IEEE NetSoft 2023 – 5th International Workshop on Cyber-Security Threats, Trust and Privacy Management in Emerging Software-defined and Virtualized Infrastructures

Information Leakages of Docker Containers: Characterization and Mitigation Strategies

Marco Zuppelli, Matteo Repetto, Luca Caviglione, Enrico Cambiaso
National Research Council of Italy, Genoa, Italy
Email: {name.surname}@cnr.it

2023 IEEE 9th International Conference on Network Softwarization (NetSoft) | 979-8-3503-9980-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/NetSoft57336.2023.10175435

***Abstract*—Compared to classic virtual machines, containers offer lightweight and dynamic execution environments. Hence, they are core building blocks for the development of future softwarized networks and cloud-native applications. However, containers still pose many security challenges, which are less understood compared to other virtualization paradigms. An important aspect often neglected concerns techniques enabling containers to leak data outside their execution perimeters, e.g., to exfiltrate sensitive information or coordinate attacks. In this paper we investigate security impacts of covert communications based on the looser isolation of memory statistics information. Our characterization indicates that the investigation of system calls should be considered a prime tool to reveal the presence of collusive attack schemes. We also elaborate on two mitigation techniques: the first entails prevention via "hardening" configurations of containers, while the second implements a run-time disruption mechanism.**

***Index Terms*—containers security, softwarization, data leak, covert channels, cybersecurity**

I. Introduction

The rapid evolution of network services and cloud-native applications requires flexible and scalable paradigms able to bypass physical boundaries. Owing to their lightweight nature and better support to the creation of micro-services, containers are now considered the prime alternative to virtual machines [1]. Despite many advantages, security is still the main barrier in their adoption in many production-quality deployments. In more detail, containers require protection to different attack scenarios, for instance from internal applications or to prevent harmful operations against the host. An emergent trend concerns inter-container attacks for bypassing secure perimeters or isolation features provided by the kernel [2]. To this aim, the creation of covert channels exploiting network traffic or the "imperfect" isolation of the various software components should be considered the most effective methods [3], [4]. In general, data can be covertly exchanged for three main goals. First, sensitive information can be leaked through multiple containers to reach an exfiltration point under control of the attacker [5]. Second, containers can implement simple communications to perform reconnaissance campaigns over the underlying hardware framework [6]. Third, malicious containers can exchange messages to orchestrate attacks or find the best timing to launch offensive routines or DoS campaigns [7].

Unfortunately, mechanisms for leaking information or implementing covert communications are seldom considered when assessing the security of softwarized/containerized infrastructures. As an example, the need of an effective isolation is well understood for the case of complex workflows, but not for threats acting with a per-container granularity [8]. Similarly, deploying advanced access control policies is a consolidated approach to reduce the abuse of co-residency properties, even if covert communications are never explicitly addressed [9]. Yet, attacks for smuggling data outside containers can be mitigated by means of different paradigms. For instance, the involved software components (e.g., the container engine) can be hardened, with the aim of reducing the exploitable attack surface [10]. Another important aspect concerns observability, i.e., the ability to perform advanced monitoring to gain detailed insights on the state of a system [11]. In this case, a rich telemetry can be used to early detect threats or identify anomalous behaviors. To trace events or inspect containers, the extended Berkeley Packet Filter (eBPF) proven to be effective in a wide-range of applications [12], including the mitigation of network covert channels [3], [13]. Yet, eBPF has been mainly used to evaluate the performances or to implement scalable traffic monitoring frameworks, rather than to enforce the security of applications based on micro-services or virtualization [12], [14]. At the best of our knowledge, there are no any previous research attempts dealing with the detection of covert communications based on the loose isolation of containers. In fact, the majority of works on inter-container communications deals with network aspects (see, [15] and the references therein).

This paper investigates security implications of containers able to leak information. With the aim of outlining possible mitigation strategies, it provides a characterization of the covert communication exploiting the loose isolation of statistics describing the memory available for the host [5], [7].

