FIREWALLING BASED ON KERNEL-ASSISTED APPLICATION IDENTITY MONITORING

Radu Mantu[1], Nicolae Țăpuș[2]

In this paper we introduce **`AppScout`**, *a kernel-based application identity monitoring solution.* **`AppScout`** *can instrument common kernel execution paths for operations such as memory mapping executable binary sections. By doing so, it can directly analyze the virtual address space of all processes and account for runtime changes. We use this system as the foundation for a firewalling solution based on application identity, implemented as an* `iptables` *plugin with negligible overhead. We note that our solution can be integrated at runtime, without requiring any kernel modification.*

Keywords: Process Identity, Network Security, Firewalling.

1. Introduction

In 2003, Gartner Research marked the transition from classic firewalls to what they coined as Next Generation Firewalls (NGFW) [1]. Prior to this, firewalls used to analyze just the network and transport layer headers. This was sufficient due to a strong correlation between port numbers and services. However, the rapid diversification of said services and web applications gave rise to a new class of more potent adversaries that the previous generation of stateful firewalls could not contend with. As a result, NGFWs were developed for the purpose of achieving application awareness through deep packet inspection.

Although current firewalling technologies need to contend with new challenges such as micro-segmentation [2] or the advancing dissolution of network demarcations, the core tenets established by Gartner back in 2003 still hold true today. However, a concerted effort dating back over ten years (e.g., Google site ranking, Let's Encrypt, etc.) rapidly lead to the widespread adoption of HTTPS in the Internet. While undoubtedly a change for the better, traffic encryption has the downside of rendering deep packet inspection ineffective.

[1]Research Assistant, Faculty of Automatic Control and Computers, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: `radu.mantu@upb.ro`

[2]Professor, Faculty of Automatic Control and Computers, National University of Science and Technology POLITEHNICA Bucharest, Romania, email: `nicolae.tapus@upb.ro`

Unfortunately, efforts towards obtaining an encrypted traffic classifier have not yet yielded a sufficiently accurate solution [3]. In lieu of better alternatives, firewall manufacturers decided to adopt SSL/TLS decryption [4], thus sacrificing user privacy in exchange for overall network security. For outgoing connections, this is achieved via a forward proxy. Each new TLS connection to an external server can be intercepted by the firewall. Next, the firewall would emit a certificate with the Distinguished Name of the external server but signed by the Certification Authority (CA) of the local organization. Since this CA is configured a priori on all internal network hosts, the TLS connection succeeds and the firewall can access the plaintext application data, then forward it to the intended endpoint via a separate TLS connection of its own. For incoming connections, the firewall is assumed to have access to the private key used by the internal server for decryption.

Although there are heuristic-based approaches such as Cisco's Encrypted Traffic Analytics (ETA) [5] or Palo Alto's App-ID [6] that rely on metrics including packet length, packet source or transmission rate, these are usually applied when encountering proprietary encryption protocols (i.e., protocols other than SSL/TLS or SSH). Furthermore, application identification on plaintext traffic is usually achieved via traffic fingerprinting. Both Cisco's Network-Based Application Recognition (NBAR) [7] and App-ID compare the payload against a database of application-specific features called signatures.

Considering these common practices in the industry today, we issue the following research questions:

RQ1: Can application awareness be achieved without sacrificing user privacy? Historically, firewalling was accomplished by placing a physical firewall on the network perimeter [8]. This decision was made for pragmatic reasons: configuring software firewalls on each individual host posed an administrative challenge. Moreover, there was no guarantee that bad actors within the network would apply the configured ruleset. Consequently, enforcing the security policies on the network egress points could be considered a sound resolution. However, due to recent BYOD [9] and remote work policies [10], these assumptions no longer hold true. We consider that adapting to these new conditions requires distributing part of the responsibility onto individual hosts [11]. This creates a unique opportunity to shift the overreliance on private user data for application identification purposes towards OS assistance.

RQ2: Is there a better method for representing application signatures? In a 2021 App-ID tech brief [12], it was revealed that the application signature database was being extended with 10-20 new entries every month. The choice of applications was based on customer feedback and industry trends. This revelation is indicative of two issues. First, a scalability issue when considering the myriad applications that can be used to generate network traffic. Second, an accuracy issue due to application versioning and polymorphic runtime behaviour. We consider whether these problems can be addressed by leveraging local OS knowledge of its running processes.

