Fast and Service-preserving Recovery from Malware Infections Using CRIU

Ashton Webster
University of Maryland

Ryan Eckenrod
University of Maryland

James Purtilo
University of Maryland

Abstract

Once a computer system has been infected with malware, restoring it to an uninfected state often requires costly service-interrupting actions such as rolling back to a stable snapshot or reimaging the system entirely. We present CRIU-MR: a technique for restoring an infected server system running within a Linux container to an uninfected state in a service-preserving manner using Checkpoint/Restore in Userspace (CRIU). We modify the CRIU source code to flexibly integrate with existing malware detection technologies so that it can remove suspected malware processes within a Linux container during a checkpoint/restore event. This allows for infected containers with a potentially damaged filesystem to be checkpointed and subsequently restored on a fresh backup filesystem while both removing malware processes and preserving the state of trusted ones. This method can be quickly performed with minimal impact on service availability, restoring active TCP connections and completely removing several types of malware from infected Linux containers.

1 Introduction

Malware attacks remain a persistent threat to computer security from year to year. Symantec alone recorded over 20 billion malware alerts across customer machines during 2010-2011, while both botnet infections and particularly damaging ransomware attacks are growing in number annually [28, 38]. In response, the security community continues to develop intrusion prevention techniques meant to stop malware from propagating to new machines and intrusion detection systems (IDS's) meant to detect malicious processes running on computer hosts [15, 26, 27, 30, 41]. Despite these efforts, many malware infections go undetected and infect new hosts daily.

Once malware has been detected on a host, removing the malware and restoring the host to a trustworthy, unharmed state proves challenging. The malware removal and remediation capabilities of many commercial malware detectors fail to completely erase a malware program's effects [34]. Other recovery solutions record meticulous logs about the processes running on a system in order to rollback and then forward restore infected hosts [25, 33]. However, while these methods are more effective at removing and recovering from malware, they prove slow, memory and monitoring intensive, and are not known to be used in practice.

Virtual Machine (VM) based approaches can quickly restore an infected host to a known trustworthy state using snapshots, but doing so will lose the state of any computations or network connections that were running on the host unless costly logging is implemented as well [18, 35]. In light of these shortcomings, we seek to develop a malware recovery technique which can preserve the state of trusted services running on an infected host without the overhead of log-based schemes.

We present a malware recovery system which extends the Checkpoint/Restore In Userspace project (CRIU) [40] to quickly restore an infected Linux container (LXC) to a safe state while removing malware and preserving running services in the process. This technique, which we call CRIU for Malware Recovery (CRIU-MR), allows CRIU to be flexibly integrated with existing malware detection systems. When malware is detected on a Linux container, the container process and its children are first checkpointed with CRIU. Malware processes are identified during this step and marked to be ignored during the subsequent container restore. The container process can then be migrated and restored on a trustworthy backup filesystem with CRIU, excluding the identified malware processes. We show that CRIU-MR only takes 2.8 seconds to complete on average regardless of the type of malware infection across several Linux malware samples. We find CRIU-MR is primarily useful for hosts with filesystems such as web servers, preserving active network connections to the host without drastically increasing response latency. By quickly restoring running services while removing malware from a system, CRIU-MR presents a lightweight alternative to log-based and VM-based malware recovery schemes.

2 Related Work

Many techniques for recovering from malware infections have been proposed over time. We group these works into the following categories: tradtional, log-based, and VM-based. We discuss these categories, as well as CRIU and LXC, which CRIU-MR relies upon.

2.1 Traditional Malware Recovery

The most basic solution to malware remediation is to reinstall the infected host's operating system and reformat any disk drives. While this method is sure to remove the malware and its effects, it is obviously undesirable as it removes all data and processes on the host. Less destructive malware remediation techniques have thus been packaged into the signature-based antivirus programs typically installed on a computer host. Unfortunately, Passerini et al. [34] find that even when these programs detect malware, they may fail to remove malware executables for over 20% of infections. Furthermore, they typically fail to reverse secondary changes to the infected filesystem or changes to registry keys in the case of Windows hosts. While more effective malware remediation techniques have been developed, these solutions remain the most commonly used.

2.2 Log-Based Malware Recovery

Log-based recovery techniques, long used in database implementations [32], restore a system's state to a known stable state by using log information to undo undesired operations, correctly reapply valid changes, or both. The Taser [20] recovery system records all file, network, and process operations performed on the system and attempts to use such logs to undo the effects of a malware program once it is flagged by an IDS. Taser however will be forced to undo all operations logged on the system if the intrusion is not caught in a timely manner, and its recovery method can take many minutes to run in the worst cases. Hsu et al. [25] attempt to differentiate trusted and untrusted applications, logging only unstrusted ones in order to rollback their operations if necessary. The downside of this method is that untrusted processes are heavily restricted in terms of their filesystem resources and ability to interact with other processes, requiring user input in most cases for any program to run successfully. It additionally incurs significant runtime and logging overhead for each untrusted process.

Palieri et al. [33] develop a technique for automatically generating a remediation executable which can be run to reverse the effects of a given malware program. While mostly successful, these executables failed to reverse effects in some cases, can accidentally reverse valid changes, and fail to reverse changes an attacker may manually make if the malware provides shell access.

