A novel network hopping covert channel using BPF filters and NTP extension fields

Rafael Ortiz <rortiz12@jhu.edu>
Ramon Benitez-Pagan <ramon.benitez@jhu.edu>
Stephen Scally <sscally@jhu.edu> Cyrus Bulsara <cbulsar1@jhu.edu>

Abstract

Many papers focus on creating covert channels for the purpose of data exfiltration. That is, they attempt to remove some information from a protected network. Less common appears to be the concept of data *infiltration*, where a covert channel is established to secretly move information *inside* a protected network. Many exfiltration oriented channels make assumptions about extant arbitrary infiltration channels being available for loading the tooling necessary to establish the outbound channel. We are proposing studying and implementing an NTP-based covert channel for infiltrating unauthorized information into a secure network.

1 Introduction

Covert channels are traditionally classified as either storage or timing channels. These classifications came about from definitions outlined in the Pink book. These definitions are listed below:

Definition 1:

"A communication channel is covert if it was neither designed or intended to transfer information at all [1]."

Definition 2:

"A communications channel is covert if it is based on transmission by storage into variables that describe resource states [1]."

Definition 3:

"Covert channels will be defined as those channels that are a result of resource allocation policies and resource management implementation [1]."

Definition 4:

"Covert channels are those that use entities not normally viewed as data objects to transfer information from one subject to another [1]."

Definition 5:

"Given a non-discretionary security policy model M and its interpretation I(M) in an operating system, any potential communication between two subjects I(S_h) and I(S_i) of I(M) is covert if and only if any communication between the corresponding subjects S_h and S_i of the model M is illegal in M [1]."

After understanding the criteria of what makes a covert channel our goal within this paper is to utilize the Network Time Protocol (NTP) to create a covert channel for data infiltration.

There are a number of network architectures that enterprises and device manufacturers utilize in order to have network connected devices synchronize their system time. We reason that most security controls within these architectures focus on ensuring that access is only allowed to appropriate network time servers, over the protocol UDP and port 123, and that public network time servers are themselves clients of higher tiered stratum layer servers. Due to the default trust associated with NTP very few operators validate the public NTP pool servers they connect to or the time data that is retrieved and propagated throughout the network from servers to clients.

In order to address these security concerns as well as control NTP delivery, enterprises have deployed dedicated vendor appliances while device manufacturers have removed the ability to change time sources on certain devices. Furthermore, adding

to the lower scrutiny of NTP usage, CVEs and exploits are typically directed at public NTP instances acting as time sources, with attacks in the form of DDoS and reflective type attacks. While it can be beneficial for an adversary to attack an internal time source as these disruptions can affect remote logins, authentication tokens, and DHCP leasing most service logging indicates clearly when a time or date synchronizations are out of skew. For an adversary attempting to keep a low profile, internal NTP attacks may create too much noise for very little gain.

Berkeley Packet Filter is a Linux Kernel subsystem that allows a user to run a limited set of instructions on a virtual machine running in the kernel. It is divided between classic BPF (cBPF) and extended BPF (eBPF or simply BPF). The older cBPF was limited to observing packet information, while the newer eBPF is much more powerful, allowing a user to do things such as modify packets, change syscall arguments, modify userspace applications, and more[2].

We intend to show that, with the above factors in mind, ingress and egress NTP communications that are not being analyzed for correctness leaves networks and devices open to covert channel utilization. We will construct a covert channel using NTP, leveraging BPF. This paper is broken down into the following sections. In the background section we will will provide basic terminology and workings of NTP and its communication structure. Related works will review past and current analysis related to NTP covert channels. The design section will explain our network architecture for implementation as well as review our expected throughput, robustness, and detection of this channel. The implementation will demonstrate our covert channel in our lab environment. Lastly, our conclusions and future work outlines further areas of expanding the NTP covert channels based on current standards specifications as well as observations during our implementation.

2 Background

2.1 NTP Modes

The network time protocol (NTP) operates in one of three modes.

