Phoenix: Surviving Unpatched Vulnerabilities via Accurate and Efficient Filtering of Syscall Sequences

Hugo Kermabon-Bobinnec*✉, Yosr Jarraya†, Lingyu Wang*✉, Suryadipta Majumdar* and Makan Pourzandi†
*CIISE, Concordia University, {hugo.kermabonbobinnec, lingyu.wang, suryadipta.majumdar}@concordia.ca
†Ericsson Security Research, Ericsson Canada, {yosr.jarraya, makan.pourzandi}@ericsson.com

***Abstract*—Known, but unpatched vulnerabilities represent one of the most concerning threats for businesses today. The average time-to-patch of zero-day vulnerabilities remains around 100 days in recent years. The lack of means to mitigate an unpatched vulnerability may force businesses to temporarily shut down their services, which can lead to significant financial loss. Existing solutions for filtering system calls unused by a container can effectively reduce the general attack surface, but cannot prevent a specific vulnerability that shares the same system calls with the container. On the other hand, existing provenance analysis solutions can help identify a sequence of system calls behind the vulnerability, although they do not provide a direct solution for filtering such a sequence. To bridge such a research gap, we propose *Phoenix*, a solution for preventing exploits of unpatched vulnerabilities by accurately and efficiently filtering sequences of system calls identified through provenance analysis. To achieve this, Phoenix cleverly combines the efficiency of Seccomp filters with the accuracy of Ptrace-based deep argument inspection, and it provides the novel capability of filtering system call sequences through a dynamic Seccomp design. Our implementation and experiments show that Phoenix can effectively mitigate real-world vulnerabilities which evade existing solutions, while introducing negligible delay (less than 4%) and less overhead (e.g., 98% less CPU consumption than existing solution).**

I. Introduction

Unpatched vulnerabilities remain an important security threat to today's businesses. According to recent studies, the average time between the discovery of a vulnerability and the release of its patch sits at around 100 days in recent years [65], and an additional 422 days on average would pass before the vendors eventually patch their vulnerable images [44]. Such delays and their impact are evident in high-profile security incidents, e.g., failure to patch CVE-2017-5638 led to a massive Equifax data breach more than 72 days after its disclosure [16], [80], and delay in patching the Log4Shell [45] vulnerabilities reportedly led to a worldwide crisis (e.g., it forced the Canadian government to shut down nearly 4,000 services [31]). Those incidents also demonstrate a dilemma faced by the businesses, i.e., either to keep the vulnerable services running, which would expose the data and underlying infrastructure to irrecoverable damages (e.g., data breach in the case of Equifax), or to temporarily shut down the services till the vulnerability is patched, which could mean significant social or financial impact (e.g., service disruption in the case of the Canadian government). This can get worse as more businesses move to container-based cloud services [9], since container images are known to be buggy with vulnerabilities [76], and the weaker isolation of containers may not only render container-based services an appealing target but also allow adversaries to escape a compromised container to cause more severe damages to the underlying cloud infrastructure [3], [53], [56].

One promising approach to address this is system call-level filtering. For instance, Seccomp-based approaches can block a set of system calls that are not normally used by the application, which can be determined through either a static (e.g., Chestnut [7], Sysfilter [11], Confine [24], and C2C [26]), dynamic (e.g., DockerSlim [12]), or temporal (e.g., Ghavamnia et al. [25] and SPEAKER [40]) analysis. Although blocking unused system calls can effectively reduce the general attack surface of a container, it cannot tackle a specific vulnerability that shares similar system calls with the container, as we will demonstrate through experiments in Section V-A. On the other hand, there exist many attack detection (e.g., Falco [17]) and analysis (e.g., CLARION [8]) techniques, and particularly the provenance graph-based attack investigation solutions (e.g., ATLAS [1], DepImpact [20], Unicorn [29], NoDoze [30], and ProvDetector [81]) can help a security analyst to capture the malicious behavior of exploiting an unpatched vulnerability, e.g., in terms of a sequence of system calls. However, such solutions do not provide a direct solution for blocking the identified sequence of system calls, which raises some novel challenges, as demonstrated in our motivating example.

Motivating Example. Figure 1 highlights the limitations of existing system call filtering mechanisms (e.g., Seccomp and Ptrace) and provides a hint of our solution. We assume the following context, i.e., using provenance analysis, a crowdsourcing campaign among affected users has indicated that we can survive an unpatched vulnerability by blocking a sequence of two system calls, `open`→`pipe(*arg1)`.

Seccomp. The left side shows two variations of applying Seccomp filters [78] (illustrated as a border officer inspecting travelers' passports). First, only blocking system calls unused by the container (e.g., [11], [24]) could allow the targeted `open` and `pipe` system calls to pass if those are also used by the container. Hence, the container stays insecure to the vulnerability. Second, Seccomp filters are inaccurate in the sense that they do not inspect the arguments in depth (e.g., `pipe(*arg2)` is a false positive, as we target `*arg1`), and they do not keep track of the order (e.g., they cannot distinguish between `open`→`pipe` and `pipe`→`open`).

✉ Corresponding authors

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24582
www.ndss-symposium.org

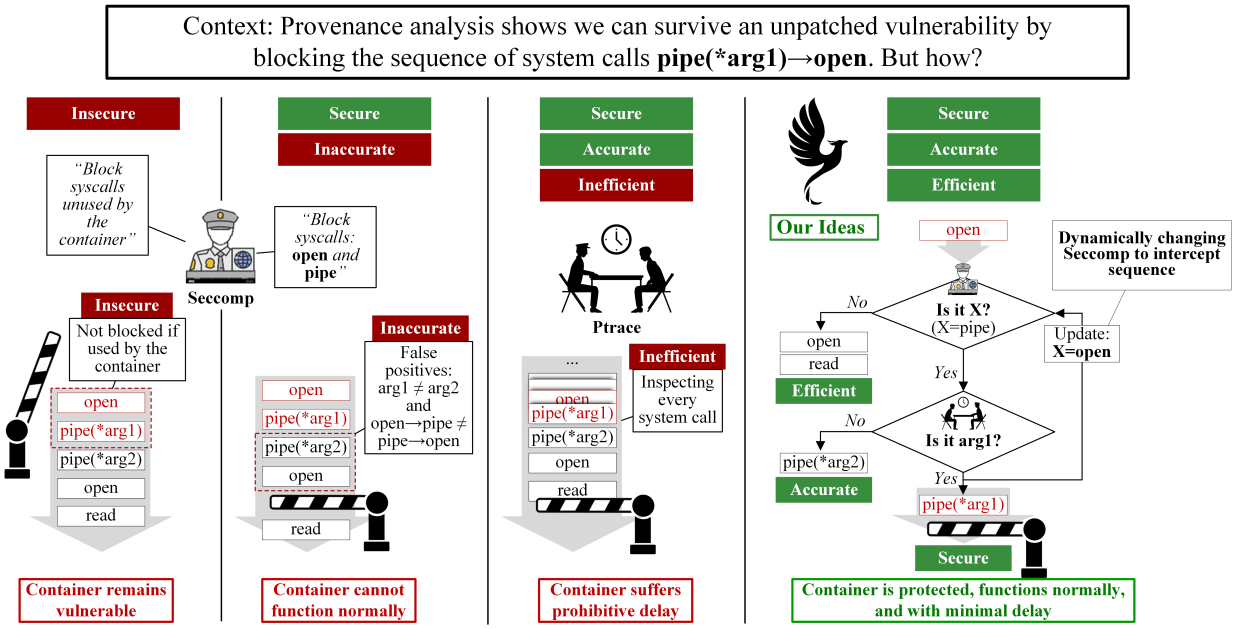


Fig. 1: A motivating example showing the limitations of existing solutions (left and middle) and our key ideas (right)

Therefore, blocking targeted system calls using Seccomp may lead to false positives that cause the container to malfunction.

Ptrace. In the middle, we show the limitation of using Ptrace [64] (illustrated as a border officer interviewing a traveler), which is a process tracer for monitoring and controlling sequences of system calls. Although Ptrace can accurately block the targeted sequence by inspecting each and every system call and its arguments, doing so could result in prohibitive delay to the container, especially considering the sheer amount of system calls (e.g., an Nginx web server may issue more than 1,800 system calls per second [71]).

Our Ideas. The right side shows our key ideas. Intuitively, we leverage Seccomp for a fast pre-screening, and only trigger Ptrace for an in-depth inspection when there is a match (illustrated as a traveler being interviewed only when his/her passport matches a given list). Therefore, most system calls (e.g., `open` and `read` in the figure) will be immediately allowed by Seccomp, which ensures *efficiency*. A matching system call (e.g., `pipe`) is further inspected by Ptrace, and either allowed if its argument mismatches (e.g., `pipe(*arg2)`), which ensures *accuracy*, or intercepted if its argument matches (e.g., `pipe(*arg1)`), which ensures *security*. Finally, we dynamically change our Seccomp filter upon an interception. For instance, once `pipe(*arg1)` is intercepted, we replace the Seccomp filter to look for the next system call in the target sequence (i.e., `open`), which provides the new capability of *intercepting sequences of system calls*.

To apply those ideas, we propose Phoenix, a solution for protecting containers against unpatched vulnerabilities. Specifically, Phoenix collects and analyzes provenance data from the victim container to identify the sequence of system calls behind the unpatched vulnerability. It then restarts the container in a hardened state[1] in which the identified sequence of system calls will be efficiently and accurately blocked, as described above. Such an approach has two potential benefits. First, as Phoenix does not require any analysis or understanding of the source code bugs causing the vulnerability, any affected user might have a chance to identify the required sequence of system calls. This makes it possible to crowdsource such "temporary patches", which could significantly reduce the waiting time compared to relying on a single vendor for the official patch. Second, since Phoenix works at a low (system call) level, it can potentially provide a "Swiss army knife" for users to survive many different attacks using a single solution. In summary, our main contributions are as follows:

- We propose a runtime protection mechanism through integrating Seccomp with Ptrace, and changing the Seccomp filters on the fly. This enables the stateful interception of a target system call sequence while benefiting from both the efficiency of Seccomp and the accuracy of Ptrace.
- This new capability allows us to bridge the gap between provenance analysis and runtime protection. Specifically, Phoenix collects and analyzes provenance data from a victim container to derive a sequence of system calls as the root cause of an incident. At runtime, Phoenix then hardens the vulnerable container against similar incidents by blocking that sequence.
- We implement and evaluate Phoenix based on real-world attacks and datasets. Our results show Phoenix can effectively mitigate vulnerabilities that are not consistently prevented by existing solutions, while introducing negligible delay (e.g., less than 4% on average) and overhead (e.g., 98% less CPU than Ptrace) to the container.

II. Preliminaries

This section provides background and our threat model.

A. Background

Containerization. Containerization offers operating system (OS)-level virtualization for running various software applications in isolated user spaces, namely, *containers*. Containers offer several unique benefits including their transient nature (i.e., fast and easy to restart) and observability (i.e., allowing to monitor system calls using Seccomp for

[1]Hence the name of Phoenix, which in mythology rises from ashes stronger.

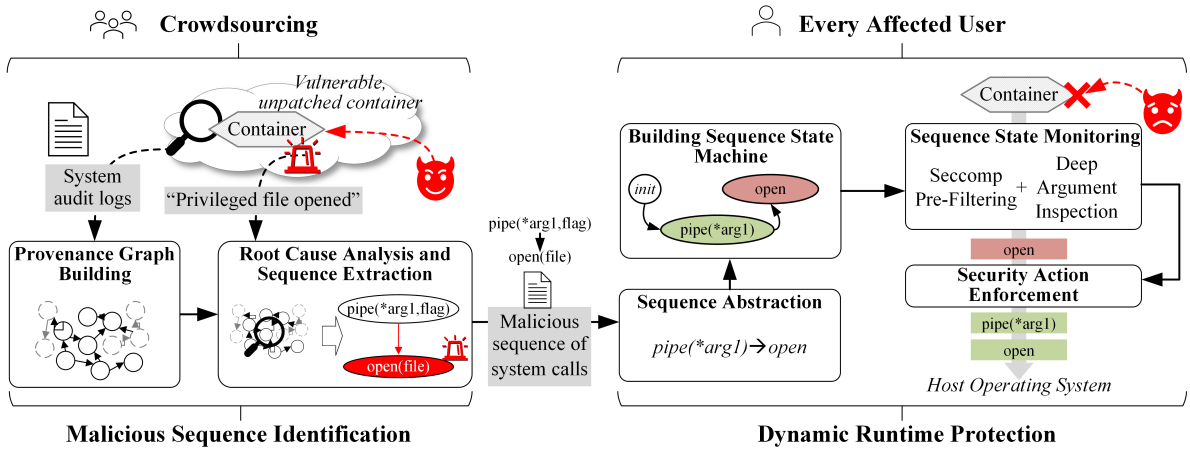


Fig. 2: Overview of Phoenix methodology

individual containers). However, those benefits may come at a cost, i.e., containerization offers relatively weaker isolation than full-fledged virtual machines, which enables kernel exploits to break the container isolation and compromise the host (e.g., CVE-2022-0847, CVE-2022-0185, and CVE-2022-0492). In this work, we leverage the transient nature and observability to restart and monitor containers, while hardening them against vulnerability exploitation (as detailed in Sec. III).

System Calls and Seccomp Filters. Like any user-space application, containers interact with kernel functionalities through *system calls*. Even though applications usually do not need all available system calls to function, these are available to the applications by default. Therefore, an attacker could exploit a vulnerable application and employ normally unused system calls to perform malicious operations (e.g., privilege escalation in the container and escaping the container isolation). To prevent such misuse of system calls, Seccomp (SECure COMPuting) [66] is a security mechanism of the Linux kernel for restricting the access of user-space programs to certain system calls. Particularly, Seccomp-BPF filters [78] (simply called Seccomp filters in this work) can be configured to perform an action (e.g., allow, deny, or send a signal [50]) upon matching system calls. Such behavior is defined using Berkeley Packet Filter (BPF) programs running in the kernel. Although Seccomp filters provide a popular protection mechanism for containers due to their negligible overhead [67], their limitations lie in (i) their stateless nature (i.e., Seccomp filters cannot "remember" previously seen system calls); and (ii) their inability to perform deep argument inspection (i.e., de-reference system call arguments that are pointers or structures). In this work, we address those limitations through integrating Seccomp with Ptrace, and dynamically changing Seccomp filters.