The rest of the paper is structured as follows. Section II introduces the threat model and the testbed, while Section III characterizes the memory-based covert channel and how some of its traits can be used for spotting the hidden data exchange. Section IV showcases two paradigms to void information leakages, and finally, Section V concludes the paper and outlines some future research directions.

II. Threat Model and Testbed

In this section, we introduce the considered threat model and the testbed used to conduct experiments.

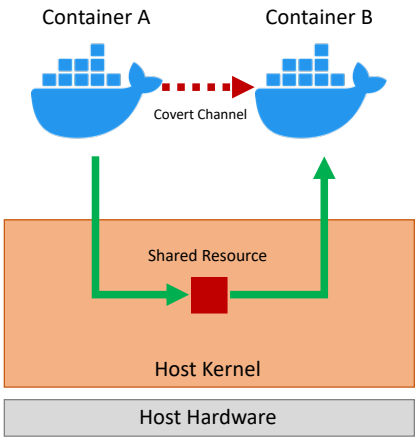


Fig. 1. Threat model: two containers exploit a shared resource to encode information and implement a covert channel.

A. Threat Model

The threat model considered in this work deals with containers residing on the same host and wanting to leak information through an hidden communication path. This scheme can be used to leak sensitive information outside the hosting node or to implement co-residency attack templates [5]. Figure 1 depicts the reference threat model. In more detail, two malicious containers (denoted in the figure as Container A and Container B) implement a covert communication by altering a resource with a "global" visibility. This allows to elude the isolation provided by the kernel, e.g., namespaces and cgroups. For instance, Container A can encode data by suitable alterations of a resource, which can be observed by Container B due to "imperfect" isolation. Among the various techniques, an effective approach exploits the manipulation of software and hardware statistics (e.g., the load of the CPU or the number of threads) observable through the `/proc` filesystem [7].

In this work, we consider a covert channel exploiting how information on the free memory is handled by Linux[1]. Specifically, memory used/freed by Container A will impact the overall memory of the host, which can be read by Container B through the `/proc/meminfo` file. Then, Container A can encode a binary message for Container B by modulating the amount of used memory. Preliminary, Container A (i.e., the sender) calculates the average time for allocating a chunk of memory: this value will be used to calibrate the bandwidth of the channel. In other words, this step allows the sender to have a symbol rate not exceeding the time frame needed to perform the required operations on the memory. The sender then encodes the binary value `1` by temporarily allocating an amount of memory large enough to impact the overall free memory of the host. Next, Container B (i.e., the receiver) periodically checks via `/proc/meminfo` the overall free memory to infer the secret information. If the difference exceeds a fixed threshold, the value `1` is decoded. Instead, when the sender wants to transmit the binary value `0`, it simply sleeps for a fixed amount of time. To have a rough synchronization mechanism, the leakage attempt starts when, for both endpoints, the number of seconds of the current "time of the day" can be divided by 20 with no remainder. The implementation of the covert communication mechanism considered in this work exploits chunks of memory of $\sim$524 MB to encode the binary `1`, which is correctly decoded if it accounts for an overall memory increase of at least 200 MB. To improve the robustness of the hidden communication attempt, the receiving endpoint checks the free memory reported in the `/proc/meminfo` once per second and decodes the binary value `1` or `0` according to the average of the last 5 samples. This reduces the impact of other processes or containers allocating/de-allocating memory, which can disrupt the encoded secret or prevent the covert communication mechanism (see, [7] for a performance evaluation considering the presence of other competing containers).

B. Experimental Testbed

For our testbed, we used Docker[2] 23.0.1 to implement the containerized architecture over a host equipped with a 3.60 GHz Intel i9-9900KF CPU with 32 GB of RAM running Ubuntu 20.04 (Linux kernel 5.15). To characterize how information is leaked, we performed tests by only considering the two colluding containers, unless differently specified. According to our setup, the Python script implementing the covert communication mechanism requires, on average, $\sim$122 ms to allocate and de-allocate the needed chunk of memory via the creation and the removal of a suitable variable, leading to a channel of a bandwidth of $\sim$0.2 bit/s. We point out that, the original implementation has been slightly modified to prevent scripts logging data to files and to reduce the presence of visible signatures. To monitor the behavior of containers as well as the hosting machine, we used `trace` and `syscount` utilities from the BPF Compiler Collection[3], which periodically report detailed statistics on the system calls. The obtained data has been collected and processed by using ad-hoc Python scripts.