2.1.1 Primary Server

The first mode is primary server which is directly synchronized from a reference clock. Reference clocks can come from multiple sources however the most common are satillite based from GPS[3], GLONASS[4], and Galileo[5] as well as regional radio based time signals provided by MSF[6] in the UK, DCF77[7] in Germany, and WWVB[8] within the United States. A primary server is utilized by secondary servers and clients. Since the primary server derives its time from a reference clock it is also categorized as a stratum 1 server. The stratum designation signifies two items, the first is the distance that the server is from a reference clock(stratum 0), in this case one hop, and that this server provides a high level of time accuracy.

2.1.2 Secondary Server

The second mode is being a secondary server. This mode operates as a client to upstream primary servers and as a server to downstream clients. There is a defined maximum of 16 stratum levels and each secondary server will reflect their level depending on how far they are from a primary server. Each increased stratum level indicates a decrease time accuracy from the the higher level time source. Any system reporting a stratum level of 16 is understood to be unsynchronized.

2.1.3 Client

Lastly the third mode is client to which most devices fall under. A client references time from multiple available time sources to synchronize its system time.

2.1.4 NTP Uses

// TODO: this section can be cut down if we need space

While accurate time is useful the question remains about why it is needed for devices such as computers, phones and within local area networks. Most if not all computing type devices contain some local clock or time keeping mechanisms. Typically they are adjusted during the setup or initialization phase to the current date and time. While the local system clock is able to keep track of time, through frequency ticks, it is subject to the same interrupt process delays as other system hardware and software. Over time the missed or delayed

interrupts can cause the system to fall behind real-time causing the system time to become skewed[9]. Putting this into perspective it can be seen how each device, on the network, if left to its own time keeping, would eventually fall out of sync with other devices. For a single device this may not be concerning as the time can be adjusted, however with logging, security, and within the network this can have cascading problems.

The first of these issues is the ability to schedule tasks. Within current operating systems tasks can be scheduled to run at specific dates and times in order to perform maintenance, updates, or monitoring. These tasks can have dependencies on other scheduled tasks running or checking dates and times for file access and updates. Externally to the system other devices may depend on the scheduled jobs placing needed files or updates within a specific time frame, as time drifts between the systems this can lead to failed job processing and race conditions.

This leads to the second issue, auditing and logging. When creating log events, troubleshooting issues or auditing security logins one of the many sources referenced is the system or application logs. When reviewing the timeline of events and seeing that the system clocks are not synchronized it places extra effort in determining when systems were accessed and at what time relative to each local system clock. Instead, if the systems were synchronized, dates and times would line up accordingly and administrators could efficiently determine root causes of errors. Other network devices such as firewalls can depend on rulesets being configured to allow or deny access based on the current date and time. Since these devices are regularly deployed in pairs security access controls would show inconsistencies as devices would allow access early or delayed depending on the firewalls system time. When combining multiple device logs such as firewalls, systems and applications to audit compliance, assurance and integrity can be brought into question as numerous time and date inconsistencies have to be re-evaluated for proper alignment. With centralized log aggregation of these entries can be dropped from processing as a previous time period has been marked as processed or because the current time is determined to be in the future.

The final item is security. Systems and services that require users to authenticate need accurate dates and times across numerous systems. With the Windows operating system, which utilizes the W32Time manager, any system looking to join the Active Directory domain or obtain needed authorization Kerberos tickets needs to have highly accurate and stable system time[10]. Accurate time reduces the ability of attackers to perform replay attacks with expired Kerberos tickets which utilize time stamps to limit the life of the ticket. Additionally, the timestamps inform applications when to request a token refresh[11]. For other web protocols, such as HTTPS, time and date settings are important for validating and generating certificates. For example each website that is secured using HTTPS presents a certificate which contains a *not valid before* and *not valid after* entries which contain a date, time and timezone as seen below:

```
Signature Algorithm: sha256WithRSAEncryption
  Issuer:
    C=US, O=DigiCert Inc,
    OU=www.digicert.com,
    CN=GeoTrust RSA CA 2018
  Validity
    Not Before: Jul  1 00:00:00 2021 GMT
    Not After : Mar  7 23:59:59 2022 GMT
```

Figure 1.: https://www.jhu.edu website certificate information

This allows systems to validate the security of the presented certificate from the webserver.