Provenance Analysis. Recent works (e.g., [4], [5], [20], [29], [30], [41], [54], [55], [81]) apply data provenance techniques to investigate the root cause of security incidents. In this work, we leverage provenance graphs to identify sequences of system calls used for exploiting unpatched vulnerabilities. Typically depicted as directed acyclic graphs (DAG), provenance graphs represent the flow of information between different subjects (e.g., processes) and objects (e.g., files, network sockets, and pipes) of a system during its execution, rather than their temporal dependencies. Therefore, working with provenance data provides richer context and often makes it easier to figure out what events are causally related, even though their temporal relationship is not clear [29]. Such causal relationships between events in an operating system can enable a root cause analysis to trace sequences of system calls that are issued while exploiting vulnerabilities. In our work, we leverage such a capability while providing a solution for blocking the identified sequences at runtime.

B. Threat Model

Like many other container security solutions (e.g., [21], [27], [47]), our in-scope threats include attacks that can be reflected in the sequences of system calls and/or their arguments. Unlike some existing solutions (e.g., [7], [11], [12], [24]–[26], [40]), we additionally consider vulnerabilities that use similar (or a subset of) system calls required by the normal behavior of the container. We focus on unpatched vulnerabilities that can be detected (e.g., using rule-based solutions, such as Falco, Nagios, and Snort, or using learning-based approaches, such as [29], [81]). We assume the integrity of Phoenix and that of the underlying infrastructure and its audit logs. Conversely, unknown or undetected zero-day attacks, attacks that do not involve system calls issued by the container to the host OS (note vulnerabilities lying in the userland may involve such system calls, either purposefully like `ret2syscall` and `sigreturn-ROP`, or as a side-effect of control flow manipulation attacks), and attacks that can tamper with Phoenix or the infrastructure are out-of-scope for this work.

III. Methodology

This section first provides an overview of Phoenix and then details each step of its methodology.

Overview. As shown in Fig. 2, Phoenix works in two major steps, namely, *malicious sequence identification* and *dynamic runtime protection*. First, the users affected by an unpatched vulnerability come together through crowdsourcing to identify the system call sequence behind the vulnerability. To facilitate this, Phoenix builds a provenance graph from system audit logs collected from a victim container and then assists users to perform root cause analysis to extract the sequence of system calls leading to the incident reported by a detection mechanism such as Falco (detailed in Section III-B). Second, the malicious sequence of system calls is shared with all the affected users to protect their containers. For this purpose, Phoenix first abstracts the sequence and builds a

corresponding sequence state machine. It then continuously monitors incoming system calls through Seccomp pre-filtering and Ptrace deep argument inspection, and enforces user-specified security actions to block any matching sequence (as detailed in Section III-A). As the dynamic runtime protection step is slightly more complex, this step will be detailed first.

A. Dynamic Runtime Protection

Figure 3 details our methodology for the *dynamic runtime protection* step. First, upon receiving as input a malicious sequence of system calls, Phoenix abstracts incident-specific parameters of system calls to construct a generic version of the sequence. Second, it builds a state machine to facilitate matching against that sequence. Third, at runtime, Phoenix first restarts the container, then monitors the incoming sequence of system calls and their parameters with Seccomp and Ptrace. Fourth, when a sequence is matched, Phoenix enforces user-specified security actions accordingly to protect the container. We detail those sub-steps in the following.

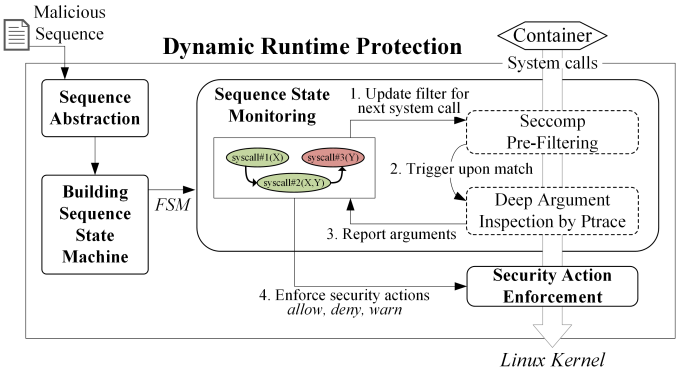


Fig. 3: Detailed view of the *dynamic runtime protection* step

1) Sequence Abstraction: During *sequence abstraction*, Phoenix builds a more generic version of the malicious sequence of system calls, since some parameters of those system calls may correspond to the specific context of one particular incident (e.g., calling process ID, arguments, and return code may all vary from one incident to another). Our idea here is to abstract such contextual parameters of system calls inside the sequence, such that Phoenix can subsequently match the sequence with other incidents exploiting the same vulnerability (which would have different values for such parameters). To facilitate the sequence abstraction, we divide system call parameters into three categories: (i) the parameters which should be precisely matched (e.g., flags and well-known connection ports). These parameters are kept as-is in the sequence; (ii) the parameters which should be matched but with their exact values ignored (e.g., IP addresses, file descriptors number, and process ID). These parameters are identified throughout the entire sequence and replaced with variables; and (iii) the parameters which should not be matched (e.g., certain strings and filenames). These parameters are removed from the sequence. The following illustrates this sub-step through an example.

Example 1. Figure 4 shows an example of *sequence abstraction*. The left table shows the system calls and their original parameters, while the right table shows the results of sequence abstraction. Abstracted parameters include file descriptor 4, which is substituted with variable **X** as it is contextual and represents the same file descriptor in both the `open` and `splice` system calls. Similarly, file descriptor 6 is abstracted as **Z** in both `pipe` and `splice`, and the PIDs of processes (12 and 13) are abstracted as variables A and B, respectively. Ignored parameters include filenames `passwd` and `su`, as well as the return values of system calls `pipe`, `splice`, and `execve`, which are removed (greyed cells in the figure). Finally, the flag `RDONLY` used by the `open` system call is an exact parameter that is left to be exactly matched.

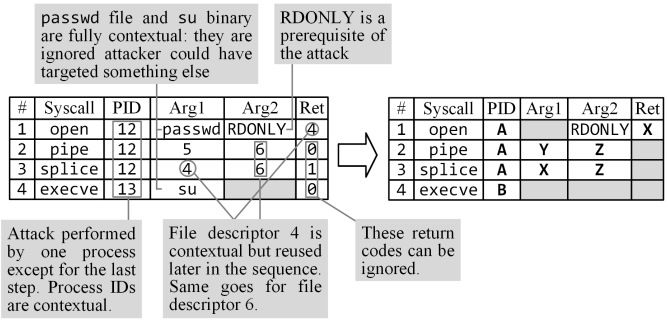


Fig. 4: Example of *sequence abstraction* step following an exploit of CVE-2022-0847

2) Building Sequence State Machine: To keep track of the attack progress in terms of intercepted system calls, we build a finite state machine (FSM) with an initial state, plus one state for each system call in the given malicious sequence. The FSM transitions from one state to the next if and only if the intercepted system call and its parameters satisfy the conditions of transition between those states (the conditions of transition are a little complex and will be explained shortly). Upon reaching a state, the FSM outputs the security action to be enforced. Such an FSM serves three purposes as follows: (i) keep track of the system calls intercepted so far (using one state per system call in the sequence); (ii) match system calls (intercepted by Seccomp) and their parameters (inspected by Ptrace) with the malicious sequence; and (iii) enforce user-specified security actions on matching system calls as the attack progresses, based on the output of the FSM.

The conditions of transition between states are used to determine whether each intercepted system call and its parameters match with the next state. Specifically, to ensure both accuracy (i.e., sequences are matched with minimal false positives) and generality (i.e., different instances of the same attack will all be matched), the conditions of transition are determined based on the parameter types (as previously described in the sequence abstraction sub-step) as follows. (i) Exactly matched parameters: Typically, these are representative and prerequisite parameters of the attack. Therefore, such parameters are always part of the conditions of transition and need to be exactly matched in order to avoid false positives (e.g., the application may open a file for reading only, whereas the attack would open it for writing). (ii) Abstracted parameters: Those parameters are shared by multiple system calls inside the sequence, while their exact values are not prerequisites to the attack. Therefore, once abstracted as variables (as described in the sequence abstraction sub-step), such parameters are part of the conditions of transition if they have been defined in a prior state (otherwise, they are defined in the current state). This can ensure different instances of the same attack will be matched (e.g., an attack may reach HTTP port 8080, while the same attack has been previously observed

on port 80). (iii) Ignored parameters: Those are typically incidental parameters that are not prerequisites to the attack. Therefore, such parameters are not considered in the conditions of transition and do not need to be matched. Ignoring those parameters can avoid false negatives when a malicious system call changes its target or tries to escape detection (e.g., an attack opens `/etc/shadow` instead of `/etc/passwd`).

Example 2. Figure 5 depicts an example of *building sequence state machine* for a sequence of three system calls involving exact parameters, ignored parameters (in grey cells), and abstracted parameters (in bold letters). Specifically, for the first system call `open`, its second argument `RDONLY` is an exact parameter, as it is a prerequisite of the attack. On the other hand, its PID and return code are abstracted parameters, as these are dependent on the execution environment, but will be re-used later in the sequence. Also, as these are observed for the first time, they are defined in this state. Therefore, the conditions of transition from the initial state to the first state `open` are composed of the system call `open` and its second argument `RDONLY`. For the second system call `pipe`, it has three abstracted parameters, PID, first argument **Y**, and second argument **Z**. Because PID **A** was defined in the previous state `open`, it now becomes part of the conditions of transition (i.e., the attack must emanate from the same process, e.g., if another process with a PID other than **A** were to issue a `pipe` call now, it would be rightfully ignored as it is out of the context of the sequence). Conversely, the two arguments **Y** and **Z** are not part of the conditions of transition, as they will only be defined in this state (`pipe`). Similar logic applies to the last state of the FSM and is omitted.

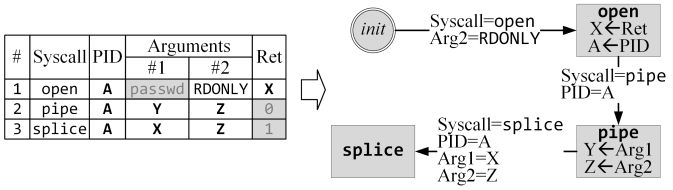


Fig. 5: Example of *building a sequence state machine* with three states (plus the initial state). System call parameters are simplified for the sake of understanding.

3) Sequence State Monitoring: During *sequence state monitoring*, Phoenix employs the previously built FSM to monitor the system calls used by the hardened container, matches them to the current state of the FSM, fires transitions accordingly, and enforces security actions as needed. First, Phoenix initializes the Seccomp filter to monitor for the first system call in the malicious sequence. Second, upon a matching, it triggers Ptrace to inspect the arguments of the system call. Third, it applies the FSM to verify whether the system call and its arguments match the current conditions of transition. If those do not match, the system call will simply be allowed; otherwise, a matching will trigger four actions in the following order: (i) the current state is updated to the next state of the FSM; (ii) the Seccomp filter is updated to match the next system call in the sequence; (iii) new abstracted parameters of the FSM are defined as needed; and (iv) any security action (i.e., allow, block, warn, etc.) output by the FSM is enforced for the current state.

Algorithm 1 summarizes the *sequence state monitoring* sub-step. Lines 3 and 4 initialize the state counter and Seccomp filter. Lines 6 and 7 leverage Seccomp and Ptrace to perform the pre-filtering and the deep argument inspection, respectively. Upon a match between the intercepted system call and the next FSM state (Line 8), we update the current state, the Seccomp filter, and the abstract parameters (if needed), and finally enforce the security action for the current state (Lines $9-12$). This is repeated until reaching the last state (Lines 5 and 12).

Algorithm 1 Phoenix procedure for *sequence state monitoring*

```
1: procedure SEQUENCE STATE MONITORING
2:     Input FSM with states {s_0, s_1, ..., s_n}, a security action for each s_i
3:     Let i = 0
4:     Update Seccomp filter to look for s_1
5:     while i < n do
6:         if Seccomp is triggered then
7:             Get interceptedState using Ptrace
8:             if interceptedState matches s_{i+1} then
9:                 i ← i + 1                                  ▷ Update state
10:                Update Seccomp filter to look for s_{i+1}
11:                Update abstracted parameters as needed
12:                Enforce security action for s_i
13:            end if
14:        end if
15:    end while
16: end procedure
```

Seccomp Pre-Filtering. During sequence state monitoring, *Seccomp pre-filtering* is the first checkpoint reached by all system calls coming from the container. As mentioned before, instead of blocking system calls right here (like in most existing works), we apply Seccomp for lightweight pre-filtering in order to take advantage of its efficiency while adding the opportunity for deep argument inspection. This is achieved through setting Seccomp to take the `SECCOMP_RET_TRACE` action [78] upon a matching, which will cause the kernel to notify Ptrace (instead of blocking the system call like with the `SECCOMP_RET_ERRNO` or `SECCOMP_RET_KILL` actions). This novel approach is the key for Phoenix to achieve both accuracy and efficiency (as illustrated in Section I).

Deep Argument Inspection by Ptrace. Upon a matching, Seccomp pre-filtering will trigger Ptrace to perform *deep argument inspection* for retrieving the value of system call arguments and the calling process ID. As explained in Section I, matching the arguments in addition to system calls allows Phoenix to avoid false positives. Even though the deep argument inspection performed by Ptrace incurs significantly more overhead (than the prior Seccomp pre-filtering), it is only performed on the few matching system calls, and hence its impact on the performance of Phoenix is still negligible, as will be shown through experiments in Section V-B.