To address these questions we present `AppScout`, a Linux Kernel Module (LKM) capable of monitoring system-wide events such as memory mapping executable binary sections or modifying permissions at runtime. `AppScout` uses these sources of information to build persistent profiles for each process. These profiles contain information that can be used to identify each executable code section that was available to any process during their entire runtime. Additionally, we created an `iptables` extension that can utilize these profiles to achieve application awareness. Finally, we integrated a rudimentary Netlink subsystem into `AppScout` in order for userspace applications to query its profile database.

We claim the following contributions:

- The implementation of `AppScout`[1], an application identity monitoring LKM.
- The development of an `iptables` extension that itegrates with `AppScout`.
- An analysis of our integration effort into live systems, without necessitating a kernel recompilation.

The remainder of this paper is organized as follows. In Section 2 we describe the architecture and threat model of our proposed system. In Section 3 we provide implementation details. Section 4 illustrates the performance impact of `AppScout` on both user-grade hardware and servers with 1Gbps and 10Gbps NICs respectively. Section 5 provides a comparison between `AppScout` and similar available solutions. Section 6 concludes this paper.

2. Architecture

In this section we describe the architecture of `AppScout`. Figure 1 offers a high-level overview of the modules that comprise our firewalling solution (i.e., the primary use case) and how they interact with other systems.

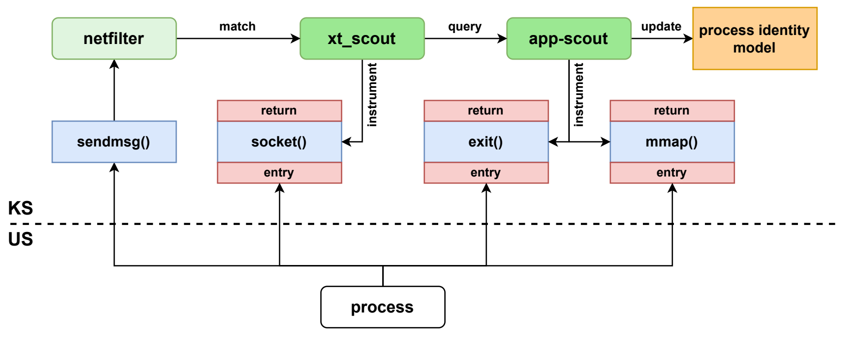

Fig. 1. `AppScout` architecture for traffic filtering.

[1]Code available at `https://github.com/RaduMantu/AppScout`

2.1. System overview

app-scout instruments certain kernel functions that are invoked during the life cycle of a process. During its initial setup phase, the dynamic linker (i.e., ld-linux) maps the necessary shared objects into virtual memory, one section at a time, with the appropriate permissions (e.g., r-x for *.text*, rw- for *.data*, etc.) These, along with any other objects that may be dynamically loaded on demand after control is ceded to the base binary, are accounted for by this module and attributed to their respective process. This information can either be relinquished after the termination of said process (i.e. on processing the exit() system call), or persist until the module is unloaded. The former alternative is preferable for memory-constrained systems, or when the identity of a process becomes unnecessary after its termination (e.g., network traffic filtering). The latter alternative is preferable when CPU resources are indispensable. Either way, at any point during the lifetime of the aforementioned process, our module will have a complete record of executable memory areas that were available to it during runtime. This information comprises the *"process identity model"*.

xt_scout is the kernel segment of an iptables extension that we developed for matching packets based on the identity of their originating process. Since app-scout is the only module with access to the process identity model, it also implements the lookup mechanism for the identity of a certain process. However, it is not immediately clear from the perspective of a Netfilter match callback what that process should be. In newer kernel versions, the socket no longer contains information regarding the process that created it. While it still retains knowledge of user and group ownership in order not to break compatibility with the owner extension in iptables, this data is insufficient for our purposes. As a result, the xt_scout module instruments the creation of new sockets by userspace tasks (through the system call socket()) and attributes them a process ownership. This relation between a socket and its owner process is then leveraged during the evaluation of an iptables rule to query the app-scout module and in turn, the *"process identity model"*. This enables the user to match network traffic to specific code objects that have at any point been accessible to the endpoint application without performing Deep Packet Inspection.