2.1.5 NTP Public Pool

Synchronized time utilizing NTP is depended upon by numerous services and security mechanisms within typical network architectures. To support this need for accurate time, public time sources are provided by the NTP pool project[12]. Servers are allocated by DNS round robin, where each client requests a forward lookup of one of the following:

- 0.pool.ntp.org
- 1.pool.ntp.org
- 2.pool.ntp.org
- 3.pool.ntp.org

The DNS based pool set of returned NTP servers IP addresses rotates every hour. Additionally, further continent zones and sub-zones exist to provide time service asa close as possible to the intended client as seen in table 1 below.

Area	Hostname
Asia	asia.pool.ntp.org
Europe	europe.pool.ntp.org
North America	north-america.pool.ntp.org
Oceania	oceania.pool.ntp.org
South America	south-america.pool.ntp.org

Table 1: Continet Zones

The available servers in the pool are volunteered by research organizations, companies, and individuals helping to support the project. In order to add a server to the available pool you must have a static IP address and a stable Internet connection. If that criteria is met, you can configure your NTP servers with a known good stratum 1 or 2 servers and make the appropriate firewall allowances. To add the severs to the available pool, account registration is needed at ntppool.org. Once you have added your server(s) to the pool they are monitored for connectivity and timing accuracy. The ntp project scores systems as part of their monitoring and once a system reaches 20 points it can be added to the public pool. If a server drops below a score of 10 it is removed from the public pool availability[13].

2.2 extended Berkeley Packet Filter (eBPF or BPF)

The extended BPF subsystem can be thought of as doing for the Linux Kernel what JavaScript does for a web browser[14]. A developer can write code that is compiled down into bytecode for a virtual machine, which is then translated into machine instructions. The virtual machine executes in a sandbox which protects the end user from a malicious or malfunctioning program. Unlike a web browser, eBPF goes a step further and verifies the applications that it executes. The eBPF verifier looks for, among other things, loops, complete execution, safe memory access, and more[15]. Because of this, there are restrictions on what BPF programs can do. They are not Linux Kernel Modules and cannot access arbitrary memory or arbitrarily modify the kernel, for example. The verifier must also be able to prove that execution terminates (i.e., solve the halting problem for the application, if it can't be verified to terminate within some number of instructions the program fails validation). The advantage of these restrictions is that an eBPF program is incredibly unlikely to crash your kernel or introduce a vulnerability.

There are many places we can attach our eBPF program to. Kprobes and uprobes allow us to run a program at an exported symbol in kernelspace and userspace, respectively[2]. The BPF program receives information on the current CPU register states in the intercepted execution and may examine or modify them as it sees fit. We can also attach ourselves to an eXpress Data Path (XDP) filter, which allows us examine and modify inbound packets[16].

A less commonly used, but no less important, attachment point is the Linux traffic control (TC) subsystem. Attaching a BPF filter to a TC classifier allows us to inspect and modify both ingressing and egressing packets[17]. This is a requirement for our covert channel, as we will need to modify outbound NTP responses as well as intercept inbound NTP responses.*

3 Related Work

While the intial specification for NTP was published in 1985[19] there has been minimal public analysis into using the NTP protocol as a covert channel. The first implementation by Halvemaan and Lahaye utilizes a NTP covert channel through tunneling, called NTPTunnel[20]. This type of channel is also observed with other common protocols such as DNS with Iodine[21] as well as TCP and ICMP with PTunnel[22]. NTPTunnel utilizes the Field Type value within the NTP header to build the initial client / server connection. A modified Pytun[23] implementation listens for a specific client NTP packet with the extension field, *field type* value of FF(hex) 00(hex) to which it will respond to the client with a *field type* value of 00(hex) FF(hex). This exchange allows the client and server to discover each other without any pre-shared details. After the tunnel is established, crafted NTP packets are sent between the server and client over the tunnel utilizing the *value* field of the NTP extension field. The *extension field* has a described use as supporting the autokey security protocol which makes it difficult to restrict or flag with IPS or IDS signature rules. However given the open nature of NTP at this time, the use of this field

*Details on the TC subsystem (such as classifiers and queueing disciplines) are beyond the scope of this paper, but if you are interested more detail can be found at [18].

or values within it can provide credible detection of this covert channel. In order to obscure the payload from analysis the data is encrypted with AES and utilizes a shared key between server and client.

