Self-Patch: Beyond Patch Tuesday for Containerized Applications

Olufogorehan Tunde-Onadele, Yuhang Lin, Jingzhu He, Xiaohui Gu
Department of Computer Science
North Carolina State University
Email: {oatundeo, ylin34, jhe16, xgu}@ncsu.edu

***Abstract*—Containers have become increasingly popular in distributed computing environments. However, recent studies have shown that containerized applications are susceptible to various security attacks. Traditional periodically scheduled software update approaches not only become ineffective under dynamic container environments but also impose high overhead to containers. In this paper, we present Self-Patch, a new self-triggering patching framework for applications running inside containers. Self-Patch combines light-weight runtime attack detection and dynamic targeted patching to achieve more efficient and effective security protection for containerized applications. We evaluated our schemes over 31 real world vulnerability attacks in 23 commonly used server applications. Results show that Self-Patch can accurately detect and classify 81% of attacks and reduce patching overhead by up to 84%.**

***Index Terms*—Container Security, Anomaly Detection, Security Patching.**

I. Introduction

Containers have become increasingly popular in distributed computing environments by providing an efficient and lightweight deployment method for various applications. However, recent studies [1] [2] have shown that containers are prone to various security attacks, which has become one of the top concerns for users to fully adopt container technology [3].

Containerized applications pose a set of new security challenges to distributed computing environments. First, container image repositories are prone to vulnerabilities. Indeed, previous study [2] reveals an alarming degree of vulnerability exposure and spread in the official Docker Hub container repository. It is complex to maintain a public or private container repository which often consists of a large number of container images and many inheritance layers. If a container is created from a base image, any vulnerability included in the base image needs to be patched in the containers that are built on top of the base image. Second, containers are often allocated with limited resources because a large number of containers often share the resources of a single physical host. Security patching might cause significant resource increase (e.g., memory bloating) in a patched container, which makes the container unable to run after patching.

Existing security patching schemes in distributed computing environments often follow a periodically scheduled whole upgrade approach, that is, updating all applications as a whole on a certain day (e.g., every Tuesday). The approach works well in stable systems consisting of long running applications. However, containers are often short-lived, which makes periodical patching schemes ineffective if the vulnerable containers miss the pre-scheduled patching day. Moreover, whole software upgrade often significantly increases the memory and storage footprint of the patched containers. As a result, those containers quickly become too heavy to fit in constrained resource allocations.

In this paper, we present Self-Patch, an intelligent self-triggering security patching framework for containerized applications. Our framework consists of three integrated components: 1) online *attack detection* module which can detect security attacks using low-cost, non-intrusive system call tracing and unsupervised autoencoder neural network models [4]; 2) *attack classification* module which classifies attack behaviors into specific vulnerability exploits by identifying most frequently appeared system calls during the attack period and 3) *targeted patch execution* module which is responsible for applying proper security patches based on the classification results. Specifically, this paper makes the following contributions.

- We present a new self-triggering targeted patching framework to achieve effective and efficient attack containment for containerized applications.
- We describe an online attack detection and classification scheme using out-of-box system call tracing and unsupervised machine learning methods.
- We have implemented a prototype of Self-Patch and evaluated it over 31 real world security attacks in 23 commonly used server applications.

Our experimental results show that Self-Patch's attack detection scheme can accurately detect and classify 81% security attacks with 16 seconds lead time on average. In comparison, other commonly used anomaly detection schemes such as k-nearest neighbor (k-NN) and k-means clustering algorithm can only detect 6% and 68% exploits, respectively. k-means also produces 7% false alarms while Self-Patch only incurs 0.7% false alarms. We further compare the memory and disk footprint change before and after patching between Self-Patch and the existing whole upgrade approach. Our results show that Self-Patch can reduce the memory footprint increase (caused by the applied patches) by up to 84% and disk size increase by up to 40%.

The rest of the paper is organized as follows. Section II de-

978-1-7281-7277-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ACSOS49614.2020.00022

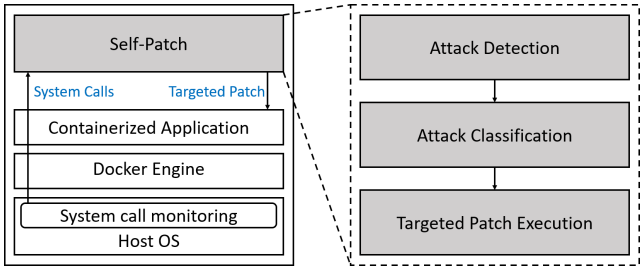

Fig. 1: System overview of Self-Patch.

scribes the system design. Section III presents our experiment setup and results. Section IV compares our work with related work. Section V concludes the paper.

