Detecting Covert Channels Through Code Augmentation

Marco Zuppelli[1], Luca Caviglione[1] and Matteo Repetto[1]

[1]IMATI - Institute for Applied Mathematics and Information Technologies, Via de Marini, 6 (Torre di Francia) - 16149 Genova, Italy

Abstract

Modern malware increasingly exploits information hiding or steganography to elude security frameworks and remain unnoticed for long periods. To this aim, a prime technique relies upon the ability of creating covert channels to bypass the limits imposed by a sandbox or to exfiltrate data towards a remote server. Unfortunately, detecting a covert channel is not a trivial task and often requires to inspect a composite set of information, e.g., the behavior of a software or statistical indicators of network traffic. Therefore, in this paper we investigate the adoption of code augmentation features offered by the Linux kernel to gather data useful to reveal the presence of covert communications. To prove the effectiveness of the approach, we tested a lightweight program to detect covert channels targeting IPv6 conversations. Results indicate that technologies like the extended Berkeley Packet Filter can offer a foundation to frameworks for spotting and mitigating covert communications.

Keywords

Covert channels, code augmentation, eBPF, network security

1. Introduction

Improvements in network defense and the increasing sophistication of modern software and services have caused the proliferation of advanced techniques allowing a malware to remain unnoticed or to bypass secure execution enclaves. Until recently, main approaches used by attackers exploited code obfuscation, multi-stage loading, file-less implementations to prevent detection from antivirus, encryption as well as anti-forensics methods to evade code and memory analysis [1]. A recent trend exploits information hiding and steganography, which allow malicious software to remain undetected, covertly communicate with a remote command & control facility, or evade execution perimeters enforced via virtualization, containerization and sandboxing [2]. For instance, steganography can be used to conceal the presence of malicious code or an additional payload within an innocent-looking image. Even if several techniques exist, the main use of information hiding to enhance malware concerns the creation of *covert channels*, i.e., hidden communication paths between two software or hardware endpoints [3].

The general concept of covert channel has been introduced by Lampson in 1973. According to [4], covert channels are *"[channels] not intended for information transfer at all"*. Malware observed

ITASEC'21: Italian Conference on CyberSecurity

marco.zuppelli@ge.imati.cnr.it (M. Zuppelli); luca.caviglione@ge.imati.cnr.it (L. Caviglione); matteo.repetto@ge.imati.cnr.it (M. Repetto)

0000-0001-6932-3199 (M. Zuppelli); 0000-0001-6466-3354 (L. Caviglione); 0000-0001-8478-2633 (M. Repetto)

"in the wild" typically exploits covert channels to implement two main attack mechanisms. The first aims at endowing two malicious processes with the capability of communicating to bypass local security policies enforced within a given host or device. The second is devoted to support Internet-wide communications, for instance to exfiltrate sensitive information, orchestrate a botnet or configure/activate a backdoor [3, 5, 6].

During the years, both researchers and cybercriminals developed a variety of mechanisms to create hidden communication paths in a wide array of scenarios, such as, virtualized environments, cloud datacenters, network infrastructures, and multi-processor frameworks [5, 6]. As a consequence of such heterogeneity, the data hiding used to create the channel is highly specialized and tightly coupled with the targeted hardware/software entity. Thus, the detection is a poorly generalizable task and the main defense pattern requires to inspect several digital artifacts (often having incompatible structures or exploiting different technologies) to spot the presence of hidden data [5]. To give an idea of the composite surface that can be targeted by information-hiding-capable threats, attackers can conceal their activity or implement an abusive communication service within: digital images, network packets, file permissions, the sequence of operations offered to a CPU/GPU, HTTP cookies, DNS queries, locks applied to Unix/network sockets, as well as the temporal evolution of the state values of hardware components like the volume or vibration settings of mobile phones [2, 5, 6].

