NTP as a Covert Channel

System and Network Engineering
University of Amsterdam

K.C.N. Halvemaan
`kees.halvemaan@os3.nl`

R.A.H. Lahaye
`rick.lahaye@os3.nl`

May 31, 2017

Abstract

Covert channels abuse a protocol by attaching extra information to it while keeping the original function of the protocol intact. They are used to hide specific data flows and send information over the network undetected. Implementations of covert channels exist based on various protocols, for example: Iodine (*Internet Protocol* (IP) over *Domain Name System* (DNS)) and Ptunnel (*Transmission Control Protocol* (TCP) over *Internet Control Message Protocol* (ICMP)). Work on *Network Time Protocol* (NTP) covert channels has been limited to a *Command And Control* (CNC) messaging system (NTP_Trojan). This paper proposes a working prototype named *NTPTunnel*, an IP over NTP covert channel. The feasibility of a covert channel over NTP is shown, and how it compares to other covert channels on terms of performance using iperf3 and detectability using the Snort *Intrusion Detection System* (IDS). NTPTunnel performs better than Iodine and Ptunnel over a direct link. However, when used on a more restricted network like Eduroam at the University of Amsterdam, Iodine outperforms both NTPTunnel and Ptunnel. Due to the blocking of ICMP and rate limiting NTP traffic, covert channels based on these protocols are inhibited in use. The only covert channel that was detected by using the IDS Snort with community rules was Ptunnel. Iodine and NTPTunnel were not detected, but it was possible to create custom rules for the detection of these covert channels based on their unique characteristics.

1 Introduction

A covert channel has been defined as "those not intended for information transfer at all, such as the service program's effect on the system load" by Lampson [1973].

Covert channels are often used to hide particular traffic and send information over the network undetected. Use-cases can be CNC servers or evading a paywall to access the Internet. A working implementation of a covert channel is one that is able to piggyback extra data on top of an existing protocol while at the same time: remaining undetected, looking like legit traffic, and keeping the original intended functionality intact.

Two implementations of a covert channel are Ptunnel (also known as Ping tunnel) and Iodine. Ptunnel is a covert channel that transfers data over ICMP, and Iodine uses DNS to transfer data. Both protocols allow enough data to be transferred for usage as a covert channel, as well as the fact they are often not

blocked on the network due the essential functionality they provide to users of the network.

NTP is a *User Datagram Protocol* (UDP) based protocol used for keeping clocks of machines synchronised [Mills et al., 2010], it is often allowed even on restricted networks, as this is an important task necessary for many time-dependant network applications. A physical device that is used to keep track of time such as an atom clock or a *Global Position System* (GPS) station is connected with a NTP server. This server can then be queried by other NTP servers and clients.

The goal of the project is to create a NTP covert channel that allows IP traffic to be tunnelled through. In this paper a NTP covert channel prototype named *NTPTunnel* is introduced. The feasibility of using NTP as covert channel is discussed and how it compares to Ptunnel and Iodine regarding the performance and detectability.

1.1 Research Question

The following research questions have been formulated.

1. Can NTP be used as a covert channel tunnel?
2. How does a NTP covert channel tunnel compare in terms of performance and detectability to a DNS (Iodone) and an ICMP (Ptunnel) covert channel implementation?

2 Related Work

There has been little work done on NTP covert channels. NTP_Trojan is a limited implementation of messaging over NTP made by Dan [2017]. NTP_Trojan is used for communicating with a CNC server and is limited to sending and receiving a predefined set of commands. Therefore it is not suitable to be used as a tunnel. Covert channels that do provide tunnelling are Iodine (IP over DNS) and Ptunnel (TCP over ICMP).

2.1 Ptunnel

Ptunnel makes use of echo and reply requests of the ICMP protocol. The ICMP protocol is used by computers and network devices to communicate operational information, error messaging, and provide troubleshooting possibilities. What information is exchanged is dependent on the type flag that is set in the ICMP header. The types echo and reply requests are used to ping devices to detect whether they are online or offline, and often is enabled on the network [Postel, 1981]. Ptunnel makes use of these types to create a tunnel for a covert channel [Stodle, 2015].

Ptunnel provides authentication and reliability for the client. Authentication is done by using a symmetric key and a challenge. When the server receives a request, it sends a challenge to the client where the client hashes the challenge including the hash of the symmetric key with the *Message Digest 5* (MD5) hashing algorithm. If the hash of the challenge and the response is the same, and therefore the symmetric key, the login will be successful and traffic will be able to flow. Reliability is provided by using sequence numbers for the client and the server. If no acknowledgement is received within 1.5 seconds, the packet will be resend [Stodle, 2015].

2.2 Iodine

Iodine makes use of response messages of DNS. The DNS protocol is used to resolve a domain to an IP address on the Internet. This IP address is then used to sent the packets to. As this protocol is essential to the Internet it is often not blocked, making it suitable for a covert channel [Ekman, 2014].

Iodine provides authentication for the client as well. Authentication is provided in the same way as Ptunnel, by using a challenge and response to connect. The advantage of using Iodine is that it is also able to communicate when only trusted DNS servers are allowed to be queried on the network. These servers would have to be recursive DNS servers which would then forward any queries for domains not in the local

zone to external DNS servers. In the case of a covert channel DNS query, it would get forwarded to the Iodine server as well.