III. Characterization and Detection

In this section, we first characterize two containers leaking information. Then, we provide some insights on possible detection strategies.

A. Characterization of the Covert Communication

To investigate information leakage attempts targeting Docker containers, we considered an hidden data exchange of 512 bits, which models two entities "colluding" to map the underlying hardware or orchestrate an attack. Results have been obtained by averaging 5 repeated trials. In the worst case, the time needed to complete the transmission was equal to 43 minutes. Figure 2 reports the characterization in

[1]https://github.com/YehudaCorsia/Docker-Covert-channel.

[2]https://www.docker.com.

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

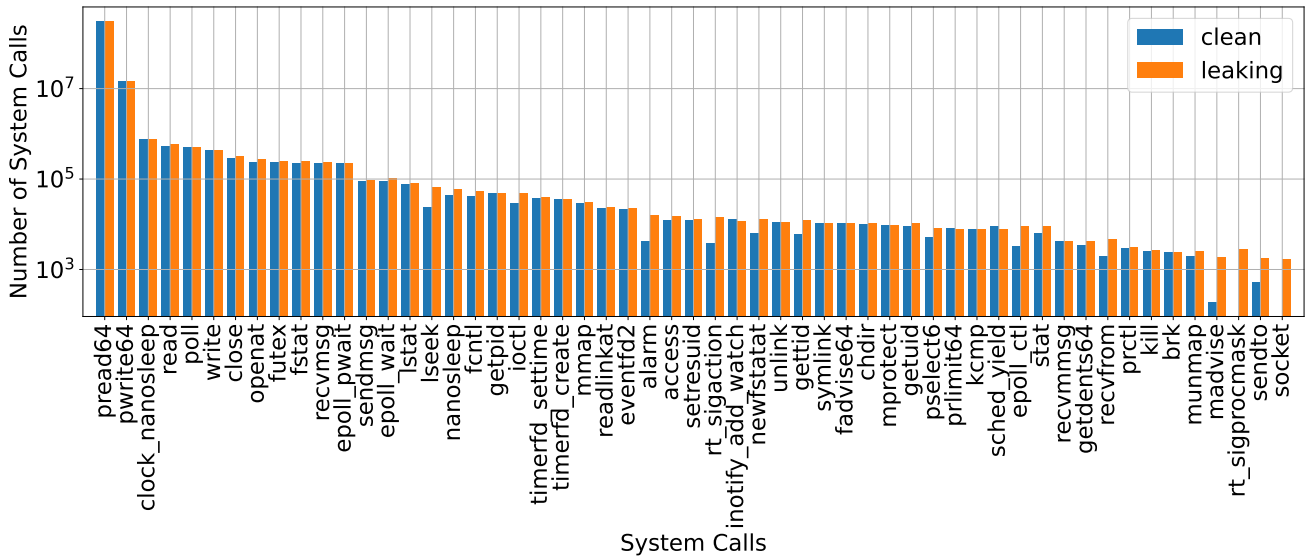


Fig. 2. Number of system calls in logarithmic scale collected host-side.

