2016 IEEE 2nd International Conference on Collaboration and Internet Computing

Smart Moving Target Defense for Linux Container Resiliency

Mohamed Azab
The City of Scientific Research and Technological Applications, Alexandria, Egypt
mazab@vt.edu

Bassem Mokhtar
Electrical Engineering Department
Alexandria University
bmokhtar@alexu.edu.eg

Amr S. Abed
ECE Department
Virginia Tech
amrabed@vt.edu

Mohamed Eltoweissy
CIS Department
Virginia Military Institute
eltoweissymy@vmi.edu

***Abstract*—Nature is a major source of inspiration for many of the inventions that we rely on to maintain our daily lifestyle. In this paper, we present ESCAPE, an evolved version of our nature-inspired game-like informed moving-target-defense mechanism for cloud containers resiliency. ESCAPE rely on a novel container mobilization framework controlled by a smart attack maneuvering module. That module drives the running containers based on real-time models of the interaction between attackers and their targets as a "predator searching for a prey" search game. ESCAPE employs run-time live-migration of Linux-containers (*prey*) to avoid attacks (*predator*) and failures. The entire process is guided by a novel host-based behavior-monitoring system that seamlessly monitors containers for indications of intrusions and attacks. To evaluate the effect of ESCAPE's container live-migration evading attacks, we extensively simulated the attack avoidance process based on a mathematical model mimicking the prey-vs-predator search game. With ESCAPE's live-migrations, results show high container survival probabilities with minimal added overhead.**

I. Introuction

Linux containers running in a commercial cloud environments share the same kernel with containers from other customers. By sharing the same kernel with the host and other containers, the attack surface is wider than the case for virtual machines where each VM has its own kernel. The light weight and efficiency of Linux container qualified it to be the future of application virtualization especially for cloud applications [1].

To protect Linux containers in a shared environment, service providers are expected to monitor the behavior of the containers running on their system for suspicious activity. Upon detection of a possible threat, the service provider should take an action to protect the guest containers. While the most straightforward action is to kill the misbehaving container and inform the owner of the detected anomaly, such action may not be cost effective especially for long running stateful applications. For instance, if an application is running for a few days, and it gets attacked by an external attacker, and due to the behavior change, recognized by the monitor system as a potential attack, the default action would be for the container to be terminated. Now, the owner will have to restart the application and have it running for more few days just to get to the same point it was already at when it got terminated.

A more cost-effective alternative was presented in [2] to take a snapshot of the current status of the running application while in safe state. Upon attack detection, the system simply rolls back to the most recent safe state saved. One drawback with this approach is when the last saved safe state is a vulnerable state and/or the attack is persistent, in which case the container will go into a continuous loop of restores.

To overcome such limitations, we introduced our preliminary version of ESCAPE in [3]. We proposed a nature inspired approach that aims at changing the container execution environment in order to mislead a persistent attack. In this paper we propose an evolved version of ESCAPE with a mature system archetecture and a more mature guidance. The proposed system aims to equip the attacker target (prey) with the tools needed to ESCAPE from the attacker (predator) "ex, moving the container to a random remote host".

In this paper, we enable runtime live-migration of cloud containers as a moving target defense (MTD) [4]–[7] mechanism against host-based persistent attacks.

The MTD mechanism is guided by a host-based intrusion detection system (HIDS) [8] [9] that monitors operating containers to detect potential anomalies or misbehaviors. The HIDS learns the behavior of all the containers running on the host, and upon detection of a change of behavior of one or more container, the HIDS signals the MTD module of the system to ESCAPE the affected container. We considered two types of applications, stateless and stateful applications.

The system reacts gradually and according to the application type. For stateful applications, the system is configured to start by using the checkpoint/restore mechanism before switching to the live-migration solution for a more persistent attack. Delaying the use of live-migration until the attack is known to be persistent (e.g. after N rollback attempts) saves the running application overhead associated with live-migration as compared to checkpoint/restore for non-persistent attacks. For a stateless application, e.g. static web servers, upon attack detection, the system simply restarts the server or ESCAPE the container to a new host while rerouting the associated network connections.

The entire process is designed to mimic the famous search-game "Prey-vs-Predator" [10]. This game describes a scenario were multiple predators are targeting specific prey moving in a forest. We adopted this model into a guidance mechanism to guide the prey in its mission to evade the predator. In our

978-1-5090-4607-2/16 $31.00 © 2016 IEEE
DOI 10.1109/CIC.2016.26

scenario, the cloud is the forest. Each host in the cloud is a potential escape location for the prey. The Prey is the attacker ("predator") targeted container-encapsulated application.

We built a simulation model to evaluate the effectiveness of the proposed approach in evading attacks. Additionally, we conducted preliminary experiments on our local ACIS cloud to evaluate the overhead of live-migration of Linux-containers. Results showed the effectiveness of our approach with a limited to no overhead due to live-migration.