II. System Design

This section describes the system design of Self-Patch. We first provide an overview about the system, followed by the design details for each component.

A. System Overview

Self-Patch aims at providing a self-triggering targeted patching framework for containerized applications, which is illustrated by Figure 1. Self-Patch consists of three coordinating components: 1) *attack detection*, 2) *attack classification*, and 3) *targeted patch execution*. The attack detection module monitors container runtime behaviors by analyzing system call traces via unsupervised autoencoder neural network learning methods. We pick system call data for our attack detection because many attacks manifest in system call invocations. We decide to use an unsupervised machine learning method in order to achieve online detection for both known and unknown attacks.

Upon detecting anomalies in container runtime behavior, we need to decide what type of vulnerability the detected attack targets. To map to a specific vulnerability, we perform offline profiling to extract the vulnerability signature by extracting the top frequently used system calls after triggering the corresponding attack. Note that we make an assumption here that attacks targeting the same vulnerability exhibit the same behavior in terms of top frequently used system calls. We find our assumption holds in our experiments. We plan to further validate this assumption using more attacks in our future work.

After the vulnerability is identified, the targeted patch execution module is dynamically triggered to contain the attack by patching the victim container to fix the vulnerability targeted by the attack. We first bring the victim container offline and then apply the proper software updates to the container in a quarantined environment. Once the patching is complete, an updated container image is committed to the repository for spawning future containers. We now present the design details of each component in the following subsections.

TABLE I: A frequency vector sample for the ActiveMQ application (CVE-2016-3088). An attack is triggered at $t = 1528903079912$.

Timestamp \ System call	access	sendto	lseek	fcntl
1528903079812	0	0	0	0
1528903079912	59	4	0	5
1528903080012	299	18	2	0
1528903080112	0	0	0	0

B. Attack Detection

Self-Patch performs attack detection by analyzing system call traces invoked by the containerized application. For robustness, we leverage an existing container monitoring tool Sysdig [5] to achieve out-of-box monitoring from the host kernel. We can collect all the system calls invoked by an application running inside the container from outside. Although system call sequences (i.e., n-gram) have been used to identify attacks in intrusion detection systems [6], they require a database of recognized sequences for detecting attacks, which cannot handle dynamic workload and mimicry attacks. In this work, we propose to first extract system call features to model runtime application behaviors and then apply unsupervised machine learning methods to detect attack behaviors. Specifically, we compute the frequencies of each system call type per sampling period to form a frequency vector. For example, Table I shows a frequency vector time series where the `access` system call is invoked 299 times within $[1528903080012, 1528903080112)$ millisecond. Intuitively, when an attack is triggered to exploit a certain vulnerability in the application, certain types of system calls are invoked more frequently than normal. For example, in Table I, we can see an abnormal frequency increase for `access` and `sendto` system calls after the attack is triggered at time 1528903079912.

To achieve online anomaly detection, we leverage unsupervised multi-variate machine learning to detect abnormal system call frequency changes. We choose autoencoder neural network as our detection model because it does not require labelled training data and can achieve good accuracy with a relatively small number of neurons with low training cost. The autoencoder neural network builds a model that learns to reconstruct training samples with minimal error. This is achieved by representing the input data in a lower dimensional space with a small number of neurons, that is called the *encode* step. Thereafter, in the *decode* step, the model attempts to regenerate the data that was compressed by the encode step. Thus, the autoencoder network typically has a symmetric architecture. The figure shows the network configuration of our autoencoder implementation with four hidden layers between the input and output layers. The encode region is fully linked to the identical but reversed decode region by their innermost layers of neurons. The length of the input vectors is the number of different system call types (e.g., read, write, futex) produced by the application as described above. The number of neurons in the input layer and output

layers of the autoencoder is determined by the number of different system call types that appear in the collected system call data. Lastly, the model classifies the frequency vector test samples with low reconstruction errors as normal and those with high reconstruction errors as abnormal. We determine the error threshold based on the reconstruction errors observed in the training data. Specifically, a certain percentile rank of reconstruction errors from the training set is selected as the threshold. We found the 99 percentile value as the threshold works well in our experiments.