2.3 VM-based Malware Recovery

Modern VM hypervisors allow for "snapshots" of a system's filesystem and process state to be taken at any time, which can later be reverted to if necessary. If an older, malware-free snapshot is available, malware can be quickly removed from an infected VM by restoring the VM to the prior snapshot. The downside is that the operations/state of any trusted processes are lost when the snapshot is restored. While not a VM-based technique, MalTRAK [39] uses the concept of "views" or system snapshots in a similar manner to undo the effects of a malware program.

ExecRecorder [18] is a VM-based recovery method which also integrates logging to restore a system to a trusted snapshot before replaying log events for non-malware processes to restore the system state. The costly logging process incurs a 4% runtime overhead and produces an average of over 5GB of logs per hour, and no analysis of how long the recovery process takes is provided. The Secom [35] system attempts to avoid such a logging overhead by first writing a process's changes to an OS-level VM. It then attempts to remove potential malware effects by clustering changes according to higher-level behavioral profiles before merging the non-malware clusters to the VM host. This method is prone to identifying false positives and still degrades program performance by intercepting each system call run on the VM. Finally, the TimeVM [19] system uses a blend of log-based recovery and live backup VMs in different time states to quickly identify a backup VM free of a detected malware infection. This VM can then be rapidly updated to a clean, up-to-date state by replaying the logs of non-malware processes. The expected recovery time using this system was still often higher than 30 seconds, and the effects of a malware process that goes undetected for a long period of time may still be unable to be reversed with this method.

2.4 LXC and CRIU

LXC is an open-source Linux project which aims to allow for the virtualization of a Linux system or process within privilege-constrained containers [6]. These containers are meant to be lightweight alternatives to virtual machines, allowing for Linux virtualization without emulating system hardware and running a separate kernel. LXC containers can be run in a privileged or unprivileged state, and it is generally recommended that containers be run as unprivileged to minimize potential system damage should an attacker discover a way to “escape” the container. Given their own recommendation for unprivileged container use, the LXC maintainers do not consider container escape exploits a serious concern, stating "as privileged containers are considered unsafe, we typically will not consider new container escape exploits to be security issues worthy of a CVE and quick fix" [8].

Checkpoint/Restore in Userspace (CRIU) is another open-source project developed for Linux [40]. CRIU allows for individual Linux processes to be checkpointed during execution, saving their allocated memory and execution progress in image files. These files can subsequently be used by CRIU to restore the process to its prior state of execution when need be. One attractive feature of CRIU is that it is able to restore TCP connections by using the `TCP_REPAIR` socket option [3]. This feature prevents interruptions for TCP connections which are established before the checkpoint/restore process. While the more obvious applications of this technology may be for the live migration of processes between hosts or load balancing, Araujo et. al previously used CRIU in the context of security by redirecting attackers attempting to use known vulnerabilities to dynamically created honeypots [14].

CRIU has been incorporated into the LXC project, allowing for an entire container and the processes running within it to be checkpointed or restored. This is done via the `lxc-checkpoint` utility, which directly calls the locally installed version of CRIU on the container host to checkpoint or restore running containers.

3 Design Objectives

After considering previous attempts at malware recovery, we seek to improve on the state of the art in several ways. To this end, we select five desirable properties to guide the design of our solution.

1. *Fast*: The method should minimize the downtime of the system.
2. *Availability-maximizing*: The method should avoid interruptions to services which are not directly affected by the malicious processes.
3. *Flexible*: The method should accept alerts from a variety of sources and make use of the information provided by them.
4. *Information-Gathering*: The method should collect information about the malicious processes to aid in detecting them more easily and quickly in the future.
5. *Comprehensive*: The method should fully remove malware traces and record which changes were reverted.

With these goals in mind, we constructed CRIU-MR. In order to achieve these goals, a few simplifying assumptions were required. First, we suppose the filesystem is “mostly-static”, meaning that updates are relatively infrequent, and when they do occur, they can be applied to both the real filesystem and the backups simultaneously. This is the case for many web servers, especially when the data is retrieved from a database on another network node instead of being locally present. This assumption allows for rapid restoration of the filesystem, as the backup can be quickly swapped back into the container root filesystem location in case of an infection without file loss. Additionally, because we make use of Linux container technology, we assume that the attacker cannot escape from the container to the host machine. With this assumption, we are able to make use of an isolated environment which can be independently checkpointed and restored. In the following sections, we describe this system and demonstrate its effectiveness before returning to challenge these assumptions in the “Discussion and Limitations” section (§6).

4 Implementation and Architecture

The majority of the implementation of our recovery method exists as modifications to the CRIU source code. Our changes are available as a fork of the CRIU repository on GitHub[1]. These changes are separated into the two main actions of CRIU: checkpoint and restore. Overall, 659 lines of C code were added to implement these features.