The rest of this paper is organized as follows. Section II provides an overview of the proposed ESCAPE system. Section III discusses the live-migration process within ESCAPE, while section IV presents the developed mathematical model for evaluating ESCAPE's live-migration processes for evading host-based attacks. Section V gives a brief summary of related work. Section VI concludes with summary and future directions.

II. System Overview

ESCAPE was built to be as general as possible with minimal application customization. The main advantage of presenting generic defense tools is to give the user/system administrator the chance to select the most appropriate tools and applications that suits their needs with no constraints or limitations.

ESCAPE supports general purpose Linux containers as a lightweight operating system virtualization technology. We used Docker [11], an LXC-based container management tool hosted on Linux operating systems to sandbox the user applications. Docker employs the resource isolation features of the Linux kernel to allow independent containers to run in total isolation from each other and the underlying host. The host kernel isolates the container and the contained applications views of the operating environment within a single Linux instance including process trees, network, user IDs and mounted file systems, while the cgroups provide resource isolation, including CPU, memory, block I/O and network. There are other Linux container technologies that can serve the same purpose such as LXC [12] or OpenVZ [13]. We selected Docker due to its small footprint and fast instantiation. However, the same concepts discussed here can still apply to LXC and OpenVZ.

To enable checkpointing of running application, we are using an experimental version of Docker that integrates a checkpointing tool named CRIU [14] to momentarily freeze the running container and its enclosed applications taking a live snapshot of the memory content and any used files. The dumped images are stored in persistent.

A. Application Encapsulation

The first step is to configure the container to host the user application. Users can deploy an empty container and customize it manually by logging into it and install all packages. They can also get that done automatically by customizing an automatic deployment configuration file. Once the container is up and running, ESCAPE uses an integrated export tool to dump the customized container into a set of files that can be executed independently from the Docker management demon service. Doing so enabled us to execute the container in an unprivileged mode in the user space for easier management. The described process is shown in figure 1.

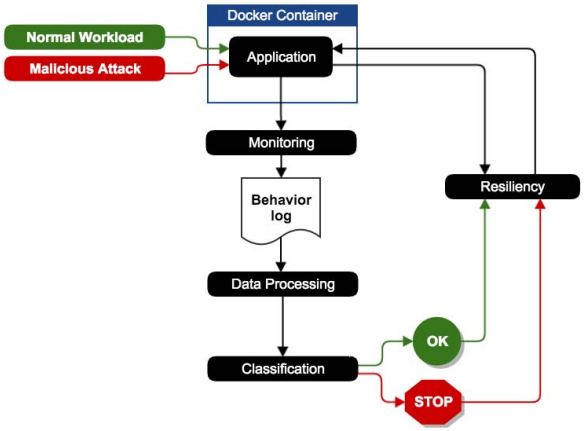

Fig. 1. Simplified Logic Flow Diagram and System Architecture

For monitoring purposes, we adopted the intrusion detection system introduced in [9] to monitor the operating containers. The system is specifically tailored for monitoring the behavior of Linux containers, and depends on the fact that Linux containers communicates with the host kernel and the outer world by using system calls issued to the host kernel.

The ESCAPE's monitoring system uses a bag of system calls (BoSC) [15] technique for learning the container behavior. The BoSC is a frequency-based technique where a sliding window of size N moves over the log of system calls, counts the frequency of each distinct system call within the current window, and creates a BoSC with the current frequencies. A bag of system calls is an array $< c_1, c_2, \ldots, c_{n_s} >$ where c_i is the number of occurrences of system call, s_i, in the current window, and n_s is the total number of distinct system calls. For a window size of 10, the sum of all entries of the array equals 10, i.e. $\sum_{i=1}^{n_s} c_i = 10$. A sample BoSC is shown below for n_s = 20 and N = 10.

```
[0,1,0,2,0,0,0,0,1,0,4,0,0,0,0,0,1,0,0,1]
```

ESCAPE?s monitoring system employs a background service running on the host kernel to monitor system calls between any Docker containers and the host Kernel. Upon start of a container, the service uses the open-source `sysdig` tool [16] to trace all system calls issued by the container to the host kernel. The full trace is written to a log file that is processed in real time and used to learn the container behavior.

The system reads the behavior log file epoch by epoch. For each epoch, a sliding window of size 10 is moved over the system calls of the current epoch, counting the number of occurrences of each distinct system call in the current window, and producing a BoSC. When a new occurrence of a system call is encountered, the corresponding index of the

BoSC is updated. If the current BoSC already exists in the normal-behavior database, its frequency is incremented by 1. Otherwise, the new BoSC is added to the database with initial frequency of 1.

