelling is between two IPs and not over the DNS server itself, which makes it fairly easy to detect. Indirect is using the use of an actual DNS server, where a client calls the C2 domain which is hosting it's own authoritative DNS and therefore resolves its own queries. When a client is running the client side of the malware, it consistently (every less than 1 sec) beacons (a 'hello' message to indicate that the client is still listening) the C2 server its associated to. When (if) the server responds, a connection is created and the server can then send commands to the client. These commands are wrapped into the TXT record of the header and is usually encoded or encrypted. When the client receives a command, its executed and a status message is sent back to C2. The reason for the constant beacons is that this protocol is over UDP (not TCP) and therefore is connectionless. It needs to maintain that connection via the beacons (Figure 5-7).



```
Time      Source     Destination Protocol Length Info
0.000000  10.0.1.15  10.0.1.14   DNS         101 Standard query 0x15f2 MX dnscat.1d390102c92f9c27ff4d2f12ee9f323091
0.001505  10.0.1.14  10.0.1.15   DNS         158 Standard query response 0x15f2 MX dnscat.1d390102c92f9c27ff4d2f12e
1.004671  10.0.1.15  10.0.1.14   DNS         101 Standard query 0x3941 MX dnscat.f10d0102c99e8ef8aa435312ef1ce0d8dc
1.006066  10.0.1.14  10.0.1.15   DNS         158 Standard query response 0x3941 MX dnscat.f10d0102c99e8ef8aa435312e
2.009067  10.0.1.15  10.0.1.14   DNS         101 Standard query 0xfeec MX dnscat.ebb10102c9e99e7b24718812f048759d30
2.010532  10.0.1.14  10.0.1.15   DNS         158 Standard query response 0xfeec MX dnscat.ebb10102c9e99e7b24718812f
```

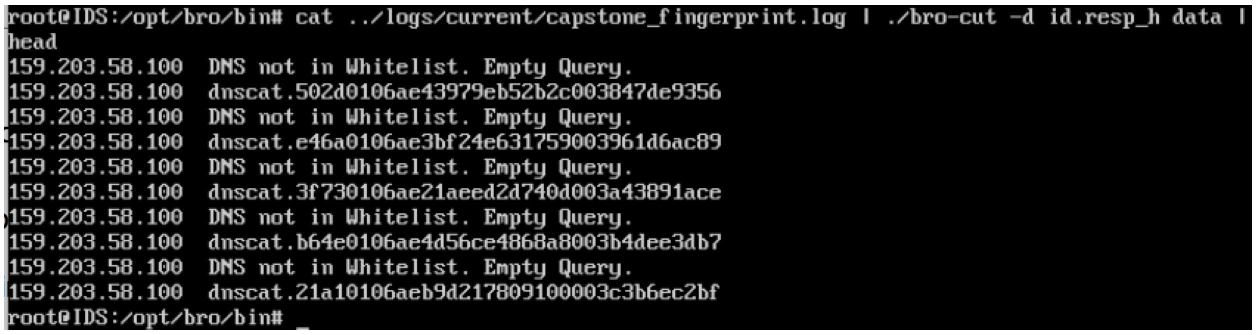
**FIGURE 5-7: WIRESHARK OF CLIENT BEACONS TO ATTACKER AND THE RETURN RESPONSE.**

**Threat Mitigation**

Research suggests an alternative approach might be to monitor and log DNS traffic within a network, with emphasis on detection of potential tunnelling, keyword blacklisting and metric analysis for abnormal behaviours.

Every network utilizes DNS for legitimate purposes; Call out to DNS to find out what a domains IP address is and follow it. In small business and home settings, this is usually the home router / default gateway that also forwards DNS resolution to (typically the ISPs DNS or) a user set known and trusted DNS. Similarly, larger companies and enterprises will do the same but may include an internal resolution of addresses and set specific addresses to override a higher level NS (this may be due to a business rule / policy). Nevertheless, those IP addresses are set and shouldn't change.