Therefore, an important requirement to fully assess the security of modern devices and digital infrastructures deals with the ability of detecting covert channels. To cope with the composite nature of hardware and software components that can be used to conceal secret information, we take advantage of code augmentation techniques made available by the Linux kernel. In more detail, we showcase the use of the extended Berkeley Packet Filter (eBPF), which supports the monitoring of a rich set of behaviors, such as system calls, page faults, memory occupation and network packets (see, e.g., [7] and the references therein). Specifically, we propose to create ad-hoc, lightweight programs to gather suitable information that can reveal the presence of a covert channel. At the best of our knowledge, code augmentation has never been used for the specific task of detecting malware endowed with steganographic capabilities, except for our previous work [8], which focused on spotting channels with a single-host scope via anomalous distributions in the invocation of `__x64_sys_chmod` syscalls. Instead, in this work, we focus on how code augmentation can be used to inspect network packets and reveal covert communications laying within traffic flows. To test our idea in a realistic scenario, we apply code augmentation to spot the presence of a malware trying to exfiltrate data via IPv6, which is expected to become a major target in next years [2, 3, 9].

The contributions of this paper are: *i*) a discussion on the main attack models exploiting covert channels, which are often neglected when addressing security of a digital infrastructure, *ii*) a framework leveraging code augmentation able to provide a technological foundation against information-hiding-capable threats, and *iii*) a preliminary performance evaluation campaign considering threats targeting IPv6 traffic.

The rest of the paper is structured as follows. Section 2 briefly introduces covert channels and how they can be used to attack computing and networking scenarios. Section 3 deals with code augmentation and how it can support the detection of network covert channels, while Section 4 showcases numerical results. Lastly, Section 5 concludes the paper and portraits potential future developments.

2. Covert Channels: Background and Attack Model

Even if covert channels can also enforce privacy of users, e.g., to preserve the anonymity of a source in investigative journalism or to prevent censorship in regimes, they are primarily used to develop offensive techniques [10]. In general, a covert channel is a hidden communication path established by two secret endpoints (often defined as the covert sender and the covert receiver) embedding data into a suited container, denoted as the carrier. Covert communications can be described via three different performance metrics: the *steganographic bandwidth* (i.e., the amount of secret information sent per time unit), the *detectatability* (i.e., how much is difficult to spot the channel), and the *robustness* (i.e., how many alterations or manipulations the hidden information can withstand). Such parameters are coupled: for instance, a high volume of secret information could require several manipulations of the carrier, thus making the channel more visible due to the presence of many alterations [6]. From the viewpoint of empowering a malware, there are two main types of channels: *local covert channels* and *network covert channels*. In the following, we will review the main attack models leveraging the aforementioned approaches.

2.1. The "Colluding Applications" Threat

In this attack model, the hidden communication happens among entities that reside on the same host/device, e.g., processes, virtual machines, and containers able to communicate by exploiting resources of the host. Some examples of techniques for creating a local covert channel include the modulation of the usage of resources such as the CPU and RAM or the manipulation of the underlying file system [11]. Figure 1 depicts the reference scenario where two entities (e.g., virtual machines, containers or processes) run independently within their sandboxes. Usually, this mechanism is sufficient to guarantee that two processes cannot communicate or share information. The colluding applications threat attempts to find a way to elude such a security architecture. For instance, let us consider that the process confined in Sandbox A, i.e., the covert sender, is able to read sensitive information. Thus, the security framework of the host prevents the process to communicate remotely (e.g., the sandbox blocks the access to the TCP/IP protocol stack). Instead, the process running in Sandbox B, i.e., the covert receiver, is considered safe, thus it can access network services. The applications can then collude by setting up a local covert channel to bypass their respective sandboxes. As a possible example, the sender process can encode the secret within usage patterns of the CPU (e.g., by increasing the load to signal a `1` or by putting itself in a sleep state to signal a `0`) in order to leak information to the receiving process.