For detection mode, the system reads the behavior file epoch by epoch. For each epoch, a sliding window is similarly used to check if the current BoSC is present in the database of normal behavior database. If a BoSC is not present in the database, a mismatch is declared. The trace is declared anomalous if the number of mismatches within one epoch exceeds a certain threshold. Upon detection the resolution mechanism is called and the migration process is triggered to mimic the simulation scenario described in section IV.

B. Container Networking

Linux containers are a software construct that can host an application and its dependencies as an isolated process on a Linux kernel. It allows containerized applications to share that kernel with other containers. The basic network primitive in Docker is a virtual bridge called docker0. When Docker boots up on a Linux server, it creates a default docker0 bridge inside the Linux kernel, and docker0 creates a virtual subnet on the Docker host so it can pass packets back and forth between containers on the same host. Docker also creates a pair of virtual interfaces on each container, randomly assigning them an IP address and a subnet from a private address range not already used by the host machine.

We prefer using a virtual network interface with a dedicated IP address for each container to facilitate runtime migration. We give the contained processes the right to directly access a dedicated network interface as its own. ESCAPE will handle the runtime mapping by local or network wide mapping of interfaces and IPs. In order to escape from Docker engine network management and enable such direct association we had to launch that container as an independent process without losing the isolation feature that Docker provides.

ESCAPE uses an integrated export tool to dumb the configured container into a set of image files. ESCAPE uses runC [17], a tiny tool for spawning and running containers according to the Open Container Protocol specification, to execute the container as a sub-process of runC. In ESCAPE, containers are started as a child process of runC facilitating executing the enclosed applications in the user space while maintaining the same level of isolation provided by the cgroups and containers. With the container running as a child process of runC, we managed to give the enclosed application the right to listen directly on any set of ports on the dedicated network interface for that application.

C. Container Checkpoint/Restore and Live-migration

Application running inside the container whether stateless or state-full use the host memory to host all runtime related libraries, calculations, and other volatile contents. it is too hard to recover these memory contents upon failure or migration events. ESCAPE were built to serve both types of applications with minor to no interference from the administrator or the programmer [5]. Our primary goal is to avoid any container customization. We leveraged the encapsulated state of the application and used CRIU [14] to dump the container memory into persistent set of files easy to share and recover. CRIU is used only on state-full applications based containers, we prefer not to use it on stateless type as the memory content and the executed states are not important for container restoration.

CRIU [14] is a tool to checkpoint/restore running tasks in user space. CRIU momentarily freezes the running (container) runC process and all its sub-processes (user apps) and checkpoint it to a collection of image files that can be used to restore the container to the exact state later. Between these dump events, containers uses the host memory to operate to maximize the application response rate.

For a failed container to be restored, the hosting server must have access to the container image files, and the memory dump files. The container image files is usually large in terms of space. In our experiments, containers with full database server can be as large as 500MB. However, the memory dumb is usually less than 10MB. The migration process for stateless type is much easier, we replicate the container on the destination server, then make a quick network switching between the source and destination. The replication process and container instantiation time is totally negligible as the original container will still be running (assuming that we are not recovering from failure), the migration process is entirely logical as the network connections are the only this that is going to ESCAPE . ESCAPE adjust the ARP table for the NAT to point to the new server instead of the old one.

For faster instantiation, quick recovery, and easy container migration, ESCAPE uses a remote shared storage as a container repository to store runC containers. Running the container from a remote storage gives instant access to multiple remote servers to instantiate such containers. Using remote repositories to host the base image of the container, massively reduced the time needed to move all the files between hosts in case of failure or live-migration. The only files that has to be synchronized between the source and destination servers are the memory dump which are so small and synchronize momentarily.

Figure 2 shows the migration process, were applications are encapsulated in containers hosted on remote Linux servers.

III. Live-Migration Process

As mentioned before, ESCAPE leverages the loosely coupled nature of the exported container defined as a set of files and uses a local shared storage service to host the running containers. Once the container starts, the entire operations of the enclosed applications will be handled on the host memory. The use of a shared storage has no effect on the container execution time. However, hosting the container files and the checkpoint dump on shared storage massively decrease the time needed to start a migration between hosts, and isolates the container files from the host attachment in case of host failure.

