PSUDP: A Passive Approach to Network-Wide Covert Communication

Kenton Born
kenton.born@gmail.com

Black Hat USA 2010

Abstract

This paper explores taking a passive approach to covert communication over DNS. By exploiting the slack space that can be created in DNS packets, data may be inserted into packets without affecting the operation of DNS resolvers and security tools. Several locations in the packet exist that allow additional data to be inserted into the network traffic without being noticed by applications before or at the destination host. Unlike many of the low-bandwidth covert channels such as port numbers and identification fields, this method is an enabler for high-bandwidth communication between multiple systems.

This method is introduced through PSUDP, a tool that creates a network-wide messaging system by piggy-backing on legitimate network DNS traffic. By creating a broker system on the DNS server, clients can communicate by injecting messages and desired recipients into DNS requests destined for the broker. These messages are then held at the broker until they may be passively delivered to the appropriate client in a legitimate DNS response. By relying on traffic that is already traveling through the network, no additional packets must be created for this messaging system.

The programs discussed in this paper are proof of concept implementations that are harmless “as is”. The techniques are demonstrated in non-malicious tools so the security community can learn from them and be able to identify this method of communication. For this reason, advanced data hiding techniques and protocol features such as reliability and sequencing were left out of the tools.

Introduction

While a large number of application layer protocols exist, they are all built from a small number of transport and network layer protocols. Currently, it is uncommon for network devices to perform deep packet inspection (DPI), analyzing the actual application layer content of the data carried by the transport layers. Instead, most devices simply use the length field specified by the lower layer protocols to forward the appropriate number of bytes to the destination host.

Passive manipulation of network traffic implies that no additional network packets are created. Instead, existing network traffic is manipulated to carry additional information (often covert) between the two systems. Because DNS is used by nearly all hosts in a network, it is ideal for demonstrating the possibilities of covert communication over protocol slack space.

The domain name system (DNS) is a hierarchical network of systems responsible for resolving domain names to IP addresses. A fully qualified domain name (FQDN) is formed through a series of labels that separate it into subdomains, each one controlled by the subdomain to its right. RFC 1035 specifies the allowable characters as a-Z, 0-9, and dashes (Mockapetris 1987). Additionally, the RFC limits the labels to 63 octets or less, with the full domain being 255 octets or less.

DNS tunnels have gained significant popularity as a mechanism for bypassing network policies and infiltrating/exfiltrating data. Similarly, they have proven ideal for establishing communication links for botnets or other malicious software. Since DNS is required on all networks requiring internet access, it can easily be used to provide storage or timing channels for restricted protocols that would not have access out of the network, otherwise. Additionally, DNS traffic is typically less monitored than protocols such as HTTP or SMTP, often only being closely examined when issues occur.

Related Work

Several DNS tunnel implementations exist that allow policy-restricted application layer protocols (ALP) to be transported inside DNS traffic. Ozyman, TCP-over-DNS, Iodine, Dns2tcp, DNScat, and DeNiSe are all examples of popular DNS tunnels that allow users to bypass firewall restrictions. While they all use similar tactics for storing data in queries (encoding data in the subdomain), many of them use slightly different strategies for responses. Some tunnels, such as TCP-over-DNS (TCP-Over-DNS 2008) and Dns2tcp (Dembour 2008), use TXT records to exfiltrate data in responses. TXT records are convenient because they allow free form text to be included in the response. Other tunnels like Iodine (Iodine 2009) use NULL record types to store data. Neither method is particularly covert since heavy amounts of TXT or NULL record types coming from standard desktop systems should throw red flags in most networks. DNScat (Pietraszek 2004) uses yet another method of tunneling data in responses by creating custom CNAME records. While this method is slightly more complex, it offers greater covertness since CNAME records are a more common resource record type than TXT or NULL records.

Reverse DNS tunneling shellcode was explored by Ty Miller (Miller 2008). In his work, DNS tunnels were used because of their ability to escape internal networks easier than HTTP, which often requires authentication. Miller forced exploited clients to probe the attacker's domain, allowing commands to be tunneled back to the internal system.

Many interesting strategies were introduced with the release of Heyuka (Revelli 2009). Revelli and Leidecker showed that many DNS servers would accept binary data in domain name labels, increasing their bandwidth from 5 bits per character to 8 bits per character. Additionally, they took advantage of EDNS0 to increase the bandwidth ceiling from 512 bytes per packet to 1024 bytes per packet. Covertness was added by spoofing packets across a range of IP addresses instead of from a single system.