C. Attack Classification

After detecting an attack, we want to classify the attack into a specific type which is linked to a vulnerability identifier (e.g. CVE ID). Once this is obtained, we can update the application to the proper version. Similarly, signatures for new attacks would also be generated, which can contribute to the development of new security updates.

The attack classification in our system is guided by the alarms raised by our detection models. Based on the detected attack period, we extract the top ranked system calls with the following algorithm. The rank is calculated by taking an average of the frequency counts for each system call during the interval. The list of system calls and their counts, sorted in descending order, serves as the rank. To extract signature patterns, we first identify the top k ranked system calls (e.g., k = 5) and then concatenate the names of those selected system calls into a string. The attack signature is denoted by the hash value from running a Secure Hash Algorithm (SHA) on the string. For example, let us consider the denial of service (DOS) attack to Network Time Protocol (NTP) vulnerability (CVE-2016-7434). The top five frequent system calls are: `rt_sigprocmask`, `gettid`, `write`, `read` and `clock_gettime`. The `rt_sigprocmask` system call checks or modifies the blocked signals of a thread, while `gettid` gets the thread ID. Furthermore, the denial of service attack is caused by sending an extremely long character to the NTP service over a socket connection. Thus, the application would need to make read and write calls to service this request. The *clock_gettime* call retrieves the time of a requested clock.

The signature is then mapped to a corresponding existing CVE ID that is collected by an offline profiling process using the same signature extraction algorithm. However, if we fail to map the signature with any existing CVE, we mark this attack as an unknown attack which requires further investigation.

D. Targeted Patch Execution

After a specific attack is detected and classified, Self-Patch triggers the targeted patching module over the victim container to contain the attack. The targeted patch execution module focuses on installing only the specific libraries needed to address the identified vulnerability. Our patching process consists of three steps: downloading new software packages, installing new software packages, and removing unnecessary files.

```
#!/bin/bash
# download files
apt-get update
apt-get -y install wget gcc make
wget https://github.com/.../ghostscript-x.xx.
    tar.gz
tar xvf ghostscript-x.xx.tar.gz

# install files
cd ghostscript-x.xx
./configure
make install

# remove files
apt-get purge -y wget gcc make
apt-get autoremove -y
cd ..
rm -r ghostscript-x.xx.tar.gz ghostscript-x.xx
```

Fig. 2: A targeted patching example for Ghostscript.

First, obtaining source files involves using tools such as *wget* or *git* or APT, depending on where the files are located. Wget is useful for downloading files provided by a URL (Uniform Resource Locator), `git clone` for GitHub repositories and `apt-get update` for retrieving packages provided by APT. Next, installation may require other tools like *make* or *pip* to compile and install the application. Applications downloaded from source with a Makefile are typically installed with `./configure` to prepare a Makefile, followed by `make` to compile source code and finally a `make install` to move the compiled files to proper locations. Those applications configured with APT can leverage `apt-get install --only-upgrade` commands, whereas those with pip can use `pip install` which handles both the download and install steps. Finally, the installation is cleaned up. Downloaded archive source files and folders extracted from them as well as their outdated counterparts can be removed with basic Linux commands. APT can handle this process with `apt-get purge` and `apt-get autoremove` commands. Figure 2 presents a basic targeted patch example for three Ghostscript vulnerabilities (CVE-2018-16509, CVE-2018-19475 and CVE-2019-6116). In the download files section, dependent *wget*, *gcc* and *make* libraries are retrieved with APT to execute the rest of the installation procedure. Whereas, *wget* downloads the tar archive that contains the new application version source files. Notice that after the file is installed, these files and packages are removed. The difficulty in the patch execution lies in the installation differences among applications. Discovering these libraries and installation steps involves extensive searches over security databases, application sites and manuals.

Targeted patching is applied to the container in a quarantined environment isolated from other normal applications. Meanwhile, various security countermeasures can be applied. For instance, further requests from the compromised container can be dropped while new trusted containers are spawned

TABLE II: List of explored real-world vulnerabilities.