terms of the 55 most used system calls collected both when the host is only performing background operations and the containers are leaking information. In the following, we will denote the presence of a covert communication attempt as "leaking", whereas with "clean" we will denote the absence of any malicious operation. In general, the containers leaking information do not account for major alterations in the volume of system calls, except for, e.g., `lseek`, `rt_sigprocmask`, `pselect6`, and `socket`. To better understand if the system calls depicted in Figure 2 can be ascribed to the leaking activity, Figure 3 showcases the system calls only generated by the sender and the receiver processes running within Docker containers. As depicted, the covert communication only accounts for the utilization of 4 main system calls for the sender process. Specifically, `openat` is executed at the very beginning of the transmission to load the required Python libraries. The `mmap` and `munmap` are invoked to allocate/de-allocate memory to encode the `1` value. The `pselect6` is used to sleep without further modifying the memory and it allows to implement a sort of "clock" for the communication process. The figure also showcases a similar analysis for the system calls generated only by the receiving process. In this case, 6 main system calls can be identified. Similarly to the sender side, the `pselect6` is invoked again to regularly capture the impact on the memory of the operations performed by the sender, whereas the `write` is due to printing the secret to standard output. Instead, the `newfstatat`, the `read`, the `openat`, and the `close` are mainly generated to access and read the `/proc/meminfo` file to obtain information necessary to decode the secret message.

Hence, system calls like `socket` and `lseek` should be considered not strictly coupled nor related to the presence of a leaking behavior. They could be then omitted when creating

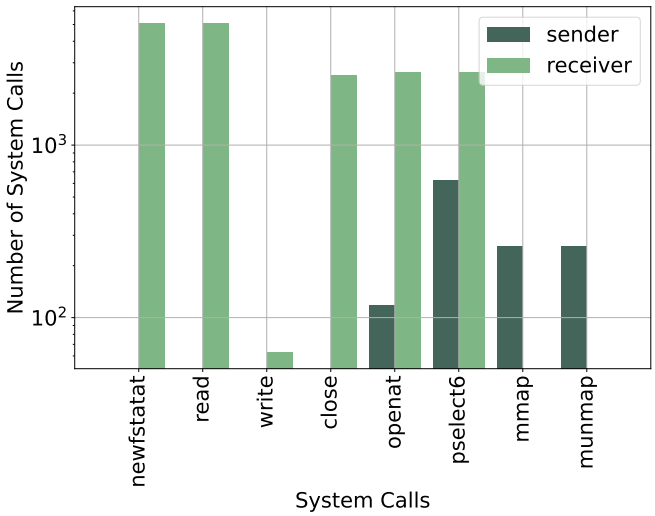
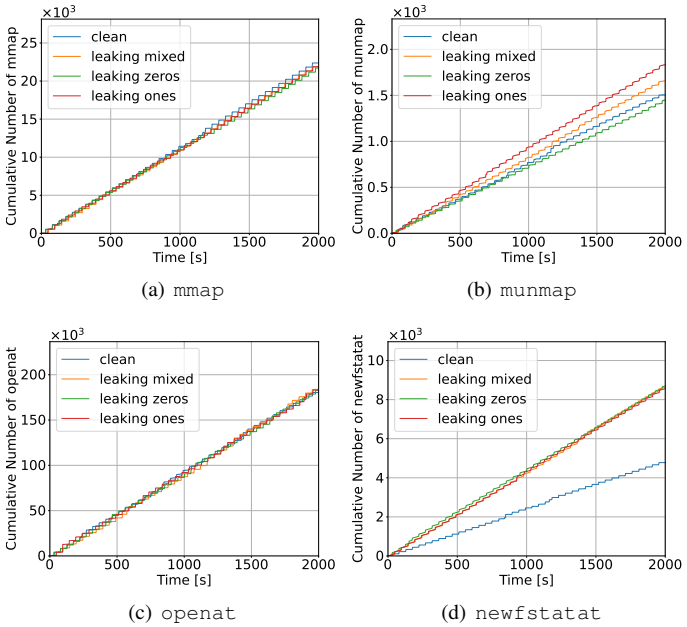


Fig. 3. Number of system calls in logarithmic scale collected for the sender and the receiver processes.

signatures. Similarly, the various `openat` invocations are mainly caused by the use of Python, thus rewriting the attack with a more effective language (e.g., C/C++) will make them not relevant for spotting collusive containers.