2.2. Exfiltration and Remote Command & Control

Network covert channels enable remote peers to secretly communicate by manipulating protocols or injecting data in network traffic, [5]. Due to their effectiveness in exfiltrating data and eluding de-facto standard security frameworks, network covert channels raised the attention of the scientific community, thus many works reviewing the literature and proposing taxonomies have been done. For instance, the authors of the work [12] proposed a possible classification

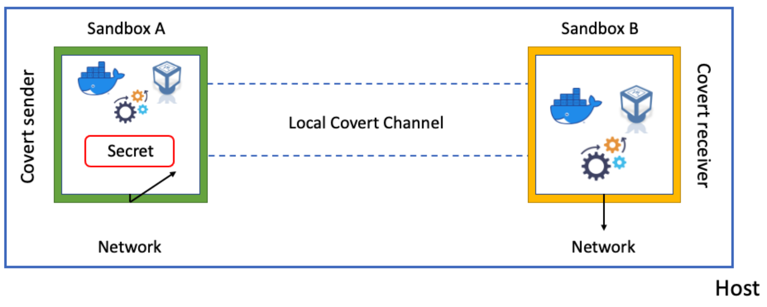


Figure 1: Reference scenario for the colluding applications threat.

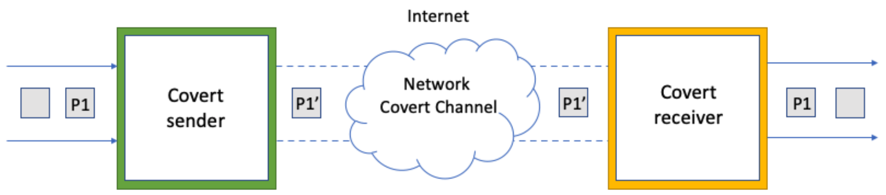


Figure 2: Reference scenario for the usage of a network covert channel.

of the types of network covert channels, distinguishing among various information-hiding patterns. Two primary groups of methods are outlined: *timing* methods and *storage* methods. The first type allows to inject the secret information by modifying the timing characteristics of the network flow, e.g., the alteration of the inter-packet time. The second type allows to inject the secret information directly in the traffic stream, e.g., in the payload of a packet or in some unused fields within the protocol header. Typically, malware exploits network covert channels to implement a wide range of attacks, such as, to implement command & control infrastructures, to send commands to a botnet, to exfiltrate secret data or to bypass the security framework of the guest operating system.

Figure 2 depicts the reference attack model. In more detail, the covert sender wants to transmit an information to a remote peer, for instance a sensitive information exfiltrated by a third-party malicious routine. To this aim, it gathers incoming packets belonging to an overt, licit flow (denoted as P1 in the figure) and, according to the used injection method, it performs suitable modifications. As a possible example, the covert sender can encode a secret by adding a predefined delay or by directly injecting data within some fields of the header. As a result, an "altered" packet containing the secret information (denoted as P1' in the figure) is sent through the network. The covert receiver can then retrieve the information by knowing in advance the feature of the traffic flow that has been used as the carrier. However, discrepancies between P1 and P1' can reveal the presence of the covert channel. To reduce the chance of detection, the covert receiver should restore the traffic flow in such a way to not arise suspects.

Even if the majority of covert channels targets the IPv4 protocol [5, 6], the increasing popularity of IPv6 makes it attractive for attackers and malware developers [9]. In fact, exploiting emerging technologies or protocols still not widely deployed or understood may give to attackers an advantage. Therefore, in the rest of this work, we will consider that the network covert channel depicted in Figure 2 has been implemented within an IPv6 conversation. In this vein, the literature proposed several techniques to inject secret information in the IPv6 protocol (see, e.g., [13] for a comprehensive analysis). However, many of them appear to be unsuitable when deployed in realistic use-cases, especially when in the presence of v6/v4 transitional mechanisms [9]. Therefore, in the following, we will focus on network covert channels built by storing data within the `Flow Label` field (i.e., a 20 bit long value supporting intermediate nodes in routing operations) and in the `Traffic Class` field (i.e., a 8 bit long value describing the type of service delivered by the network). For the sake of completeness, we will also consider a channel injecting the secret information via a modulation of the value of the `Hop Limit` field (i.e., a 8 bit long counter limiting the number of nodes that a datagram can traverse). In this case, the secret information is encoded by increasing or decreasing the value of the `Hop Limit` for two consecutive datagrams, e.g., bits `1` and `0` are encoded by increasing or decreasing the field by a fixed value.