Threat Impact	CVE ID	CVSS Score	Application	Attack Duration (seconds)
Return a shell and execute arbitrary code	CVE-2012-1823	7.5	PHP	1
	CVE-2014-3120	6.8	Elasticsearch	9
	CVE-2015-1427	7.5	Elasticsearch	60
	CVE-2015-2208	7.5	phpMoAdmin	2
	CVE-2015-3306	10	ProFTPd	4
	CVE-2015-8103	7.5	JBoss	30
	CVE-2016-10033	7.5	PHPMailer	125
	CVE-2016-3088	7.5	Apache ActiveMQ	49
	CVE-2016-9920	6	Roundcube	121
	CVE-2017-11610	9	Supervisor	2
	CVE-2017-12615	6.8	Apache Tomcat	13
	CVE-2017-7494	10	Samba	36
	CVE-2017-8291	6.8	Ghostscript	1
Execute arbitrary code	CVE-2014-6271	10	Bash	2
	CVE-2015-8562	7.5	Joomla	1
	CVE-2016-3714	10	ImageMagick	4
	CVE-2017-12794	4.3	Django	1
	CVE-2017-5638	10	Struts	29
	CVE-2018-16509	9.3	Ghostscript	2
	CVE-2018-19475	6.8	Ghostscript	2
	CVE-2019-6116	6.8	Ghostscript	2
Disclose credential information	CVE-2014-0160	5	OpenSSL	14
	CVE-2015-5531	5	Elasticsearch	2
	CVE-2017-7529	5	Nginx	1
	CVE-2017-8917	7.5	Joomla	1
	CVE-2018-15473	5	OpenSSH	2
Consume excessive CPU	CVE-2014-0050	7.5	Apache Tomcat	45
	CVE-2016-6515	7.8	OpenSSH	20
Crash the application	CVE-2015-5477	7.8	BIND	6
	CVE-2016-7434	5	NTP	1
Escalate privilege level	CVE-2017-12635	10	CouchDB	1

to replace compromised ones. After a successful update, an image is saved from the container with a `docker commit`. The resulting image is then used to deploy new containers.

III. Experimental Evaluation

In this section, we present our evaluation methodology and experimental results.

A. Evaluation Methodology

1) Real world vulnerabilities: We evaluate Self-Patch using 31 real world vulnerabilities discovered in 23 commonly used server applications, highlighted in Table II. We especially focus on vulnerabilities of medium to high severity reported in the last five years. These applications include widespread web services (e.g. Apache Tomcat, Nginx, Elasticsearch) and database services (e.g. CouchDB), which are currently popular containerized applications [7], [8]. Attacks to application vulnerabilities result in threat impacts that fall into six categories classified by a recent study [2]: 1) return a shell and execute arbitrary code; 2) execute arbitrary code; 3) disclose credential information; 4) consume excessive CPU; 5) crash the application and 6) escalate privilege level. Such attacks and vulnerable container images are obtained from exploit databases and repositories (e.g. VulHub [9]).

2) Experiment setup: We run workload generated with Apache JMeter [10] on the container of each target vulnerability, to approach real world system operation. Specifically, JMeter quickly delivers appropriate requests to the applications. The supplied request rate increases to the maximum value that the application can accommodate. After running the container for a period of normal operation to train the detection model, an attack is triggered to exploit the security vulnerability. The attack then executes for a subsequent period until no further attack activity is made. Meanwhile, we use Sysdig [5] to record the system calls invoked by the running containerized applications. We separate the entire system call trace into two halves. We use the first half of the data to train the detection model as it consists of enough samples of normal operation. However, we use the whole trace to extract detection and attack signature results. The whole trace consists of seven minutes of activity except for the attacks that crash the application. Given our sample interval of 0.1 milliseconds, this contributes about 4200 samples.

To evaluate the results of targeted patching, we repeat the above process. Immediately after the vulnerability is triggered, we execute the targeted patching. At the same time, we monitor the memory utilization and disk size. In particular, we track the memory usage by leveraging the APIs exposed by cAdvisor [11]. Meanwhile, the container disk size is collected with native Docker commands. The sizes of the read-only image layers and writable container layers are summed and given by the `docker ps -s` command. After the completion of each patching experiment, we determine whether patching is successful. First, we save the image of the patched container. We then start a new container using the image just created and then execute the attack commands. If the commands continue to work as before, we mark this patching experiment as unsuccessful and successful otherwise.