2.3 Detection of Covert Channels

Covert channels are often detected in two ways: signature detection and anomaly detection. Signature detection uses rules with distinct characteristic (e.g. hard coded string or packet size) for a specific covert channel. This rule will then get triggered when the covert channel is used on the monitored network [Couture, 2010].

Anomaly detection identifies traffic as normal or not normal. To identify traffic as normal, a baseline needs to be known that identifies as normal. If the traffic deviate from this baseline, it is considered an anomaly. Characteristics to base a baseline on for a covert channel can be: amount of packets, length of packets, and used options and flags in protocols [Couture, 2010]. Multiple techniques can be used for anomaly detection like machine learning, statistics, and knowledge-based (description of anomaly) [Garca-Teodoro et al., 2009].

3 Implementation

The NTP prototype was written in Python based on the Pytun *Network TUNnel* (TUN) wrapper by Github user montag451 [2012][1]. An IP tunnel is set up between two machines, with the underlying stream being UDP. Pytun was chosen as it is based on UDP thus allowing for a straightforward conversion of the tunnel packets to valid NTP packets by prepending just a NTP header and popping it at the other side of the tunnel. Scapy for Python3 version 0.20 [Biondi and Dobelis, 2017] was used to create NTP packets. There is a client and a server setting for NTPTunnel, they differ in the values that are set for certain NTP header fields.

[1]Also see reference [montag451, 2012].

In Figure 5 the layout of the NTP packet can be seen. The extension fields are optional, as well as the *Message Authentication Code* (MAC) consisting of the Key Identifier field and the Message Digest field. The risk of packets being dropped by middle boxes has been minimised by creating the most authentic possible NTP packets whilst carrying the payload of the tunnel.

The Value field of the Extension Field will be used to carry the tunnelled data as it has the largest size of all NTP fields. The purpose of the Extension Field has been specified in Mills et al. [2010] for optional capabilities such as, but not restricted to, the Autokey security protocol. See Figure 6 for the layout of the Extension Field. In the following sections a rationale will be given for the choice of the values of the header fields, the length of the payload and on the reliability of the protocol.

The Pytun implementation requires a predefined remote address for both sides of the tunnel. In some situations a client might not know its own *Internet Protocol address* (IP address) before setting up the connection. For this scenario a basic handshake has been implemented in NTPTunnel. The server starts by listening for an incoming NTP packet that has an Extension Field with Field Type set to $\mathrm{FF}_{hex}00_{hex}$. If such a packet has been received, a reply will be sent with an Extension Field with Field Type set to $00_{hex}\mathrm{FF}_{hex}$ and a tunnel will be set up with the remote address of the packet it had just received. The client will start by listening for the reply after sending the request and set up a tunnel when it has received the reply. This system allows for a tunnel to be set up without a predefined remote address on the client side of the NTPTunnel.

3.1 NTP Header

This section describes for each header field how a realistic value was chosen for NTPTunnel based on Mills et al. [2010], and based on a packet capture that was done of NTP traffic of an Ubuntu LTS 16.04 client using the default NTP daemon.

3.1.1 LI

Leap Indicator (LI), which warns "of an impending leap second to be inserted or deleted in the last minute of the current month" [Mills et al., 2010].

For NTPTunnel it has been set to a value of 3, which has a meaning of "unknown (clock unsynchronized)" [Mills et al., 2010]. The choice of value was based on the observed values of NTP traffic of the Ubuntu LTS 16.04 client.

3.1.2 VN

Version Number (VN), NTPTunnel uses NTP version 4 as specified in Mills et al. [2010].

3.1.3 Mode

The mode is either Client or Server, which are set respectively for both sides in NTPTunnel.

3.1.4 Stratum

The Stratum is the indication of how close a NTP server is to the clock device. The lower the Stratum, the closer the server is, with the exception of Stratum zero which is the Kiss-o'-Death Packet used to signal a client to stop sending requests [Mills et al., 2010]. Stratum one would be a primary server which is connected to a precise timekeeping device such as an atom clock or a GPS station. A secondary (Stratum 2 to 15) NTP server will communicate with other servers in a lower stratum to get a more accurate time.

In NTPTunnel the client Stratum was set at three and the server Stratum at two. Due to the high cost of accurate precise timekeeping devices, there are more secondary NTP servers than primary NTP servers which make this a common setup.

3.1.5 Poll

The Poll field indicates "the maximum interval between successive messages, in log2 seconds" [Mills et al., 2010].

Mills et al. [2010] suggest a value between 6 and 10 for Poll, however, during some initial testing by the authors with various secondary NTP servers only a value of 3 was seen. In NTPTunnel the value was set to 3 for both the client and the server.

3.1.6 Peer Clock Precision

"The clock precision is defined as the running time to read the system clock, in seconds" [Mills et al., 2010].

For the client the Precision has been hard coded to 0.015625 second as this concurred with the packets the Ubuntu LTS 16.04 machine sends on its regular NTP packets. The server packets have their Precision set to 0.000001 second which was based on a value observed on various NTP servers.

3.1.7 Root Delay

The Root Delay is the "total round-trip delay to the reference clock" [Mills et al., 2010].