Recently, I showed how data could be exfiltrated over DNS without additional software or privileges by executing a local JavaScript file in a browser. In this work, it is demonstrated how DNS queries may be separated from their respective HTTP requests, creating a covert DNS storage channel. Additionally, it was

shown how low-bandwidth bidirectional tunnels could be created through both storage and timing channels (Born 2010).

The tools demonstrated in this work differ from all past work in that it uses a new, higher-bandwidth method of creating storage channels. Additionally the method proposed may be used passively by piggy-backing on legitimate traffic instead of actively generating new packets.

DNS Packet Length

When network devices receive a packet, they are able to use the length field specified at the IP layer to determine the size of the transport layer data. While not present in TCP, the UDP protocol additionally has its own length field that allows programs to calculate the size of the application layer data. According to RFC 1035 (Mockapetris 1987), DNS packets are restricted to a UDP length of 512 bytes. However, this can be extended to 1024 bytes by using the EDNS0 extension (Pixie 1999).

Every DNS query and response has two parts: the header section and the resource record sections. While the resource record sections are variable in length, the header is always contained in the initial 12 bytes. The header first contains an identification field followed by various parameters and flags describing the packet. Lastly, several fields describe how many of each of the four resource record types will be found in the resource record sections. This count information is used by the DNS parser to determine when the end of a section type is reached.

Each label in a stored domain name is preceded by an eight bit field specifying the length of the label. A parser knows it has reached the end of a domain when it reads a 0 from the length field. When a domain or a list of labels has been declared by a previous resource record, a pointer to that location may be used instead to reduce the size of messages.

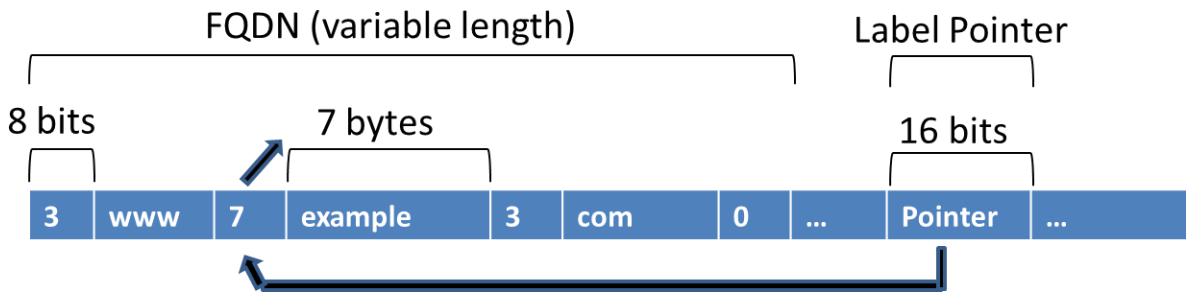

Figure 1: Label Format

Use of the compressed DNS form is optional for servers, making the packet length of domain names variable. For this reason, the parser has no way of easily calculating how much storage space is actually required for a set of domains. Similarly, many resource record types have a variable RDATA length (retrieved from the RDLENGTH field). Both the domain length and the RDLENGTH are required before knowing the total resource record size.

Unlike the IP and UDP headers, The DNS header does not store the total length of the packet. This information must be derived through the UDP and IP headers which store the total length of both their headers and data, respectively.

Defeating the Parser

Because DNS headers do not contain information about the length of the resource records or total packet, parsers rely on the number of resource records specified in the DNS header to determine when to stop parsing the data. When the last resource record specified in the header has been parsed, it is assumed that the end of the data has been reached. However, by manipulating the IP and UDP headers to account for additional length, it is possible to append any amount of binary data to the end of the DNS packet without adversely affecting how DNS servers and resolvers react to manipulated DNS queries and responses.

Since the storage channel is not restricted to label-based characters or special formatting like many channels used in DNS tunnels, a significantly higher ceiling of data storage may be reached. Additionally, the use of non-ASCII characters is much less identifiable when the raw packets are examined. By modifying the IP and UDP length headers, the additional data will not be dropped as the packet is passed through the network.