3) Attack detection setup: We perform attack detection with an autoencoder neural network consisting of just four hidden layers. The encode region is made up of 256 neurons in the first layer and 128 in the second layer while the decode region is an exact reflection of the encode. The weights among neurons are updated with the sigmoid activation function. Furthermore, we attain our results with lightweight online training of 10 iterations. We execute back-propagation with the Tensorflow root mean square propagation (RMSProp) optimizer to minimize the mean squared error (MSE).

4) Alternative Schemes: To evaluate Self-Patch's performance, we compare it with several baseline methods. Self-Patch consists of three phases, i.e., attack detection, attack classification, and targeted patch execution. For the attack detection phase, we compare Self-Patch against k-nearest neighbors (k-NN) [12] and k-means [13] techniques. For the targeted patch execution phase, we evaluate our approach against the *whole upgrade* method. We describe each alternative method in detail below.

k-NN for anomaly detection: We take the system call frequency vectors as input and return the outliers at their corresponding timestamps. The k-NN algorithm typically involves assigning a label to a data point based on the majority vote from its k closest neighbors. Abnormal samples are those too far away from their neighbors. We calculate the average distance of each point to its nearest neighbors and determine the anomalous ones with larger distances. In our experiment, we empirically select k as 5. In addition, we choose the samples with the top 10% largest average neighbor distance as the outliers.

k-means for anomaly detection: We customize the k-means algorithm for the anomaly detection phase in a similar way to the k-NN algorithm. To be specific, data samples are distributed to one of the k randomly initialized cluster centers. Thereafter, the cluster centers are recalculated based on the average position of its members and then cluster memberships are reassigned. This process is performed iteratively until no more change occurs. Here, the algorithm identifies abnormal samples as those that belong to isolated clusters with a little membership. Similar to that of k-NN, we tune k-means parameters to achieve a good tradeoff between true positive rate and false positive rate in our experiments. The resulting number of clusters is equal to ten (k = 10), while the cluster threshold for determining anomalous clusters is set to 100.

Whole upgrade for patch execution: This approach refers to the conventional manner in which security updates are performed in Debian-based Linux systems that containers run on. Old versions of all packages found by the package manager are updated to their newest versions. In our study, this is accomplished by an `apt-get update` followed by an `apt-get upgrade` of the APT package manager. The update command refreshes the package source lists to find the latest available packages while the latter installs the newly found software versions. We also employ the corresponding commands for containers based on Alpine Linux (i.e. replacing `apt` with `apk`).

B. Results and Analysis

In this subsection, we discuss our experimental results of each component of Self-Patch.

1) Attack Detection Results: We present the detection results of Self-Patch over three evaluation metrics, i.e., true positive rate, false positive rate (FPR) and lead time. For detection coverage, we measure whether the attack is detected by checking whether the alarm is raised after the attack is triggered and *before* the attack is successful. The detection coverage is also referred as true positive rate (TPR) in this paper calculated by the following standard true positive rate equation, where TP is number of attacks that are detected and FN is the number of attacks that are undetected.

$$TPR = \frac{TP}{TP + FN} \tag{1}$$

Next, we use the standard false positive rate FPR as the second evaluation metric. FP represents the number of false alarms and TN represents the number of normal data samples that Self-Patch correctly does not generate alarms on.

$$FPR = \frac{FP}{FP + TN} \tag{2}$$

Lastly, we assess detection performance using lead time as the third metric. Lead time is defined to be the amount of time between the first alert from the detector after the malicious command is executed and completed. This represents the amount of flexibility the system has to initiate security countermeasures before the container is fully compromised.

TABLE III: Detection result of Self-Patch and alternative approaches.