3. Visibility Through Code Augmentation

Due to the different nature of the models outlined in Section 2, each attack could require a specific detection technique. For instance, colluding applications can be revealed via correlation metrics, e.g., processes wanting to communicate become active in an overlapped manner to alter/read the used carrier before it is disrupted by other competing entities or the OS [11]. In the case of network covert channels, the prime approach leverages traffic monitoring, e.g., deep packet inspection frameworks to reveal anomalous patterns or to spot information injected in the header.

Given the broad range of potential methodologies for creating covert channels, a promising idea to perform their detection concerns the use flexible and easily extensible frameworks. In fact, being able to collect threat-independent measurements using the same technology is desirable since the carrier exploited by the attacker is usually not known in advance. However, the ability of generalizing the detection process should not be considered the only objective: another desired goal deals with performances and the detection process must limit the consumption of processing, memory, and networking resources. In this respect, code augmentation stands out as the best approach to gain visibility over multiple functions and processes within a host and its operating system [8]. Put briefly, code augmentation is a technique that pushes additional bytecode into a running application, in order to dynamically extend its functionality. One fundamental advantage of code augmentation is that multiple complementary "hooks" can be developed to monitor and trace different subsystems, without requiring major changes to the whole design. However, it is important to verify the code before the insertion, and to avoid creating instability or new vulnerabilities.

In this section, we briefly introduce the used code augmentation framework. Then, we present how data can be organized to guarantee performances and mitigate the consumption of resources.

3.1. Code Augmentation via eBPF

Originally developed as an efficient mechanism for packet monitoring and filtering, the eBPF has been recently extended into a more complex framework. For example, it can be used to inspect network packets, to trace specific kernel functions or to monitor the CPU or the memory usage. In this extent, eBPF can provide a basic mechanism to collect threat-independent measurements for detecting stegomalware. To this aim, eBPF provides a wide set of functionalities, including key/value maps to store data, supporting routines to allow kernel to user-space communications and mechanisms to concatenate multiple programs. The eBPF programs may be executed on the reception of packets in raw sockets, queues, xdp driver or can be attached to kprobes, tracepoints and perf events. To enforce security, eBPF provides a dedicated virtual machine within the Linux kernel, with a limited access to system resources. Thus, eBPF is intrinsically safer and more robust than other mechanisms (e.g., kernel modules) even if executed in kernel mode. Due to performance and security constraints, the only way to interact with an eBPF program is through maps, which are shared-memory regions. For this reason, the typical development pattern includes both the eBPF program and a user-space utility for its loading and to exchange data through maps. An in-kernel verifier validates the code to avoid kernel deadlocks.

So far, the primary usage of eBPF has been tracing and monitoring the Linux kernel for investigating performance issues. As an example, the toolkit (e)BPF Compiler Collection (BCC)[1] contains several working tools to do such operations. The tools are based on a user-friendly Python class for compiling, loading and attaching eBPF programs to several hooks.

Owing to is flexibility, the eBPF framework can collect a wide range of data, addressing both local covert channels and network covert channels.

3.2. Data Collection

Since the range of techniques for creating covert channels is very broad and the used carrier could be not known a priori, anomaly detection approaches appear suitable to spot the presence of stegomalware. For the case of colluding applications, anomalies can be searched for by evaluating usage patterns of kernel functions. As a possible example, a kernel function can be traced to spot the presence of colluding applications exchanging data through a local covert channel manipulating file permissions. Thus, eBPF can monitor the kernel function `__x64_sys_chmod` to identify anomalies in its distribution and evolution [8]. For the case of network covert channels, a similar approach can be used for counting the possible values assumed by fields in the header of the various network flows. The gathered data can be then compared with expected usage patterns, to identify anomalies. For instance, when this approach is applied to the `Flow Label` field, it can provide a rough estimation of active flows in the network. An anomaly is then raised when the number of different `Flow Label` values seen in a given period is different from the number of active flows, which can be provided by a third-party security tool. Counting the various values can be straightforward when in the presence of fields with a limited range (e.g., the `Traffic Class` and `Hop Limit` are characterized by 256 different values) but can pose

[1]https://github.com/iovisor/bcc.