4) Security Action Enforcement: During *security action enforcement*, Phoenix enforces user-specified security actions on the intercepted system calls. First, the user has the flexibility to pre-define a different security action for each of the system calls inside the malicious sequence, such as: (i) **Step**: the system call is allowed without notice; (ii) **Warn**: the system call is allowed, but an additional action is taken (e.g., alert the user or log to a file); (iii) **Block**: the system call is not allowed, but the calling process is not forced to terminate; (iv) **Exit**: the system call is not allowed, and the calling process is forced to terminate; (v) **Kill**: the system call is not allowed, and the calling container is shut down. Second, the user can combine those different actions in a particular way to achieve a better trade-off between availability and security (e.g., 'step' the first few system calls, 'warn' on

the next ones, and then 'block' the truly damaging ones at the end). Third, the user can also dynamically change those actions over time, based on his/her confidence in the malicious sequence, and/or any observed impact on security or availability (e.g., the user can start very conservatively with only 'warn' and no 'block', and then transit to more aggressive actions once his/her confidence in the sequence grows, or when he/she observes no impact on availability).

Example 3. Figure 6 depicts an example of both *sequence state monitoring* and *security action enforcement* using a sequence of two system calls (same as in Example 2), i.e., `open` (❶) and `pipe` (❷). First, the Seccomp filter is set to look for `open` system calls (❶a). In ❶b, a container calls `open` with first argument `file1` and second argument flag `RDONLY`. Suppose the FSM's conditions of transition require to match `RDONLY` but not `file1`. In ❶c, the `open` system call is matched by our Seccomp filter, which triggers Ptrace for deep argument inspection (while all other calls would simply be allowed by Seccomp). The two arguments are inspected in ❶d, and the second argument is exactly matched with the sequence. As now the first system call in the sequence has been intercepted, a state transition is fired, and a series of actions are performed in ❶e. First, we update the Seccomp filter at runtime to intercept the next system call (`pipe`) in ❶f. Second, the current system call (`open`) is allowed to enforce the user-specified action of 'step' in ❶g. Third, Phoenix updates the current state to the next state `pipe`. Finally, it defines the abstracted parameters that were observed (i.e., variable **A** is set to 85, the PID of the process in the sequence, and **X** is set to 16, the return code of the system call) in ❶h. The next system calls in the sequence will be intercepted in a similar fashion (details omitted).

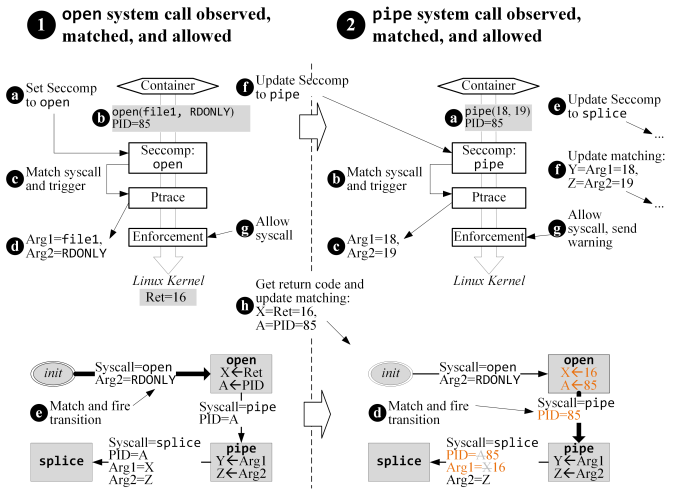


Fig. 6: Example of *sequence state monitoring* and *security action enforcement* upon matching `open` and `pipe`

B. Malicious Sequence Identification

The aforementioned *dynamic runtime protection* step allows Phoenix to block a malicious sequence of system calls accurately yet efficiently. Identifying such a sequence of system calls from security incidents exploiting an unpatched vulnerability would allow other affected users to prevent similar incidents from reoccurring. Although there exist fully automated approaches (e.g., host-based intrusion detection system (HIDS) [43], [49]), the lack of human involvement usually means inaccurate results for practical applications [70]. Therefore, we adopt a *human-in-the-loop* approach as follows. First, Phoenix continuously monitors abnormal behavior in the containers and constructs a provenance graph using historical data. Second, upon receiving an alert, users can leverage existing root cause analysis solutions (e.g., [20], [29], [30]) and their knowledge to identify the corresponding attack subgraph(s), and finally Phoenix automatically extracts candidate sequences of system calls from the subgraph(s) for further validation by human experts. Those are detailed in the following.

1) Provenance Graph Building: During *provenance graph building*, Phoenix generates a provenance graph based on events collected from the victim container. First, it continuously collects system audit data leveraging CLARION [8] (as part of its provenance tracking process). Auditing data consists of system calls and other details including the timestamps and process identifiers (PID) of the calling process and corresponding commands, user identifiers and group identifiers of the user running the process (UID and GID), etc. (a user may decide to rotate the auditing data as its size increases, i.e., to overwrite the oldest data or export it to external persistent storage). Unlike other approaches, our solution can make use of pointer arguments since it also performs deep argument inspection during sequence state monitoring. For this reason, we thoroughly consider the arguments of the audit data by de-referencing structures and pointers. We collect the arguments of the system calls such as defined by the system call interface of the Linux kernel. Second, upon receiving an alert, Phoenix leverages system provenance tools such as CLARION [8] and CamFlow [60], [61] to transform the collected auditing data into a provenance graph, i.e., a directed graph connecting system resources (as nodes) through system calls (as edges). Specifically, those tools parse the system calls, their arguments, and additional metadata (e.g., namespaces, which are necessary for working with containers) in the auditing data in order to establish the relationships between different resources such as files, processes, and network connections.

Example 4. Figure 7 depicts an example of the *provenance graph building* sub-step for an attack on a web application. A corresponding alert is received at 6:24:12. The collected auditing data from the victim container is saved. As an example of de-referencing, the argument `sockaddr` of the `accept` system call points to memory address `0xff52`. This address completely depends on the context of execution of the program, and hence should not be used for matching. Therefore, it is de-referenced and replaced by the corresponding values of the structure it points to, i.e., the address family `AF_INET`, the port number `80`, and the IP address `10.0.0.15`. Next, the provenance graph is built with the following relationships identified between resources: process `12` opens the file `passwd` as file descriptor `4`; then the same process calls `splice` between the file descriptors `4` and `6`, a newly created pipe. These are represented in the provenance graph as process `12` $\xrightarrow{\text{open}}$ `passwd` $\xrightarrow{\text{splice}}$ `pipe`. The rest of the provenance graph is built in a similar manner (details omitted).

2) Root Cause Analysis and Sequence Extraction: The goal of *root cause analysis* is to analyze the previously constructed provenance graph to identify the subgraph(s) encompassing system calls related to the received alert (i.e., attack subgraph(s)). First, upon receiving an alert, Phoenix

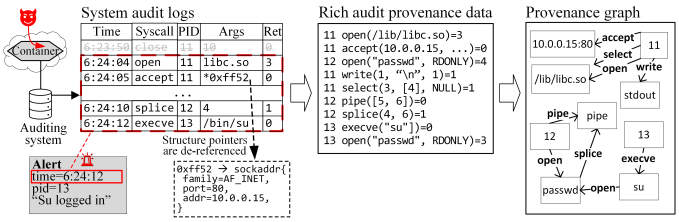


Fig. 7: Example of *provenance graph building* (the structures have been simplified for simplicity reasons)

automatically maps it to node(s) in the provenance graph by matching various parameters and metadata present in the alert, such as timestamp, PID, UID, GID, and alert message (e.g., as provided by Falco [17]). Second, users can leverage existing root cause analysis approaches (e.g., weighting backward dependencies by interest [20], reconstructing malicious behaviors using supervised learning [1], [54], [55], or unsupervised learning [29], [81]) and their domain knowledge to identify the attack subgraph(s).

The goal of *sequence extraction* is to identify the sequence of system calls from the attack subgraph(s) to be given as input to our *dynamic runtime protection* step. Because provenance graphs store causal (instead of temporal) dependencies, Phoenix must trace back the events to their original system calls to build a chronological sequence of system calls and their parameters (arguments, calling processes, and return values). First, Phoenix automatically maps the system calls present in the attack subgraph to the original system audit logs. Second, Phoenix extracts the system calls and their parameters as a sequence in which the system calls are ordered chronologically. Finally, Phoenix displays such candidate sequence(s) inside GUI for them to be validated by an expert in order to identify the final sequence to be used by Phoenix for *dynamic runtime protection*. The administrator may further assign an action plan, i.e., a series of security actions to be performed upon intercepting each system call in the sequence (as detailed in Section III-A).

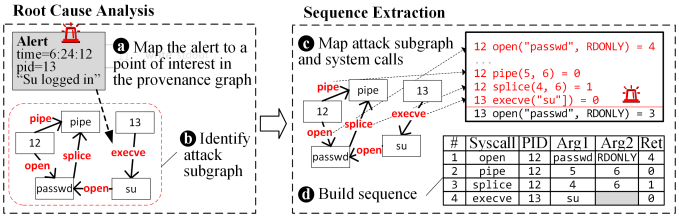


Fig. 8: Example of *root cause analysis* and *sequence extraction* following an exploit of CVE-2022-0847

Example 5. Following Example 4, Fig. 8 shows an example of *root cause analysis* and *sequence extraction*. First, during root cause analysis, the alert "Super user logged in" is mapped with the system call `execve` based on its timestamp, the calling process PID, and additional metadata provided by Falco (a). Then, the attack subgraph is identified by leveraging root cause analysis solutions [20] (b). Second, during sequence extraction, in c, Phoenix identifies all system calls present in the attack subgraph *before* the alert (i.e., `execve`, `open`, `pipe`, and `splice`) and retrieves them in the original system audit logs. Then, in d, those system calls and their parameters are chronologically ordered to reconstruct the malicious sequence of system calls, i.e., `open`→`pipe`→`splice`→`execve`.

IV. Implementation

This section details the implementation of Phoenix. We implement Phoenix in C and C++ for Linux kernel v5.10 in approximately 2,000 lines of code (excluding the sources of other existing projects that are leveraged in Phoenix). Phoenix can also function on other kernel versions that implement a recent version of Seccomp-BPF (≥v4.14). It is important to note that Phoenix requires *no* modification to the applications, the container runtime, or the Linux kernel (instead, a kernel module is needed).

A. Implementation of Dynamic Runtime Protection

The *dynamic runtime protection* module is divided between user-space components (*sequence preprocessor*, *sequence state monitor* and *Ptrace*) and kernel-space components (*Seccomp filter* and *kernel module*), as illustrated in Fig. 9). In the following, we detail each of these components.

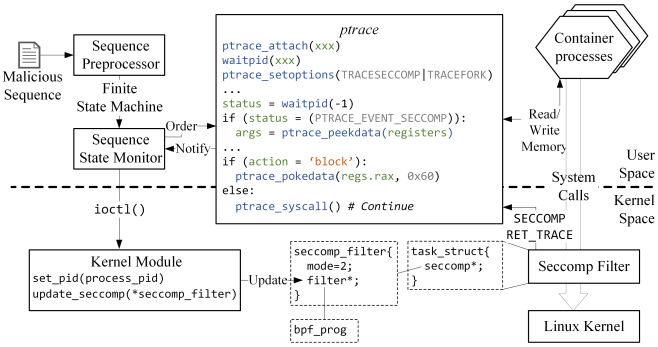


Fig. 9: Implementation of *dynamic runtime protection*

Sequence Preprocessor. Our sequence preprocessor is to abstract the malicious sequence of system calls identified during *malicious sequence identification*, and construct an FSM based on the abstracted sequence. This module is written in C++ and takes the malicious sequence of system calls in JSON format as input. We use the JSON library for Modern C++ [46] to parse the input malicious sequence and we develop our own implementation of the FSM.

Sequence State Monitor and Ptrace. This component runs as a privileged process in the host OS to monitor system call sequences for all processes in a container. Specifically, the sequence state monitor first collects information from the container runtime interface (CRI) (e.g., using `docker inspect` or `crictl inspect`)[2]. It then attaches Ptrace to all the processes in a container with `PTRACE_ATTACH` and traces child processes using the following options: `PTRACE_EVENT_FORK`, `PTRACE_EVENT_CLONE`, and `PTRACE_EVENT_VFORK`. Additionally, it monitors Seccomp signals using the `PTRACE_EVENT_SECCOMP` flag. At runtime, this component waits for a `trap` signal from any of its tracees.

A challenge upon the receipt of a signal is to differentiate between the signal from a forked process and from a process getting interrupted by Seccomp. To address the challenge, we compare the status of the stopped process (obtained with `wait(-1, &status))` with conditions of the Ptrace event, i.e., `(status » 8 == (SIGTRAP`

[2]We implement our solution for Docker and Containerd, however it can work on other CRI (e.g., CRI-O) using the `crictl` CLI.

| (PTRACE_EVENT_FORK « 8))) and (status » 8 == (SIGTRAP | (PTRACE_EVENT_SECCOMP « 8))), respectively [64]. When the first condition is satisfied, we use PTRACE_GETEVENTMSG to retrieve the PID of the new child process, and attach to it. Otherwise, we inspect and retrieve arguments in depth using the PTRACE_PEEKUSER and PTRACE_PEEKDATA functions on the corresponding registers. Additionally, to inspect arguments with pre-defined, and per-architecture rules, our solution leverages Strace [73].

After Seccomp and Ptrace intercept a system call and its arguments matching with the FSM state, we perform the following actions. (i) The FSM transition to the next state is implemented by updating a variable in the FSM. (ii) The update of Seccomp filter for intercepting the next system call is implemented using the libseccomp library (seccomp_init, seccomp_rule_add, seccomp_load functions), and updated using our kernel module, as detailed later. (iii) The abstracted parameters of the FSM (if any) are updated with the new values just observed. A challenge here is that sometimes the return code is also a parameter to be observed. To address this, we repeat that action after the system call has been allowed by sending PTRACE_SYSCALL and waiting for the calling process to pause on the system call exit. After the system call exits, the return code can then be read from the rax register. (iv) Finally, enforcing user-specified security actions (step, warn, block, and kill) is implemented using Ptrace (this step is left to the end to ensure consistency and avoid race conditions). Specifically, the step or warn actions (both allow the system call) are implemented by resuming the calling container process with PTRACE_CONT. The block action is implemented by replacing the system call with the exit system call (using PTRACE_SETREGS), and the kill action is implemented by sending the SIGKILL signal. Those actions are input through a GUI and stored in the sequence file.