Threat Impact	CVE ID	CVSS Score	k-NN	k-Means	Self-Patch
Return a shell and execute arbitrary code	CVE-2012-1823	7.5	✗	✗	✗
	CVE-2014-3120	6.8	✗	✓	✓
	CVE-2015-1427	7.5	✓	✓	✓
	CVE-2015-2208	7.5	✗	✗	✓
	CVE-2015-3306	10	✗	✓	✓
	CVE-2015-8103	7.5	✗	✓	✓
	CVE-2016-10033	7.5	✗	✓	✓
	CVE-2016-3088	7.5	✗	✓	✓
	CVE-2016-9920	6	✗	✗	✓
	CVE-2017-11610	9	✗	✓	✓
	CVE-2017-12615	6.8	✗	✗	✓
	CVE-2017-7494	10	✗	✓	✓
	CVE-2017-8291	6.8	✗	✗	✓
Execute arbitrary code	CVE-2014-6271	10	✗	✗	✓
	CVE-2015-8562	7.5	✗	✓	✗
	CVE-2016-3714	10	✗	✓	✓
	CVE-2017-12794	4.3	✗	✓	✗
	CVE-2017-5638	10	✗	✓	✓
	CVE-2018-16509	9.3	✗	✗	✓
	CVE-2018-19475	6.8	✗	✗	✓
	CVE-2019-6116	6.8	✗	✗	✓
Disclose credential information	CVE-2014-0160	5	✗	✓	✓
	CVE-2015-5531	5	✗	✓	✓
	CVE-2017-7529	5	✗	✓	✗
	CVE-2017-8917	7.5	✗	✓	✓
	CVE-2018-15473	5	✗	✓	✓
Consume excessive CPU	CVE-2014-0050	7.5	✗	✓	✓
	CVE-2016-6515	7.8	✗	✓	✓
Crash the application	CVE-2015-5477	7.8	✗	✗	✗
	CVE-2016-7434	5	✓	✓	✗
Escalate privilege level	CVE-2017-12635	10	✗	✓	✓
Average Results			6.45%	67.74%	80.65%

Table III shows the detection results for each detection approach (i.e. k-NN, k-means and Self-Patch). The detection coverage results show that k-NN performs much more poorly than k-means and Self-Patch. k-NN detects 2 of 31 attacks (6.45%), whereas, k-Means and Self-Patch detection recognize 21 (67.74%) and 25 (80.65%), respectively. Self-Patch also demonstrates superior performance with a lower average FPR of 0.72% than the 7.16% k-Means result. The average lead time of Self-Patch is the longest (16.38 seconds), compared with both k-means (13.53 seconds) and k-NN (0.15 seconds). Although the attacks have varied attack periods, noted in Table II, Self-Patch more consistently yields higher lead time.

We express the detection coverage, FPR and lead time of each method over the attacks in each threat impact category in Figure 3, 4 and 5, respectively. Figure 3 shows that Self-Patch achieves the highest detection coverage in all but two categories: *disclose credential information* and *crash the application*. Although k-means outperforms Self-Patch in these areas, it suffers from a much higher false positive rate. Furthermore, Self-Patch as well as the other detection approaches struggle with attacks that crash the application. This is likely because the crash causes the container to end abruptly and lose data before an alarm is confidently raised. We plan to improve the accuracy of Self-Patch in future work with strategies that leverage system call arguments.

2) Attack Classification Results: We examine the patterns of top system calls, generated from attack classification, that correspond to the attacks against each vulnerability. We observe that Self-Patch produces unique patterns for 29 of the 31 CVEs. In particular, the duplicates are only observed among three of four containers of the GhostScript application used for image processing. However, other applications with

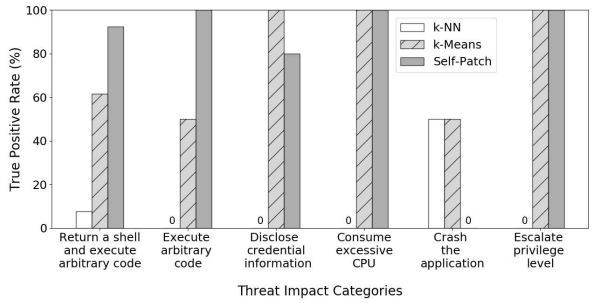

Fig. 3: True positive rate result of anomaly detection approaches.

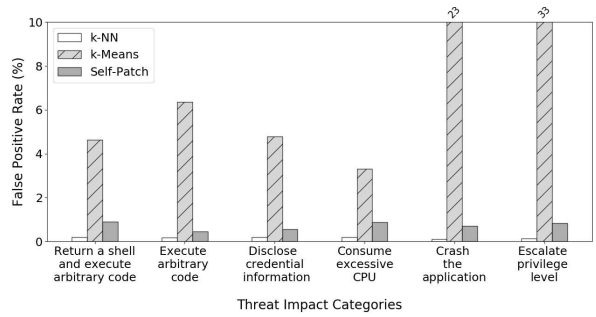

Fig. 4: False positive rate result of anomaly detection approaches.