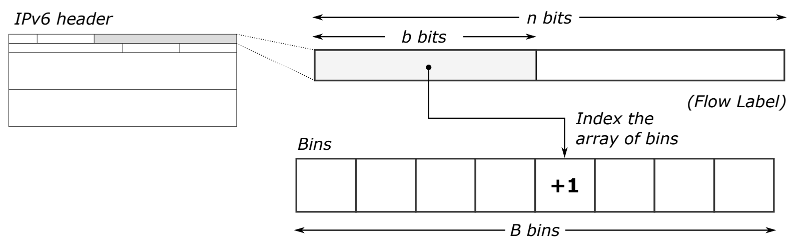


Figure 3: Mapping field values to bins.

some challenges for the case of `Flow Label`, which generates a space of 2^{20} different values. To overcome performance issues, we split the whole range values into a smaller number of groups (called "bins"), and used a counter for each group.

The implementation of this mechanism within an eBPF program is depicted in Figure 3, with the `Flow Label` as an example. For each incoming packet, the value of the `Flow Label` is placed in the corresponding bin, and the associated counter is incremented. To make the implementation more efficient, the number of bins is always a power of 2, so that the association of a field value to the corresponding bin is a simple bitwise operation to match the prefix.

The bin-based structure reduces the memory usage of the eBPF program, but at the expense of coarser-grained granularity of the collected statistics. The proper number of bins must be selected according to the specific use case, seeking an optimal balance between resource consumption and precision of the detection. In this case, the anomaly could be searched for by evaluating how the volume of changing bins evolves during time. To have a condensed indicator, our user-space utility periodically collects the values for all bins and considers the number of bins that change between two consecutive reads.

4. Numerical Results

To prove the effectiveness of the proposed approach based on code augmentation, we prepared a testbed composed of two virtual machines running Debian GNU/Linux 10 (kernel 4.20.9), which communicate through a third virtual machine. The latter, with the same technical specifications, is in charge both of routing traffic and running the eBPF program to gather information about the traffic exchanged between the two peers. To test our idea in realistic network conditions, we replayed legitimate IPv6 traffic conversations collected on a OC192 link between Sao Paulo and New York on January 17, 2019 from 14:00 to 15:00 CET, made available by the Center for Applied Internet Data Analysis. Specifically, for this work we used the CAIDA Anonymized Internet Traces Dataset collected in the April 2008 - January 2019 period[2]. Traffic traces has been replied with the `tcpreplay` tool and resulting flows represented the bulk overt traffic. To have a suitable degree of freedom, we generated via the `iPerf3` tool an additional conversation between the two endpoints wanting to secretly communicate with a bandwidth of 500 kbps. The covert channel has been implemented via an ad-hoc Python script using Scapy 2.4.3 and

[2]Used traces: CAIDA dataset, Jan. 17th 2019.
Available online: https://www.caida.org/data/monitors/passive-equinix-nyc.xml [Last Accessed: March 2021].

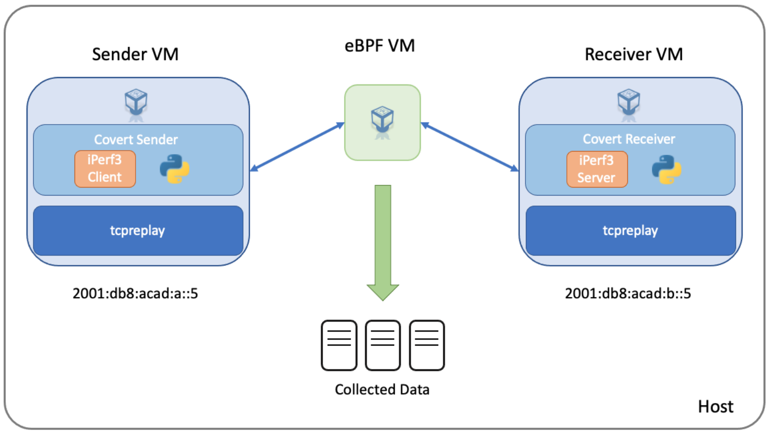
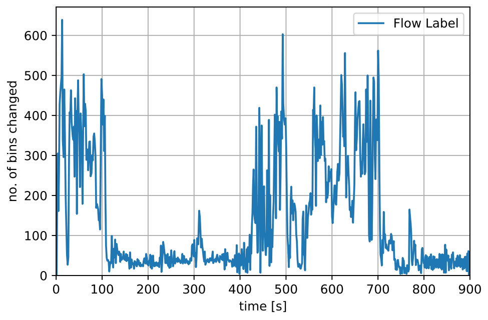


Figure 4: Reference testbed used in this work.