4.1 Checkpoint

The changes made to the checkpoint process mostly center around reading a “policy” file and using this policy to build a list of container processes which should not be restored. The policy is read into CRIU using Protocol Buffers (also known as protobuf) [21], which is a binary serialization format developed by Google. Protobuf was selected based on its high performance serializing and deserializing data relative to other formats, such as XML or JSON [31], and also because it was already used extensively for the image files generated by CRIU checkpoints. The policy can be composed of a variety of

[1]https://github.com/ashtonwebster/criu

user-defined or dynamically generated rules that are used to omit processes from being restored, including:

- *Executable Name Match:* Whether the executable filename of a process matches a given string
- *File Match:* Whether any opened file of a process matches a given string
- *TCP IP Match:* Whether the IP address for any established TCP connection of a process matches a given IP address
- *Memory Match:* Whether the process contains the specified ASCII or Hex encoded string
- *PID Match:* Whether the PID of a process matches the given PID
- *Parent PID Match:* Whether the parent PID of a process matches the given PPID
- *Parent Executable Name Match:* Whether the parent executable filename matches the given string

In choosing these rule types, we seek to provide a flexible policy language which can identify malware to omit during the restore process based upon alerts provided by various intrusion detection triggers, such as potentially malicious TCP connections, executables which match a virus signature, and flagged PIDs. Using these rules, both *dynamic* and *static* policy elements can be created. Dynamic policies are created from alerts generated by other systems, such as IDS's or antivirus scanners. For instance, outbound firewall rule violations might trigger the generation of a policy to terminate any process attempting to communicate which a suspicious IP address. Users can also define static policies which have a base of assertions that are always enforced, regardless of the type of malware. For example, perhaps some processes should never have child processes under normal execution, or perhaps it is not expected for any process to have a sensitive file open. These assertions can be encoded as static policies, to which dynamic policies are added as attacks are detected. The combination of static and dynamic policy rules allows for detection of a wide variety of malware, including malware which may run exclusively in memory, such as the `meterpreter` metasploit payload [9].

These changes to CRIU source code are mostly additions at the point when information about files, connections, or process identifiers are being dumped to disk. Essentially, we check if there are any matching policy elements for each of these resources, and if there are, the PIDs of relevant processes are written to an additional protobuf formatted file named `omit.img`. It is important to note that no process dump information is discarded in this phase; it is simply logged for later action. This is so that information about potentially malicious processes can be forensically analyzed at a later time, but not restored.

Modifications were also made to the `lxc-checkpoint` command to accept the same parameters as the ones that were added for CRIU. Specifically, parameter processing for the `-policy` (path to the policy to use) and `-base-path` (path to the container filesystem) parameters was added. This required 44 lines of C code added across 3 files. The modified version of LXC is available on GitHub as well[2].

4.2 Restore

The core modifications for the CRIU restore process ensure that malicious processes flagged by the checkpoint process in `omit.img` are not restored. This is as simple as iterating over this list of omitted processes and removing the corresponding PIDs. Additionally, the way that missing files are handled by vanilla CRIU is changed. Vanilla CRIU will crash immediately if any process is missing a referenced file. Instead, CRIU-MR is adjusted to simply omit any process with a missing file reference. This is performed by checking to ensure files referenced by file descriptors are actually present on the target filesystem during the reconstruction of the container process tree. In the case of a process with an omitted parent, the child is omitted as well. These changes ensure that as the container is restored on the backup filesystem, processes referencing potentially malicious files that are no longer present will be gracefully omitted during the restore, even if these processes were not directly flagged via a policy rule. This will not harm non-malware processes given the "mostly-static" filesystem assumption.

4.3 Architecture

In order to allow for manual or automatic triggering of the checkpoint/restore process and dynamic generation of policies, we write a simple program to act as the recovery agent. The recovery agent listens on a given TCP port for JSON formatted information specifying how to construct a policy. It can receive these JSON alerts from other processes or even other hosts, as demonstrated in Figure 1. These JSON alerts constitute the dynamic policy rules and can be added to static policies which are defined in the agent and used as the default. This combined policy is written in the protobuf format and passed to CRIU for execution. The recovery agent also handles the filesystem restore, preparation, and cleanup operations needed to perform quick malware recovery, which

[2]https://github.com/ashtonwebster/lxc

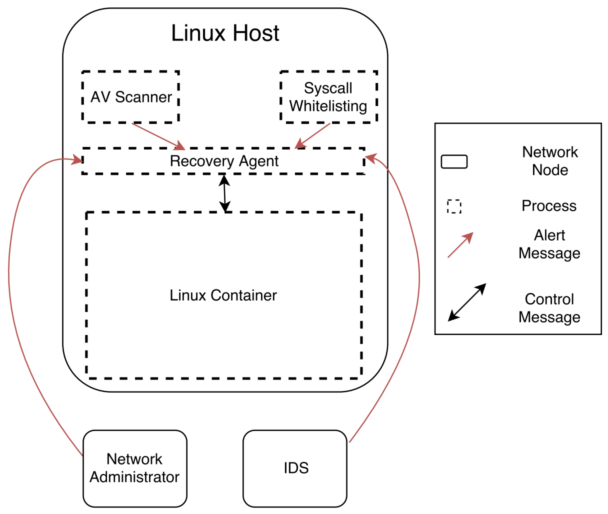

Figure 1: The CRIU recovery agent can receive alerts from a variety of sources, both at the host and network level.

will be covered in more detail below. The agent code was implemented in python and is available as a separate open source GitHub repository[3]. With these components, a typical malware recovery follows these steps:

Infection: Malware is introduced to the system. This may be through a backdoor, network exploit, or other method. At some point it begins executing and may modify the filesystem.

Detection: As a result of the malware on the system, one or more "triggers" may send an alert to the recovery agent. The recovery agent accepts a JSON file specifying the trigger type (e.g. AV scanner, IDS, IPS) and relevant information (e.g. filename, TCP connection). This JSON file is used to build the policy used by CRIU for malicious process removal. We have created example JSON generators for Snort [17] (a rule based IDS) and ClamAV [16] (an antivirus scanner). The example code used to generate the JSON alerts and send them to the recovery agent are shown in Appendix A.