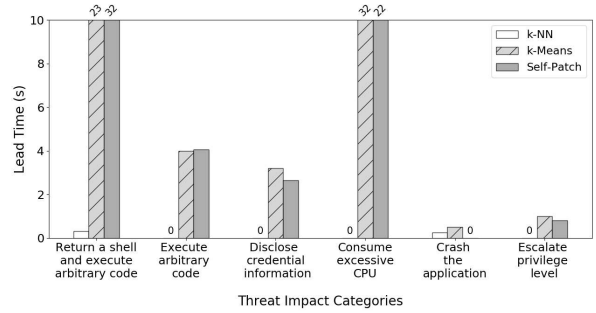

Fig. 5: Lead time result of anomaly detection approaches.

multiple containers of distinct vulnerabilities do not yield identical signatures. The GhostScripts attacks exploiting CVE-2018-16509, CVE-2018-19475 and CVE-2019-6116 are of a similar fashion. They involve uploading vulnerable image files embedded with malicious content to bypass the GhostScript security sandbox and execute commands. Thus, one can expect similar behavior from these attacks. Indeed, the GhostScript vulnerabilities can all be addressed by the same targeted patch.

3) Patching Results: We discuss the patching results, including success status and patching costs, i.e., memory and disk costs. Table IV summarizes the success rate of the patching approaches. Self-Patch achieves 80.65% success rate. Note that this is a 100% of the cases where the attack was detected by Self-Patch. However, only 6.45% of whole upgrade trials

TABLE IV: Overall comparison result of different patching approaches.

Patching Approach	Success Rate	Memory Cost	Disk Cost
Whole Upgrade	6.45%	10.13x	1.49x
Self-Patch	80.65%	4.79x	1.16x

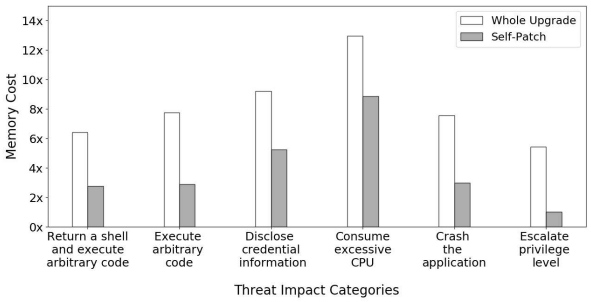

Fig. 6: Container memory cost of patching.

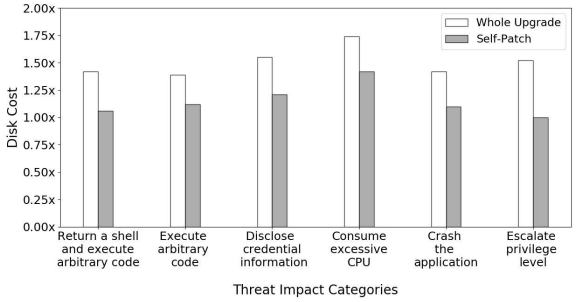

Fig. 7: Container disk cost of patching.

are successful. This demonstrates the superiority of targeted patching over periodic updates. The reason why whole upgrade achieves such low success rate is because many applications are not configured to work with package managers. There are many programs whose developers have not prepared the files that will be handled by the manager (e.g., debian files for APT) or have provided other installation means that users prefer. Therefore, for applications not managed by APT, the periodic update process will upgrade libraries other than those needed to address the vulnerability of the containerized application in question.

In addition, the patching cost results show lower memory and disk size footprint when performing targeted patching rather than the whole upgrade approach. On average, the memory size grew to 4.79 times its original size with targeted patching versus 10.13 times with whole upgrade. Similarly, targeted patching multiplied the disk size by a factor of 1.16, whereas whole upgrade increased the size by a factor of 1.49. Self-Patch attains lower memory and disk costs over the whole upgrade approach across all the threat impact categories as shown in Figure 6 and Figure 7.