This value might vary depending on the latency between the reference clock and the NTP server. For the NTPTunnel server it was set to a random value between 0 and 0.1 for each packet. The idea behind this was that NTPTunnel would show realistic behaviour by having varying delay to reflect the imaginary latency to the root server. The client has the Root Delay hard coded to 1 as this field only has significance for packets coming from the NTP server.

3.1.8 Root Dispersion

The Root Dispersion is the "total dispersion to the reference clock, in NTP short format" [Mills et al.,

2010].

In a similar fashion as the Root Delay, the NTP-Tunnel server will set a random value between 0.002 and 0.003 as the Root Dispersion for each packet. The client has the Root Dispersion hard coded to 1 as this field only has significance for packets coming from the NTP server.

3.1.9 Reference ID

For the client the Reference ID is always empty, and for the server it is the IP address if the Stratum is higher than one or the name of the clock device if the Stratum is one.

In NTPTunnel the client will put no value in the Reference ID field, and the server will put its global IP address in there.

3.1.10 Timestamps

There are four different kinds of timestamps included in the NTP header, which are explained below. The NTPTunnel server has all of them set to the current time on its system. The client only has the Transmit Timestamp set to the current time, in the other cases it has the value of the field set to zero which equals to the start of the Unix Epoch. These settings correspond to the first packet a NTP client will send in order to synchronise its clock with a NTP server.

Reference Timestamp The Reference Timestamp is the "time when the system clock was last set or corrected" [Mills et al., 2010].

Origin Timestamp The Origin Timestamp is the "time at the client when the request departed for the server" [Mills et al., 2010].

Receive Timestamp The Receive Timestamp is the "time at the server when the request arrived from the client" [Mills et al., 2010].

Transmit Timestamp Transmit Timestamp is the "time at the server when the response left for the client, in NTP timestamp format" [Mills et al., 2010].

3.2 NTP Extension Field

The explanation of the fields in the NTP Extension Field will be given in this section. The additions specific for NTPTunnel are discussed here as well.

3.2.1 Field Type

The Field Type represents the type of the Extension Field. In order to differentiate the covert channel traffic from regular NTP packets, Field Types that are unassigned[2] have been be used. This choice will be elaborated on in Section 8.

3.2.2 Length

The Length field reflects the total length of the Extension Field, including headers. It is a 16 bit unsigned integer, thus the maximum length of the Extension Field is 65536 bytes.

3.2.3 Encryption/Payload Padding Length

The Encryption Padding Length and Payload Padding Length fields are added for the NTPTunnel format of the Extension Field, as can be seen in Figure 7. For *Advanced Encryption Standard* (AES) in

[2] The assigned Field Types can be found on the *Internet Assigned Numbers Authority* (IANA) website: `https://www.iana.org/assignments/ntp-parameters/ntp-parameters.xhtml` as well as in Haberman and Mills [2010]. Note that "Field Type" and "OPcode" are used interchangeably in this paper.

Electronic Code Book (ECB) mode and *Cipher Block Chaining* (CBC) mode the plain text has to be a multiple of 16, for this the Encryption Padding is used. The Payload Padding is used to make the total length of the Extension Field a multiple of 4 as this is a requirement of NTP[3].

3.2.4 Value

The Value field contains the actual payload that is being sent over the tunnel, it is encrypted with AES in ECB mode, as will be described in Section 3.5.

3.2.5 Encryption/Payload Padding

The Encryption Padding is done by using zero bytes 00_{hex}, and the Payload Padding is done with random bytes. Whether padding is used depends on the size of the Value field and the total size of the Extension Field.

3.3 Payload Size

Mills et al. [2010] state that the "maximum field length remains to be established", however, in Haberman and Mills [2010] the following is stated: "the reference implementation discards any packet with a field length more than 1024 octets". Theoretically, the Extension Field length could be extended to match the NTP packet length in the maximum UDP payload size. The Extension Fields are variable in size as defined by the 16 bit length field with a minimum of 16 bytes and a maximum of 65536 bytes [Mills et al., 2010]. The length of the value field has a maximum of 65532 bytes since the Field Type and Length fields are included in the calculation of the length.

[3]Note that if the Extension Field is not used, there is also no need for padding.

3.4 Reliability

Reliability is an issue for a covert channel based on UDP as it is a connectionless protocol. NTP does not have any functionality to introduce this kind of reliability by resending packets [Mills et al., 2010][4]. A similar problem occurs when doing a covert channel over DNS and ICMP. Iodine does not solve the issue as it does IP over DNS. Ptunnel fixes this problem by doing TCP over ICMP. In the former case a design choice was made to have the layers running on top of IP to solve the problem. In the latter case TCP will take care of the reliability that the lower layers are lacking. The implementation of NTPTunnel is similar to that of Iodine as Pytun creates an IP tunnel. Reliability can thus only be enforced if a reliable protocol such as TCP is used in combination with NTPTunnel.

3.5 Encryption

As NTP does not provide encryption as part of the protocol, NTPTunnel uses AES in ECB mode to encrypt the payload. The key is generated by hashing a user generated password using *Secure Hash Algorithm 2* (SHA-256). This password should be the same on the client side as the server side, otherwise the ciphertext cannot be decrypted. In the prototype the choice for ECB was made even though it is significantly less secure than CBC mode, the motivation for this will be discussed in Section 6.