Within Bro, a rule was created to whitelist DNS IPs (Appendix A-1) that are trusted (this list can be modified, as required, for policy requirements). The rule then identifies ALL DNS traffic that is not to or from a trusted source as suspect and an alert is generated (Figure 5-8). This same rule has been modified to allow for a more



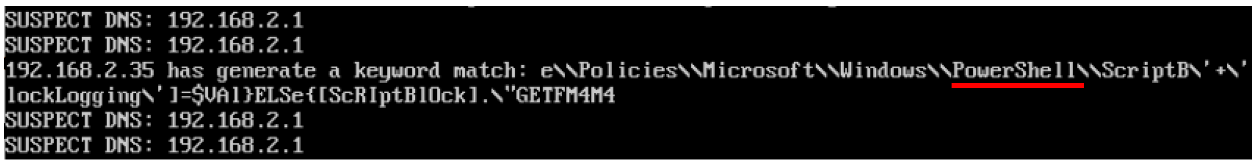
```
root@IDS:/opt/bro/bin# cat ../logs/current/capstone_fingerprint.log | ./bro-cut -d id.resp_h data |
head
159.203.58.100  DNS not in Whitelist. Empty Query.
159.203.58.100  dnscat.502d0106ae43979eb52b2c003847de9356
159.203.58.100  DNS not in Whitelist. Empty Query.
159.203.58.100  dnscat.e46a0106ae3bf24e631759003961d6ac89
159.203.58.100  DNS not in Whitelist. Empty Query.
159.203.58.100  dnscat.3f730106ae21aeed2d740d003a43891ace
159.203.58.100  DNS not in Whitelist. Empty Query.
159.203.58.100  dnscat.b64e0106ae4d56ce4868a8003b4dee3db7
159.203.58.100  DNS not in Whitelist. Empty Query.
159.203.58.100  dnscat.21a10106aeb9d217809100003c3b6ec2bf
root@IDS:/opt/bro/bin# _
```

**FIGURE 5-8: ALERT GENERATION FROM NON-WHITELISTED DNS.**

granular identification and has been successful for fingerprinting a particular DNS tunnelling software (in this instance, DNSCAT2) through the malware's use of TXT records within the response query. The confidence level of such a rule is high since policy will dictate known, legitimate, trusted Name Servers and anything else would be considered either a NS that a user is attempting to use against policy or an actual DNS tunnel.

Those same TXT records are used for legitimate purposes; for example, a mail server may be setup and TXT record(s) may indicate DMARC, DKIM and SPF settings (for that service). However, again, this record can be used to pass other info as the TXT record is an up to 255 user supplied character string, with no structure requirements. Armed with this information, if an attacker was to create a macro embedded document with a reverse shell payload, they may choose to embed that payload directly into the macro but the signature of such an attack can be caught through endpoint protections. Another option is to put that encoded payload into several TXT records and have the macro call in it once the document is delivered to the correct destination (As described in the section labelled Malicious Payload, Figures 5-5 and 5-6).

Another rule was created in Bro to combat such a scenario. After the threshold is reached (an overuse of TXT record queries within a small timeframe), the event checks the TXT payloads to verify certain keywords (powershell, veil, empire, etc) are detected (Figure 5-9); if they are generate an alert (Appendix A-2). Again, like the previous rule,



```
SUSPECT DNS: 192.168.2.1
SUSPECT DNS: 192.168.2.1
192.168.2.35 has generate a keyword match: e\\Policies\\Microsoft\\Windows\\PowerShell\\ScriptB\'+\'
lockLogging\']=$VAl}ELSe{[ScRIptBlOck].\"GETFM4M4
SUSPECT DNS: 192.168.2.1
SUSPECT DNS: 192.168.2.1
```

**FIGURE 5-9: ALERT DETECTING KEYWORD MATCH TO POWERSHELL.**

the confidence level for such a rule is high since there isn't a legitimate reason for one of the keywords to be in a TXT record. Note, this rule is not 100% detectible and requires further manipulation to increase it's capability. If an encode string is broken on the detectible word itself, it will not generate an alert. This is due to the word being incomplete.

## Way Ahead / Next Steps

There were areas within this project that were deemed out of scope from the beginning; included in this list is Intrusion Prevention Systems (IPS). Adding IPS functionality to this project, within the short timeframe, would have been too much to tackle but that doesn’t mean there isn’t a space for it. Adding the ability to automatically block or redirect traffic to a honeypot or some other endpoint solution would be extremely valuable. Detection can only be taken so far but to mitigate the attack, it has to be stopped. IDS doesn’t accomplish this.