B. Detection: Challenges and Opportunities

To elaborate a detection strategy, a possible approach concerns the identification of suitable patterns or specific behaviors characterizing the information leakage. To this aim, we evaluated the cumulative number of the most relevant system calls used by the containers to implement the covert communication. We considered again the "clean" case and the leakage of 512 bits. To have lower and upper bounds



(a) `mmap` (b) `munmap`

(c) `openat` (d) `newfstatat`

Fig. 4. Cumulative number of the main system calls used to leak information.

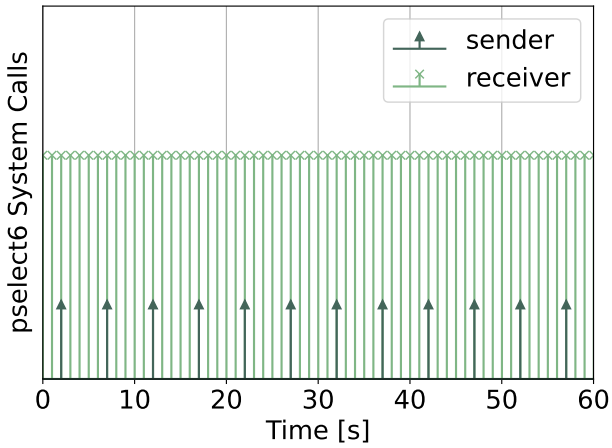


Fig. 5. Behavior of the `pselect6` for the sender and the receiver processes.

we used messages only containing either `1` or `0` digits. We also used messages composed of alternating `0` and `1` digits. In the following, we will denote such use cases as "leaking ones", "leaking zeros", and "leaking mixed", respectively. Figure 4 showcases the results. Specifically, containers trying to exchange information impact on the behavior of both the `mmap` and `munmap`, i.e., the transmission of `1` values accounts for a relevant amount of memory allocations/de-allocations (notice the different scale for Figure 4(a) and Figure 4(b)). Apart the challenges to effectively utilize this indicator to spot hidden communications, exploiting such an information entails to known in advance the resource used to encode the data. Luckily, many leaking techniques require the receiver to infer information via the `/proc` filesystem [7]. Figure 4(c) and Figure 4(d) depict values for the `openat` and the `newfstatat`. As shown, the evolution of `openat` does not lead to any useful indications as it is invoked only to open the `/proc/meminfo` file. Instead, to decode data, the receiver operates in a sort of busy-waiting/spinlock fashion, i.e., it "polls" the `/proc/meminfo` every second. Thus, the volume of `newfstatat` experiences a higher increase during a covert communication. Accordingly, the behaviour of the cumulative number of `newfstatat` is insensitive to the type of message, since it does not involve any memory allocation/de-allocation for encoding `1` or `0` values.

In this perspective, the detection of information leakages between containers should exploit some form of abstraction. Typically, covert channels leveraging a shared resource (e.g., the memory) require the sender and receiver to act close in time. This prevents that other processes or containers will disrupt the encoded values or the hidden communication path. An effective covert channel also requires a sort of synchronization between the sender and the receiver in order to minimize possible errors during the transmission. For these reasons, a possible idea is to inspect all the containers deployed in the host to spot tight time correlations [16]. To this aim, we investigated the `pselect6` system calls for both the sending and receiving processes hosted by the respective containers. Recalling that the `pselect6` is used to put a process into a sleep state, finding time correlations in the usage of such a system call may lead to the identification of the potential leaking endpoints. Figure 5 reports the temporal evolution for the `pselect6` of "colluding" containers over a time window of one minute. As shown, the sender sleeps every `5` seconds, i.e., to wait after transmitting a single bit. The receiver instead continuously exhibits a sleep/sampling pattern over the `/proc/meminfo` file to decode the message by computing the average amount of free memory and infer the received bit. As a consequence, this type of analysis could be used to implement detection mechanisms independent of the specific resource used for the hidden communication (e.g., memory, CPU load, or threads enumeration). Alas, containers legitimately interacting in a client-server manner or cooperating through a micro-service paradigm could lead to high correlations as well. To mitigate the number of false positives, suitable whitelists could be created.