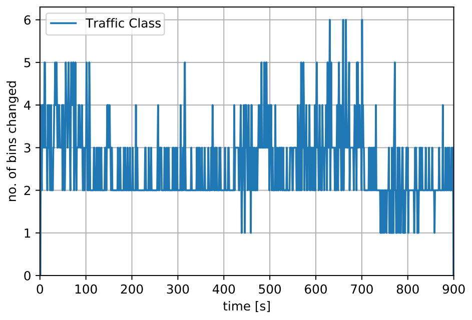
NetfilterQueue 0.8.1. Figure 4 depicts the resulting testbed.

In the first set of trials, we wanted to evaluate the effectiveness of the eBPF-based approach in gathering statistical information on the traffic useful to spot the presence of a network covert channel. To this aim, we run a script able to capture, on a per-packet basis, the value of the `Flow Label`, `Traffic Class`, and `Hop Limit` fields and populate the data structure described in Section 3.2. The sampling time used to gather information obtained from the traffic, i.e., the timeframe between two different measurements, was set to 1 second. In other words, the user-space programs retrieve from the eBPF counterparts the collected values for the bins every second. The number of bins used in the case of `Flow Label` was set to 2^{15}, while for the remaining fields the value was set to 2^{8}.

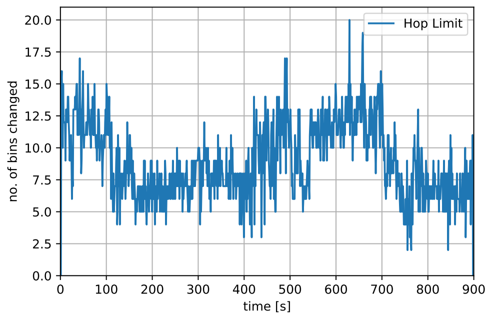
Figure 5 depicts the obtained results. For the sake of clarity, we limited traces to 15 minutes of traffic. In more details, the figure shows the ability of our lightweight code augmentation framework to capture the temporal evolution of the values characterizing a specific field of the IPv6 header. Recalling that to have a "condensed" metric we inspect the number of bins changed between two consecutive measurements provided by the eBPF filter, the depicted trends offer several insights on the observed IPv6 traffic. As an example, if an attacker exploits the `Traffic Class` field to contain secrets, the evolution of the number of bins changed will differ significantly from the expected behavior. In fact, as it is possible to notice from Figure 5(b), the number of different `Traffic Class` observed is always limited to only few values. Therefore, the presence of a hidden communication could reflect into a sort of "signature" in the number of changing bins and allow to spot the covert communication. A similar consideration can be done for the `Flow Label`, even if Figure 5(a) depicts a greater variety in terms of observed values. In this case, additional inputs from the network could be needed. For instance, this trace could be checked against the number of active flows present at a given time step. Since each flow is characterized by a unique value for the `Flow Label`, discrepancies between the number of active conversations and the volume of changing bins could reveal the presence of the hidden channel. Instead, for the case of `Hop Limit`, a more sophisticated approach could be needed as the information-modulating nature of the embedding methods makes the covert communication



(a) Evolution of the `Flow Label`



(b) Evolution of the `Traffic Class`



(c) Evolution of the `Hop Limit`

Figure 5: Number of changing bins of the observed traffic when gathering data for different fields.

harder to spot.