A challenge here is to monitor processes created from outside of the container. For instance, processes created using docker exec or kubectl exec commands are essentially created by the container shim process and thus are not children of an existing container process, but rather its siblings. To ensure Phoenix can monitor and trace such processes as well, we implement a *shim tracer* to observe the container's shim process and wait for clone or fork events. Upon such events, our shim tracer interrupts the newly created process, detaches from it, and sends a SIGSTOP signal to the main Phoenix process to transmit the PID of the new process using a message queue. Therefore, Phoenix can now handle the SIGSTOP to resume and trace this new process as usual.

Seccomp Filters. Seccomp filters are installed by default on *Docker* and *containerd* (can otherwise be enabled, also for Kubernetes [38]). To implement Seccomp pre-filtering in Phoenix, we set them to monitor (instead of blocking) a system call with the SECCOMP_RET_TRACE action [78]. To implement the dynamic update of Seccomp filters, the new filter is compiled in the user space and sent to the kernel module with an ioctl directive (detailed later). Upon intercepting the system call, the Seccomp filter sends a PTRACE_EVENT_SECCOMP signal that triggers Ptrace for deep argument inspection.

Kernel Module. A key implementation challenge is for Phoenix to update Seccomp filters at runtime. Seccomp filters are applied to the processes of a container at its startup, and cannot be trivially modified during the container runtime. Even though the prctl and seccomp system calls can be used to update the Seccomp filter of a process, there exist two major limitations: (i) these can only be called from inside the container, which means we would have to change the container's behavior; and (ii) these can only update the filters by making them *more* restrictive, which cannot support our need to remove old rules and add new ones. Therefore, we need to update Seccomp filters directly in kernel space. As our monitor runs in user space, we implement a kernel module to update the Seccomp filter through two functions that are called successively by the monitor:

- get_seccomp(process_pid): This function retrieves the address of the current Seccomp filter to later modify its content. To do so, it retrieves the task_struct (i.e., a kernel structure containing all the information about a process, including its current Seccomp filters represented as a bpf_prog structure) associated with the process identified by its process_id.
- update_seccomp(*seccomp_filter): This function compiles the Seccomp filter received from the monitor (via a pointer to seccomp_filter in user space) into a BPF program using the bpf_prog_create_from_user function. The kernel module additionally checks that the given Seccomp filter is correct for the current architecture with seccomp_check_filter, and retrieves the new BPF program address as a pointer. Then, it replaces the BPF program at the current process (previously obtained by the get_seccomp function) with the new BPF program by updating the pointer to the bpf_prog structure to point to the new BPF program address.

Our kernel module is written in C (around 400 source lines of code) and is loaded in the Linux kernel at runtime. The kernel module functions are invoked by Phoenix from the user space using ioctl directives.

B. Implementation of Malicious Sequence Identification

We implement the *malicious sequence identification* module as a framework integrating existing tools for provenance data collection, attack detection, provenance graph construction, and root cause analysis. First, to detect the impact of vulnerability exploitation in containers, we deploy a popular monitoring solution, Falco [17], with both the default rules and custom rules developed for detecting particular exploits (e.g., DirtyPipe [18]). Later, to implement the correlation of Falco alerts with a point of interest in the provenance graph, we extract metadata in the alert [19], particularly the timestamp (evt.time) and the system call (syscall.type). We also extract the PID and parent PID (PPID) of the process generating the alert to narrow down the candidate points of interest in the provenance graph. As containers run in different namespaces, they often share the same virtual (i.e., from the point of their PID namespace) process ID. Therefore, using the virtual PID of processes would lead to one-to-many mappings. Instead, we attach both the real PID and PPID of the container generating the event (proc.pid and proc.ppid, respectively) to the alert.

Second, to collect and track system provenance data, we leverage CLARION [8] (a namespace-aware and container-aware extension of SPADE [23]). To enable namespace-aware

and container-aware provenance data tracking, we rely on the Auditd Linux auditing daemon [28] and the custom kernel module from CLARION. We successfully compiled the custom kernel module on Linux kernel versions >5.7, even though vendors do not officially support it (the required `kallsyms_lookup_name` function is not exported anymore [10] in such versions). To overcome this issue, we plant a Kprobe into the kernel and retrieve the `kallsyms_lookup_name` function's symbol. Finally, we enable CLARION's `fileIO`, `localEndpoints`, `IPC`, and `namespace` options to capture fine-grained details regarding file paths, local network ports and addresses, and container namespace contexts, respectively. We enable all options that give us in-depth information about arguments, as these can be leveraged by Phoenix during *dynamic runtime protection*. We store the provenance data captures using a Neo4j database. Our solution can also potentially be integrated with other existing provenance tracking tools, such as CamFlow [60].

Finally, for root cause analysis, users can leverage existing provenance-based detection tools including DepImpact [20], ATLAS [1], ProvDetector [81], Unicorn [29] as needed. To help users identify the malicious sequence from subgraphs, we developed a GUI tool using the graph library `Neovis.js` to facilitate manual exploration and analysis through zooming, panning, filtering (nodes and edges), time-slicing, etc. Once the sequence is identified, its extraction is performed in JSON format with a custom JavaScript code.

V. Evaluation

This section evaluates Phoenix in terms of both security and performance by answering the following research questions:

- **RQ1**: How well does our solution prevent the exploit of real-world vulnerabilities in real-life applications? How does it compare to existing solutions in this regard?
- **RQ2**: What is the overhead introduced by Phoenix on the performance of the containers? How does it compare to the overhead induced by existing solutions?
- **RQ3**: How feasible is the sequence identification approach? How does it compare to existing solutions?
- **RQ4**: What are the additional benefits (in terms of accuracy) and overhead of deep argument inspection?

Experimental Setup and Dataset. We run Phoenix on a Kubernetes cluster composed of one master node and two worker nodes, each equipped with 4 vCPUs and 8GB of RAM. The nodes are VMs running Ubuntu 20.04 on Linux kernel v5.10 (except while testing vulnerabilities that require a particular kernel version). For our security evaluations, we borrow two existing datasets of container system calls, namely, the Container Breakout dataset (CB-DS) [15] and the DongTing dataset [14]. We conduct the response time experiments over a local network to minimize the impact of network delay on the measurement of the application's response time.

A. Security

1) Comparison of effectiveness for blocking CVEs: To answer RQ1, we perform a case study using real-world CVEs with a CVSS (severity) score ranging from 2.1 to 7.8 (out of 10) on popular applications, as listed in Table I. In this experiment, we compare our solution to other state-of-the-art solutions, including Confine [24], Sysfilter [11], and Docker's default Seccomp filter [67], in terms of their capability for consistently preventing the exploitation of the CVEs for all applications. Thus, for each CVE, we first download a proof-of-concept (PoC) exploit from public code repositories. Next, for each combination of the CVE and application, we create a separate container, with the corresponding exploit code mounted in the container next to the application. Then, while the application is running, the exploit code is manually triggered through a terminal opened inside the container (note all the CVEs are kernel-related and independent of applications). Additionally, the exploit code is executed under Strace [73] to record the set of system calls and arguments involved in the exploit. For each application, we apply existing solutions following their documentation and official code base[3] to obtain the list of system calls that can be blocked under that application (i.e., the ones unused by the application). Finally, we compare the set of system calls required by the exploit code to the set of system calls that can be blocked by existing solutions.

CVE	Severity	Application 1	2	3	4	5	6	7	8	9	10
2017-18344	2.1	--p	csp	csp	csp	csp	csp	csp	csp	-sp	csp
2017-5123	4.6	--p	--p	csp	--p	c-p	--p	--p	c-p	-sp	c-p
2019-5489	5.5	--p	csp	csp	csp	c-p	csp	c-p	-sp	--p	csp
2022-1015	6.6	--p	csp	csp	csp	csp	csp	csp	csp	--p	csp
2017-17053	6.9	-sp	csp	csp	csp	csp	csp	csp	csp	--p	csp
2022-0492*	6.9	-sp	-sp	csp	csp	-sp	-sp	csp	csp	-sp	-sp
2022-2602	7.0	--p	csp	csp	-sp	-sp	csp	csp	csp	-sp	-sp
2017-11176	7.2	--p	csp	csp	csp	csp	csp	csp	csp	-sp	csp
2018-14634	7.2	--p	--p	--p	--p	--p	--p	--p	--p	-sp	--p
2021-3347	7.2	-sp	--p	csp	--p	c-p	--p	csp	-sp	-sp	-sp
2021-4154	7.2	-sp	-sp	c-p	-sp	csp	-sp	-sp	csp	-sp	csp
2022-0847	7.2	--p	c-p	-sp	--p	-sp	--p	-sp	-sp	-sp	-sp
2016-9793	7.8	-sp	csp	csp	csp	--p	csp	csp	csp	-sp	csp
2017-6074	7.8	-sp	csp	csp	-sp	--p	csp	csp	-sp	-sp	-sp
2017-7308	7.8	-sp	-sp	-sp	-sp	-sp	-sp	-sp	--p	-sp	-sp
2022-0995	7.8	-sp	csp	csp	csp	c-p	csp	csp	csp	-sp	csp
2022-2588	7.8	--p	csp	csp	csp	csp	csp	csp	csp	-sp	csp
2022-2639	7.8	--p	csp	csp	csp	csp	csp	csp	csp	-sp	csp
2023-0386	7.8	--p	-sp	-sp	-sp	-sp	-sp	csp	--p	-sp	-sp
2023-32233	7.8	-sp	csp	csp	-sp	csp	csp	csp	csp	-sp	csp

1: CRIU†, 2: Django, 3: Httpd, 4: Nginx, 5: Postgres, 6: Python, 7: Redis, 8: Tomcat, 9: Wine†, 10: Wordpress.
c: blocked by Confine [24], s: blocked by Sysfilter [11], p: blocked by Phoenix, -: not blocked.
*blocked by default Seccomp filter [67], †Confine not tested (not a container).

TABLE I: Comparison of the effectiveness of Confine [24], Sysfilter [11], and Phoenix for blocking 20 CVEs without affecting the normal operation of 10 popular applications.

Table I summarizes the results of this experiment. It can be observed that, while Phoenix can consistently block all the studied CVEs on all the applications, none of the existing solutions can achieve this. In particular, Docker's default Seccomp filter can only block one CVE (2022-0492), while Confine and Sysfilter show varying degrees of success on different combinations of CVEs and applications. An extreme example is CVE-2018-14634 (an integer overflow in the Linux kernel), which almost completely escapes Confine and Sysfilter on all the applications (except that Sysfilter works on one application, Wine). This is due to the fact that those existing solutions by design do not block system calls that are needed by an application (as their main purpose is to

[3]We make best efforts in following the recommendations of the tools' authors to perform our study though some inaccuracies or usage in a different context than the authors' are still possible.

reduce the general attack surface). In contrast, Phoenix shows superior effectiveness for blocking specific vulnerabilities due to its added capabilities of considering the sequence in which a system call appears and deep argument inspection.

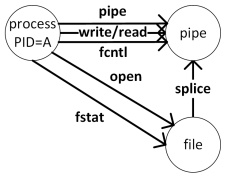
		CB Dataset [15]						DongTing Dataset [14]					
		1-call		2-call		3-call		1-call		2-call		3-call	
		mean %	*max* %	*mean* %	*max* %	*mean* %	*max* %	*mean* %	*max* %	*mean* %	*max* %	*mean* %	*max* %
CVE	**2017-18344**	1.64	11.2	0.12	1.37	0.03	1.06	1.44	6.38	0.22	2.68	.057	1.44
	2017-5123	1.77	11.2	0.15	1.37	0.09	1.35	1.66	6.38	0.05	1.38	.003	0.05
	2019-5489	1.27	11.2	0.11	3.39	0.05	1.35	1.0	6.38	0.07	2.63	.01	0.72
	2022-1015	0.29	4.84	0.01	0.1	0.0	0.01	0.97	4.28	0.14	4.28	0.0	.001
	2017-17053	1.66	11.2	0.12	1.37	0.06	1.35	1.13	6.38	0.05	1.38	.008	0.15
	2022-0492	0.91	11.2	0.10	3.45	0.03	1.35	0.86	6.38	0.05	2.63	.004	0.72
	2022-2602	1.11	6.96	0.13	3.32	0.04	1.81	1.12	6.38	0.01	0.06	.001	.004
	2017-11176	1.30	16.4	0.07	1.37	0.04	1.35	0.97	6.38	0.08	2.63	.011	0.72
	2018-14634	1.71	11.2	0.14	1.37	0.09	1.35	1.44	6.38	0.06	1.38	.006	0.05
	2021-3347	1.79	16.4	0.13	4.32	0.04	1.35	1.01	6.38	0.03	1.38	.002	0.05
	2021-4154	1.62	16.4	0.05	1.37	0.01	1.35	0.90	6.38	0.05	2.63	.004	0.60
	2022-0847	1.80	11.2	0.17	1.37	0.11	1.35	1.64	6.38	0.18	2.63	.004	0.05
	2016-9793	1.06	16.3	0.08	3.45	0.03	3.11	0.7	6.38	0.05	2.63	.005	0.57
	2017-6074	1.45	6.96	0.01	0.1	0.0	0.00	1.85	6.38	0.04	0.37	.001	0.01
	2017-7308	1.16	11.2	0.0	0.0	0.0	0.0	1.18	6.38	0.0	0.01	0.0	0.0
	2022-0995	0.81	4.84	0.0	0.0	0.0	0.0	2.62	4.28	0.54	4.28	0.0	0.0
	2022-2588	1.0	11.2	0.07	1.37	0.03	1.35	0.86	6.38	0.06	2.63	.01	0.59
	2022-2639	0.96	11.2	0.07	3.45	0.01	1.06	1.26	6.38	0.16	4.28	.004	0.04
	2023-0386	1.18	11.2	0.05	1.36	0.02	1.06	1.23	6.38	0.06	2.68	.004	0.05
	2023-32233	1.28	16.4	0.08	3.29	0.01	1.06	0.96	6.38	0.08	2.68	.007	0.60
	Average	1.29	11.4	0.08	1.86	0.03	1.13	1.24	6.17	0.1	2.26	.007	0.32

TABLE II: Comparison of false positive rates between set-based (1-call) and sequence-based (2-call and 3-call) solutions for blocking CVEs with system calls needed by the application

The following provides more details for two of those CVEs (others are omitted due to space limitations) and explains how Phoenix works in each case.