IV. Mitigation Strategies

In this section, we briefly present two mitigation strategies that can be deployed against containers trying to leak data using information on memory. First, we showcase how to act on the configuration to reduce the chances of abusing a shared resource. Second, we demonstrate an approach to "disrupt" at runtime the channel.

A. Prevention via Configuration Hardening

As commonly happens for many cybersecurity threats, hardening of the configuration is the preliminary prevention policy [10]. When considering attacks leveraging information

ΔT [s]	Memory Range [MB]	Average Bit Error Rate [%]
0.1	$100-800$	7.08%
	$100-1,000$	12.92%
	$100-1,200$	18.33%
	$100-1,400$	32.50%
1	$100-800$	7.08%
	$100-1,000$	16.25%
	$100-1,200$	16.67%
	$100-1,400$	31.67%
5	$100-800$	37.92%
	$100-1,000$	39.59%
	$100-1,200$	56.25%
	$100-1,400$	54.17%
[1-5]	$100-800$	23.33%
	$100-1,000$	36.25%
	$100-1,200$	44.17%
	$100-1,400$	58.75%

TABLE I
AVERAGE BER FOR THE COVERT CHANNEL WHEN AN ARTIFICIAL NOISY PATTERN IS ADDED THROUGH A THIRD CONTAINER.

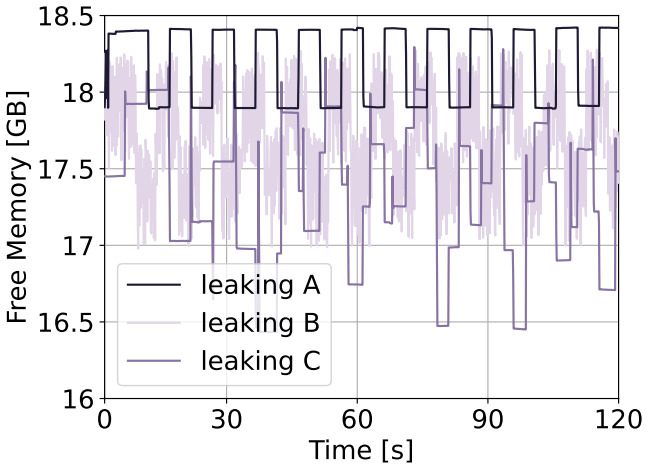


Fig. 6. Evolution of the overall free memory when containers leak information and the disruption mechanism is used for different memory ranges and ΔT.

leakages through covert communications, hardening can be implemented by controlling the use of resources. For our case, the goal is to not assign a container enough memory to encode hidden bits of information or to not achieve a satisfactory small Bit Error Rate (BER), i.e., memory variations encoding bits could not be correctly decoded by the receiving container.

To this aim, hard constraints on user-space and kernel memory can be set when configuring a container or at a later time. This is not trivial, since out-of-memory management "kills" the container whenever one of these limits is exceeded. The real challenge is then the selection of the memory thresholds, especially since memory usage may vary quite significantly under the different workload conditions characterizing many realistic deployments. Loose limits are likely to be ineffective for preventing the hidden communication since the attacker could trade both bandwidth and robustness of the channel to stay "under the radar" imposed by memory constraints [17]. On the contrary, excessively tight limits may affect the stability and availability of the whole service. In fact, aggressive limitations of the memory available for a single container would render the protection mechanism into a threat itself, i.e., a denial of service caused by unsuccessful attempts of encoding bits when hitting the memory limit. Hence, the main challenge is being able to provide suitable (run-time) policies and enforcement of constraints without impairing the performance. Yet, this is a still an open research question [18].