Within alerting, there is room for adding support to send alert(s) to an email(s), bringing immediate attention to a potential threat. Bro offers such an ability however time constraints prevented this implementation and left the current iteration with just console alerts and file logging. The overall hangup for this project was the lack of understanding to mitigate the potential spam it would have caused.

Since this project is a proof of concept, fingerprinting other known DNS tunnels could eventually be done. Concentration towards fully printing DNSCAT2 was the goal. That said, there are several other tunnelling malware softwares out there however this project was left to those that are not fingerprinted, are merely identified as suspicious.

Being able to integrate WHOIS data (date created / registered) and verification of domain reputation (was xyz.com a known C2 Server via some generated list?) as a way to further amplify / solidify a risk rating, providing an enhanced dataset / probability scale was also attempted. Issues arose around how to call that data into Bro and more importantly the time constraint on each check. Further development in this area would be warranted.

A last area that was attempted was surrounding entropy. An attempt to threshold a subdomain's randomness in the name as a way to verify validity was deemed to be infeasible. If a subdomain was too small, the entropy would be too high, causing too many false positives. If the idea was to look at only subdomains that are greater than a threshold set in length, then too much would be skipped over, leaving a false sense that the rule was working but really wasn't. This could be reinvestigated in the future for an additional parameter to check against.

## Conclusion

The goal was to detect C2 traffic over DNS and it was demonstrated several ways. Before detection of such traffic, a significant learning curve was overcome; that traffic / malicious payloads was required to be crafted. A complete understanding of the DNS protocol was necessary to comprehend how to exploit it, then create a payload that can be detected.

Bro IDS itself was a challenge as it uses it's own syntax, which is not well documented for implementation. Learning this syntax was another obstacle and afforded the knowledge to write rulesets for the required detections.

Detection of DNS tunnelling, and fingerprinting of types of know tunnelling software was successful. Detection of malicious payloads within macro embedded documents that have a payload recall through DNS was also successful. While there is still significant room for enhancement to this project, the requirements were met and the code base has been left with room for those improvements to be easily integrated, as required.

## References

Anderson, Blake. “Detecting Encrypted Malware Traffic (Without Decryption).” *Blogs@Cisco - Cisco Blogs*, 7 July 2017, blogs.cisco.com/security/detecting-encrypted-malware-traffic-without-decryption.

Brumaghin, Edmund, and C. Grady. “Covert Channels and Poor Decisions: The Tale of DNSMessenger.” *Cisco's Talos Intelligence Group Blog: Take the RIG Pill: Down the Rabbit Hole*, 2 Mar. 2017, blog.talosintelligence.com/2017/03/dnsmessenger.html.

Dietrich, Christian J., et al. "On Botnets that use DNS for Command and Control." *2011 seventh european conference on computer network defense*. IEEE, 2011.

Farnham, Greg, and A. Atlasis. "Detecting DNS tunneling." *SANS Institute InfoSec Reading Room* 9 (2013): 1-32.

Five Ways Cybercriminals Conceal Command-and-Control Communications. Vectra, 2016, vectra.ai/assets/cybercriminals-conceal-command-and-control.pdf.

Gunadi, Hendra, and S. Zander. “Comparision of IDS Suitability for Covert Channels Detection.” Murdoch University, 2017.

Jammal, Wasseem. "Multi-Stage Detection Technique for DNS-Based Botnets." (2017).

Kara, A. Mert, et al. "Detection of malicious payload distribution channels in DNS." *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 2014.

Nadler, Asaf, A. Aminov, and A. Shabtai. "Detection of Malicious and Low Throughput Data Exfiltration Over the DNS Protocol." *arXiv preprint arXiv:1709.08395* (2017).

Sakib, Muhammad N., and C. Huang. "Using anomaly detection based techniques to detect HTTP-based botnet C&C traffic." *Communications (ICC), 2016 IEEE International Conference on*. IEEE, 2016.