2.2. Security considerations

In order to facilitate the easy adoption of AppScout we decided against modifying the kernel source and instead implementing it as a collection of loadable modules. A consequence of this decision is that the process state prior to their insertion into the kernel cannot be accounted for. Thus, we recommend that the AppScout modules be included in the initial ramdisk of the system and be loaded prior to the init process pivoting the root filesystem.

In this scenario, we consider the initramfs to be part of the Trusted Computing Base (TCB) of the system.

In order to guarantee the authenticity and integrity of the modules, we suggest employing the module signing facility of the kernel (based on X.509 ITU-T standard certificates). In lieu of this option, the entire initial ramdisk should be verified by its bootloader. E.g., on ARM systems that adhere to the ARM Trusted Firmware Design and use U-Boot as BL33, the Firmware μImage Package that contains the kernel and ramdisk can be signed by the developer and verified at boot time. The public key used in the signature verification can either be hardcoded in the U-Boot binary or loaded from its Flattened Device Tree, both being themselves verified by the Original Equipment Manufacturer (OEM) bootloader.

In our threat model, we consider the attacker capable of gaining access to the system but require that the `AppScout` modules had already been loaded at that time. If the attacker is considered to be able of escalating his privilege level to that of *root*, we require that the kernel be put in lockdown via *securityfs* immediately after loading the `AppScout` modules. Otherwise, the attacker can either reload the `app-scout` module or interfere with it via a module of his own, with the purpose of obfuscating his intrusion attempt during auditing or staging further attacks.

3. **Implementation**

3.1. **An argument for instrumentation**

It was clear from the outset that in order to build a system-wide, long-term process identity model, we needed to collect certain information at runtime. This information is not always readily accessible (i.e., stored in kernel structures, but instead transient - in the local context of a call frame) and must be obtained and indexed before the process state change that it represents propagates to userspace. For example, if a process maps a file in its virtual address space, our solution must intercept this operation and analyze the contents of the mapped file before said process has a chance to acknowledge that the operation finalized successfully and potentially modify the memory, or unmap it.

There are two approaches to this problem: either manually adding hooks in certain key functions, or instrumentation. While the former is a valid approach (e.g., the integration method of the Netfilter system into the Linux kernel), unless merged into upstream it will deter adoption of this system, primarily by teams that develop their own Linux fork; a fairly common practice in the embedded industry. Consequently, an instrumentation-oriented approach would be preferable.

Initially, kprobes were computationally expensive due to the use of software interrupts to trap the execution of certain kernel code regions. Incidentally, this approach is also used by debuggers via the `ptrace()` system call.

However, debuggers are required to employ it due to their inferiors (i.e., programs under test) being separate processes. In 2010, a patchset by the author of Djprobe [13] implemented the jump optimization for kprobes, reducing their overhead to almost negligible margins. At the time of writing, this optimization is available on x86, ARM and PowerPC. Taking into consideration the fact that over ten years later, improvements are still being researched [14], we surmised that kprobes are a viable choice for runtime data collection via instrumentation.

3.2. Construction of process identity

The goal of the `app-scout` module is to gradually create a process identity model based on data obtained from instrumentation. This model is organized as an associative array using process IDs as keys for easy retrieval. Ideally, the process identity information would have been stored in its respective `task_struct` instance. However, this structure has very few fields that could be repurposed and doing so could very well interfere with other mechanisms. E.g., the `security` field would be a prime candidate if not for the fact that it is reserved for LSMs such as Tomoyo. Moreover, the `task_struct` instance is released after the termination of the process, meaning that the reference to the process identity information would have been unavailable for tha analysis of delayed packets.

The data associated to each PID in our model consists primarily of a list of SHA256 digests computed based on the contents of each executable memory-mapped region. Writable data sections have been excluded based on their volatile nature. While an argument could be made in favor of including *.rodata* sections in our model, we consider that the *.text* sections are sufficient for demonstrating the capabilities of this prototype. Furthermore, we acknowledge the possibility of encountering writable, non-file-backed executable sections in processes that implement JIT compilers or translators. Efficiently labeling a continuously mutating memory region at any given time is in and of itself a non-trivial task, let alone tracking its change history during the runtime of the process. Other solutions such as `Dymo` [15] reason that code generated by a trusted binary should itself be trusted. Based on these considerations, we decided to take a similar stance and deem the identification of JIT-ed code outside the scope of this paper.