A second NTP covert channel utilizes the NTP Timestamp Format[24], specifically the 32 bits representing the fraction of seconds within the timestamp format. The establishment of this covert channel does require shared information between the sender and receiver, however the receiver does not have to be the NTP server and can be any host capable of listening to NTP traffic between the server and its clients. The receiver listens on the network for the predetermined Initiation pattern ($0x00000001$), Sequence pattern ($0xe9$, $0xab$, $0xcb$) of three 32 bit segments, and an end of message pattern ($0xeb$). In order to track the messages between the client and receiver the *Peer Clock Precision*[25] field is used. The detection within this channel is difficult without knowing the sequences to look for. Additionally the data flows within the existing NTP communication flow only using a small amount of storage within the existing channel. However, based on the limitation of data that can be sent per message analysis of NTP message volume may expose this channel.

Lastly is a more recent covert channel that uses NTP as a *Dead Drop* utilizing both the information NTP stores as part of its client / server communication, such as its most recently used (MRU) list and retrieval through NTP query and control messages[26]. NTP has a number of service query and monitoring commands that be used to acquire information about the status of the service. It was noted by the authors that these requests were typically disabled by default however testing with the public NTP pool revealed that queries were answered leading to the likelihood that other environments would also respond to these queries. To implement the first possible covert channel the peer list, which is used to keep track of data about upstream time sources, such as poll, offset, jitter, delay, refid, etc, sets it stratum level equal to 14. This indicates to any querying client that there is a covert message stored within the reference ID(*refid*) field of the peer list. In the authors implementation the message is within the IP address, stored in the *refid* field, of which each octet represents an ASCII value. A covert client can obtain this information by querying the peer status of the covert server. The second covert channel utilizes the MRU list available on every NTP instance. A crafted NTP packet is sent to a listening NTP instance, which if not configured with restrictions any instance can respond to NTP queries, where data can be set / updated within that instances MRU table. Once these values are set a covert client can query for the instances MRU data which it can then decode the covert message from. In each instance of these covert channels any monitoring or warden device would observer this to be expected NTP communications.

4 Standard Implementation

As previously outlined, being able to synchronize as well as utilize reliable time sources is important across many divergent network architectures and device types. Large organizations can accomplish reliable time by aquiring their own reference clock which can directly access satilite, radio, or atomic time sources. For smaller organizations, vendors, and individuals the NTP Pool Project is provided as a public time resource[12]. Both the NTP Pool Project and RFC8633[27] outline best common practices(BCP) when employing the use of pool.ntp.org resources. Below we review two such implementations.

4.1 DMZ Access

In secure network architectures there is generally a zone which allows for restricted external network access to approved destinations. In this zone, also referred to as a DMZ[28], an organization places the systems that will become the primary time sources for the ecompassing network. Deploying this configuration for distributed time to internal hosts provides a number of benefits. First, it reduces diplicate queries to ntp pool resources from multiple hosts within the network. Second, is the minimization of both external ntp quieries and network egress traffic. Lastly, if external access to pool.ntp.org was disrupted, network devices would still be able to source and continue to synchronize time within the network.

The figure below demonstrates the DMZ configuration where the primary NTP servers act as clients to pool.ntp.org and as servers to clients within the network. This configuration can allow for the DMZ NTP servers to admit covert data from the

unconditional trust placed in pool.ntp.org's public pool servers.

//TODO add image and figure out the latex figure numbering scheme

```
Placeholder for direct
access image / diagram.
```

Figure 2. The DMZ Access NTP scenario, where an NTP server in a DMZ serves replies to clients in a secure inner network.