CVE-2022-0847. Figure 10 details an exploit of this vulnerability (a.k.a. *Dirty Pipe*) [52] and the corresponding provenance graph, as well as the sequence of system calls identified. The vulnerability lies in the Linux `pipe` mechanism, where an unprivileged user can write to a normally read-only file and escalate privileges. To exploit the vulnerability, an attacker creates a pipe, and then fills it with arbitrary data. S/he then opens a read-only file (the target file) and splices data from the file into the pipe. Finally, by writing again to the pipe, s/he overwrites cached memory pages and successfully writes to the read-only file. A simple example of privilege escalation using this technique is to write to the */etc/passwd* file and add a new privileged user to the system. The vulnerability can also be exploited to escape a container isolation (e.g., by overwriting the *runc* binary on the host [56]).

Existing solutions, such as Confine and Sysfilter, may face challenges in blocking this CVE. Specifically, with certain applications (e.g., a high-performance HTTP proxy using web servers such as Nginx and Tomcat), a container might need to manipulate pipes and perform zero-copy operations using the `pipe` and `splice` system calls [51]. For instance, applying Confine on a Tomcat container apparently cannot prevent this CVE from being exploited due to system calls needed by the application. Although Sysfilter reportedly blocks the `splice` system call, it also blocks the `sendfile` system call, a call needed for the normal behavior of Tomcat. In contrast, Phoenix can prevent CVE-2022-0847 on every application by accurately blocking the sequence of system calls described in Fig. 10. An admin may decide the best action plan for preventing the vulnerability. For example, the corresponding action chosen for system calls #1 to #5 is `step` (i.e., observe and continue in the sequence) because these calls are frequently seen when manipulating Linux pipes. The `read` and `splice` system calls immediately following will send a warning, whereas the last `write` (i.e., the actual exploit of the vulnerability) should be blocked to prevent the attack. Alternatively, the admin may decide to block the sequence or send warnings earlier, or be more conservative by only sending warnings.

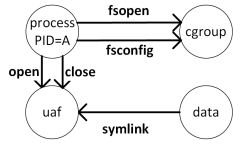


Action	#	Syscall	PID	Arguments #1	Arguments #2	Arguments ...	Ret
step	1	open	A	file	RDONLY		X
step	2	fstat	A	X			
step	3	pipe	A	Y	Z		
step	4	fcntl	A	Z			
step	5	write	A	Z			
warn	6	read	A	Y			
warn	7	splice	A	X	Z		
block	8	write	A	Z			

Fig. 10: CVE-2022-0847 with corresponding provenance graph (left) and system calls sequence identified by Phoenix (right)

CVE-2021-4154. This is a vulnerability in the Linux kernel's `cgroup` mechanism. Using a privilege escalation technique (e.g., DirtyCred [42]), an attacker can exploit this vulnerability to escape a container's isolation, and compromise the underlying host. The vulnerability lies in a use-after-free (UAF) bug that can be triggered using the `fsopen` and `fsconfig` system calls on a `cgroup` filesystem. We use a proof-of-concept leveraging the DirtyCred technique to exploit this vulnerability, and Fig. 11 depicts the corresponding provenance graph and system call sequence identified.

According to our results, Confine blocks CVE-2021-4154 on three applications out of ten, while Sysfilter does not prevent the CVE on the *httpd* web server. Phoenix prevents the exploit of CVE-2021-4154 on every application by blocking the sequence of system calls described in Fig. 11. As the exploit is performed, Phoenix identifies the first argument of `fsopen` as `cgroup`, to be matched as-is. The file descriptor returned is later used by the `fsconfig` system call, and abstracted as **X** by our solution. Next, the exploit creates a `symlink` later used by `open` (abstracted as **Z**), then returns another file descriptor abstracted as **W**. After the call to `fsconfig` with the first and third abstract arguments **X** and **W**, and the flag `SET_FD`, the UAF is triggered by closing the file descriptor **X**.



Action	#	Syscall	PID	Arguments #1	Arguments #2	Arguments ...	Ret
step	1	fsopen	A	cgroup	0		X
step	2	symlink	A	Y	Z		
step	3	open	A	Z			W
warn	4	fsconfig	A	X	SET_FD	W	
block	5	close	A	X			

Fig. 11: CVE-2021-4154 with corresponding provenance graph (left) and system calls sequence identified by Phoenix (right)

2) General comparison of false positives between Phoenix and set-based blocking: Our next experiment performs a more general comparison between a *sequence-based* system call blocking solution like Phoenix, and a *set-based* blocking solution such as the direct application of Seccomp filters. The goal is to assess the impact of considering (or ignoring) the sequence in which a system call appears, on false positives, while preventing the exploitation of a specific vulnerability. For this purpose, we perform the experiment using two attack-free datasets of container system calls, i.e., the Container Breakout dataset (CB-DS) [15] (more than 3 million normal system calls extracted from 1,700 traces of an Apache application)

and the DongTing dataset [14] (25 million normal system calls from 6,850 traces of four Linux kernel regression test suites). We measure the false positive rate of each solution as the percentage of normal system calls that are blocked as a side-effect of blocking different CVEs (same as in Table I).

Table II reports the percentage of false positives observed for the set-based solution (blocking one system call regardless of the sequence it appears in) and the sequence-based solution for blocking two and three system calls, respectively. The results show that ignoring the order between system calls (set-based solution) may result in significantly more false positives, i.e., around 1.29% false positives on average, with certain cases going as high as 16.4% (e.g., CVE-2021-3347 on the CB-DS dataset). The same system calls blocked in a sequence of length two results in less than 0.1% false positives, while increasing the sequence size to three reduces the false positives to less than 0.03% in both cases. Those results not only show the effectiveness of blocking system calls based on the sequence in terms of fewer false positives, but also demonstrate that even a partial sequence can lead to a significant reduction in false positives.

3) Comparison with existing stateful solutions: In this experiment, we compare Phoenix with existing solutions that can also perform stateful inspection of system calls [21], [34], [57] (these are re-implemented as no working code is publicly available). Specifically, VtPath [21] and Mutz et al. [57] both leverage the call stack between pairs of consecutive system calls (namely, virtual paths) to learn the normal behavior and detect anomalous calls, while the latter also considers system calls arguments. PoLPer [34] blocks anomalous system calls in the *setuid* family by learning the calling process hierarchy, call context, and its arguments. For a fair comparison, we evaluate them both for a purpose similar to Phoenix, i.e., blocking a particular vulnerability (blacklisting), and for their intended usage, i.e., anomaly detection (whitelisting).

Solution	Vulnerability learned	Same vulnerability exploit (TP)	Modified vulnerability exploit (TP)	Normal behavior (FP)
VtPath	2023-32233	100%	15%	0%
	2017-6074	100%	17%	0%
	2022-0847	100%	63%	0%
	2021-4154	100%	64%	0%
	2023-0386	100%	94%	0.01%
Mutz et al.	2023-32233	0.57%	0.42%	0%
	2017-6074	1.83%	1.22%	0%
	2022-0847	7.59%	3.80%	0%
	2021-4154	7.89%	7.16%	0%
	2023-0386	1.34%	1.29%	0%
PoLPer	2023-32233	100% (2/2)	0% (0/2)	0% (0/2)
	2017-6074	0*%	0*%	0*%
	2022-0847	0*%	0*%	0*%
	2021-4154	0*%	0*%	0*%
	2023-0386	100% (3/3)	33% (1/3)	0% (0/3)
Phoenix	2023-32233	100%	100%	0%
	2017-6074	100%	100%	0%
	2022-0847	100%	100%	0%
	2021-4154	100%	100%	0%
	2023-0386	100%	100%	0%

*: exploit does not invoke *setuid* calls

TABLE III: Comparison of Phoenix with existing stateful solutions for blocking vulnerabilities (blacklisting)

Blacklisting. In this first experiment, we apply each solution to learn the exploit of each vulnerability based on the used characteristics (i.e., the malicious virtual paths for [21] and [57], the string arguments for [57], and the malicious *setuid* calls context, hierarchy, and arguments for [34]). We then evaluate their effectiveness in capturing the same exploit executed again, a slightly modified exploit (detailed below) of the same vulnerability, and the normal behavior, respectively.

First, as Table III shows, while VtPath [21] successfully detects the same exploit used during training, it cannot accurately detect slightly modified exploit codes. For instance, directly invoking the system calls instead of using the C library wrappers can drastically change the virtual path between two consecutive system calls. As a result, only 15% and 17% of the exploit code for the first two CVEs can be captured. Moreover, adding dummy system calls in the exploit code can also deceive VtPath, as shown by the next three CVEs. Second, the approach of Mutz et al. [57] shows worse results (even on the same exploit). Our investigation shows this is due to their design choice of limiting the learning to 36 system calls (which is not sufficient to capture the exploits). Third, PoLPer [34] can only capture two out of five CVEs due to the fact that it is specifically designed to match system calls from the *setuid* family, which are only used by two of those exploits (the number of successfully matched *setuid* calls are reported between parenthesis). Moreover, like the other two works, PoLPer cannot capture the slightly modified exploit codes. Finally, all three solutions cause almost zero false positives on the normal behavior. In contrast, Phoenix shows perfect results in all three cases since the sequence of system calls and arguments it relies on is essential for the CVE to function, and hence cannot be easily manipulated by attackers. In summary, although those existing works can also perform stateful inspection of system calls, they are not suitable for blocking unpatched vulnerabilities (blacklisting), as they all rely on context-specific information, which is not essential to a CVE, and hence cannot detect slight variants of attacks or evasive attack behaviors.

Whitelisting. In this experiment, we apply those existing solutions for their intended purpose, i.e., learning the normal behavior of applications to detect anomalies. The experiment is based on three widely used applications, i.e., Nginx, Tomcat, and Redis. For each application, we collect several minutes of normal behavior data (90% for training and 10% for testing) to measure the FP (second column). Then, we perform anomaly detection on the same application while executing the exploits (same as in Table III) to measure the TP (third column).

As Table V shows, all the solutions can successfully detect the anomalous system calls (*setuid* calls detected by PoLPer are reported between parenthesis). However, VtPath [21] and Mutz et al. [57] both cause false positives on the normal behavior (training data). VtPath has the highest rate of false positives on the Tomcat application, where almost 27% of the data is misinterpreted as an attack (likely due to the multi-threaded nature of Tomcat). The approach of Mutz et al. has a lower false positive rate for all the applications (which can be explained by its use of arguments comparison and the fact that it monitors a smaller number of system calls). PoLPer achieves zero false positives for all the applications (which can be explained by its focus on the eight system calls from the *setuid* family, and the fact that our test involves a single application running inside a container). In summary, those existing works are more effective for their designed purpose, i.e., anomaly detection, which is complementary to the objective of Phoenix

(blacklisting). Our experiences also show that, by collecting call stack for every single system call, those existing solutions may generate significant overhead (in fact, the experiment could not be performed for Flask, MySQL, and Django, as those applications simply stopped working properly under the overhead). Moreover, as shown next, our experiments indicate those solutions incur significantly longer response time per system call. In contrast, as we will show next, the performance overhead of Phoenix remains negligible for all applications.

Comparison with Existing Stateful Solutions in terms of Response Time. We compare the overhead of VtPath [21], Mutz et al. [57], and PoLPer [34] with that of our solution. For VtPath [21], the authors assume "the program counter and call stack can be visited with low runtime overhead when each system call is made" and thus do not provide a performance evaluation of the actual implementation (only the algorithm is evaluated). Mutz et al. [57] implement system call interception and stack unwinding by modifying a SNARE kernel module. PoLPer [34] plants a Kprobe [37] at the *setuid*-family system call handlers. While this has the advantage of not requiring any user space implementation, it is not designed for a container environment since the handler will be invoked by any *setuid*-family system calls in the operating system, most of which will be irrelevant to the targeted container (in contrast, Phoenix will only be triggered when a particular system call coming from a particular container is invoked).

We adapt the available data and code of those existing solutions according to changes in technology as follows (e.g., the SNARE kernel module has been progressively replaced by the Auditd subsystem). For VtPath, we assume the use of Strace (Ptrace and `libunwind`) to access the call stack, as the authors in [21] mention they use the "same user-space level mechanism to intercept system calls as [68]", which is Ptrace. For Mutz et al., we could not find the source code of the deprecated SNARE kernel module; instead, we utilize a kernel module that performs stack unwinding of a user space program at each of the 36 system calls monitored, similarly to Kunwind [39]. For PoLPer, we develop a kernel module and plant a Kprobe at each system call handler of the *setuid*-family system calls.

We evaluate the overhead of each approach by measuring the average response time on all system calls over 1,000 calls (note that not all system calls are called with the same frequency in reality). As Table IV shows, the baseline (i.e., no protection) takes around 3,000 ns per system call. In contrast with the assumption made in [21], using user space mechanisms such as Strace (with the `-k` option for stack unfolding) is very inefficient and incurs prohibitive (over 12,000%) overhead. Using Kunwind and Kprobe is more efficient, but still represents a 116% and 1,080% overhead, respectively. Phoenix only incurs negligible overhead (less than 3%) over the baseline. Finally, for a fairer comparison, we modify Phoenix to add a stack unwinding mechanism (using the `libunwind` library) although Phoenix does not really require call stack to function. Our results show that this only adds a 15% extra overhead over the regular version of Phoenix.