B. Channel Disruption

As discussed, searching for signatures at run-time or hardening configurations could be difficult, especially in Internet-scale deployments. Hence, an alternative technique exploits the perturbation of the shared resource to disrupt the hidden communication [17], [19]. For the case of information leakages targeting the memory and the loose isolation of its status, an additional process randomly allocating/de-allocating memory can be used to add a noisy signal on the values reported in the `/proc/meminfo` file. To this aim, we devised an additional container in charge of performing operations on the memory of the host with different patterns, which can be adjusted via Docker environment variables[4]. To evaluate the effectiveness of the approach, we quantified the additional BER experienced by the leaking containers when such a "sanitizer" is deployed. For this round of tests, we limited to containers leaking 24 bits of data: this allows to consider the original channel as error-free, e.g., due to synchronization issues possibly occurring in longer transmissions. Concerning the memory allocation patterns, we varied both the amount of required/released memory (denoted as "Memory Range"), as well as the time interval between two allocations (denoted as "ΔT"). Table I summarizes the outcome in terms of the average BER obtained over 10 different trials. In general, increasing the amount of memory used by the container for the allocation/de-allocation cycle will yield to a higher BER experienced by the leaking endpoints. This is due to larger fluctuations of the overall free memory that will exceed the threshold for decoding the bits, i.e., 200 MB in our setting. As an example, increasing the memory range from $100-800$ MB to $100-1,400$ MB, increments the BER from 7.08% to 32.50%, when $\Delta T = 0.1$ s. As regards the impact of temporal intervals at which memory operations happen, higher values of ΔT lead to higher BERs. Even if this could be counterintuitive, this can be explained by the synced behavior characterizing containers leaking information (see, Figure 5). Specifically, the more frequent the noisy pattern is applied to the overall memory (i.e., ΔT decreases), the higher the chances that the receiver will compute the "correct" average amount of free memory even by using wrong values. Another possible pattern could exploit some form of de-correlation, e.g., by randomly selecting an integer value for ΔT from the $[1-5]$ s range. In this case, we obtained the highest BER, i.e.,

[4]https://github.com/cybersecurity-cnr/docker-stego-protector.

58.75% for the $100 - 1,400$ MB memory range. This is due to the "spread" of several allocations across the 5 s interval used by the receiving container to compute the average, hence prevent to perform a sort of filtering of the noisy signal.

Regarding the evolution of the total free available memory, Figure 6 depicts some trends. In particular, with "leaking A" we refer to an alternating pattern of `1` and `0` digits without any additional perturbations, whereas with "leaking B" and "leaking C" we denote the cases where the "sanitizing" container alters the memory in the ranges of $100-800$ MB and $100-1,400$ MB, with ΔT equals to 0.1 s and 5 s, respectively. Coherently with results of Table I, high-frequency allocations still allow to recognize the memory evolution imposed by the encoding of secret data. Instead, slower but larger allocations turn out to be more effective, i.e., the "original" trend is mostly disrupted. However, increasing the memory used to impair the channel will also impact the amount of resources that can be used by licit containers, thus a suitable trade off should be searched for.

As a final remark, we point out that disrupting the covert channel between malicious containers often requires to know in advance the targeted shared resource. This would need a suitable set of containers that can be deployed to perturb the most abused resources, which can be identified during the design of the softwarized infrastructures, e.g., via shared-resource-matrix-like techniques [20]. At the same time, sanitization could be already implemented in the Docker engine, for instance by rate-limiting the access to the `/proc` filesystem for some entries [7] or through some form of access control.

V. Conclusions and Future Works

In this paper, we presented a prime analysis of threats based on the hidden exchange of data between Docker containers. Especially, we characterized a channel exploiting the loose isolation of memory allocation information. As shown, a subset of system calls could be considered a valuable starting point to identify the presence of containers leaking data. Unfortunately, this approach lacks of abstraction, which could be partly compensated by inspecting some periodic patterns, e.g., the `pselect6`. Other possible mitigation mechanisms exploit the elimination/limitation of the channel during the configuration phase or the deployment of some disruption mechanism, i.e., to add noise on the data encoded in the shared resource.

Unfortunately, there is not a one-size-fits-all solution against information leakages between containers, since the attack surface caused by the "imperfect" isolation is too broad (e.g., many of the entries exposed via the `/proc` filesystem can be used) and the resource encoding the secret data is not known *a priori*. Therefore, our future work aims at abstracting threats based on leakages among containers, for instance by identifying behaviors not bounded to a specific covert communication mechanism. Another part of our ongoing research deals with the use of AI to find features and correlations that can lead to early detect leakages or containers orchestrating attacks.