IV. Related Work

Previous work has been done in applying machine learning techniques in intrusion detection systems. DeepLog [14] utilize Long Short-Term Memory (LSTM) to learn log patterns during normal run and detect anomalous system events in production systems. Tiresias [15] leverage Recurrent Neural Networks (RNNs) to predict future security events, in order to detect malicious activities. Marinescu et al. [16] automatically learn authorization rules and extract invariants in modern online social networks. Banescu el al. [17] apply regression models to predict the time period that software protection transformations are able to withstand various attacks. In comparison to the existing work, Self-Patch proposes to perform feature extraction over system call trace data and apply unsupervised autoencoder neural network to achieve robust online attack detection.

In addition, previous work has been done on modifying the application binaries on-the-fly to quarantine the system without experiencing down times. FIBER [18] analyzes open source security patches and generates binary signatures that are used to provide new patches for similar vulnerabilities. KARMA [19] patches Android kernels at multiple levels to filter malicious inputs with little runtime overhead. Piston [20] takes control of software on an embedded device and modifies the binary code on-the-fly to protect the system. Compared with the existing work, Self-Patch leverages online anomaly detection and attack classification to achieve self-triggering targeted patching.

V. Conclusion

In this paper, we have presented Self-Patch, a new self-triggering targeted patching framework for container-based distributed computing environments. Self-Patch aims at providing effective and efficient solutions to protect containerized applications from security attacks. To achieve this goal, the Self-Patch framework consists of three coordinating components: 1) an online attack detection module which can dynamically detect abnormal attack activities by extracting feature vectors from system call traces and applying unsupervised machine learning methods over the extracted features; 2) an attack classification scheme which classifies a detected attack into a specific type linked to a certain CVE; and 3) a targeted patch execution module which can install proper software patches to fix the vulnerability. We have implemented a prototype of Self-Patch and evaluated it over 31 real-world vulnerabilities discovered in 23 common server applications. Our initial experimental results are promising, which shows we can increase detection rate to over 80% and reduce false alarm rate to 0.7%. In contrast, traditional schemes can either only detect 6% attacks or incur more than 20% false alarms. Compared to the whole software upgrade approach, Self-Patch can reduce the memory overhead by up to 84% and disk overhead by up to 40%.

VI. Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is supported by the NSA Science of Security Lablet: Impact through Research, Scientific Methods, and Community Development under the contract number H98230-17-D-0080. Any opinions, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] "Docker image vulnerability research," Federacy, 2017. [Online]. Available: https://www.federacy.com/docker_image_vulnerabilities
[2] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 269–280.
[3] A. Bettini, "Vulnerability exploitation in docker container environments," *FlawCheck, Black Hat Europe*, 2015. [Online]. Available: https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments-wp.pdf
[4] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.
[5] Sysdig, 2019. [Online]. Available: https://github.com/draios/sysdig
[6] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Symposium on Security and Privacy*. IEEE, 1996, pp. 120–128.
[7] E. Carter, "2018 docker usage report," Sysdig, 2018. [Online]. Available: https://sysdig.com/blog/2018-docker-usage-report
[8] "8 surprising facts about real docker adoption," Datadog, 2018. [Online]. Available: https://datadoghq.com/docker-adoption
[9] VulHub: Pre-Built Vulnerable Environments Based on Docker-Compose, 2019. [Online]. Available: http://vulhub.org/
[10] Apache JMeter, 2018. [Online]. Available: https://jmeter.apache.org/
[11] Google cAdvisor, 2019. [Online]. Available: https://github.com/google/cadvisor
[12] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.
[13] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *Transactions on Pattern Analysis & Machine Intelligence*, no. 7, pp. 881–892, 2002.
[14] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1285–1298.
[15] Y. Shen, E. Mariconti, P. A. Vervier, and G. Stringhini, "Tiresias: Predicting security events through deep learning," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 592–605.
[16] P. Marinescu, C. Parry, M. Pomarole, Y. Tian, P. Tague, and I. Papagiannis, "Ivd: Automatic learning and enforcement of authorization rules in online social networks," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 1094–1109.
[17] S. Banescu, C. Collberg, and A. Pretschner, "Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning," in *26th USENIX Security Symposium*, 2017, pp. 661–678.
[18] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th USENIX Security Symposium*, 2018, pp. 887–902.
[19] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1253–1270.
[20] C. Salls, Y. Shoshitaishvili, N. Stephens, C. Kruegel, and G. Vigna, "Piston: Uncooperative remote runtime patching," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 141–153.