	Average Response Time per System Call
Baseline	3,113 ns
Strace + stack unwinding (VtPath)	393,361 ns
Kunwind (Mutz et al.)	6,726 ns
Kprobes (PoLPer)	36,748 ns
Seccomp + Ptrace (Phoenix)	3,201 ns
Phoenix + stack unwinding	3,690 ns

TABLE IV: Performance comparison with existing stateful solutions

B. Performance

To answer RQ2, we deploy Phoenix and existing solutions with different applications (popular web servers and databases, namely, Django, Flask, MySQL, Nginx, Redis, and Tomcat) in our testbed and then measure different performance metrics (i.e., response time, CPU usage, and memory consumption) of the applications. We test an *idle case* (i.e., the security solutions are not blocking any attack, as no attack is actually performed), a *normal case* where we simulate an attack rate of around 100 attacks per day [2], and a *denial-of-service (DoS) case* in which the solutions are assumed to be under DoS by receiving one attack on every request. Note we consider the last case only to assess how resilient Phoenix is under stress, while such a case is extremely unlikely to occur in reality, as it would mean all system calls were coming from the attacker and none from legitimate requests. During our experiments, due to their different designs, Seccomp blocks one single system call per request, while Ptrace and Phoenix each block the same sequence of four system calls per request. We compare the performance of Phoenix (combining Ptrace and Seccomp), with (i) the baseline application without protection (No Seccomp); (ii) the direct application of Seccomp which blocks every system call appearing in the attack sequence regardless of the actual sequence it appears in (Default Seccomp); and (iii) a naive implementation of Ptrace to block attack sequences (Ptrace). We average the results of 10,000 requests over the span of 100 seconds.

1) Overhead on response time: Figure 12 compares the response time of different applications deployed with Phoenix and the other solutions. We run different containers and measure the average user-experienced response time (without considering the network delay) over repeated queries. In idle and normal cases (Fig. 12a and Fig. 12b), Phoenix introduces almost no extra overhead compared to *No Seccomp* (which provides no protection) and the *Default Seccomp*, which generates significantly more false positives, since its blocking is based on a set (instead of sequence) of system calls, as shown in Table II (1-call). On the other hand, the naive Ptrace solution at least doubles the response time (e.g., 160% increase from 12.7 ms to 33 ms on Django). As shown in Fig. 12c, in the DoS case, the overhead introduced by Phoenix largely depends on the application but remains acceptable in most cases. The overhead ranges from negligible, less than Ptrace (e.g., Django), to higher than Ptrace (e.g., MySQL). Particularly, we observe that database applications, such as MySQL or Redis, usually experience larger overhead than web servers (especially Python-based servers, such as Django and Flask). The reason is that the DoS case assumes one attack for every request (which is already unrealistic), which is further amplified by high-performance applications such as MySQL or Redis, as these typically make use of multithreading and can cause a very high rate of system calls being issued. Such an extremely high rate of attacks causes the otherwise negligible overhead of updating the Seccomp filters to become more dominant than even deep argument inspection. Therefore, a viable solution here is for Phoenix to automatically switch to a Ptrace-only mode once

the rate of attacks exceeds a pre-determined threshold, which we regard as future work. Additionally, Fig. 12d depicts the average response time of different solutions over all applications. On average, Phoenix introduces only 4% additional delay (when compared to the Default Seccomp) whereas Ptrace increases it by almost 145%. Therefore, we conclude that using Phoenix to block sequences of system calls is practical in terms of the overhead on response time.

2) CPU/Memory consumption: We evaluate the resource consumption of both the protected application and our solution. We measure CPU and memory consumption using the `psutil` Python library. CPU consumption comprises both user and system CPU time, and memory usage is measured as the Unique Set Size (USS, i.e., the amount of memory that is unique to the process) as recommended in [63]. Table VI depicts our results. We study the baseline (*No Seccomp*) and three security solutions (*Default Seccomp*, Ptrace, and Phoenix). For each solution, we evaluate both the footprint of the solution itself (in the *Solution* column) and the application (in the *Application* column). We observe that none of the solutions introduces significant CPU or memory consumption on the protected application. In the idle case and under normal attack rates, Phoenix CPU consumption remains around 0%, while Ptrace continuously consumes more than 5% (i.e., as much as the application itself). This represents a save of 98% by Phoenix. During the DoS simulation, the CPU consumption of Phoenix peaks at almost 7%. Note that this happens only exceptionally (in case of DoS attack), while Ptrace CPU consumption is slightly better but constant regardless of the environment. Memory consumption is constant and mostly similar for Ptrace and our solution (where we test both solutions for memory leaks to ensure memory consumption does not increase over time). We conclude that our solution incurs negligible overhead in terms of resource consumption in most cases.

	App.	Normal (% FP)	Attack (% TP)
VtPath	Nginx	7.37	100
	Tomcat	26.84	100
	Redis	0.91	100
Mutz et al.	Nginx	0.56	100
	Tomcat	0.34	100
	Redis	0.11	100
PoLPer	Nginx	0 (8/8)	100 (6/6)
	Tomcat	0*	100 (6/6)
	Redis	0*	100 (6/6)
Phoenix	Nginx	0	100
	Tomcat	0	100
	Redis	0	100

*: program does not invoke *setuid* calls

TABLE V: Comparison of Phoenix with existing stateful solutions for anomaly detection (whitelisting)

		App.		Solution	
		CPU (%)	Mem. (MB)	CPU (%)	Mem. (MB)
No Seccomp		5.01	58.01	N/A	N/A
Default Seccomp		5.03	58.01	N/A*	N/A*
Ptrace	(idle)	4.98	57.75	5.18	0.58
	(normal)	5.02	57.90	5.18	0.57
	(DoS)	5.01	57.82	5.24	0.59
Phoenix	(idle)	4.97	57.73	0.03	0.63
	(normal)	5.02	57.82	0.1	0.64
	(DoS)	5.01	57.71	6.82	0.66

*: not collected as Seccomp does not execute in a separate kernel thread

TABLE VI: Average CPU and memory consumption of the application and the solutions (Seccomp, Ptrace, and Phoenix)

3) Impact of sequence on overhead: To protect containers, Phoenix takes as input a sequence of system calls of various lengths (depending on the CVE and on the results of the investigation). In this experiment, we assess how the length of such sequences impacts the performance of Phoenix. Figure 13 shows the average response time when protecting an application from sequences with sizes varying from two to nine system calls. The length of the sequence has no visible effect in the idle case (light grey) as the average response time remains at 5.3 ms, while a normal attack rate (black) increases the response time by less than 0.1 ms. Under DoS (dark grey), the average response time remains stable at 6.8 ms for sequences of length two, three, and four, then increases slowly to reach 10.4 ms with nine system calls. This increase in response time is mostly due to Phoenix updating the Seccomp filter for each system call in the sequence. Therefore, filtering longer sequences implies more updates per request. We conclude that the performance overhead of Phoenix remains acceptable even for relatively long sequence lengths (nine system calls).

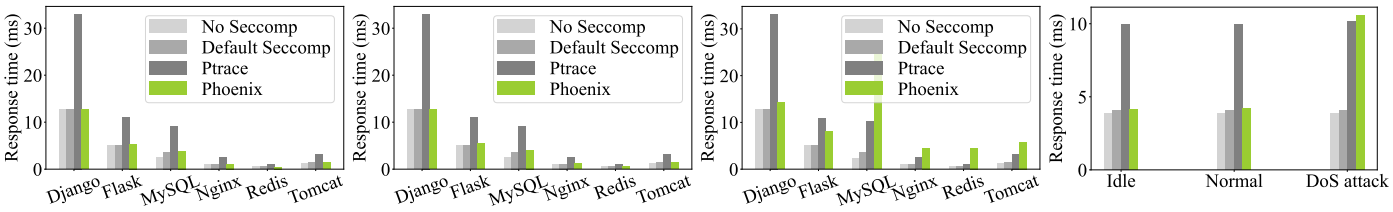
C. Evaluation of Malicious Sequence Identification

To answer RQ3, we perform two experiments. First, we compare the sequences of system calls identified following the Phoenix approach (as described in Section III-B) with the results of several existing solutions in the literature. Second, we perform a user study on the usability of the Phoenix approach.

1) Comparison with existing solutions: We compare Phoenix with several existing tools, signature-based solutions involving system calls (note the malicious sequence of system calls is essentially used by Phoenix as an attack signature, whereas our experimental results in Section V-A3 already show anomaly detection is not suitable for this purpose), and provenance analysis solutions. First, Strace/Auditd/Sysdig [75] are common tools that can be used to identify system calls invoked by an exploit, which provides a baseline for comparison. Second, Nimos [72] identifies system call sequences by applying Generalized Sequential Patterns (GSP) mining to learn n-grams from a database of kernel exploits. Third, Madani et al. [48] adopt a similar approach to extract malicious system calls from malware. Finally, CLARION/SPADE [8], [23] and CamFlow [60] generate provenance graphs that can be used for identifying system calls, either manually or using semi-automated tools such as DepImpact [20] and ATLAS [1].

To facilitate the comparison to Phoenix, we adapt the available data and code of those existing solutions as follows. For Strace/Auditd/Sysdig (which provide the same result), we extract the system call sequence generated while running the exploit code (in contrast, Phoenix does not require to have/know the exploit code). For Nimos, as it is based on sequence mining [72], we repeat the original exploit to generate a large number of exploit traces such that it can extract the longest frequent pattern (with a support of 0.5). For Madani et al., we remove duplicate consecutive system calls as recommended by the authors [48]. Finally, we extract system calls from both the original provenance graphs generated by CLARION/SPADE, and the subgraphs after pruning by DepImpact [20] to compare to Phoenix.

Table VII reports the number of system calls produced by each solution as well as the false positives and false negatives (i.e., the system calls not relevant to the CVE, and the relevant ones missing in the result, respectively) to demonstrate how close those automated solutions can get to the result of Phoenix. First, for all the five CVEs, Strace/Auditd/Sysdig produce a large number of system calls (ranging from 80 to more than 18,000), since they consider all system calls invoked by the exploit code as relevant to the attack (which explains why these solutions cause no false negative). However, this causes a high false positive rate, as most of the system calls are not representatives of the attack (e.g., memory allocation, library loading, and process cloning). Therefore, even by



(a) Idle (b) Normal (c) DoS attack (d) Average on all applications

Fig. 12: Overhead of different solutions in terms of response time on various container applications

assuming the exploit code is available, those tools still do not provide a viable solution for human analysts to identify the malicious system calls. Second, while both Nimos and Madani et al. produce a small number of system calls, they also cause a relatively large number of false negatives, e.g., Madani et al. misses 7 out of the 8 relevant system calls for CVE-2022-0847. This can be explained by the fact that they focus on identifying the most frequent pattern of system calls in the exploit code, which is not necessarily relevant to the attack. As none of the exploit codes in this experiment involves a repetitive pattern of invoking system calls, those solutions become inaccurate. Also, unlike Strace/Auditd/Sysdig, the small output size of those solutions leaves little room for analysts to refine their results.

	2017-6074			**2021-4154**			**2022-0847**			**2023-0386**			**2023-32233**		
	Size	*FP*	*FN*	*Size*	*FP*	*FN*	*Size*	*FP*	*FN*	*Size*	*FP*	*FN*	*Size*	*FP*	*FN*
S/A/S	329	323	0	457	453	0	80	72	0	>18k	>18k	0	>7k	>7k	0
Nimos	4	4	6	3	2	3	4	1	5	4	4	8	4	3	5
Madani	3	0	0	3	3	4	3	2	7	3	1	5	4	3	5
CLARION	310	304	0	>1k	>1k	0	777	769	0	>29k	>29k	0	>4k	>4k	0
DepImpact	25	16	0	N/A			44	33	0	149	117	0	N/A		
Phoenix	6	0	0	4	0	0	8	0	0	8	0	0	8	0	0

TABLE VII: Comparison of system calls identified using Phoenix and existing solutions (S/A/S: Strace/Auditd/Sysdig)

Third, CLARION reports an even larger output size than Strace/Auditd/Sysdig due to the fact that it performs system-wide provenance data collection (at the kernel level), whereas Strace/Auditd/Sysdig only collect system calls related to the (exploit) process. Consequently, although CLARION's output contains no false negatives, it contains a large number of false positives corresponding to background activities unrelated to the exploit code. On the other hand, as CLARION (and other provenance graph-based tools) does not require the exploit code, it is more practical than Strace/Auditd/Sysdig in a real-world environment where the exploit code is not available. Finally, DepImpact produces the closest results to those of Phoenix, i.e., smaller output sizes, fewer FPs, and no FN, which provides a good basis for analysts to further refine the results. Nonetheless, we were able to run DepImpact on only 3 out of the 5 CVEs (for the other two CVEs, a Java error beyond our control was returned). In summary, those results clearly support the semi-automated approach of Phoenix, which combines the strength of automated tools (e.g., CLARION for graph generation and DepImpact for graph pruning) and the knowledge of human experts.

2) User study: To assess the usability of the Phoenix approach for identifying malicious sequences of system calls, we conduct a user study[4] in which participants unfamiliar with the chosen vulnerabilities are asked to identify candidate sequences of relevant system calls (as subgraphs) following the Phoenix approach. All participants have a general background in security but limited knowledge/experience in provenance analysis and Linux kernel security (similar to real-world administrators). Each participant is given the original provenance graph (displayed in Neo4j [58]) with the alert highlighted, the access to our GUI tool and DepImpact [20], and an explanation of the Phoenix approach and the task. The first two columns (for each CVE) of Table VIII report the time taken by each participant (T1) and the size of the identified subgraph (number of nodes plus edges), respectively. Also, to measure the quality of those subgraphs, the last two columns (for each CVE) report the time taken by an expert (another participant who has more expertise and prior knowledge about the chosen vulnerabilities) to extract the correct sequence from the subgraph, and the ratio of extracted system calls (when the correct sequence is not fully present in the subgraph). The results indicate that most participants take less than 30 minutes (only one instance of >30 minutes) to identify reasonably small subgraphs (ranging between 62 to 634 nodes/edges) for all four CVEs. Those subgraphs represent good results as the expert only takes a few minutes (<6 minutes) to extract the correct sequence in most cases. Some vulnerabilities seem slightly more challenging for certain participants (evidenced by the lower ratios), which clearly motivates (and can be easily addressed by) crowdsourcing.