In order to retrieve the aforementioned data, we instrumented the `vm_mmap_pgoff()` function as part of the synchronous and asynchronous `mmap()` common path. In the pre-call probe, we exclude kernel threads from our analysis and ensure that the region is supposed to be mapped with execute permissions. In the post-call probe, we determine the virtual address where a known number of pages have been mapped. Having obtained this address does not necessarily mean that the actual data resides in memory. Consequently, we pin those pages in memory, forcing the kernel to fault them into RAM if any are missing. Herein lies a problem: depending on the underlying filesystem driver or backing storage device driver, this operation can force the current task to yield its remaining CPU time. We have first observed this behavior in our development environment, using the Plan 9 filesystem protocol. Because the kprobe instrumentation callbacks are invoked in atomic context, we were required to re-enable preemption for the duration of this operation. We reasoned that as opposed to classical kprobes (i.e., those that are based on software interrupts), the jump-optimized version that we use should be able to safely execute even outside an atomic context. As a precautionary measure, we arbitrarily increased the maximum number of concurrently active kprobe instances for this function. At the time of writing, we are still debating reimplementing this functionality as deferred work to preempt any unforeseen bugs. After all pages are guaranteed to be present in main memory, we calculate the SHA256 sum and store it alongside the name of the backing file in our data structure. Finally, the memory pin is released in order to prevent an unsustainable growth of the Resident Set Size.

3.3. Xtables integration

The `iptables` extension consists of two components. One is the shared object `libxt_scout.so` that implements the necessary functionality for the userspace tool (e.g., argument parsing, rule printing, etc.) `iptables` loads this plugin on demand from certain well known paths (e.g., `/usr/lib/xtables`) or from the paths stored in the `XTABLES_LIBDIR` environment variable.

The second component is the `xt_scout` kernel module. This module implements the `checkentry()` and `match()` callbacks for the Xtables framework. The former performs sanity checks on newly inserted rules. The latter is invoked as part of a potentially larger rule evaluation on a specific packet or rather, a `sk_buff` object. In Section 2 we mentioned that this socket buffer instance can be linked back to the socket that it belongs to. However, that socket is not itself linked to any given process in newer kernel versions. Nonetheless, there is a mechanism in place for this.

A process can be assigned as the owner of socket in order for the kernel to send signals under certain conditions. One of the more relevant examples is being notified when urgent data is available as part of a TCP stream. Nonetheless, even Urgent Pointers are largely unsupported in modern applications, many TCP/IP network stacks preferring to handle urgent data in the same queue as normal data instead of implementing a fast path. As a result, the `xt_scout` module instruments the `soc_alloc_file()` function. This function is invoked after the successful instantiation of a socket in order to allocate a `file` structure and bind the socket to a file descriptor. In our instrumentation callback, we initialize the ownership field with the PID of the current process (i.e., the one that created the socket). Thus, we create a persistent link between a socket object and a process entry in our `app-scout` model.

The benefit of integrating `app-scout` with the `Xtables` framework is that all other `iptables` extensions are readily available to the user. For example, Listing 1 illustrates a rule that makes use of the `conntrack` module to block the creation of a new connection based on application identity. Without it, the match callback function in `xt_scout` would be invoked for every single outgoing packet, even though the verdict is predictable and should be memoized. Note however that the order in which the modules are loaded is more important than the order in which the match criteria is specified. The only reason why the `scout` module is not invoked on subsequent packets is that the `ctstate` check failed beforehand.

```
$ iptables                                                  \
    -m conntrack -m scout   `# load extension modules   `  \
    -I OUTPUT               `# insert on OUTPUT chain   `  \
    -j DROP                 `# DROP verdict on match    `  \
    --digest 9986...7ae7    `# SHA256 digest (abridged)`  \
    --ctstate NEW           `# new connection            `
```

Listing 1. iptables rule with scout and conntrack modules

4. Evaluation

4.1. Experimental setup

The experiments described in this section have been carried out on the following hardware:

- **Intel NUC:** Intel Core i7-7567U CPU @3.50GHz, Intel I219-V 1-Gigabit Ethernet Controller, 8GB DDR4 memory @2400MT/s.
- **IBM System x3550 M4:** Intel Xeon E5-2650Lv2 CPU @1.70GHz, Intel 82599ES 10-Gigabit SFP+ Ethernet Controller, 32GB DDR3 memory @1600MT/s.