4.2 Direct Access

There are instances where manufactuer or vendor hardware enforces the time source to be used. Numerous reasons for this includ user experience, respecting the public ntp pool available resources, and uncertainty with device deployment. In the instances where an appliance or embedded device is deployed to a customer network, the possibility exists that there are no available time servers, the version of time server is incompatable, or a customer may want to avoid dependancy on their network resource for the device.

In direct access, devices with the network synchronize their clocks directly via NTP to a preconfigured time source. Any information that the NTP server can embed into an NTP reply may reach devices inside this network.

//TODO add image and figure out the latex figure numbering scheme

```
Placeholder for direct
access image / diagram.
```

Figure 3. The Direct Access NTP scenario, where clients in a secure inner network are allowed direct access to a pool of NTP servers.

5 Covert Channel Implementation

The covert channel we implemented can transfer an arbitrary file from an NTP server, through a DMZ NTP server, to an NTP client on an internal network using a combination of BPF filters and user space applications.

5.1 Applications

The channel can be broken down broadly into three separate components, a client program, a server program, and a DMZ program. Each component uses some combination of BPF filter and user space application.

5.1.1 Client

The client resides entirely in uer space. It listens for NTP packets with extension fields. When it finds an extension field, it saves it to an in-memory buffer. When a certain sequence of bytes is received (a full transmistion of only `0xBE` bytes for our POC code) it trims off the sequence and any extra bytes, and saves the received file in the current working directory. This allows the client to receive any arbitrary file, including an executable, via NTP.

5.1.2 Server

The server uses a combination of BPF filter and userspace application. The userspace application's primary purpose is to load and configure the BPF filter. The application takes a given file (in our POC, a compiled `hello` Hello World program), breaks it into chunks for the NTP extension field, and saves it to the BPF program's ring buffer. The BPF program then sits and waits for outbound NTP messages, which it then appends the chunked file to as an extensison field. After all the chunks have been sent, it stops appending data until more information is written to the ringbuffer.

The use of a BPF filter means we do not need to modify the NTP server code itself. We simply modify existing packets generated by existing NTP requests and responses. This has multiple advantages: the NTP server code will always match legitimate NTP server code; the NTP server does not need to be modified or recompiled; our channel will survive an update and restart of the NTP server daemon; the server is unaware that the response has been modified.

5.1.3 DMZ

The DMZ, similar to the server, uses a combination of two BPF filters (one ingress and one egress) and one userspace application. The userspace application's purpose is, again, to load and configure the BPF filters. The egress filter behaves the same as

the server's egress filter. In fact, it is the exact same filter. It reads information from a ringbuffer and attaches it to outbound NTP responses. The ingress filter, however, is new. It's job is to read incoming NTP responses (e.g., from the server) and extract their extension fields. If the extension field matches the expected length, it is added to the ringbuffer so that it may be transmitted further down-stratum to additional clients. Using this technique we may chain an arbitrary number of intermediary NTP servers together, each saving and passing along bits of information from the initiating server to the ultimate client. For our POC we have limited ourselves to a single stratum hop.

5.2 Throughput

The throughput of our implementation is fairly flexible. In a test environment, we found about one NTP roundtrip request and response per 70 seconds was fairly typical. However, we are able to vary the size of the NTP extension field at will. Large packets will stand out so we want to keep our NTP extension to a reasonable size. Our implementation limits the extension field to 91 bytes, due to BPF's 512 byte stack limit[15]. However, there are many techniques we could have applied to circumvent this limit. With our current implementation we are able to send 91 bytes per 70 seconds, or approximately $91/70 = 1.3$ bytes per second (10.4 bits per second). Again, this can be scaled as needed, balancing the extension field's size with how well it blends in on the network.