	2021-4154				**2022-0847**				**2023-0386**				**2023-32233**			
	T1	*Size*	*T2*	*TP*	*T1*	*Size*	*T2*	*TP*	*T1*	*Size*	*T2*	*TP*	*T1*	*Size*	*T2*	*TP*
#1	22′	249	2′	13/13	9′	62	1′	12/12	21′	233	4′	10/12	23′	128	3′	17/17
#2	24′	154	2′	11/13	34′	218	3′	11/12	8′	203	4′	10/12	10′	232	3′	15/17
#3	25′	250	4′	13/13	18′	63	1′	9/12	12′	495	6′	4/12	11′	182	2′	17/17
#4	15′	187	4′	12/13	25′	413	5′	5/12	14′	376	3′	2/12	17′	92	2′	11/17
#5	11′	169	3′	12/13	12′	498	6′	9/12	18′	162	2′	10/12	5′	62	1′	1/17
#6	14′	413	7′	11/13	24′	245	2′	8/12	7′	634	6′	10/12	9′	585	5′	17/17
#7	18′	278	5′	13/13	29′	376	5′	12/12	30′	491	6′	10/12	18′	258	3′	17/17
Avg.	18′	243	4′	12/13	22′	268	3′	9/12	16′	370	4′	8/12	13′	220	3′	14/17

T1: time taken by a user to identify candidate sequences (subgraph)
T2: time taken by an expert to extract sequence from the user's result
Original graph sizes: 1.7k, 227k, 123k, 4k

TABLE VIII: Results of a user study on the usability of Phoenix approach for identifying malicious sequences of system calls

D. Deep Argument Inspection (DAI)

To answer RQ4, we study the impact of deep argument inspection (DAI) both in terms of security benefit and overhead. While our previous experiments show the combined results of Seccomp and Ptrace, here we study the impact of DAI itself, as DAI is a unique feature of Phoenix not shared by other solutions (i.e., default Seccomp, Confine, and Sysfilter).

1) Security: We measure the impact of DAI (in inspecting up to two arguments of each system call in the sequence) on

[4]The Office of Research Ethics of our university has identified this study as for quality assurance purpose and hence exempted it from ethics approval.

security by evaluating the number of false positives of our solution on the CB-DS dataset for different vulnerabilities. For each vulnerability, we measure the false positive rate when matching system calls and their arguments (using DAI) and compare it to the false positive rate when matching system calls only (as presented in Table II). We report the percentage of reduction between these two experiments (i.e., how many fewer false positives we obtain due to DAI). We perform this experiment both when blocking individual system calls (1-syscall), and when blocking sequences of two and three system calls. Table IX depicts the corresponding results for eight of the previously studied CVEs (as described in Table I). On average, matching each individual system call and its arguments results in a false positive reduction of 77.4% on average. When blocking sequences of two and three system calls, the use of DAI allows to greatly reduce the false positives by 94% and 97.2% on average, respectively. Finally, blocking a sequence of three system calls with DAI completely removes the false positives for CVE-2017-5123 and CVE-2022-0847. We conclude that the added DAI feature of Phoenix is critical for reducing false positives compared to existing Seccomp-based solutions.

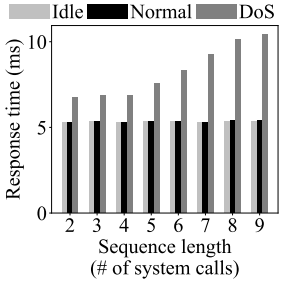


Fig. 13: Impact of sequence length on Phoenix performance

CVE	CB Dataset 1-syscall	2-syscall	3-syscall
2017-5123	77.3%	99.9%	100%*
2017-17053	74.1%	97.5%	99.3%
2022-0492	72.2%	75.4%	88.4%
2017-11176	76.4%	90.5%	97.0%
2018-14634	77.9%	97.1%	99.1%
2021-3347	79.2%	92.9%	94.6%
2021-4154	84.9%	98.5%	99.4%
2022-0847	77.1%	99.9%	100%*
Average	77.4%	94.0%	97.2%

*: 0 false positive using DAI

TABLE IX: Reduction of the false positive rate due to DAI

2) Performance: We study the impact of DAI on the performance of the protected application. Figure 14 depicts the response time of applications protected by Phoenix and the Ptrace solution. While idle or under normal attack rate (Fig. 14a and Fig. 14b), our solution shows substantially less overhead than its Ptrace counterpart, and the use of DAI does not significantly affect its performances. Similar to Fig. 12, the overhead of Phoenix under DoS attacks (Fig. 14c) depends on the application, although the use of DAI consistently introduces negligible overhead (e.g., at most 0.32 ms on MySQL). Fig. 14d shows that on average, our solution introduces 50% less and 1% more delay than Ptrace in the normal and DoS cases, respectively. We conclude that the added DAI feature allows Phoenix to benefit from the security advantage of DAI (as shown in Table IX) while keeping the response time low.

3) CPU/Memory usage: We measure the CPU and memory usage incurred by the use of DAI. Specifically, we measure the CPU and memory usage of both Ptrace and Phoenix when using DAI. Table X presents our results as the difference of CPU consumption (in % of CPU time) and memory consumption (in MB) for both the application and the solution employed, with and without DAI. Results show that the use of DAI in both Ptrace and Phoenix incurs little or no extra consumption in the idle and normal cases, although Ptrace introduces an extra +0.9% of CPU usage. However, consistent with previous results, the overhead incurred by Phoenix's DAI under DoS attacks (+1.21) is higher than that of Ptrace (+0.14) but still negligible.

Solution		Application CPU (%)	Application Memory (MB)	Solution CPU (%)	Solution Memory (MB)
Ptrace	(idle)	−0.04	−0.17	+0.12	+0.02
	(normal)	−0.04	−0.17	+0.12	+0.03
	(DoS)	+0.05	−0.28	+0.14	+0.02
Phoenix	(idle)	−0.06	−0.07	+0.02	−0.04
	(normal)	−0.07	−0.06	+0.03	−0.03
	(DoS)	−0.07	+0.21	+1.21	−0.09

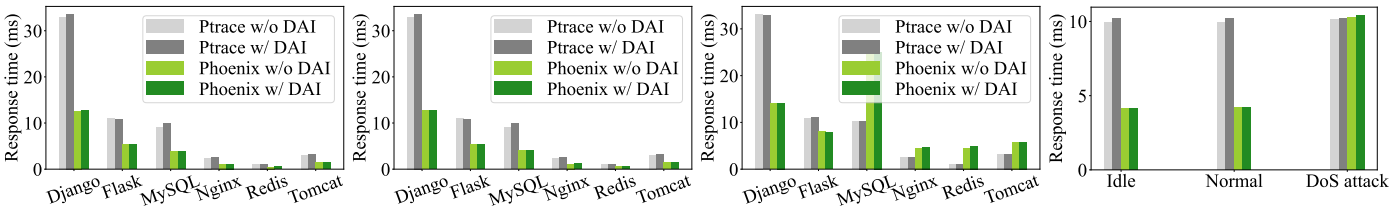
TABLE X: Additional CPU and memory consumption of DAI on the application and the solutions (Ptrace and Phoenix)

VI. Discussions

Positioning and Interoperability. Phoenix can work in tandem with other security solutions in surviving unpatched vulnerabilities. For instance, it can complement existing works that block unnecessary system calls [11], [24]–[26], [36], [40] by further blocking common system calls shared between vulnerabilities and applications, and it will not interfere with those as it only updates (instead of replacing) Seccomp filters. Phoenix also complements attack detection (e.g., Falco [17]) and analysis [1], [8], [20], [29], [30], [81] solutions by turning their results into security actions to prevent recurring attacks. Finally, the temporary protection of Phoenix is not meant to replace, but to help techniques for developing security patches through gaining more time for the vendors.

Evasive and DoS Attacks. Attackers aware of Phoenix may attempt to defeat it by evading its protection through varying the attack behavior (e.g., slowing down the attack or mixing different attacks) or mimicking normal behavior (e.g., with equivalent system calls, modified arguments, or injected irrelevant system calls). Attackers may also target Phoenix itself by causing unacceptable performance overhead or false positives. First, Phoenix already offers some protection against such attacks, e.g., the existing sequence abstraction feature can tolerate variation in contextual parameters such as file names or port numbers, and the deep argument inspection feature can avoid false positives with similar system calls but different arguments (Section III-A). Second, although our discussions focus on a single sequence of system calls, the fact that Phoenix employs FSM for matching means it can support more general forms of attack representation such as regular expressions to defeat mimicry attacks employing equivalent system calls [79]. Moreover, Phoenix can leverage provenance techniques specifically designed for long-lasting APTs [1], [29], [30], [54], [55] to tackle slow attacks, and deploy multiple agents to monitor mixed or concurrent attacks. Finally, our experimental results (Section V-B) show Phoenix is resilient against DoS attacks employing high-rate requests.

However, Phoenix may also be improved in this regard. First, the FSM model used by Phoenix may be replaced with other more expressive models and well-known mechanisms for capturing attack signatures (e.g., colored Petri nets, hidden Markov models). This can help Phoenix address more general cases such as multi-threading applications in which the system calls to be blocked may be partially ordered instead of appearing in a sequence (total order). Second, our abstraction mechanism is only designed for variations naturally induced by OS (e.g., PIDs and filenames), but not designed for addressing malicious variations induced in evasive attacks. An interesting direction is to leverage more powerful abstraction



(a) Idle (b) Normal (c) DoS attack (d) Average on all applications

Fig. 14: Overhead of DAI on various container applications

mechanisms to cover a broader range of cases including evasive attacks in which the exploits are deliberately modified to vary the system calls and arguments. Third, crowdsourcing between all affected users (similarly to the way the Snort community shares rules) provides another promising solution to address evasive and evolving attacks since it allows the affected users to timely exchange information about the latest variants of an exploit, and consequently update the sequences of system calls and arguments to capture such variants, until the official patch is released by the vendor.

Practicality of Sequence Identification. Our experiments and user study clearly indicate that the Phoenix approach for identifying malicious sequences of system calls is practical. Moreover, through crowdsourcing, it only takes one expert or vendor to succeed in identifying that sequence, which can then be shared with all the users affected by this vulnerability to block it with Phoenix (even if they are not yet aware of this vulnerability). Finally, as explained in Section III-A, potential inaccuracies in the malicious sequence can be mitigated with a more conservative action plan (e.g., warning instead of blocking) to avoid service disruption. Therefore, Phoenix provides a practical approach that could significantly reduce the attack window, compared to relying on the vendor to develop its official security patch.

Scalability. Phoenix is scalable for large-scale applications due to the following facts. First, the size of its FSM model is bounded by the number of system calls involved in each CVE (our observation shows most real-world CVEs involve no more than 10 system calls). Second, the number of sequences monitored by Phoenix at the same time (which is equal to the number of FSM models) is bounded by the number of unpatched CVEs inside a given system, which is typically small in practice. Third, Phoenix only looks for one system call per CVE and per FSM at any given time, and most system calls will be quickly dismissed by Seccomp.

Beyond Containers. Although our implementation/experiments focus more on containers (as they have become both popular among providers and attractive to attackers), our general methodology can potentially apply to any OS process, and hence an interesting future direction is to explore its application beyond container environments. Particularly, Phoenix can function as long as a Seccomp filter can be applied to the process (which is the case in all Linux kernels no earlier than v4.14). Our methodology can also be extended to other operating systems that implement an in-kernel system call restriction mechanism, such as `pledge()` in OpenBSD [62].

Vulnerability Coverage. Although we have mostly focused on kernel-related vulnerabilities in our evaluation, Phoenix can potentially be applied to any vulnerability that invokes a sequence of system calls. There is strong evidence that kernel exploitation usually requires multiple system calls, in the kernel fuzzing literature (e.g., [22], [59], [74]), kernel exploit literature (e.g., [13], [82], [83]), and IDS literature (e.g., [32], [69]), e.g., "even the state-of-the-art kernel exploits would require multiple system calls to compromise the execution" [13].

VII. Related Work

This section reviews related works on container security, system call security, and provenance. First, container security has recently attracted attention due to its widespread adoption. Existing works [24], [26] block unused system calls in containerized applications by first analyzing the applications beforehand to generate an over-approximation of the required system calls and then blacklisting the rest. Sysfilter [11] and ChestNut [7] employ a similar approach but to protect applications in general. SPEAKER [40] and Ghavamnia et al. [25] reduce the attack surface of containers by considering different Seccomp profiles during their booting and running phases. The authors of C2C [26] and [36] alternatively propose configuration-driven hardening solutions. While those approaches are effective in reducing the attack surface by restricting unnecessary system calls, none of them is designed to block vulnerabilities exploiting common system calls required by the benign applications, which is the main focus of Phoenix.

SFIP [6] captures the transition of applications' system calls using digraphs through static analysis, and then ensures the integrity of such transitions using a modified Linux kernel (in contrast, Phoenix simply requires a kernel module to be loaded). Sifter [33] filters system calls for Android kernel drivers based on eBPF through a whitelisting approach (in contrast, Phoenix focuses on blacklisting). There also exist solutions that perform stateful inspection of system calls based on call stack information [21], [34], [57]. Specifically, VtPath [21] leverages the so-called virtual paths (call stack between pairs of consecutive system calls) to learn the normal behavior and detect anomalous calls, while Mutz et al. [57] also considers system calls arguments. PoLPer [34] blocks anomalous system calls in the *setuid* family by learning the calling process hierarchy, call context, and its arguments.