The reported throughput was derived from the `iperf3` output and was based on the application data transfer, excluding any headers. Both systems are running a minimal installation of Arch Linux with kernel version 6.11.0.

4.2. Throughput impact

Our throughput evaluation of `AppScout` consists of `iperf3` tests performed between directly linked systems. For each testing campaign we have varied the Maximum Transmission Unit (MTU) between 100 and the maximum value supported by the NIC. For each `iperf3` experiment, we have used the reported average data transfer rate over a TCP connection lasting ten seconds. The reported values do not include the header lengths of the underlying network layers.

Figure 2 consists of three campaigns of tests, illustrating the impact of adding one `iptables` rule consisting of strictly one `xt_scout` match function invocation. The match argument was selected so that each query to `app-scout`

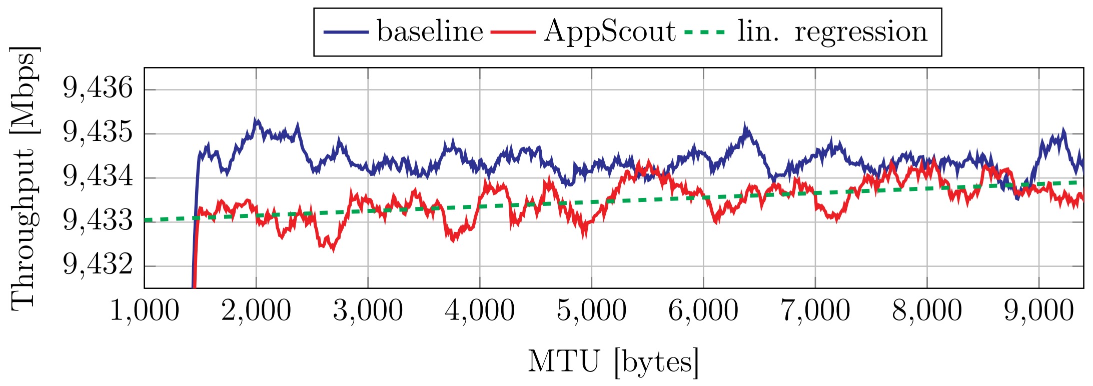

FIG. 2. `AppScout` throughout on IBM system.

would parse the executable sections record in its entirety, thus simulating a worse case scenario. For `iperf3`, this record consists of 13 entries. Because the average throughput difference between the baseline and the filtered case is exceedingly low (0.15 Mbps) within the MTU range 1500-9710, we performed a moving average with a window of 30 samples, equating to 300 bytes. This was necessary in order to demonstrate that the low overhead was not coincidental. Moreover, we observe a tendency of the `AppScout` throughput to grow and converge towards the baseline for increasingly higher MTU values (see the linear regression of the raw data in the 1500-9710 MTU range). This phenomenon can be motivated by a decrease in the number of Packets Per Second (PPS). For example, a six-fold increase over the default MTU size of 1500 translates results in a similar decrease in the number of packets necessary to saturate the network controller. Because the process identity inference cost does not depend on the packet size, it stands to reason that the `AppScout` overhead should also decrease sixfold.

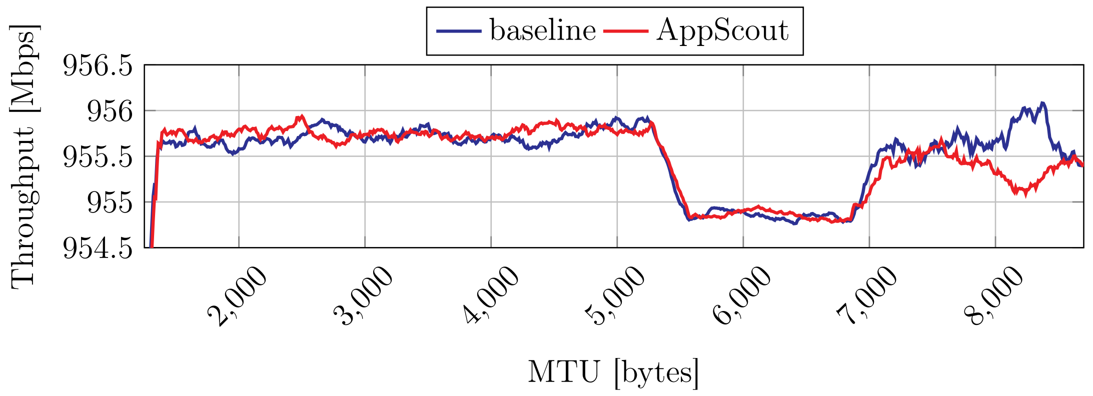

FIG. 3. `AppScout` throughout on Intel NUC.