4 Method

The performance experiments are discussed in Section 4.1 and detectability experiments using the IDS are explained in Section 4.2.

[4]Note that reliable protocols such as TCP would decrease the accuracy of NTP due to the possible (delayed) resending of outdated information.

4.1 Bandwidth Benchmark

The covert channels Iodine, NTPTunnel and Ptunnel have been benchmarked on their performance by doing three different experiments. For all experiments iperf3 [Matthews and Cottrell, 2001] was used to measure the throughput.

Table 1 contains the software versions used during the benchmarks.

Software	Version
iperf3	3.1.3
Iodine	0.7.0
NTPTunnel	Prototype
Ptunnel	0.72

Table 1: Software versions.

For each experiment the same client and server was used. The hardware of these nodes can be seen in Table 2 and 3.

Role	Client
Model	Asus UX303LN
OS	Ubuntu 17.04 (Zesty Zapus)
Kernel	4.11.0-041100
CPU	i7-4510U CPU 2.00GHz
RAM	12 GB
Network	TP-UE300 1Gbit/s USB3.0
Wireless	Intel Wireless-AC 7260-AC

Table 2: Client hardware.

Role	Server
Model	Dell PowerEdge R210
OS	Ubuntu 16.04.2 LTS (Xenial Xerus)
Kernel	4.4.0-77
CPU	2x Intel Xeon L3426 1.87GHz
RAM	8 GB
Network	NetXtreme II BCM5716 1Gbit/s

Table 3: Server hardware.

Each benchmark uses a *Maximum Transfer Unit* (MTU) of 1500 bytes on the physical interface. This is the MTU defined for ethernet networks [Mogul and Deering, 1990]. Some covert channels like NTPTunnel and Iodine allow setting the MTU of the tunnel which impacts the bandwidth. However, during preliminary testing Iodine was dropping packets when the default MTU is not used, rendering it unstable and unusable. Therefore only NTPTunnel is able to function with a changed MTU. The MTU of each covert channel and whether they can be set or not can be viewed in Table 4.

Covert Channel	Default Tunnel MTU	Settable
Iodine	1130B	Yes, but not stable
NTPTunnel	1500B	Yes
Ptunnel	1472B	No

Table 4: Covert channel MTU.

The following experiments were done:

- **Experiment #1:** benchmark of NTPTunnel with increasing MTU values over a direct 1Gbit/s link.
- **Experiment #2:** benchmark over a direct 1Gbit/s link for each covert channel.
- **Experiment #3:** realistic scenario with benchmark over the wireless network Eduroam[5] using 802.11n 5Ghz.

4.2 Detectability

Each covert channel has characteristics on which it can be detected in a network. To check the detectability of the covert channels, the open source IDS Snort has been used together with the so-called community rules, and manual analysis of network traffic [Roesch et al., 1999]. The traffic used for the detection was generated by running *Secure Shell* (SSH), ping and iperf3.

Snort is using version 2.9.9.0, and the community rules[6] of the 15th of May 2017[7].

[5] At the University of Amsterdam location Science Park 904.

[6] All rules were used, even experimental ones that are disabled by default.

[7] Community rules do not use any versioning except for date on which the rules were retrieved.

If the covert channel has not been detected by Snort, a custom rule will be created. The requirement for a custom rule is that it only triggers on a covert channel, and not on other valid traffic for which the protocol was intended.

5 Results

The results of the bandwidth benchmark and detectability of each covert channel can be seen respectively in Section 5.1 and Section 5.2.

5.1 Bandwidth Benchmark

This section contains the results of the performance experiments which were done. Experiment #1 is a benchmark of NTPTunnel with an increasing value of the MTU, experiment #2 a benchmark over a direct 1 Gigabit connection for each covert channel, and experiment #3 a benchmark in a realistic scenario over a wireless network.

5.1.1 Experiment #1

Figure 1 shows a benchmark with changed MTU values for NTPTunnel. The NTPTunnel MTU value of 65440 is the maximum because of the added NTP headers which create a small overhead. It shows the higher the tunnel MTU is, the higher the bandwidth will be. This is because the a high MTU will allow more data to be be put in the tunnel, resulting in the maximum size of an UDP packet minus the NTPTunnel overhead. In addition, it means less segmentation and overhead for the tunnel to transfer a particular amount of data into the UDP tunnel.

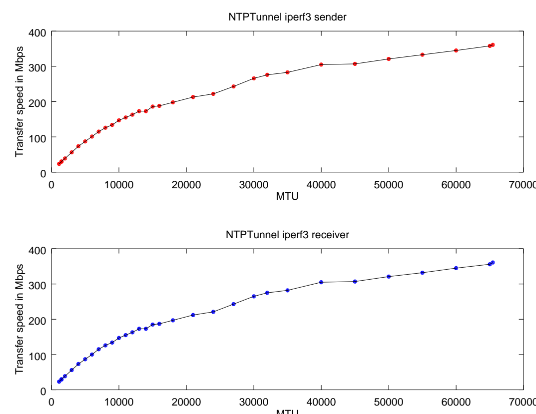

Figure 1: Effect of increasing the NTPTunnel MTU on the transfer speed.