5.3 Robustness

Our implementation is fairly naive. There is no error checking or attempt to hide that data is being transmitted. However, we are confident these techniques can easily be layered on top due to the flexibility we have in how much data we send per packet. Additionally, the official RFC guidance states that middleboxes should not attempt to modify, clean, or otherwise act as an active warden on NTP extension fields[29]. This stems from NTP extension fields being open for vendor-specific implementations (e.g., authenticated NTP) and attempting to "normalize" these fields could break NTP on a network, which would have disastrous consequences. For example, a Fortinet firewall attempting to normalize an authentication extension coming out of Windows Server might desync time for an entire domain, which would lead to chaos throughout anything that relies on Kerberos authentication. From the section 4, Security Considerations, of RFC7822:

> Middleboxes such as firewalls MUST NOT filter NTP packets based on their extension fields. Such middleboxes should not examine extension fields in the packets, since NTP packets may contain new extension fields that the middleboxes have not been updated to recognize.[29]

5.4 Detection

Detection of our specific implementation is fairly straightforward, simply look for an NTP packet that contains the ELF magic number. This marks the beginning of the binary we are sending over, and any further connections to this NTP server can be blocked. However, the use of encryption would render the contents opaque to any signature matching.

Another way to detect the channel would be to exhaustively list the known and allows NTP extensions on your network and flag anything not on that list. You would need to ensure that these extensions are strongly recognizable and have some structure. Keep in mind that an attacker can set the field type and structure at will, so you need a way of positively identifying known good extensions. If your network relies on an extension that appears the same as random encrypted data, you may be forced to rearchitect NTP or you will not be able to block this channel.

6 Conclusions and Future Work

Deploying content to one service and exploiting automated idempotence to further the propagation of the exploit. You could use puppet or ansible which is configured to return a directory or file to a previously well-known hashedstate. The replacing of our covert file

//TODO: roughtime securinig time with digital signatures [30] [31] [32]

6.1 Native Receiver

// TODO: talk about how we might implement a native receiver, using only physical access to the machine, our brains, and no internet besides NTP

References

[1] V. Gligor, "A guide to understanding covert channel analysis of trusted systems, version 1 (light pink book)," NCSC-TG-030, Library No. S-240,572, 1993.

[2] "What is ebpf? An introduction and deep dive into the ebpf technology," *What is eBPF? An Introduction and Deep Dive into the eBPF Technology.* [Online]. Available: https://ebpf.io/what-is-ebpf.

[3] "Timing," *GPS.gov: Timing Applications.* National Coordination office for Space-Based Positioning, Navigation,; Timing., Nov. 2019, [Online]. Available: https://www.gps.gov/applications/timing/.

[4] ESA, "GLONASS general introduction," *GLONASS General Introduction - Navipedia.* European Space Agency, 2011, [Online]. Available: https://gssc.esa.int/navipedia/index.php/GLONASS_General_Introduction.

[5] E. galileo, "Galileo general introduction," *Galileo General Introduction - Navipedia.* European Space Agency, 2011, [Online]. Available: https://gssc.esa.int/navipedia/index.php/Galileo_General_Introduction.

[6] NPL, "MSF radio time signal," *Time and Frequency.* National Physics Lab, UK, 2021, [Online]. Available: https://www.npl.co.uk/msf-signal.

[7] A. Bauch, "DCF77," *PTB.de.* PHYSIKALISCH-TECHNISCHE BUNDESANSTALT, May 2017, [Online]. Available: https://www.ptb.de/cms/en/ptb/fachabteilungen/abt4/fb-44/ag-442/dissemination-of-legal-time/dcf77.html.

[8] NIST, "Radio station WWVB," *NIST.* Physical Measurement Laboratory / Time; Frequency Division, Mar. 2010, [Online]. Available: https://www.nist.gov/pml/time-and-frequency-division/time-distribution/radio-station-wwvb.

[9] "Clock skew," *Wikipedia.* Wikimedia Foundation, Apr. 2021, [Online]. Available: https://en.wikipedia.org/wiki/Clock_skew.

[10] D. M. Havey, M. Blodgett, P. McPhee, E. Ross, and dknappettmsft, "Support boundary for high-accuracy time," *Microsoft Docs.* Microsoft, Oct. 2018, [Online]. Available: https://docs.microsoft.com/en-us/windows-server/networking/windows-time-service/support-boundary.

[11] A. Docs, "Maximum tolerance for computer clock synchronization," *Microsoft Docs.* Microsoft, Aug. 2016, [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/jj852172(v=ws.11).