Concerning the resources consumption, we measured the CPU usage and the amount of memory needed by both the eBPF program and the user-space utility. Since the eBPF program is executed only when a packet is processed, defining the amount of CPU used is not straightforward, whereas the memory occupation strictly depends on the size of the used kernel map. Owing to the proposed bin-counting architecture, it remains bounded to few megabytes. As regards the CPU usage for the user-space utility, measurements indicate that, for the `Traffic Class` and `Hop Limit` cases, it is ∼3%, while for the `Flow Label` the consumption increases to ∼12%. We point out that these values may change according to the selected sampling interval (1 second, in our tests). Summing up, the eBPF-based approach demonstrated to introduce only minimal overheads especially in terms of delays added to the network traffic. Thus it has to be considered a suitable technology for the development of countermeasures to be deployed in realistic scenarios (see, e.g., [14] for a work considering its deployment through containerization).

To assess the real capacity of using bin-based behaviors as effective indicators to spot the presence of a channel, we conducted an additional round of tests. To this aim, we hold the same traffic conditions described above and we added a covert channel exfiltrating data within an

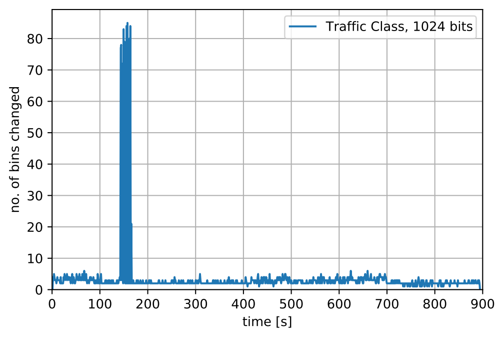


Figure 6: Evolution of `Traffic Class` when a covert communication is present.

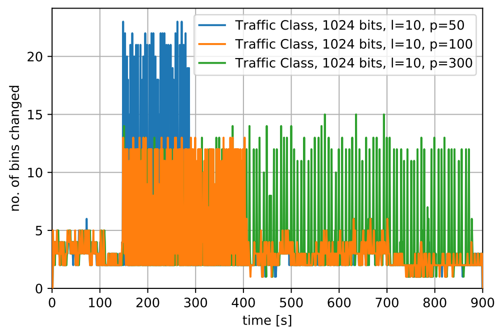


Figure 7: Evolution of `Traffic Class` when a covert communication is present with different injection policies.

IPv6 conversation. We considered a hidden transmission targeting the `Traffic Class`, which has been used to exfiltrate a secret message of 1, 024 random bits. This can represent the exfiltration of a cryptographic key or of a sensitive information [2, 5, 6]. Figure 6 depicts the results. As shown, the attacker utilizes all the packets composing the overt stream, thus creating a channel with the maximum steganographic bandwidth achievable. The secret message is then sent in few seconds but at the expense of a poor undetectability. Figure 6 clearly shows that our metric can spot the hidden communication attempt. In fact, the "spike" in the number of changed bins clearly appears as an anomaly.

In order to make the covert communication more difficult to spot, an attacker can choose, for example, to limit the steganographic bandwidth of the hidden channel. To prove this idea, we conducted an additional round of tests. Specifically, we considered an attacker able to reduce the bandwidth of the hidden channel by alternating the amount of consecutive stego-packets (i.e., packets that are injected with the secret, denoted with l in the following) with the amount of legitimate packets (denoted with p in the following). In this vein, we conducted trials with $l = 10$ and $p = 50, 100,$ and 300 packets.

Figure 7 depicts the outcome when different patterns of injection are applied. As the steganographic bandwidth decreases, i.e., p increases, the time needed by the attacker to transmit the secret in its entirety rises. As regards detectability of the covert channel, when the attacker uses a value for $p = 50$ (i.e., the blue line in Figure 7), the rate of hidden data is still not adequate compared to the rest of the traffic, thus making the channel more detectable. This is due to the superimposition of two causes: the few different values for the `Traffic Class` characterizing the overt traffic, and a too aggressive injection procedure. To mitigate such effect, the attacker can apply some form of encoding, e.g., map the secret into a reduced number of values, possibly observed in the overt bulk of data [9]. As hinted, another idea concerns slowing the steganographic bandwidth by interleaving stego-packets with a greater amount of unmodified data units. In this case, the values of the "spikes" are closer to the rest of the evolution of the `Traffic Class`, which lead to less detectable covert channels (see, Figure 7). However, the resulting steganographic communications will last longer (141 seconds compared to 23 seconds), potentially causing more visible alterations of the overt traffic flow. Similar considerations can be done also for the case of `Flow Label`. Instead, for the case of the `Hop Limit`, the number of changing bins can not be employed "out of the box" and needs further investigations: this is part of our ongoing research.