Preparation: The recovery agent for CRIU-MR listens on a TCP port for a JSON message. Upon receipt of a message, a new rule for process omission will be generated and added to the policy file of existing rules. The policy is then compiled as a protobuf formatted file which is read by our modified version of CRIU. Next, a folder is created for storing the checkpoint/restore data generated by CRIU.

During the subsequent checkpoint/restore, the container will be unavailable for a few seconds. To avoid the loss of any packets arriving during this time, it may be necessary tp use the iptables target NFQUEUE on the container host to buffer packets. Essentially, NFQUEUE allows traffic to be sent to userspace for processing, and in this case it can be used to buffer packets while the malware recovery process is being executed. We provide a code listing and further description in appendix B.

[3]https://github.com/ashtonwebster/CRIU-MR-agent

CRIU Checkpoint: CRIU dumps the relevant image data for all processes (including malicious processes) on the container. Processes will be flagged as malware if they match a specified policy and are written to disk in a protobuf file named `omit.img`.

Filesystem Restore: In order to allow for fast filesystem restore, CRIU-MR maintains two backups. One backup, which we denote the "swap backup", is simply renamed to match the Linux container root filesystem path via the `mv` command. The other backup, denoted the "master backup", is used to restore the swap backup so this process can be repeated. Using these backups, the filesystem for the container is restored with a few simple shell commands:

```
mv $lxc_path/rootfs $infected_fs_dir/infectedfs
mv $backup_path/rootfs.swap_backup \
        $lxc_path/rootfs
```

One benefit of this method is that the infected filesystem can be later inspected (with the assistance of the CRIU-MR log files) to collect malware samples and detect malicious filesystem changes.

CRIU Restore: At this point, the CRIU restore of the checkpointed non-malware processes begins. During construction of the process tree, processes may be omitted for two reasons. First, processes which reference missing files are omitted. Next, processes contained in the `omit.img` file previously created are omitted. Any children of these processes are also ignored. Restore then continues as normal, with established TCP connections also being restored.

Cleanup: Finally, a few cleanup tasks are performed to return the system to its normal state. If NFQUEUE was used, the process is stopped so that buffered packets are forwarded along to the container. The swap backup for the container is also restored from the "master" backup to allow for a quick filesystem restore in the event of another breach using the following command:

```
cp $backup_path/rootfs.master_backup \
        $backup_path/rootfs.backup_swap
```

The preparation, CRIU checkpoint, filesystem restore, CRIU restore, and cleanup steps are all automated via the CRIU-MR recovery agent program. Thus, the response to malware can be completely independent of human interaction for rapid recovery from attacks.

5 Experiments

We conduct experiments to address two questions. First, we seek to answer the question "How long does it take to successfully remove various malware from the system?" In order to answer this question, we measure the recovery time for six different malware programs. Then, we address the question "What is the availability impact of the recovery process on a running service?". To answer this question, we devise an experiment using Apache Benchmark [1] to simulate HTTP requests to an Apache web server running on the container. We observe the impact of the checkpoint/restore process on the active connections and find that no connections fail while the maximum response time increases by only a few seconds. All experiments were run on a Virtual Machine with 4 Intel Xeon 2.4GHz cores and 4 GB RAM running Ubuntu 16.04 hosting a linux container. The container used ran Ubuntu 16.04 with AMD64 architecture.

5.1 Experiment I: Malware Recovery Duration

For our first experiment, we measure the duration of the recovery process and ensure that all malware processes and files are removed. To conduct the experiment, we collect six Linux malware samples.

- *linux_lady*: This malware was written in Go and attempts to mine bitcoin using the resources of infected computers. It primarily works by downloading the mining script payload and adding itself as a cronjob to the victim host. This sample was collected from the Contagio malware repository[2].
- *ms_bind_shell*: This is a simple payload from the Metasploit framework [9] which binds on a specified port and IP and provides shell access to the attacker.
- *ms_reverse_shell*: This is another malware from the Metasploit framework which creates a *reverse shell* by initiating a connection with the specified host. The reverse shell method is often more effective than the bind shell method in practice because it can more easily evade firewalls by initiating the connection rather than accepting a connection to an unused port.
- *wipefs*: This malware was found on the Hybrid Analysis website [5]. It uses the stratum mining protocol to mine bitcoin on the victim's machine.
- *Linux.Agent*: This malware, first discovered by Tim Strazzere [29] attempts to exfiltrate either the `/etc/shadow` file with encrypted passwords (if root access is available) or the `/etc/passwd` file (otherwise).
- *goahead_ldpreload*: This is actually a vulnerability in GoAhead [36], a lightweight embedded webserver and not a malware sample. However, we are able to inject a long-running malware script via the remote code execution vulnerability explained by Daniel Hodson of Elttam [24] with associated CVE-2017-17562 [10]. Unlike the other samples, this is an example of a benign process being infected with a malicious payload (instead of a malware binary being executed). To simulate a long-running malicious payload, we remotely execute commands which create a file each second on the filesystem, but any arbitrary C code can be executed.