5.1.2 Experiment #2

Table 5 shows a benchmark comparison of the bandwidth of each tunnel over a 1 Gigabit connection. Experiment #1 in Section 5.1.1 has shown that the performance increases when a higher MTU is used, so this experiment includes the highest possible MTU for NTPTunnel as well. Table 5 shows that NTPTunnel with the maximum MTU performs better than Iodine or Ptunnel.

Covert Channel	Tunnel MTU	Sending	Receiving
None	None	942 Mbps	941 Mbps
NTPTunnel without encryption	65440B	920 Mbps	919.4 Mbps
NTPTunnel with encryption	65440B	359 Mbps	359 Mbps
Iodine	1130B	99.8 Mbps	98.3 Mbps
NTPTunnel without encryption	1500B	33.4 Mbps	32.8 Mbps
NTPTunnel with encryption	1500B	30.5 Mbps	30 Mbps
Ptunnel	1472B	7.3 Mbps	0.517 Mbps

Table 5: Benchmark covert channels over direct link.

5.1.3 Experiment #3

Table 6 shows a benchmark comparison of each covert channel in a realistic environment over a wireless 5Ghz network. This experiment includes NTPTunnel with a MTU value of 65440 as experiment #1 in Section 5.1.1 had shown that a higher MTU value

leads to a higher throughput. Table 6 shows that Iodine performs better than NTPTunnel in a realistic environment. The reason for this will be discussed in Section 6.

Covert Channel	Tunnel MTU	Sending	Receiving
None	None	95.5 Mbps	93 Mbps
Iodine	1130B	81.1 Mbps	79.2 Mbps
NTPTunnel without encryption	65440B	3.2 Mbps	1.1 Mbps
NTPTunnel with encryption	65440B	3.2 Mbps	1.1 Mbps
NTPTunnel without encryption	1500B	162 Kbps	60.7 Kbps
NTPTunnel with encryption	1500B	162 Kbps	59.6 Kbps
Ptunnel	1472B	n/a	n/a

Table 6: Benchmark covert channels over wireless network.

5.2 Detection

This section contains the results of the detectability of each covert channel by using manual analysis on network captures and Snort.

Ptunnel was detected by Snort by using signature ID 29456. The message for this signature ID is *"PROTOCOL-ICMP Unusual PING detected"* Snort [2017]. The signature can be seen in Figure 2 in Section 9.

Iodine was not detected by Snort with the community rules. However, Snort custom rules can be created for detection. The created custom rules can be seen in Figure 3. The custom rules include one for the request to open a tunnel, and one for the response.

NTPTunnel was not detected by Snort with the community rules. Custom rules for detection can be seen in Figure 4. These custom rules include one by triggering on the fixed Extension Field OPcode value of NULL, and one when the NTP packet length does not conform to the reference implementation in Mills et al. [2010]. The Extension Field OPcode of NULL has not been assigned by IANA, and the maximum size of an NTP packet should not be larger than 2216 bytes. These 2216 bytes contain 48 bytes of the header, 2048 bytes of the two extension fields, 4 bytes of the Key Identifier, and 16 bytes of the digest.

6 Discussion

In the wireless Eduroam experiment, the decrease in performance of NTPTunnel is due to the fact that NTP traffic is rate limited on the wireless network to prevent *Denial of Service* (DoS) attacks[8]. Ptunnel did not work in the realistic scenario due to ICMP traffic being blocked.

According to the *Federal Communications Commission* (FCC), a minimum download speed of 1 Mbps is needed for browsing, 4 Mbps for streaming, and 4 Mbps for gaming [FCC, 2017]. For watching Netflix with Ultra High Definition quality, 25 Mbps is required [Netflix Inc., 2017]. Even with all these activities combined, and if no NTP rate limiting takes places, NTPTunnel seems fast enough with a performance of 359 Mbps as can been seen in Table 5. In the realistic experiment where rate limiting took place, the minimum requirements were met only for browsing. The more aggressive the rate limiting is, the less bandwidth the covert channel tunnel has.

The Python Pytun TUN implementation is a wrapper for a C library, which in turn is a wrapper for system calls to configure a TUN interface. This setup allowed for fast transfer due to the low-level bindings. The NTPTunnel extends the Python part with code for parsing the NTP header fields. As Python is an interpreted language it has a negative affect on

[8]This was confirmed by a University of Amsterdam system administrator. It is not completely blocked as some Apple devices have hard coded IP addresses for external NTP servers.

the performance on the NTPTunnel prototype. The encryption and decryption operations also decreased the performance of NTPTunnel significantly.

Running a NTP covert channel might interfere with the default operation of the protocol. Since NTP packets have as their main function to update the clock at a client, the incoming NTPTunnel packets might interfere. While doing the experiments described in Section 4, the authors did not notice any interference on the client's clock. A NTP daemon will periodically request an update from NTP servers, after which it will wait for these specific servers to respond. In case the daemon is running in parallel with a NTPTunnel client, it was assumed that there should be no interference as the NTPTunnel server will have a different IP address from the actual NTP servers. Further research should be done to confirm that this assumption is correct.

The traffic generated by covert channels will often have certain characteristics which make it stand out from regular traffic. These deviations from the norm can then be used in methods to detect the covert channel. In the case for NTPTunnel, it was the non-assigned Field Type, for Ptunnel a specific string in the content of the ICMP packet, and for Iodine it was a specific byte string which occurred during the handshake phase. In addition, the length of covert channel packets is often significantly longer than regular traffic which can also be used in a rule based detection system.