5. Conclusions and Future Work

In this work, we presented a lightweight approach based on a "counting" scheme designed to detect network covert channels targeting IPv6 traffic. The approach take advantage of the code augmentation mechanism based on eBPF and can help to reveal the presence of covert channels. Results have indicated the effectiveness of the approach.

Future work aims at refining the idea and carry out a thorough performance evaluation campaign. A relevant part of our effort is devoted to understand the feasibility of using code augmentation to implement an "active warden", i.e., an intermediate node able to disrupt the covert channel. For the case of network covert channels, this can be a middlebox overwriting suitable fields in the header of packets. Instead, for the case of colluding applications, this approach is less obvious and requires further investigations. Another part of our ongoing research concerns the design of a threat-independent metric able to reveal the presence of both local and network covert channels as well as an effective mechanism to automatically raise an alarm when detecting the hidden communication attempt.

6. Acknowledgments

This work has been supported by the EU Project GUARD, Grant Agreement No 833456, and by the EU Project SIMARGL, Grant Agreement No 833042.

References

[1] A. Qamar, A. Karim, V. Chang, Mobile malware attacks: Review, taxonomy & future directions, Future Generation Computer Systems 97 (2019) 887–909.

[2] W. Mazurczyk, L. Caviglione, Information hiding as a challenge for malware detection, IEEE Security & Privacy 13 (2015) 89–93.

[3] K. Cabaj, L. Caviglione, W. Mazurczyk, S. Wendzel, A. Woodward, S. Zander, The new threats of information hiding: The road ahead, IT Professional 20 (2018) 31–39.

[4] B. W. Lampson, A note on the confinement problem, Communications of the ACM 16 (1973) 613–615.

[5] S. Zander, G. Armitage, P. Branch, A survey of covert channels and countermeasures in computer network protocols, IEEE Communications Surveys & Tutorials 9 (2007) 44–57.

[6] W. Mazurczyk, L. Caviglione, Steganography in modern smartphones and mitigation techniques, IEEE Communications Surveys & Tutorials 17 (2014) 334–357.

[7] S. Miano, M. Bertrone, F. Risso, M. Tumolo, M. V. Bernal, Creating complex network services with eBPF: Experience and lessons learned, in: 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), IEEE, 2018, pp. 1–8.

[8] A. Carrega, L. Caviglione, M. Repetto, M. Zuppelli, Programmable data gathering for detecting stegomalware, in: Proceedings of the 2nd International Workshop on Cyber-Security Threats, Trust and Privacy Management in Software-defined and Virtualized Infrastructures (SecSoft), IEEE, 2020.

[9] W. Mazurczyk, K. Powójski, L. Caviglione, IPv6 covert channels in the wild, in: Proceedings of the 3rd Central European Cybersecurity Conference, 2019, pp. 1–6.

[10] J. Saenger, W. Mazurczyk, J. Keller, L. Caviglione, VoIP network covert channels to enhance privacy and information sharing, Future Generation Computer Systems 111 (2020) 96–106.

[11] M. Urbanski, W. Mazurczyk, J.-F.-. Lalande, L. Caviglione, Detecting Local Covert Channels Using Process Activity Correlation on Android Smartphones, International Journal of Computer Systems Science and Engineering 32 (2017) 71–80.

[12] S. Wendzel, S. Zander, B. Fechner, C. Herdin, Pattern-based Survey and Categorization of Network Covert Channel Techniques, ACM Computing Surveys (CSUR) 47 (2015) 1–26.

[13] N. Lucena, G. Lewandowski, S. Chapin, Covert channels in IPv6, in: Int. Workshop on Privacy Enhancing Technologies, Springer, 2005, pp. 147–166.

[14] C. Liu, Z. Cai, B. Wang, Z. Tang, J. Liu, A protocol-independent container network observability analysis system based on eBPF, in: 2020 IEEE 26th International Conference on Parallel and Distributed Systems, IEEE, 2020, pp. 697–702.