Similarly, Figure 3 illustrates the same experiment performed on the Intel NUC system, with a 1Gbps NIC. Although less noisy due to the lower throughput, the same moving average window was used for consistency. Despite a number of anomalies that can for the most part be attributed to the `e1000e` driver, we notice that the reduced number of Packets Per Second (PPS) in relation to the core count as well as the increase base CPU frequency contribute to further reduce the overhead of `xt_scout`.

4.3. System tuning based on processor resources

During our experiments, we encountered an issue where `AppScout` caused the network device watchdog timer to trigger, leading to cascading resets of the NIC. Normally, this problem is caused by faulty hardware that is unable to service the transmission queue in a reasonably timely manner. However, this bug can also be triggered on Intel network controllers by setting a high value for the `InterruptThrottleRate` driver argument. This command line kernel argument is used to limit the number of interrupts generated by the NIC in situations where most of the CPU time is dedicated to network traffic processing. In our case, the process identity lookup operation introduced a delay that had the same effect as the throttle mechanism. We identified three solutions to this problem:

(1) **Adjust the frequency governor:** When we first triggered this bug, the effective CPU frequency was set to approx. 1.1GHz. By setting the scaling governor to *performance* mode, we increased it to a mostly constant 2.1GHz, thus compensating for the identity lookup delay.
(2) **Disable garbage collection:** Avoiding scheduling more delayed work in the bottom half is a good method of freeing up clock cycles. While still using the default *userspace* scaling governor, we noticed an approx. 80% decrease in watchdog timer triggers. While this was not a permanent solution in our testing environment, it can have a greater impact on servers with high system load and many short-lived processes.
(3) **Employ conntrack:** The `conntrack` module can be used to improve the efficiency of `iptables` by reducing the number of match callback invocations based on the state of each connection. This type of optimization is widely used in both hardware and software firewalls and can have a greater impact than any of the previous solutions.

4.4. Identity auditing using the Netlink protocol

Traditionally, arbitrary communication between a userspace process and a specific kernel module was done via the Virtual File System (VFS) layer. Lately, this functionality has been partially duplicated via Netlink sockets, with specific modules exposing different subsystems as protocols in the `AF_NETLINK` domain. For example, `netstat` is superseded by `ss` which uses the Netlink Socket Diagnostics subsystem. Similarly, `nftables` uses the Netlink Netfilter subsystem which was backported to the `iptables` backend to maintain compatibility. As a result, we decided to implement support for a new Netlink subsystem in the `app-scout` module. We offer `probe_appscout` as a userspace application capable of querying the kernel module for information regarding a certain process, based on its PID.

5. **Related Work**

`VMWall` [16] is a Xen-based application firewall first introduced in 2008. In their architecture, dom0 is the driver domain (i.e., the VM that controls the actual hardware) and all other VMs access the networks through dom0 via a Virtual Network Interface (VNI). Their firewall comes in the form of a kernel module that attaches itself to the network bridge between the VNI backend and the NIC driver in dom0. This module performs Virtual Machine Introspection (VMI) to access the kernel memory of other VMs, iterate over the list of active processes, and find the one that has access to the socket that is related to a specific flow. The process information that it retrieves for use in filtration rules consists of PID, process name and the full path of the base executable. Although `VMWall` ensures better isolation to its kernel module by means of hardware virtualization extensions, `AppScout` provides more fine-grained identity matching criteria with significantly lower overhead (approx. 1.065E-5 vs 1-7%).

Released in 2011, `Dymo` [15] is a Windows XP kernel module that implements a dynamic code identity primitive. By subscribing to the NT kernel notification system for memory allocation, file mapping and memory permission changes, `Dymo` identifies all memory regions that *could* be executed. By leveraging this information in an instrumented Page Fault handler, it tracks all pages that contain code that was *actually* executed and labels them using cryptographic hashes of their content. While similar in that it also computes cryptographic hashes over virtual memory ranges, `AppScout` faults in the memory mapped executable section in order to reduce variability due to runtime execution paths. Another important distinction between the two solutions stems from their intended applications. E.g., `AppScout` implements long-term retention of the process identity in order to contribute to audits following security incidents.