## Appendices

### A-1: Whitelist Rule

```
@load ./main
@load ./whitelist

module WHITELIST;

event bro_init() {

}

global dns_whitelist = {
		8.8.8.8,		#Google
		8.8.4.4,		#Google
		2.2.2.2,		#L3 Communications
		2.2.2.3,		#L3 Communications
		10.0.1.1,		#Gateway / Defined DNS by Company Policy.  Add others, as needed.
		10.0.1.255,		#Broadcast Address
		224.0.0.251,	#mDNS
		224.0.0.252
};

global dns_ports = {
		53/udp,
		53/tcp,
		5353/udp,
		5355/udp,
		137/udp
};
```

## A-2: TXT Rule

```
@load ./main
@load ./txt

@load base/frameworks/sumstats

module TXT;

export {
        redef enum Log::ID += { LOG };

        type Info: record {
                evt: string &log;
                ts: time &log;
                id: conn_id &log;
                data: string &log;
        };
}

const excessive_limit: double = 15  &redef;
const time_interval = 30 secs &redef;
const scripting_languages = /veil|python|powershell/ &redef;
const base_64_string = /^[a-zA-Z0-9V"$+.]*={0,2}$/ &redef;

global r_queries = 0.0;
global r_unique = 0;

event bro_init() {
        Log::create_stream(TXT::LOG, [$columns=Info, $path="/opt/bro/spool/manager/capstone_txt"]);
        local r1 = SumStats::Reducer($stream="dns.observe", $apply=set(SumStats::SUM,
                 SumStats::HLL_UNIQUE));
        SumStats::create([$name="dns.queries",
            $epoch = time_interval,
            $threshold = excessive_limit,
            $reducers=set(r1),
            $threshold_val(key: SumStats::Key, result: SumStats::Result) = {
                return result["dns.observe"]$sum;
            },
            $threshold_crossed(key: SumStats::Key, result: SumStats::Result) = {
                local r = result["dns.observe"];
                print fmt("%s has made more than %.0f DNS queries and %d unique DNS queries.", key$host,
                        r$sum, r$hll_unique);
                r_queries = r$sum;
                r_unique = r$hll_unique;
            }
        ]);
}

event dns_TXT_reply(c: connection, msg: dns_msg, ans: dns_answer, strs: string_vec) {
        SumStats::observe("dns.observe", [$host=c$id$orig_h], [$str=c$dns$query]);
        if (r_queries != 0) {
                Log::write(TXT::LOG, [$evt="Excessive TXT Queries", $ts=network_time(), $id=c$id,
                        $data=fmt("Queries: %.0f. %d are unique", r_queries, r_unique)]);
                r_unique = 0;
                r_queries = 0;
        }
        local txt_str = split_string(c$dns$answers[0], / /)[2]; #TXT Data
        local txt_len = |txt_str|; #Length of the TXT Record as INT
        while ((txt_len % 8) != 0) { #Pads the string to 8 byte boundary for base64 decoding
                txt_str += "0";
                txt_len += 1;
        }
        local base_64 = match_pattern(txt_str, base_64_string);
        if (base_64$matched == T) {
                local s1 = decode_base64(txt_str); #Base64 decodes the string (attempts, regardless of encoding)
                local s2 = to_string_literal(s1); #Coverts String to literal string (changes hex values to string)
                local s3 = split_string(s2, /\\x[a-fA-F0-9]{2}/); #splits the string into chars
                local s4 = join_string_vec(s3, ""); #removes the split (weird but whatever)
                if (scripting_languages in to_lower(s4)) {
                        Log::write(TXT::LOG, [$evt="Malicious Keyword Match Detected", $ts=network_time(),
                                $id=c$id, $data=s4]);
                        print fmt("%s has generate a keyword match: %s", c$id$orig_h, s4);
                }
                else {
                        Log::write(TXT::LOG, [$evt="Base64 TXT Record Detected", $ts=network_time(),
                                $id=c$id, $data=txt_str]);
                        print fmt("Base64 Detected: %s", txt_str);
                 }
        }
}
```