Each experiment consists of the following: first, an *ssh* session is started, and the malware is started as root in the background and using the unix command *nohup* to avoid termination when the *ssh* session ends. The exception is the *goahead_ldpreload* exploit, which begins by running the GoAhead server as root and remotely executing the malicious payload). Next, detection is simulated by triggering the checkpoint/restore process with a JSON file specifying the executable file to omit[4]. After 3 seconds of allowing the malicious processes to execute, the recovery process is triggered, as described in §4. The timing measurements are taken by using the *timeit* library in Python [11]. Each malware is removed 10 times with timing results shown in Figure 2, and the time for each stage of checkpointing is shown in Table 1. In addition to an experiment for each malware sample, we also run the malware recovery process with no malware present for comparison (labeled as "None").

By restoring the infected filesystem to a safe backup state, we observe that any file state changes made by the malware were undone. We also observed that for each experiment, each malware process was successfully omitted and no longer running on the restored container. We acknowledge that it is possible that the malware also changed memory or features of other restored processes, and we discuss this in more detail in the Discussion section (§6).

The results for this experiment suggest that the type of malware does not affect the time for recovery in a noticeable way. In fact, the removal of malware appears to match the time taken for a checkpoint and restore actions even in the absence of malware or policies (denoted "None" in Figure 2). This suggests that our modifications to the underlying CRIU checkpoint and restore

[4]For *goahead_ldpreload* we observe that the remote code execution occurs in a separate process `/root/goahead/test/cgi-bin/cgitest` handling CGI scripts, which is the executable name used in the policy for that exploit.

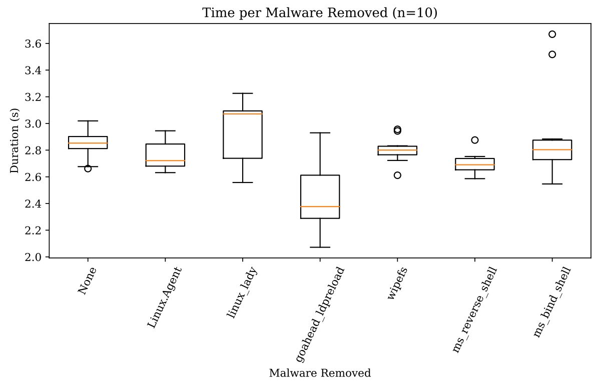

Figure 2: Boxplots summarizing duration of malware recovery process for six different malware.

Table 1: Mean (Std. Dev.) Time per Recovery Step

Step	Time (s)
Prep. Time	0.022 (0.003)
Checkpoint Time	2.157 (0.202)
FS Swap Time	0.012 (0.005)
Restore Time	0.572 (0.110)
Total Time	2.763 (0.265)

methods do not have a significant impact on their performance in terms of duration. Furthermore, we see that the time taken for the recovery process (see Table 1) is mostly dominated by the checkpoint process, with the restore process taking only about a fifth of the total time.

5.2 Experiment II: Availability Impact

Next, we address the question of the recovery process's impact on the availability of trusted services running on the host that is infected with malware. To evaluate this, we measure the impact of removing malware on a web server container with many active clients. For this, we use the *ab* tool (Apache Benchmark), which is able to simulate repeated HTTP connections and measure their duration and the number of failed requests. In order to mimic a realistic setting with a variety of request/response durations, we serve seven different pages ranging in size from 1kB to 1GB by powers of 10. We execute one instance of `ab` for each file size in parallel and vary the number of concurrent requests per process at 1, 5, 10, and 20 for a total concurrency across all processes of 7 to 140. For each experiment, we first start the apache benchmark script. After 30 seconds of normal execution of `ab`, we start the `linux_lady` malware on the host using `ssh` and trigger the recovery process in a method similar to Experiment I. The results of this process are summarized in Table 2.

In each experiment, all requests complete successfully. We observe that relative to the median request completion time, the "Max Request Time" for each file tends to increase by an amount of time comparable to the time it takes for CRIU-MR to execute as measured in Experiment I. These results show that while some connections are subjected to a latency increase of 3-6 seconds by the checkpoint/restore process, CRIU-MR still ensures that each request succeeds.

6 Discussion and Limitations

Overall, our experiences using CRIU-MR confirm that it is a viable strategy for removing malware while preserving a variety of services. For example, we manually observed that `ssh` connections interrupted by the CRIU-MR recovery process continue gracefully after the recovery completes, even without the use of NFQUEUE. However, when the recovery process occurs during downloads of large files using `curl` or browsers, NFQUEUE usage is required in order to preserve the download process. We therefore conclude that our modifications for CRIU-MR do not impact CRIU's underlying TCP restore abilities.

Beyond faster system restore, one benefit of the CRIU-MR method over methods which log every filesystem modification, such as Taser [20], is that there is no overhead for writing to logs during normal execution of the container. However, there is some performance overhead associated with using Linux containers that should be considered. Work comparing LXC to native performance and other virtualization techniques reveals that it often performs similarly to the native operating system [37]. This is likely due to the fact that Linux containers rely mostly on partitioning resources using Linux namespaces and control groups instead of more complex solutions, such as hardware virtualization used by conventional VMs.

Another concern is the security of the container in terms of isolation. Is it possible to escape the Linux container and infect the host operating system? Unfortunately, some proof of concept attacks have been found for Linux containers. Two whitepapers from the NCC Group explore this problem, one focusing on attacks [23] and one focusing mostly on mitigations [22]. This research reveals that `ptrace(2)` can be used to escape Linux containers, and an escape from the security boundary of the container can be executed via direct communication with the hardware. Fortunately, mitigations for these attacks are available, and the simplest method (which will fix both of these issues) is to simply use unprivileged Linux containers.