6. **Conclusion**

In this paper we explored a method of attaining application awareness without sacrificing user privacy (**RQ1**). To this end, we have implemented a kernel module that collects SHA256 message digests of all executable sections that were at any time accessible to any given process during its lifetime. This approach allows a user to match network traffic based not on the content of each packet (i.e. DPI) but instead on the relationship between its endpoint socket and the process that created it. Consequently, traffic can be either blocked or allowed based on whether the endpoint application was "tainted" by a specific binary (identified by its SHA256) or library version. The identity of the application is established without requiring knowledge of the protocols that it implements (**RQ2**), as is usually the case in contemporary solutions based on protocol dissectors. To illustrate this functionality, we have implemented an `iptables` extension that presents negligible overhead (i.e., less than 2Mbps slowdown on a 10Gbps link) in both desktop and server-grade environments.

REFERENCES

[1] *Stiennon R.*, "Four paths to true network security," Commentary COM-20-0571, Gartner Research, Tech. Rep., 2003.

[2] *Sheikh, Nabeel and Pawar, Mayur and Lawrence, Victor*, "Zero trust using network micro segmentation," in IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2021, pp. 1–6.

[3] *Husák, Martin and Čermák, Milan and Jirsík, Tomáš and Čeleda, Pavel*, "Https traffic analysis and client identification using passive ssl/tls fingerprinting," EURASIP Journal on Information Security, vol. 2016, pp. 1–14, 2016.

[4] *Radivilova, Tamara and Kirichenko, Lyudmyla and Ageyev, Dmytro and Tawalbeh, Maxim and Bulakh, Vitalii*, "Decrypting ssl/tls traffic for hidden threats detection," in 2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT). IEEE, 2018, pp. 143–146.

[5] *Manning, Derek and Li, Peilong and Wu, Xiaoban and Luo, Yan and Zhang, Tong and Li, Weigang*, "Aceta: Accelerating encrypted traffic analytics on network edge," in ICC 2020-2020 IEEE International Conference on Communications (ICC). IEEE, 2020, pp. 1–6.

[6] *Malmgren, Andreas and Persson, Simon*, "A comparative study of palo alto networks and juniper networks next-generation firewalls for a small enterprise network," 2016.

[7] *Headquarters, Americas*, "Classifying network traffic using nbar," cit. on, p. 13, 2006.

[8] *Keromytis, Angelos D and Prevelakis, Vassilis*, "Designing firewalls: A survey," Network Security: Current Status and Future Directions, pp. 33–49, 2007.

[9] *Morrow, Bill*, "Byod security challenges: control and protect your most sensitive data," Network Security, vol. 2012, no. 12, pp. 5–8, 2012.

[10] *Malecki, Florian*, "Overcoming the security risks of remote working," Computer fraud & security, vol. 2020, no. 7, pp. 10–12, 2020.

[11] *Ioannidis, Sotiris and Keromytis, Angelos D and Bellovin, Steve M and Smith, Jonathan M*, "Implementing a distributed firewall," in Proceedings of the 7th ACM conference on Computer and communications security, 2000, pp. 190–199.

[12] "App-id tech brief," Palo Alto Networks, Tech. Rep., 2021.

[13] *Hiramatsu, Masami and Oshima, Satoshi*, "Djprobe—kernel probing with the smallest overhead," in Linux Symposium, 2007, p. 189.

[14] *Jia, Jinghao and Le, Michael V and Ahmed, Salman and Williams, Dan and Jamjoom, Hani and Xu, Tianyin*, "Fast (trapless) kernel probes everywhere," in 2024 USENIX Annual Technical Conference (USENIX ATC 24), 2024, pp. 379–386.

[15] *Gilbert, Bob and Kemmerer, Richard and Kruegel, Christopher and Vigna, Giovanni*, "Dymo: Tracking dynamic code identity," in Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings 14. Springer, 2011, pp. 21–40.

[16] *Srivastava, Abhinav and Giffin, Jonathon*, "Tamper-resistant, application-aware blocking of malicious network connections," in Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings 11. Springer, 2008, pp. 39–58.