7 Conclusion

The NTPTunnel prototype has shown that NTP can successfully be used as a covert channel. It is not detected by the Snort IDS using the default community rules, however, custom rules can be added to detect the NTPTunnel traffic based on the unassigned Extension Field type or packet length. The tunnel has a high enough performance and reliability allowing for simple tasks such as browsing.

The performance in bandwidth is higher for NTPTunnel than Iodine and Ptunnel when a direct link is used. Using the covert channels in the realistic scenario of the wireless network Eduroam, it was shown that Iodine performs better than NTPTunnel, while Ptunnel did function at all. Out of the three covert channels, only Ptunnel was detected by the community rules of Snort. However, it was possible to write rules to detect the other two covert channels based on their unique characteristics.

8 Future Work

The NTPTunnel prototype presented in this paper has shown positive results. However, the implementation is lacking on a number of points which could be looked into as future work.

A basic handshake has been implemented in NTPTunnel, though there is no authentication in place. Any NTPTunnel client can connect to any NTPTunnel server without having to authenticate itself. Both parties do need to have the same key for the encrypted payload, otherwise the tunnel is not functional. Future work might look into implementing an authentication similar to Ptunnel which uses a hashed challenged as authentication method. When a client would send a tunnel request it would receive a timestamp based challenged from the server which it would have to hash in combination with a password: hash(challenge + hash(password)). The server then validates the response by calculating the hash itself, and compares it with what the client had sent. The authentication password in the case of NTPTunnel could be the same as the encryption key.

The AES encryption used to encrypt the payload was done in ECB mode, a more secure mode would be CBC mode [Bellare et al., 1997] . However, this would require reordering of the UDP packets before decryption can take place. In CBC, an *Initialisation Vector* (IV) is used as initial key to encrypt a block, and is sequentially updated after each block. When UDP packets can arrive in any order, the key used to decrypt the packet might not be valid for the current packet. A solution would be to use *Datagram Trans-*

port Layer Security (DTLS) which takes care of the reording and lost packets. However, by default DTLS can only encrypt the entire payload of a UDP packet, which would include the NTP header. The NTPTunnel prototype should be extended in future work to implement DTLS but only for the payload the NTP packet carries. In addition, as explained in section 6, the encryption and decryption of the payload significantly decreased the performance of NTPTunnel. PyCrypto version 2.6.1 by Litzenberger [2017] was used to do the AES encryption. Future work might look into optimising the current Python codebase for performance or porting it to a lower level language such as C or C++.

The authenticity of the NTP packets could be further increased by setting more appropriate values for the length, field type, and timestamps.

To not exceed the maximum recommended size of an NTP packet, the inserted data of the tunnel should be limited to the size of the recommended Extension Field length of 1024 bytes [Haberman and Mills, 2010]. A straightforward solution using the current implementation with one Extension Field would be to lower the tunnel MTU so the maximum recommended length is not exceeded. As two Extension Fields are allowed in an NTP packet, both could be used to allow for a larger payload. The maximum amount of data that can be added as payload to a NTP packet is 1024 bytes per Extension Field used, so 2048 bytes in total. The downside of this modification would be that the MTU of the tunnel cannot be higher than 2016 bytes[9].

Some additional experiments might be necessary to find a Type Field from [Haberman and Mills, 2010] which can easily avoid IDS rules. An in depth review of the workings of commonly used NTP implementations could reveal a packet format which allows for high amounts of data to be transferred while also conforming to the RFCs. A candidate for this could be the Field Types related to Autokey authentication which is one of the few protocols using the Extension Field.

[9] There are two NTP Extension Field headers of 16 bytes each, so 32 bytes are subtracted from the 2048 bytes.

The timestamps in the client packets that are set to the start of the Unix Epoch are in accordance with the first packet a NTP client would send to synchronise its clock. After the client has received its first packet it will have synchronised and thus newer packets would show the current timestamp in the fields. This change of values has not been implemented in NTPTunnel and this should be done to further improve the authenticity.

9 Credits

Figure 5 and Figure 6 are taken from in Mills et al. [2010]. Figure 7 is based on Figure 14 of Mills et al. [2010].

References

M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 394–403. IEEE, 1997.

P. Biondi and E. Dobelis. Scapy. *URL: https://github.com/phaethon/scapy*, 2017. Python3 port of Scapy.

E. Couture. Covert channels. Technical report, Escal Institute of Advanced Technologies, 2010. URL `https://www.sans.org/reading-room/whitepapers/detection/covert-channels-33413`.

A. Dan. NTP_Trojan, 2017. URL `http://lockboxx.blogspot.nl/2015/04/python-ntp-trojan.html`.

E. Ekman. Iodine, 2014. URL `http://code.kryo.se/iodine/`.

P. Garca-Teodoro, J. Daz-Verdejo, G. Maci-Fernndez, and E. Vzquez. Anomaly-based

network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(12):18 – 28, 2009. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2008.08.003. URL http://www.sciencedirect.com/science/article/pii/S0167404808000692.

FCC. Broadband speed guide, 2017. URL https://www.fcc.gov/reports-research/guides/broadband-speed-guide. Federal Communications Commission.