As alluded to previously, the malware process that triggered an intrusion detection alert might not be found

Table 2: Connection Stress Test

Concurrent Requests	File Size	Median Request Time (s)	Max Request Time (s)	Completed Requests	Failed Requests
7	1 kB	1	3,695	51,973	0
	10 kB	1	3,695	50,568	0
	100 kB	1	3,697	36,937	0
	1 MB	4	3,701	11,823	0
	10 MB	34	3,731	1,580	0
	100 MB	393	4,081	146	0
	1 GB	5,415	8,777	11	0
35	1 kB	4	3,776	51,803	0
	10 kB	4	3,776	58,479	0
	100 kB	5	3,782	41,953	0
	1 MB	20	3,821	11,385	0
	10 MB	130	4,115	1,776	0
	100 MB	1,256	6,066	205	0
	1 GB	12,482	26,098	19	0
70	1 kB	7	6,307	60,647	0
	10 kB	7	6,307	59,976	0
	100 kB	10	6,310	40,300	0
	1 MB	42	6,343	11,595	0
	10 MB	242	6,343	1,810	0
	100 MB	2,474	10,047	207	0
	1 GB	43,088	43,097	12	0
140	1 kB	13	4,614	78,377	0
	10 kB	13	4,614	77,497	0
	100 kB	19	4,641	53,338	0
	1 MB	77	4,706	14,494	0
	10 MB	583	5,351	1,953	0
	100 MB	5,712	10,933	191	0
	1 GB	62,474	62,474	1	0

by the specified policies in some cases. For example, if botnet malware is detected via an IDS based on a TCP connection to a command and control server, the connection may end before the alert is processed and CRIU-MR begins the checkpoint process, meaning the malware will fail to be flagged for omission during restore. In such a case, if the malware runs from an executable placed on the system via a malicious channel, CRIU-MR will still successfully remove it from the container during the restore process since the botnet executable isn't located on the safe filesystem backup. Such an event can be verified by checking the logs of CRIU-MR, which report which policy elements were triggered and any missing files that resulted in the removal of a process.

Nonetheless, the system may be infected with malware that both runs entirely within memory via code injection and evades being flagged during a checkpoint event as just described. In such an instance, it is prudent for the user to not only restore the filesystem to a safe point but to also restart the system and bring services back online. Users with active connections to services may experience an interruption in this case, but such mitigation will be necssary if no malware process was found. Similarly, there may exist malware which interfere with the memory and connections of other processes. These changes will not be detected by the current CRIU-MR system as they are not directly a part of the malware process (unless the interference somehow triggers another policy rule). The best solution for avoiding this issue is to use containers which have only one main service to reduce the potential attack surface. Alternatively, assertions about the memory spaces of benign processes could be checked during the restore process to verify their integrity, an idea we consider future work (§7).

This method is specific to Linux operating systems as it relies heavily on CRIU and LXC, which are obviously specific to that operating system. However, Linux is a popular operating system for web servers, with approximately 66.8% of web servers from the Alexa top ten million sites using some flavor of it, according to a survey conducted by W3Tech in February 2018 [12]. It might also be possible to extend the main ideas of this method using container technology for other operating systems, such as Docker, by using or creating the appropriate checkpoint/restore methods.

Finally, it is important to take appropriate actions even after malware removal. Namely, any vulnerability that resulted in a malware payload being delivered or executed needs to be patched. For example, the `goahead_ldpreload` exploit can be immediately exploited again after the first malware recovery if the GoAhead web server is not patched. Therefore, CRIU-MR needs to be coupled with a patching process in order to avoid repeated exploitation of the same vulnerabilities.

7 Future Work

One avenue for future work is in the verification of the integrity of processes. We previously noted that it is possible that malware may seek to change the memory spaces of benign processes outside of its process tree. One way to check if this has occurred is to instrument these processes with additional code to verify they are still executing properly. We refer to these checks as "dynamic assertions", where the processes are expected to dynamically respond to queries about execution state in order to verify the integrity of the process. Research into this area may reveal more robust ways of ensuring that malware effects have been reverted even if it interfered with other processes.

Because any maliciously uploaded files are archived in a separate filesystem, CRIU-MR could also be used as part of a framework which discovers and analyzes new malware. For example, checkpointed malware processes with corresponding executables could be executed in sandboxes to collect more information. Cuckoo [4] is one option for local analysis, or, if an external service is preferred, VirusTotal [13] or Hybrid Analysis [5] can be used to learn more about the nature of the collected payload. These results could then be integrated into other systems responsible for malicious activity alerts to more rapidly detect attacks of this type.

In addition to improvements to this particular component, we intend to explore how CRIU-MR can fit into a broader framework of intrusion detection. Related work we are currently conducting seeks to use machine learning techniques to analyze payloads of network traffic and could act as a trigger for this malware cleaning operation. We are also considering employing elements of moving target defense, such as changing the IP address, passwords, or even the physical host machine of a restored container to complicate and delay attacks while more robust defenses can be deployed.