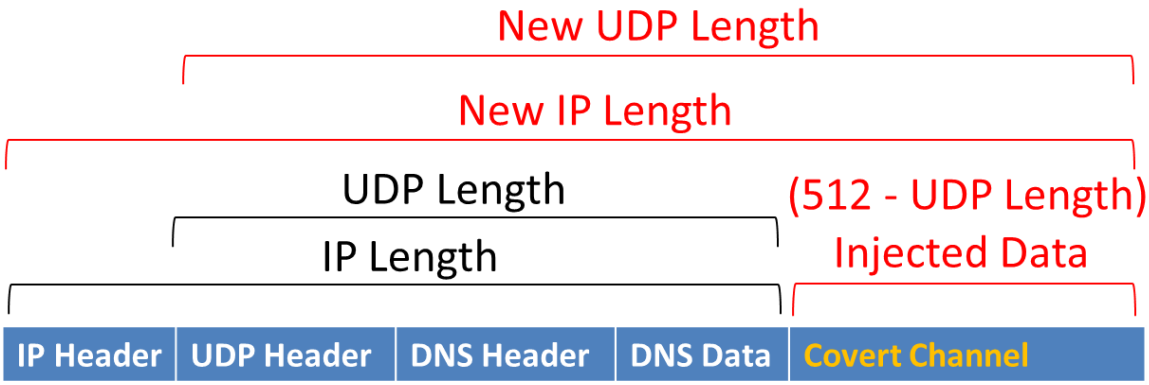

Figure 2: Injected Packet

The storage channel capacity of each packet may be calculated by looking at the UDP header's length field. For example, if the UDP packet length is 200 bytes, the storage channel capacity can be calculated as 512-200 = 312 bytes. The ceiling for DNS packet length may be extended to 1024 bytes by combining this strategy with the EDNS0 extension.

One problem with this strategy is that the covert channel is always located at the end of the data, simplifying detection. An ideal channel would allow the data to be more covertly hidden in the middle of the packet, making it more difficult to identify when looking at the packets through analysis tools such as Wireshark.

Raising the Bar

Previously, it was shown how DNS label compression may be used to save space in DNS packets. When a pointer is used, it points to a previous location in the packet describing similar labels. However, tests show that DNS parsers do not require pointers to point only to previous positions in the packet. This was seen in past DDOS attacks where DNS packets were crafted to contain self-referencing or cross-referencing pointers, often crashing applications (Securiteam 1999). Using a similar strategy, a label may be copied to a position after the injected data, adjusting the relevant pointer to account for the new position.

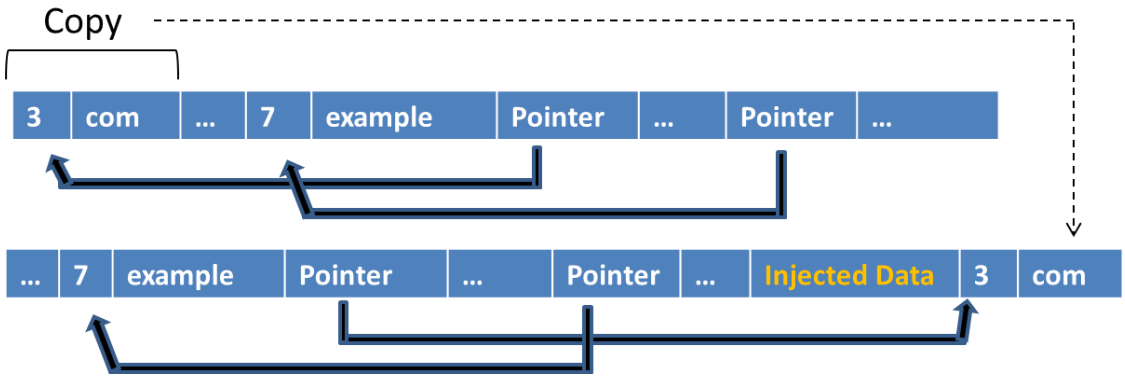

Figure 5: Forward pointer with injected data

Detection

To programmatically detect this type of covert communication, one might initially approach the problem by keeping track of the furthest position in the packet analyzed while parsing the DNS resource records. If the UDP size extends past this point, then it is a good sign that there may be data injected into the packet. However, this strategy fails against the more sophisticated injection method where pointers are manipulated to point to labels following the injected data. This will require the parser to read from the furthest point in the manipulated packet, registering the UDP length as legitimate.

Instead of simply keeping track of the furthest point accessed in the packet, a detection agent must mark every position legitimately accessible and accounted for during DNS parsing. If any unmarked positions exist in the packet, then it is highly likely that data was injected into the packet. Alternatively, this can be simplified by verifying two properties: all pointers must point backwards, and the last legitimate point in the DNS packet must match the length given in the UDP packet.