[12] B. Hansen, "Pool.ntp.org," *pool.ntp.org: the internet cluster of ntp servers.* NTP Pool Project, 2014, [Online]. Available: https://www.ntppool.org/en/.

[13] D. Ziegenberg, "How to configure NTP for use in the NTP pool project on ubuntu 16.04," *Community Tutorial.* DigitalOcean, May 2017, [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-configure-ntp-for-use-in-the-ntp-pool-project-on-ubuntu-16-04.

[14] T. Graf, "eBPF - rethinking the linux kernel." InfoQ, Apr. 08, 2020, [Online]. Available: https://www.infoq.com/presentations/facebook-google-bpf-linux-kernel/.

[15] "BPF design q&a," *BPF Design Q&A - The Linux Kernel documentation.* [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/bpf_design_QA.html.

[16] "BPF and xdp reference guide - CILIUM 1.10.3 DOCUMENTATION," *BPF and XDP Reference Guide - Cilium 1.10.3 documentation.* [Online]. Available: https://docs.cilium.io/en/stable/bpf/.

[17] "Tc-bpf(8) - linux manual page," *Tc-Bpf(8) - Linux manual page.* [Online]. Available: https://man7.org/linux/man-pages/man8/tc-bpf.8.html.

[18] M. A. Brown, "Traffic control HOWTO," *Traffic control howto*. Oct. 2006, [Online]. Available: https://tldp.org/HOWTO/Traffic-Control-HOWTO/.

[19] "Network Time Protocol (NTP)." RFC 958; RFC Editor, Sep. 1985, doi: 10.17487/RFC0958.

[20] K. Halvemaan and R. Lahaye, "NTP as a covert channel system and network engineering university of amsterdam," 2017.

[21] E. Ekman, B. Anderson, and A. Bezemer, "Yarrick/iodine: Official git repo for iodine dns tunnel," *GitHub*. May 2016, [Online]. Available: https://github.com/yarrick/iodine.

[22] D. Stodle, "Ping tunnel," *Ping Tunnel - Send TCP traffic over ICMP*. Dec. 2004, [Online]. Available: https://stuff.mit.edu/afs/sipb/user/golem/tmp/ptunnel-0.61.orig/web/.

[23] montag451, D. Campelo, O. Gasser, T. Kain, and cofob, "Montag451/pytun: Linux TUN/TAP wrapper for python," *GitHub*. Mar. 2012, [Online]. Available: https://github.com/montag451/pytun.

[24] A. Ameri and D. Johnson, "Covert channel over network time protocol," in *Proceedings of the 2017 international conference on cryptography, security and privacy*, 2017, pp. 62–65.

[25] J. Martin, J. Burbank, W. Kasch, and P. D. L. Mills, "Network Time Protocol Version 4: Protocol and Algorithms Specification." RFC 5905; RFC Editor, Jun. 2010, doi: 10.17487/RFC5905.

[26] T. Schmidbauer and S. Wendzel, "Covert storage caches using the NTP protocol," in *Proceedings of the 15th international conference on availability, reliability and security*, 2020, pp. 1–10.

[27] D. Reilly, H. Stenn, and D. Sibold, "Network Time Protocol Best Current Practices." RFC 8633; RFC Editor, Jul. 2019, doi: 10.17487/RFC8633.

[28] "What is a DMZ and why would you use it?" *Fortinet*. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/what-is-dmz.

[29] T. Mizrahi and D. Mayer, "Network Time Protocol Version 4 (NTPv4) Extension Fields." RFC 7822; RFC Editor, Mar. 2016, doi: 10.17487/RFC7822.

[30] C. Patton, "Roughtime: Securing time with digital signatures," *The Cloudflare Blog*. The Cloudflare Blog, Sep. 2018, [Online]. Available: https://blog.cloudflare.com/roughtime/.

[31] G. Git, "Roughtime," *Google Git*. [Online]. Available: https://roughtime.googlesource.com/roughtime/.

[32] "Radclock," *Network Time Foundation*. Jan. 2014, [Online]. Available: https://www.nwtime.org/projects/radclock/.