## A-3: Direct Tunnelling Rule

```
@load ./main.bro
@load ./direct_tunnel.bro

module TUNNELLING;

export {
        redef enum Log::ID += { LOG };

        type Info: record {
                evt: string &log;
                ts: time &log;
                id: conn_id &log;
                data: string &log;
        };
}

event bro_init() {
        Log::create_stream(TUNNELLING::LOG, [$columns=Info, $path="/opt/bro/spool/manager/
                capstone_tunnelling"]);
}

event dns_message(c: connection, is_orig: bool, msg: dns_msg, len: count) {
        if (!(c$id$resp_h in dns_whitelist)) {
                if (c$id$resp_p in dns_ports) {
                        if (c?$dns) {
                                if (c$dns?$query) {
                                        #DNSCAT
                                        if (|(find_all(c$dns$query, /dnscat/))| == 1) {
                                                print fmt("Fingerprinted! DNSCAT Detected -- Attacking IP: %s
                                                        | Victim IP: %s", c$id$resp_h, c$id$orig_h);
                                                Log::write(TUNNELLING::LOG, [$evt="DNSCAT Detected",
                                                        $ts=network_time(), $id=c$id, $data=c$dns$query]);
                                        }
                                        #DNS2TCP -- Failed attempt to Fingerprint. Needs work.
                                        else if (|(find_all(c$dns$query, /dn8AAAA/))| == 1) {
                                                print fmt("Fingreprinted! DNS2TCP Detected -- Attacking IP:
                                                        %s | Victim IP: %s", c$id$resp_h, c$id$orig_h);
                                                Log::write(TUNNELLING::LOG, [$evt="DNS2TCP Detected",
                                                        $ts=network_time(), $id=c$id, $data=c$dns$query]);
                                        }
                                        #Unable to Fingerprint
                                        else {
                                                print fmt("SUSPECT DNS: %s", c$id$resp_h);
                                                Log::write(TUNNELLING::LOG, [$evt="Suspect DNS
                                                        Detected", $ts=network_time(), $id=c$id, $data="DNS
                                                        not in Whitelist. Unable to Fingerprint."]);
                                        }
                                }
                                else {
                                        print fmt("SUSPECT DNS: %s", c$id$resp_h);
                                        Log::write(TUNNELLING::LOG, [$evt="Suspect DNS Detected",
                                                $ts=network_time(), $id=c$id, $data="DNS not in Whitelist.
                                                Empty Query."]);
                                }
                        }
                        else {
```

```
                                        print fmt("SUSPECT DNS: %s", c$id$resp_h);
                                        Log::write(TUNNELLING::LOG, [$evt="Suspect DNS Detected",
                                                $ts=network_time(), $id=c$id, $data="DNS not in Whitlist. Port not in
                                                Whitelist."]);
                                }
                        }
                }
        }
```