For this method of covert communication to be effective, it must be able to pass by intrusion detection systems and packet analyzers such as Wireshark without being flagged as suspect. At this point, no tool has been found that identifies the packet as unusual, making detection only possible when examining the raw data manually. However, it is very rare for raw DNS data to be investigated without first being alerted to an issue. Below, Figure 4 depicts a screenshot of Wireshark displaying a heavily injecting DNS packet as having no errors and containing the appropriate resource records.

```
192.168.0.104   192.168.0.107   DNS   Standard query response CNAME mt.l
▷ Flags: 0x8180 (Standard query response, No error)
  Questions: 1
  Answer RRs: 5
  Authority RRs: 4
  Additional RRs: 0
00a0  02 00 01 00 02 a1 bf 00  06 03 6e 73 31 c0 10 c0   ........ ..ns1...
00b0  10 00 02 00 01 00 02 a1  bf 00 06 03 6e 73 33 c0   ........ ....ns3.
00c0  10 c0 10 00 02 00 01 00  02 a1 bf 00 06 03 6e 73   ........ ......ns
00d0  32 c0 10 c0 10 00 02 00  01 00 02 a1 bf 00 06 03   2....... ........
00e0  6e 73 34 c0 10 4e 65 76  65 72 20 67 6f 6e 6e 61   ns4..Nev er gonna
00f0  20 67 69 76 65 20 79 6f  75 20 75 70 2c 20 4e 65    give yo u up, Ne
0100  76 65 72 20 67 6f 6e 6e  61 20 6c 65 74 20 79 6f   ver gonn a let yo
0110  75 20 64 6f 77 6e 2c 20  4e 65 76 65 72 20 67 6f   u down,  Never go
0120  6e 6e 61 20 72 75 6e 20  61 72 6f 75 6e 64 20 61   nna run  around a
0130  6e 64 20 64 65 73 65 72  74 20 79 6f 75 2e 20 4e   nd deser t you. N
```

Figure 4: Wireshark output with injected lyrics

PSUDP: Network-Wide Covert Communication

It has been shown how additional data may be appended to the end of legitimate DNS traffic. With this knowledge, it can be quickly seen that having a control point at a DNS server gives passive communication access to most machines in the network. This strategy is used in PSUDP, a program that allows a system (typically a DNS server) to act as a broker between systems running the client program.

The name PSUDP (pronounced sūdēpē) was arrived at for several reasons. Firstly, it is playing off of the "postscript" (p.s.) abbreviation found on written works meaning "that which comes after the writing". Secondly, it is short for "Pseudo UDP", since it uses an alternative UDP-like protocol on top of the existing UDP/DNS packets. Lastly, it is playing off of "sudo UDP", because it is manipulating UDP to allow for a little extra power!

PSUDP clients route all communication through the broker by appending a custom protocol on the end of legitimate traffic. Firstly, the client may send messages to any other client by appending the recipient address and custom message on the end of a DNS query. When the DNS server receives and forwards this query, the broker strips out the message and recipient data, storing it for later delivery. At this point, the broker will wait until it has the opportunity to append the message to a response destined for the recipient.

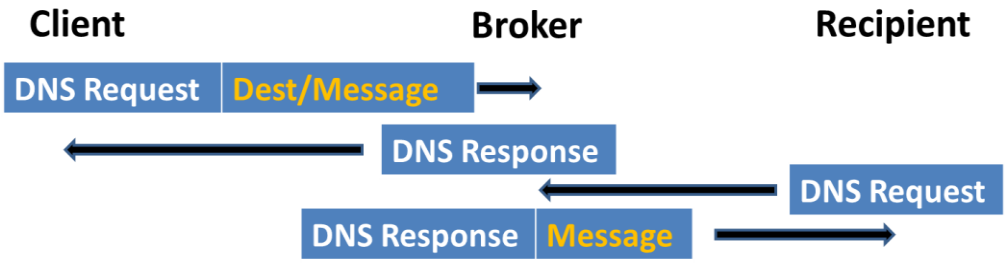

Figure 3: PSUDP Flow

Clients and brokers are able to detect when the protocol has been injected with additional data by parsing the DNS data and determining the actual end of the resource record sections in relation to the specified UDP length.

Implementation Details

The primary PSUDP program is separated into three executables: broker, client, and psudp. Both the broker and client use libnetfilter_queue to manipulate packets going in and out of systems. This library allows packets to be inspected and mangled through userspace programs by providing an API into the kernel packet filter. Combined with the interception and rerouting capabilities of iptables, this allows the client and broker to examine and strip packets before they reach the application layer, and similarly manipulate the packets after they are formed at a resolver or server. Both the client and the broker identify injected packets by parsing the DNS resource records to determine the end of the legitimate packet data. If this does not match the length specified in the UDP header, then the remaining information is stripped and interpreted as injected data.

While stripping the injected packets in this scenario makes it unnecessary for DNS parsers to properly handle the injected packets, other scenarios such as data exfiltration may require this. It is not uncommon to only be able to listen to exfiltrated data instead of directly being able to manipulate it. This would require the manipulated packets to be properly accepted and undetected at the end host.