Acknowledgment

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

References

[1] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[2] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: making linux security frameworks available to containers," in *27th {USENIX} Security Symposium*, 2018, pp. 1423–1439.

[3] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, and M. Zuppelli, "Kernel-level tracing for detecting stegomalware and covert channels in linux environments," *Computer Networks*, vol. 191, p. 108010, 2021.

[4] N. Boskov, N. Radami, T. Tiwari, and A. Trachtenberg, "Union buster: A cross-container covert-channel exploiting union mounting," in *Cyber Security, Cryptology, and Machine Learning: 6th International Symposium, Be'er Sheva, Israel, June 30–July 1, 2022, Proceedings*. Springer, 2022, pp. 300–317.

[5] Y. Luo, W. Luo, X. Sun, Q. Shen, A. Ruan, and Z. Wu, "Whispers between the containers: High-capacity covert channel attacks in docker," in *2016 IEEE trustcom/bigdatase/ispa*. IEEE, 2016, pp. 630–637.

[6] W. Mazurczyk and L. Caviglione, "Cyber reconnaissance techniques," *Communications of the ACM*, vol. 64, no. 3, pp. 86–95, 2021.

[7] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "A study on the security implications of information leakages in container clouds," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 1, pp. 174–191, 2018.

[8] L. Miller, P. Mérindol, A. Gallais, and C. Pelsser, "Towards secure and leak-free workflows using microservice isolation," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing*. IEEE, 2021, pp. 1–5.

[9] H. Zhu and C. Gehrmann, "Lic-sec: an enhanced apparmor docker security profile generator," *Journal of Information Security and Applications*, vol. 61, p. 102924, 2021.

[10] A. R. MP, A. Kumar, S. J. Pai, and A. Gopal, "Enhancing security of docker using linux hardening techniques," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology*. IEEE, 2016, pp. 94–99.

[11] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A survey on observability of distributed edge & container-based microservices," *IEEE Access*, 2022.

[12] H. Sharaf, I. Ahmad, and T. Dimitriou, "Extended berkeley packet filter: An application perspective," *IEEE Access*, 2022.

[13] A. Carrega, L. Caviglione, M. Repetto, and M. Zuppelli, "Programmable data gathering for detecting stegomalware," in *2020 6th IEEE Conference on Network Softwarization*. IEEE, 2020, pp. 422–429.

[14] S. Magnani, F. Risso, and D. Siracusa, "A control plane enabling automated and fully adaptive network traffic monitoring with eBPF," *IEEE Access*, vol. 10, pp. 90 778–90 791, 2022.

[15] J. Nam, S. Lee, P. Porras, V. Yegneswaran, and S. Shin, "Secure inter-container communications using XDP/eBPF," *IEEE/ACM Transactions on Networking*, 2022.

[16] M. Urbanski, W. Mazurczyk, J.-F. Lalande, and L. Caviglione, "Detecting local covert channels using process activity correlation on android smartphones," *International Journal of Computer Systems Science and Engineering*, vol. 32, no. 2, pp. 71–80, 2017.

[17] W. Mazurczyk and L. Caviglione, "Steganography in modern smartphones and mitigation techniques," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 334–357, 2014.

[18] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar, "Securing docker containers from denial of service (DoS) attacks," in *2016 IEEE International Conference on Services Computing*. IEEE, 2016, pp. 856–859.

[19] S. Zander, G. Armitage, and P. Branch, "A survey of covert channels and countermeasures in computer network protocols," *IEEE Communications Surveys & Tutorials*, vol. 9, no. 3, pp. 44–57, 2007.

[20] R. A. Kemmerer, "Shared resource matrix methodology: An approach to identifying storage and timing channels," *ACM Transactions on Computer Systems*, vol. 1, no. 3, pp. 256–277, 1983.