8 Conclusion

The main contribution of our work is a new method for malware recovery. Rather than using logging or VM-based methods for removing malware, CRIU-MR uses Linux containers and CRIU to quickly restore a system to a safe state in the event of an infection. Furthermore, our method improves upon prior work by very quickly recovering the state of trusted services after recovery with minimal impact to clients. We conduct two experiments to test the speed and availability of CRIU-MR and find promising results. Our test of the duration of the malware recovery process finds that malware recovery does not take significantly more time than a CRIU checkpoint/restore with no policy. Furthermore, our second experiment indicates that CRIU-MR is capable of restoring container processes and TCP connections after malware recovery, even when many concurrent connections are present. The success of this tool is dependent on its use in the context of other systems, such as IDS's, firewalls, and antivirus scanners. Information from these systems, along with static application-specific knowledge, can form a robust policy for malware removal. CRIU-MR can now be used by both administrators and researchers to build systems which are responsive and service-preserving when faced with malware infections.

9 Acknowledgments

This work was supported by the United States Office of Naval Research under Contract N000141612107.

References

[1] ab - Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`. Accessed: 2018-02-05.

[2] Contagio malware dump. `http://contagiodump.blogspot.com/`. Accessed: 2018-02-05.

[3] CRIU: TCP Repair.

[4] Cuckoo sandbox. `https://cuckoosandbox.org/`. Accessed: 2018-02-05.

[5] Hybrid analysis. `https://www.hybrid-analysis.com/`. Accessed: 2018-02-05.

[6] Linux containers - lxc. `https://linuxcontainers.org/lxc/`. Accessed: 2018-02-05.

[7] Logstash. `https://www.elastic.co/products/logstash`. Accessed: 2018-02-05.

[8] Lxc security. `https://linuxcontainers.org/lxc/security/`. Accessed: 2018-02-05.

[9] Metasploit. `https://www.metasploit.com/`. Accessed: 2018-02-05.

[10] National vulnerability database: Cve-2017-17562 detail. `https://nvd.nist.gov/vuln/detail/CVE-2017-17562`. Accessed: 2018-02-05.

[11] Timeit. `https://docs.python.org/2/library/timeit.html`. Accessed: 2018-02-05.

[12] Usage of operating systems for websites. `https://w3techs.com/technologies/overview/operating_system/all`. Accessed: 2018-02-05.

[13] Virus total. `https://www.virustotal.com`. Accessed: 2018-02-05.

[14] ARAUJO, F., HAMLEN, K. W., BIEDERMANN, S., AND KATZENBEISSER, S. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 942–953.

[15] BARTOS, K., SOFKA, M., AND FRANC, V. Optimized invariant representation of network traffic for detecting unseen malware variants. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 807–822.

[16] CISCO. ClamAV.

[17] CISCO. Snort. `https://www.snort.org/`. Accessed: 2018-02-05.

[18] DE OLIVEIRA, D. A. S., CRANDALL, J. R., WASSERMANN, G., WU, S. F., SU, Z., AND CHONG, F. T. Execrecorder: Vm-based full-system replay for attack analysis and system recovery. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability* (New York, NY, USA, 2006), ASID '06, ACM, pp. 66–71.

[19] ELBADAWI, K., AND AL-SHAER, E. Timevm: A framework for online intrusion mitigation and fast recovery using multi-time-lag traffic replay. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (New York, NY, USA, 2009), ASIACCS '09, ACM, pp. 135–145.

[20] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The taser intrusion recovery system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 163–176.

[21] GOOGLE. Protocol buffers. `https://developers.google.com/protocol-buffers/`. Accessed: 2018-02-05.

[22] GRATTAFIORI, A. Understanding and hardening linux containers. *Whitepaper, NCC Group* (2016).

[23] HERTZ, J. Abusing privileged and unprivileged linux containers. *Whitepaper, NCC Group* (2016).

[24] HODSON, D. Remote LD_PRELOAD exploitation. `https://www.elttam.com.au/blog/goahead/`. Accessed: 2018-02-05.

[25] HSU, F., CHEN, H., RISTENPART, T., LI, J., AND SU, Z. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the 22Nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), ACSAC '06, IEEE Computer Society, pp. 257–268.

[26] JORDANEY, R., SHARAD, K., DASH, S. K., WANG, Z., PAPINI, D., NOURETDINOV, I., AND CAVALLARO, L. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 625–642.

[27] KHARAZ, A., ARSHAD, S., MULLINER, C., ROBERTSON, W., AND KIRDA, E. UNVEIL: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 757–772.

[28] LABS, M. 2017 state of malware report. `https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf`. Accessed: 2018-02-05.

[29] LABS, S. O. Hiding in plain sight? `https://www.sentinelone.com/blog/hiding-plain-sight/`. Accessed: 2018-02-05.

[30] LEVER, C., KOTZIAS, P., BALZAROTTI, D., CABALLERO, J., AND ANTONAKAKIS, M. A lustrum of malware network communication: Evolution and insights. In *2017 IEEE Symposium on Security and Privacy (SP)* (May 2017), pp. 788–804.

[31] MAEDA, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on* (May 2012), pp. 177–182.

[32] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17*, 1 (Mar. 1992), 94–162.

[33] PALEARI, R., MARTIGNONI, L., PASSERINI, E., DAVIDSON, D., FREDRIKSON, M., GIFFIN, J., AND JHA, S. Automatic generation of remediation procedures for malware infections. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 27–27.