The psudp executable is used to pass messages and destinations to the client program over a Unix domain socket. Once the client receives a message over the socket, it is stored in a linked list of messages until the client sees a legitimate DNS query it can inject the message into. When the broker detects injected data, it first strips out the message and destination. This destination is used as a key into a hash table of linked lists, each list storing messages for that particular destination. Each time a DNS response is sent from the broker, it examines the hash table to determine whether there are any messages for that particular destination. If a message exists, it is injected into the response before being sent.

Data Exfiltration and DNS Tunnels

For data exfiltration or DNS tunnel scenarios, it is more appropriate to have a point-to-point data channel between two systems, ignoring the broker aspect. This architecture can be seen in the "injector" and "listener" tools that are packaged with PSUDP. The injector passively exfiltrates a file over DNS packets, while the listener analyzes DNS packets for injected data.

The injector differs from the client in that it takes a file as input and splits it into as many pieces as necessary to exfiltrate it over a series of DNS packets without exceeding the bandwidth limit. Unlike the client and broker seen earlier, the listener is implemented using libpcap. As DNS packets are received and injected content is identified, it is dumped into a file on the listening system, eventually rebuilding the original document. While libnetfilter_queue would have worked fine, libpcap was used to show another possible solution to solving the message detection problem. Additionally, this shows how end applications will still work appropriately when receiving, unstrapped, manipulated packets.

While a high-bandwidth bi-directional DNS tunnel has not been provided, it is easily seen how one could be implemented without significant effort. If the end objective is to build a replacement for the popular IP/TCP tunnels in use today, there are several things that must be kept in mind. Firstly, an approach using active packet generation would become a necessity. Secondly, it is important to ensure that no machines intercept and rewrite the packet between the two systems. For instance, if a DNS server analyzes the DNS query and creates a new request, the appended data will be lost in the future request. Therefore, it is important to send DNS requests directly to a controlled system.

Conclusions and Future Work

The tools discussed in this paper are proof of concept implementations that are harmless "as is". The techniques are demonstrated in non-malicious tools so the security community can learn from them and be able to identify this method of communication. For this reason, advanced data hiding techniques and protocol features such as reliability were left out of the tools.

While the technique of building covert communication in the slack space of another protocol was explored using DNS, it is highly like that it will apply to many other protocols commonly used in networks. It will be important to discover all protocols where this technique applies. Similarly, work must be done in detecting and mitigating this strategy for all protocols vulnerable to this storage channel. Additionally, the exploitation of slack space should be applied to actively generated packets to appropriately analyze bandwidth and covertness compared to existing DNS tunnel implementations.

Deep packet inspection (DPI) will be necessary to properly identify this type of covert communication in security tools. Firstly, it must be ensured that the legitimate data ends at the correct UDP packet length. Secondly, it must be verified that all pointers refer to previous positions in the packet. These two checks (and similar checks in other protocols) should be sufficient to detect slack space covert channels.

Works Cited

Born, K. (2010), “Browser-Based Covert Data Exfiltration”, In proceedings of 9th Annual Security Conference, Apr 7-8, Las Vegas, Nevada.

Dembour, O. (2008), 'Dns2tcp', http://www.hsc.fr/ressources/outils/dns2tcp/index.html.en. Nov 2008.

'Dnstop', http://dns.measurement-factory.com/tools/dnstop, 2009.

'Dsc', http://dns.measurement-factory.com/tools/dsc, 2009.

'Iodine', http://code.kryo.se/iodine/. June 2009.

Libnetfilter_queue. http://www.netfilter.org/projects/libnetfilter_queue/index.html.

Pixie, V (1999), ‘Extension Mechanisms for DNS (EDNS0)’, http://tools.ietf.org/html/rfc2671, Aug 1999

Mockapetris, P. (1987), 'RFC1035 - Domain names - implementation and specification', http://www.faqs.org/rfcs/rfc1035.html, Nov 1987.

'TCP-over-DNS tunnel software HOWTO', http://analogbit.com/tcp-over-dns_howto. July 2008.

Pietraszek, T, http://tadek.pietraszek.org/projects/DNScat/, 2004.

Miller, T, “Reverse DNS Tunneling Shellcode”, In proceedings of Black Hat 2008, Aug 2008.

Revelli A., Leidecker, Nico, “Introducing Heyoka: DNS Tunneling 2.0”, In proceedings of CONFidence 2009, May 2009.

Securiteam (1999), “Weaknesses in DNS label decoding can cause a Denial of Service”, http://www.securiteam.com/exploits/2CVQ4QAQNM.html, June 1999.

Wireshark (2010), www.wireshark.org, Apr 2010.