## A-4: Indirect Tunnelling Rule

```
@load ./main.bro
@load ./indirect_tunnel.bro

module TUNNELLING;

export {
	redef enum Log::ID += { LOG };

	type Info: record {
		evt: string &log;
		ts: time &log;
		id: conn_id &log;
		data: string &log;
	};
}

event bro_init() {
	Log::create_stream(TUNNELLING::LOG, [$columns=Info, $path="/opt/bro/spool/manager/
		capstone_tunnelling"]);
}

global nums: table[int] of count &default=0 &write_expire=15secs;

event dns_request (c: connection, msg: dns_msg, query: string, qtype: count, qclass: count) {
	local elements = split_string(c$dns$query, /\./);
	local size_of_elements = 0;
	local string_size = 0;

	for (i in elements) {
		size_of_elements = size_of_elements + 1;
	}

	if (size_of_elements > 2) {
		for (i in elements) {
			if (i < (size_of_elements - 2)) {
				string_size = (string_size + |elements[i]|);
			}
		}
	} else {
		return;
	}

	if (string_size > 17) {
		if ((string_size == 146) && (nums[0] == 0)) {
			nums[0] = string_size;
		} else if ((string_size == 34) && (nums[2] == 1)) {
			nums[2] = 1;
		}
	} else {
		return;
	}
}

event dns_TXT_reply(c: connection, msg: dns_msg, ans: dns_answer, strs: vector of string) {
	local elements = split_string(c$dns$query, /\./);
	local size_of_elements = 0;
```

```
        local string_size = 0;
        for (i in elements) {
                size_of_elements = size_of_elements + 1;
        }
        if (size_of_elements > 2) {
                for (i in elements) {
                        if (i < (size_of_elements - 2)) {
                                string_size = (string_size + |elements[i]|);
                        }
                }
        } else {
                return;
        }

        if (string_size > 17) {
                if ((nums[0] == 146) && (nums[1] == 0) && (string_size == 82)) {
                        nums[1] = string_size;
                } else if ((nums[0] == 146) && (nums[1] == 82) && (string_size == 34)) {
                        print fmt("DNSCAT2 TUNNELLING Detected! Beacon out.");
                        Log::write(TUNNELLING::LOG, [$evt="DNSCAT2 Detected", $ts=network_time(),
                                $id=c$id, $data=c$dns$query]);
                        nums[0] = 0;
                        nums[1] = 0;
                        nums[2] = 1;
                } else if (!(nums[2] == 1)) {
                        print fmt("SUSPECTED TUNNELLING -- TXT Response: %d characters in a subdomain",
                                string_size);
                        Log::write(TUNNELLING::LOG, [$evt="SUSPECTED TUNNELLING -- TXT",
                                $ts=network_time(), $id=c$id, $data=c$dns$query]);
                }
        } else {
                return;
        }
}

event dns_CNAME_reply(c: connection, msg: dns_msg, ans: dns_answer, name: string) {
        local elements = split_string(c$dns$query, /\./);
        local size_of_elements = 0;
         local string_size = 0;
         for (i in elements) {
                size_of_elements = size_of_elements + 1;
        }
        if (size_of_elements > 2) {
                for (i in elements) {
                        if (i < (size_of_elements - 2)) {
                                string_size = (string_size + |elements[i]|);
                        }
                }
        } else {
                return;
        }

        if (string_size > 17) {
                if ((nums[0] == 146) && (nums[1] == 0) && (string_size == 82)) {
                        nums[1] = string_size;
                } else if ((nums[0] == 146) && (nums[1] == 82) && (string_size == 34)) {
                        print fmt("DNSCAT2 TUNNELLING Detected! Beacon out.");
                        Log::write(TUNNELLING::LOG, [$evt="DNSCAT2 Detected", $ts=network_time(),
                                $id=c$id, $data=c$dns$query]);
                        nums[0] = 0;
                        nums[1] = 0;
                        nums[2] = 1;
                } else if (!(nums[2] == 1)) {
                        print fmt("SUSPECTED TUNNELLING -- CNAME Response: %d characters in a
                                subdomain", string_size);
                        Log::write(TUNNELLING::LOG, [$evt="SUSPECTED TUNNELLING -- CNAME",
                                $ts=network_time(), $id=c$id, $data=c$dns$query]);
                }
        } else {
                return;
        }
}

event dns_MX_reply(c: connection, msg: dns_msg, ans: dns_answer, name: string, preference: count) {
        local elements = split_string(c$dns$query, /\./);
        local size_of_elements = 0;
        local string_size = 0;
        for (i in elements) {
                size_of_elements = size_of_elements + 1;
        }
        if (size_of_elements > 2) {
                for (i in elements) {
                        if (i < (size_of_elements - 2)) {
                                string_size = (string_size + |elements[i]|);
                        }
                }
        } else {
                return;
         }

        if (string_size > 17) {
                if ((nums[0] == 146) && (nums[1] == 0) && (string_size == 82)) {
                        nums[1] = string_size;
                } else if ((nums[0] == 146) && (nums[1] == 82) && (string_size == 34)) {
                        print fmt("DNSCAT2 TUNNELLING Detected! Beacon out.");
                        Log::write(TUNNELLING::LOG, [$evt="DNSCAT2 Detected", $ts=network_time(),
                                $id=c$id, $data=c$dns$query]);
                        nums[0] = 0;
                        nums[1] = 0;
                        nums[2] = 1;
                } else if (!(nums[2] == 1)) {
                        print fmt("SUSPECTED TUNNELLING -- MX Response: %d characters in a subdomain",
                                string_size);
                        Log::write(TUNNELLING::LOG, [$evt="SUSPECTED TUNNELLING -- MX",
                                $ts=network_time(), $id=c$id, $data=c$dns$query]);
                }
        } else {
                return;
        }
}
```

## A-5: DNSCAT2 Rule (Proof of Concept)

```
@load ./main
@load ./dnscat2

module DNSCAT2;

event bro_init() {}

event dns_request (c: connection, msg: dns_msg, query: string, qtype: count, qclass: count) {
	local elements = split_string(query, /\./);
	if ((|elements| > 1) && (elements[0] == "dnscat")) {
		print fmt("Proof of Concept - DNSCAT detected!");
	}
}
```