[34] PASSERINI, E., PALEARI, R., AND MARTIGNONI, L. How good are malware detectors at remediating infected systems? In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2009), DIMVA '09, Springer-Verlag, pp. 21–37.

[35] SHAN, Z., WANG, X., AND C. CHIUEH, T. Malware clearance for secure commitment of os-level virtual machines. *IEEE Transactions on Dependable and Secure Computing 10*, 2 (March 2013), 70–83.

[36] SOFTWARE, E. Goahead: Simple, secure embedded web server. `https://www.embedthis.com/goahead/`. Accessed: 2018-02-05.

[37] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev. 41*, 3 (Mar. 2007), 275–287.

[38] SUBRAHMANIAN, V., OVELGONNE, M., DUMITRAS, T., AND ADITYA PRAKASH, B. *The Global Cyber-Vulnerability Report*. 01 2015.

[39] VASUDEVAN, A. Maltrak: Tracking and eliminating unknown malware. In *Proceedings of the 2008 Annual Computer Security Applications Conference* (Washington, DC, USA, 2008), ACSAC '08, IEEE Computer Society, pp. 311–321.

[40] VIRTUOZZO. CRIU. `https://criu.org`. Accessed: 2018-02-05.

[41] ZHU, Z., AND DUMITRAS, T. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 767–778.

Appendices

A Logstash Pipelines for Snort and ClamAV Triggers

Logstash [7] can be a useful tool for parsing and forwarding alerts from a variety of sources. The "grok" filter, a filtering action in the Logstash pipeline, can be used to parse alerts from arbitrary sources (such as files, network ports, etc.) into easily parseable JSON. Listing 1 shows an example of using Logstash with the grok filter to parse Snort alerts and send them to the CRIU-MR agent. The Snort command used to generate the alerts is `snort -c snort.conf -i lxcbr0 -A full -k none`, where `-A full` denotes full alert syntax. The `-k none` parameter indicates no checksums should be calculated, which we anecdotally observe is required for obtaining alerts on both inbound and outbound traffic.

The ClamAV parsing is very similar. The command to execute the scanner is `clamscan path/to/scan -no-summary -infected > output.log`. The Snort example is modified slightly for the different output format. Namely, the path is changed to point to `output.log`, the multiline code is not needed (each line of `output.log` corresponds to one alert), and the `add_field` codec is modified for the appropriate trigger type. Finally the grok parsing code in the filter step simply becomes:

```
%{GREEDYDATA:filepath}:
        %{GREEDYDATA:malwarename} FOUND
```

B NFQUEUE Buffer

Listing 2 shows an example implementation of a buffer for packets intended for the interface `lxcbr0`, which is the default interface used for the Linux container networking. This simple python script uses the netfilterqueue library (available via `pip`) to hold packets until the program terminates via a kill signal. Packets are then released to the kernel and forwarded along to or from the container.

Listing 1: Logstash Pipeline for Snort Alert Parsing

```
input {
  file {
    # standard path for snort alerts
    path => "/var/log/snort/alert"
    # combines multiple lines as a single log event
    codec => multiline {
      pattern => "^\[\*\*\] "
      negate => true
      what => "previous"
    }
    # adding a field to the parsed json so that CRIU-MR knows
    # how to parse it
    add_field => { "trigger_type" => "snort" }
  }
}

filter {
  grok {
    # Parsing output of snort into JSON
    # newlines added for readability
    match => { "message" => "\[\*\*\] \[%{NUMBER:version}:%{NUMBER:sid}:
        %{NUMBER:revision}\] %{GREEDYDATA:rule} \[\*\*\]*\n\[Priority:
        %{NUMBER:priority}\] \n%{MONTHNUM:month} \/%{MONTHDAY:day}-%{HOUR:hour}
        :%{MINUTE:minute}:%{SECOND:second} %{IP:src_ip}:%{NUMBER:src_port}
        -> %{IP:dst_ip}:%{NUMBER:dst_port}*\n*\n*\n*" }
  }
}

output {
  tcp {
    host => <CRIU-MR_HOST_IP>
    port => <CRIU-MR_PORT>
  }
}
```

Listing 2: NFQUEUE Python Buffer

```python
import os
from netfilterqueue import NetfilterQueue
import signal

def send_packets(signal, frame):
    print("sending packets and shutting down")
    os.system("iptables -D INPUT -i lxcbr0 -j NFQUEUE --queue-num 1")
    os.system("iptables -D OUTPUT -o lxcbr0 -j NFQUEUE --queue-num 1")
    os.system("iptables -D FORWARD -o lxcbr0 -j NFQUEUE --queue-num 1")

    for packet in packets:
        packet.accept()
    nfqueue.unbind()

def hold_packet(pkt):
    global packets
    print("holding " + str(pkt))
    packets.append(pkt)

packets = []
signal.signal(signal.SIGTERM, send_packets)

os.system("iptables -I INPUT -i lxcbr0 -j NFQUEUE --queue-num 1")
os.system("iptables -I OUTPUT -o lxcbr0 -j NFQUEUE --queue-num 1")
os.system("iptables -I FORWARD -o lxcbr0 -j NFQUEUE --queue-num 1")

nfqueue = NetfilterQueue()
nfqueue.bind(1, hold_packet)

try:
    nfqueue.run()
except KeyboardInterrupt:
    send_packets(None, None)
```