B. Haberman and D. Mills. Network time protocol version 4: Autokey specification. RFC 5906, RFC Editor, June 2010. URL http://www.rfc-editor.org/rfc/rfc5906.txt.

B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

D.C. Litzenberger. Pycrypto-the python cryptography toolkit. *URL: https://www.dlitz.net/software/pycrypto*, 2017.

W. Matthews and L. Cottrell. Achieving high data throughput in research networks. Technical report, Stanford University, 2001. No. SLAC-PUB-8903. 2001.

D. Mills, J. Martin, J. Burbank, and W. Kasch. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor, June 2010. URL http://www.rfc-editor.org/rfc/rfc5905.txt.

J. Mogul and S. Deering. Path mtu discovery. RFC 1191, RFC Editor, November 1990. URL http://www.rfc-editor.org/rfc/rfc1191.txt.

montag451. Pytun, 2012. URL https://github.com/montag451/pytun. Github repository, name of the author is the user account at Github.

Netflix Inc. Internet connection speed recommendations, 2017. URL https://help.netflix.com/en/node/306.

J. Postel. Internet control message protocol. STD 5, RFC Editor, September 1981. URL http://www.rfc-editor.org/rfc/rfc792.txt.

M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

S. Sheridan and A. Keane. Detection of dns based covert channels. In *ECCWS2015-Proceedings of the 14th European Conference on Cyber Warfare and Security 2015: ECCWS 2015*, page 267. Academic Conferences Limited, 2015.

Snort. Sid 1-29456, 2017. URL https://www.snort.org/rule_docs/1-29456. Reference to the Ping tunnel detection Snort community rule.

D. Stodle. Ping tunnel, 2015. URL https://stuff.mit.edu/afs/sipb/user/golem/tmp/ptunnel-0.61.orig/web/.

Appendix

```
alert icmp $HOME_NET any -> $EXTERNAL_NET any (msg:"PROTOCOL-ICMP Unusual PING
    detected"; icode:0; itype:8; fragbits:!M; content:!"
   ABCDEFGHIJKLMNOPQRSTUVWABCDEFGHI"; depth:32; content:!"0123456789
   abcdefghijklmnopqrstuv"; depth:32; content:!"
   EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE"; depth:36; content:!"WANG2"; content:!"
   cacti-monitoring-system"; depth:65; content:!"SolarWinds"; depth:72;
   metadata:ruleset community; reference:url,krebsonsecurity.com/2014/01/a-
   closer-look-at-the-target-malware-part-ii/; reference:url,krebsonsecurity.
   com/2014/01/a-first-look-at-the-target-intrusion-malware/; classtype:
   successful-recon-limited; sid:29456; rev:2;)
```

Figure 2: Snort community rule for Ptunnel [Snort, 2017].

```
alert udp any any -> any 53 (content:"|01 00 00 01 00 00 00 00 00 00|"; offset
   :2; depth: 10; msg:"Covert Channel Iodine Request"; threshold: type limit,
    track by_src, count 1, seconds 300; sid: 1122233; rev: 1;)
alert udp any 53 -> any any (content:"|84 00 00 01 00 01 00 00 00 00|"; offset
   :2; depth: 10; msg:"covert iodine tunnel response"; threshold: type limit,
    track by_src, count 1, seconds 1; sid: 9619501; rev: 2;)
```

Figure 3: Snort custom rules for Iodine based on Sheridan and Keane [2015].

```
    alert udp any 123 -> any 123 (content:"|00|"; offset:49; depth: 1; msg:"
       NTP Extension Opcode is NULL, potential NTPTunnel use"; threshold:
       type limit, track by_src, count 1, seconds 10; sid: 111111; rev: 1)
    alert udp any 123 -> any 123 (dsize: > 2116; msg: "NTP Packet length does
       not conform to RFC5905 standard"; threshold: type limit, track by_src,
        count 1, seconds 10; sid: 9121111; rev: 1)
```

Figure 4: Snort custom rules for NTPTunnel.