SPADE [23] and CamFlow [60] can collect, process, and store system provenance data, while CLARION [8] extends such capabilities to containers by adding Linux namespaces handling and addressing potential fragmentation and ambiguity issues. Following the idea of backtracking intrusions [35], numerous works (e.g., [1], [20], [29], [30], [54], [55], [77], [81]) explore the use of provenance data to investigate and understand complex attacks. Particularly, HOLMES [55], POIROT [54], ATLAS [1], Unicorn [29], and NoDoze [30]

focus on identifying advanced persistent threats (APT) attacks using provenance graphs. DepImpact [20] analyzes large provenance graphs to extract a subgraph given an initial point of interest (e.g., an alert). ProvTalk [77] correlates provenance data at multiple semantic levels. ProvDetector [81] detects stealthy malware by measuring a local outlier factor in provenance graphs. As discussed in Section VI, Phoenix is complementary to those provenance-based approaches as it leverages them to identify malicious sequences of system calls with a different objective of preventing unpatched vulnerabilities.

VIII. Conclusion

We proposed Phoenix, a novel solution for temporarily protecting containers against unpatched vulnerabilities while awaiting for an official patch. Specifically, we developed a solution for accurately and efficiently blocking a sequence of system calls by combining Seccomp and Ptrace. We also designed an approach for identifying such a sequence from vulnerabilities through provenance analysis. Our implementation and evaluation using CVEs and real data showed that Phoenix could effectively mitigate vulnerabilities under popular applications with negligible delay and overhead.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under the Industrial Research Chair in SDN/NFV Security, and the Canada Foundation for Innovation under JELF Project 38599.

References

[1] A. Alsaheel, Y. Nan, S. Ma, L. Yu, G. Walkup, Z. B. Celik, X. Zhang, and D. Xu, "ATLAS: A Sequence-based Learning Approach for Attack Investigation." in *USENIX Security Symposium*, 2021.

[2] "Aqua Security Cloud Native Threat Report." 2021. [Online]. Available: https://info.aquasec.com/hubfs/Threat%20reports/AquaSecurity_Cloud_Native_Threat_Report_2021.pdf

[3] Avrahami, Yuval, "Container Escape - CVE-2022-0492." 2022. [Online]. Available: https://unit42.paloaltonetworks.com/cve-2022-0492-cgroups/

[4] M. Barre, A. Gehani, and V. Yegneswaran, "Mining Data Provenance to Detect Advanced Persistent Threats." in *International Workshop on Theory and Practice of Provenance (TaPP)*, 2019.

[5] G. Berrada and J. Cheney, "Aggregating Unsupervised Provenance Anomaly Detectors." in *International Workshop on Theory and Practice of Provenance (TaPP)*, 2019.

[6] C. Canella, S. Dorn, D. Gruss, and M. Schwarz, "SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems." *arXiv preprint arXiv:2202.13716*, 2022.

[7] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating Seccomp Filter Generation for Linux Applications." in *ACM SIGSAC Cloud Computing Security Workshop (CCSW)*, 2021.

[8] X. Chen, H. Irshad, Y. Chen, A. Gehani, and V. Yegneswaran, "CLARION: Sound and Clear Provenance Tracking for Microservice Deployments." in *USENIX Security Symposium*, 2021.

[9] CNCF, "CNCF Annual Survey 2022." 2022. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2022/

[10] Corbet, Jonathan, "Unexporting kallsyms_lookup_name." 2020. [Online]. Available: https://lwn.net/Articles/813350/

[11] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated System Call Filtering for Commodity Software." in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[12] "Docker Slim," 2023. [Online]. Available: https://slimtoolkit.org/

[13] Y. Du, Z. Ning, J. Xu, Z. Wang, Y.-H. Lin, F. Zhang, X. Xing, and B. Mao, "Hart: Hardware-Assisted Kernel Module Tracing on ARM." in *European Symposium on Research in Computer Security (ESORICS)*, 2020.

[14] G. Duan, Y. Fu, M. Cai, H. Chen, and J. Sun, "DongTing: A Large-scale Dataset for Anomaly Detection of the Linux Kernel." *Journal of Systems and Software*, vol. 203, no. C, p. 111745, 2023.

[15] A. El Khairi, M. Caselli, C. Knierim, A. Peter, and A. Continella, "Contextualizing System Calls in Containers for Anomaly-Based Intrusion Detection." in *ACM SIGSAC Cloud Computing Security Workshop (CCSW)*, 2022.

[16] "2017 Equifax Data Breach." 2022. [Online]. Available: https://en.wikipedia.org/wiki/2017_Equifax_data_breach

[17] "Falco." 2018. [Online]. Available: https://falco.org/

[18] "CVE-2022-0847: Linux Local Privilege Escalation." [Online]. Available: https://sysdig.com/blog/cve-2022-0847-dirty-pipe-sysdig/

[19] "Falco Rules Supported Fields." 2023. [Online]. Available: https://falco.org/docs/reference/rules/supported-fields/

[20] P. Fang, P. Gao, C. Liu, E. Ayday, K. Jee, T. Wang, Y. F. Ye, Z. Liu, and X. Xiao, "Back-Propagating System Dependency Impact for Attack Investigation." in *USENIX Security Symposium*, 2022.

[21] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly Detection using Call Stack Information." in *IEEE Symposium on Security and Privacy (S&P)*, 2003.

[22] M. Fleischer, D. Das, P. Bose, W. Bai, K. Lu, M. Payer, C. Kruegel, and G. Vigna, "ACTOR: Action-Guided Kernel Fuzzing." in *USENIX Security Symposium*, 2023, pp. 5003–5020.

[23] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments." in *ACM/IFIP/USENIX International Middleware Conference*, 2012.

[24] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction." in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[25] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal System Call Specialization for Attack Surface Reduction." in *USENIX Security Symposium*, 2020.

[26] S. Ghavamnia, T. Palit, and M. Polychronakis, "C2C: Fine-grained Configuration-driven System Call Filtering." in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[27] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient Context-Sensitive Intrusion Detection." in *Network and Distributed System Security Symposium (NDSS)*, 2004.

[28] Grubb, Steve, "Linux Audit Daemon." 2023. [Online]. Available: https://linux.die.net/man/8/auditd

[29] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats." in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[30] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage." in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[31] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch, "The Race to the Vulnerable: Measuring the Log4j Shell Incident." in *IFIP Network Traffic Measurement and Analysis Conference (TMA)*, 2022.

[32] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion Detection using Sequences of System Calls." *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[33] H.-W. Hung, Y. Liu, and A. A. Sani, "Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction." in *Annual International Conference on Mobile Computing And Networking (MobiCom)*, 2022.

[34] Y. Jeon, J. Rhee, C. H. Kim, Z. Li, M. Payer, B. Lee, and Z. Wu, "Polper: Process-Aware Restriction of Over-Privileged Setuid Calls in Legacy Applications." in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2019.

[35] S. T. King and P. M. Chen, "Backtracking Intrusions." *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 1, pp. 51–76, 2005.

[36] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-Driven Software Debloating." in *European Workshop on Systems Security (EuroSec)*, 2019.

[37] "Linux Kernel Probes (Kprobes)." 2023. [Online]. Available: https://docs.kernel.org/trace/kprobes.html

[38] "Restrict a Container's Syscalls." 2023. [Online]. Available: https://kubernetes.io/docs/tutorials/security/seccomp/

[39] "Kunwind." 2016. [Online]. Available: https://github.com/giraldeau/kunwind

[40] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "SPEAKER: Split-phase Execution of Application Containers." in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[41] Z. Li, Q. A. Chen, R. Yang, Y. Chen, and W. Ruan, "Threat Detection and Investigation with System-level Provenance Graphs: A Survey." *Computers & Security*, vol. 106, p. 102282, 2021.

[42] Z. Lin, Y. Wu, and X. Xing, "DirtyCred: Escalating Privilege in Linux Kernel." in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[43] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Host-Based Intrusion Detection System with System Calls: Review and Future Trends." *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[44] P. Liu, S. Ji, L. Fu, K. Lu, X. Zhang, W.-H. Lee, T. Lu, W. Chen, and R. Beyah, "Understanding the Security Risks of Docker Hub." in *European Symposium on Research in Computer Security (ESORICS)*, 2020.

[45] "Log4Shell." Nov 2022. [Online]. Available: https://en.wikipedia.org/wiki/Log4Shell

[46] Lohmann, Niels, "JSON for Modern C++." 2022. [Online]. Available: https://json.nlohmann.me/

[47] S. Lv, J. Wang, Y. Yang, and J. Liu, "Intrusion Prediction with System-call Sequence-to-Sequence Model." *IEEE Access*, vol. 6, pp. 71 413–71 421, 2018.

[48] P. Madani and N. Vlajic, "Towards Sequencing Malicious System Calls." in *IEEE Conference on Communications and Network Security (CNS)*, 2016.

[49] F. Maggi, M. Matteucci, and S. Zanero, "Detecting Intrusions through System Call Sequence and Argument Analysis." *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 7, no. 4, pp. 381–395, 2008.

[50] "`Seccomp_rule_add` Man Page." [Online]. Available: https://man7.org/linux/man-pages/man3/seccomp_rule_add.3.html

[51] Maurer, Norman, "Netty Java Library Supports Splice() System Call." 2015. [Online]. Available: https://github.com/netty/netty/commit/ca250c8a4b1308fb8ff5263ee03cd1bde761861d

[52] Max Kellermann, "The Dirty Pipe Vulnerability." 2022. [Online]. Available: https://dirtypipe.cm4all.com/

[53] McCune, Rory, "Container Escape - CVE-2022-0185." 2022. [Online]. Available: https://blog.aquasec.com/cve-2022-0185-linux-kernel-container-escape-in-kubernetes

[54] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "POIROT: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting." in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[55] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: Real-time APT Detection through Correlation of Suspicious Information Flows." in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[56] Mountain, Eric and Tafani-Dereeper, Christophe and McCormick, Tommy and Baguelin, Frederic, "Container Escape with CVE-2022-0847." 2022. [Online]. Available: https://securitylabs.datadoghq.com/articles/dirty-pipe-container-escape-poc/

[57] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer, "Exploiting Execution Context for the Detection of Anomalous System Calls." in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[58] Neo4j, "Neo4j - the world's leading graph database," 2012. [Online]. Available: http://neo4j.org/

[59] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation." in *USENIX Security Symposium*, 2018.

[60] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical Whole-System Provenance Capture." in *Symposium on Cloud Computing (SoCC)*, 2017.

[61] T. Pasquier, J. Singh, D. Eyers, and J. Bacon, "CamFlow: Managed Data-sharing for Cloud Services." *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 472–484, 2015.

[62] "OpenBSD's pledge()." 2023. [Online]. Available: https://man.openbsd.org/pledge.2

[63] "*psutil* Memory Full Info." 2023. [Online]. Available: https://psutil.readthedocs.io/en/latest/index.html#psutil.Process.memory_full_info

[64] "Ptrace Man Page." 2023. [Online]. Available: https://man7.org/linux/man-pages/man2/ptrace.2.html

[65] Y. Roumani, "Patching Zero-day Vulnerabilities: an Empirical Analysis." *Journal of Cybersecurity*, vol. 7, no. 1, 2021.

[66] "Seccomp Man Page." 2021. [Online]. Available: https://man7.org/linux/man-pages/man2/seccomp.2.html

[67] "Seccomp for Docker." 2023. [Online]. Available: https://docs.docker.com/engine/security/seccomp

[68] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-based Method for Detecting Anomalous Program Behaviors." in *IEEE Symposium on Security and Privacy (S&P)*, 2001.

[69] A. Sharma, A. K. Pujari, and K. K. Paliwal, "Intrusion Detection Using Text Processing Techniques with a Kernel Based Similarity Measure." *Computers & Security*, vol. 26, no. 7-8, pp. 488–495, 2007.

[70] X. Shu, D. Yao, and B. G. Ryder, "A Formal Framework for Program Anomaly Detection." in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[71] J. Simonsson, L. Zhang, B. Morin, B. Baudry, and M. Monperrus, "Observability and Chaos Engineering on System Calls for Containerized Applications in Docker." *Future Generation Computer Systems*, vol. 122, pp. 117–129, 2021.

[72] S. Song, S. Suneja, M. V. Le, and B. Tak, "On the Value of Sequence-Based System Call Filtering for Container Security." in *IEEE International Conference on Cloud Computing (CLOUD)*, 2023.

[73] "Strace." 2022. [Online]. Available: https://linux.die.net/man/1/strace

[74] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "Healer: Relation Learning Guided Kernel Fuzzing." in *ACM SIGOPS Symposium on Operating Systems Principles (SOPS)*, 2021.

[75] "Sysdig." 2023. [Online]. Available: https://sysdig.com/

[76] "Sysdig Docker Vulnerability Scanning." [Online]. Available: https://sysdig.com/learn-cloud-native/container-security/docker-vulnerability-scanning

[77] A. Tabiban, H. Zhao, Y. Jarraya, M. Pourzandi, M. Zhang, and L. Wang, "ProvTalk: Towards Interpretable Multi-Level Provenance Analysis in Networking Functions Virtualization (NFV)." in *Network and Distributed System Security Symposium (NDSS)*, 2022.

[78] The kernel development community, "Seccomp BPF." 2022. [Online]. Available: https://www.kernel.org/doc/html/v5.10/userspace-api/seccomp_filter.html

[79] D. Wagner and P. Soto, "Mimicry Attacks on Host-Based Intrusion Detection Systems." in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2002.

[80] P. Wang and C. Johnson, "Cybersecurity Incident Handling: a Case Study of the Equifax Data Breach." *Issues in Information Systems*, vol. 19, no. 3, pp. 150–159, 2018.

[81] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter, and H. Chen, "You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis." in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[82] W. Wu, Y. Chen, X. Xing, and W. Zou, "KEPLER: Facilitating Control-Flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities." in *USENIX Security Symposium*, 2019.

[83] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities." in *USENIX Security Symposium*, 2018.