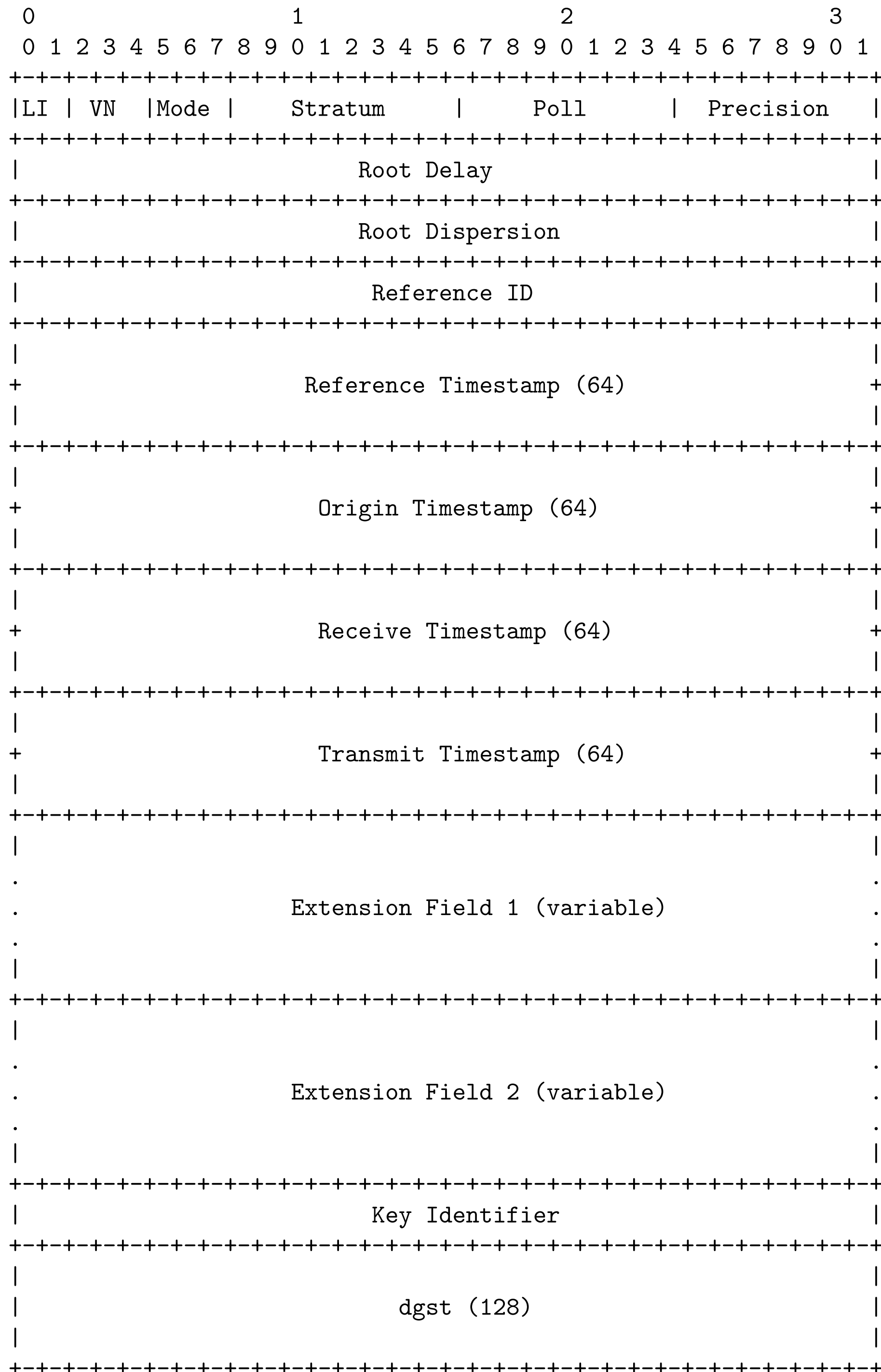

Figure 5: NTP packet header format.

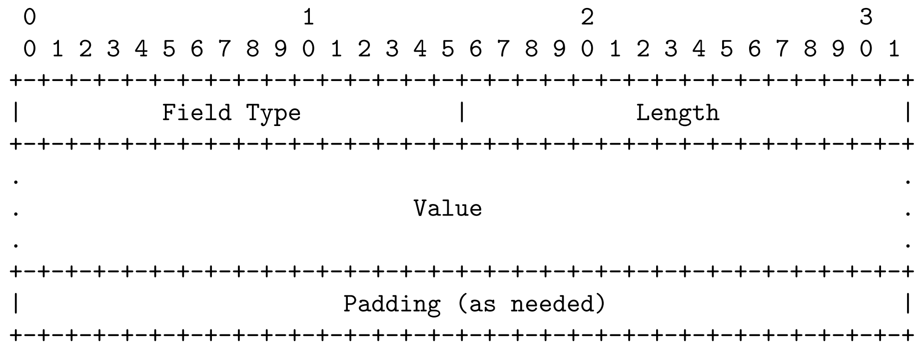

Figure 6: NTP Extension Field format.

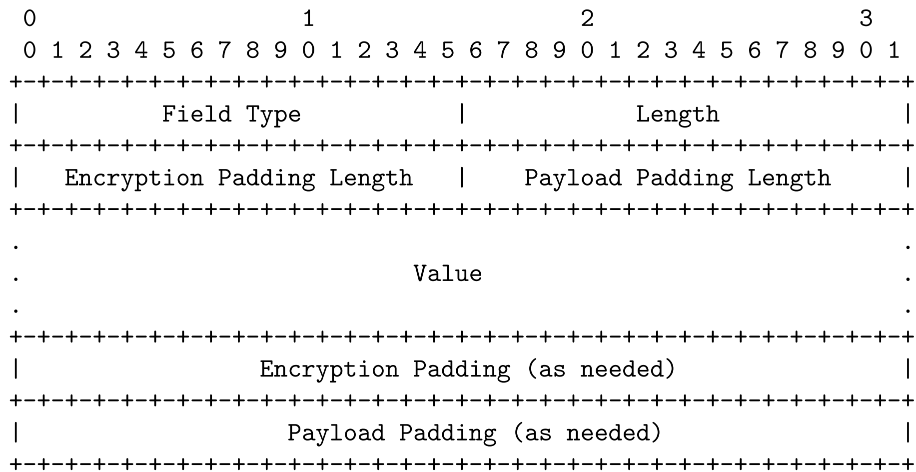

Figure 7: Format of the NTP Extension Field in NTPTunnel.