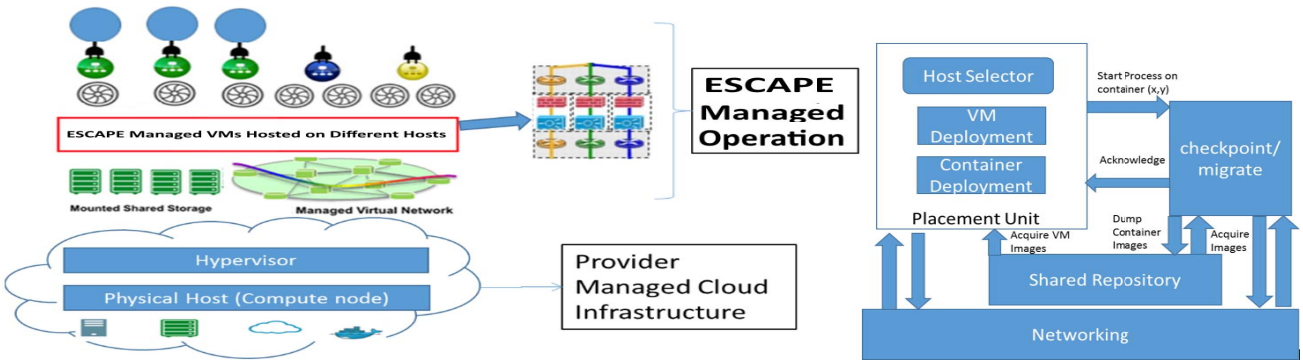

Fig. 2. Migration Process

For stateless-application containers, there is no need for checkpointing as the memory content is irrelevant for application re-execution. The relaxation of that constraint made it much easier for ESCAPE to recover such container in case of failure/attack.

ESCAPE starts the migration process by instantiating/selecting a suitable destination. The current implementation follows the prey-vs predator model introduced in section V to select the next destination for the migrating container. ESCAPE can also select a destination with far logical distance from the source to evade attackers. The logical distance is defined as based on a heterogeneity scale. The more different the destination is in terms of configuration, network, and datacenter association the more likely for ESCAPE to select. ESCAPE aims to select such host to complicate the reapplication of the same cause of failure. However, considering that level of heterogeneity might induce resurrection conflicts. The details of that complicated migrations will be included in our sequel papers.

Upon selecting an appropriate destination, ESCAPE mounts the shared storage drive that holds the running container, and adjust the network settings to facilitate network relocation. ESCAPE isolate the destination from the active network by disabling all interfaces, and start the container startup process. Once the container is on, ESCAPE alters the ARP table to redirect the network traffic from the source to the destination host, and disable the source host at the same process.

The migration process starts by checkpointing the container, killing the process on the original host, make an ARP update to change the MAC/IP assignment of the old server network interface to match the new one while mainlining the IP value , and restore the container and all enclosed applications on the destination server. The entire process occurs in matter of milliseconds. Following the aforementioned process guarantee almost zero downtime unless the source host fails completely during the process.

IV. Mathematical Model

A. Model Design

We have generated a mathematical model for evaluating the efficiency of the live-migrations mechanism of application containers adopted by ESCAPE to evading host-based cyber attacks. The developed model depends on theories and models of evolutionary search games [18] [10]. Since many cyber security-related problems are considered as complex problems, a lot of such problems are abstracted based on game theories to basic fundamental problems, such as hide-and-seek and predator-prey problems [19]. We adopted the predator-prey model as a reference model to develop our mathematical model assuming an evasive prey (i.e., mobile prey) [20]. Our model simulates and evaluates ESCAPE in mobilizing targeted application Linux-Containers (i.e., prey) across a set of attack-prone Virtual/Physical hosts within a virtual network considering some of application containers implemented at those hosts are under attacks (i.e., predators).

As discussed before, ESCAPE is designed to mitigate various host-based cyber security attacks that might exit in cloud computing environments with various application containers. In our developed model, we consider the operation of ESCAPE to protect running applications and their related containers against generic host-based cyber attacks. We assume a networking context that comprises malicious and benign users. The attacker goal is to malfunction targeted application containers and extract valuable information. The attacker will succeed in case of spending a time with the victim container on the same host machine shorter than the time required to detect its harmful or abnormal effects.

We target in our ESCAPE performance evaluation study the survival probability of a targeted application container implemented on a host when operating over a cloud of networking hosts. Our model will calculate the survival probability assuming that there is a capability of migrating the valuable application container from a host to another one. We assume that there is one targeted application container that should be protected and the studied networking area is a two-dimensional square area. The following points discuss the developed model.

1) Survival Probability of Targeted Application Containers: The survival probability P_{static} of a static application container in a host is defined in equation 1

$$P_{static}(t) = e^{-\rho S(t)} \quad (1)$$

Where

- ρ: the hacked host density, $\rho = N/V$ that N is the number of hacked hosts and V is the number of all hosts in the studied network.
- $S(t)$: the mean number of distinct hosts visited by mobile attackers up to time t and hosts' application containers malfunction. As t increases, number of visited hosts increases (referring to the mobility of attacks and the possibility of having attacks widely spreading at many hosts). $S(t) \approx \pi t/\big(ln(t)\big)$ for a two-dimensional square area [20].

The survival probability $P_{mobile}(t)$ of a mobile application container is defined in equation 2

$$ln\Big(P_{mobile}(t)\Big) \approx \Big(\frac{N}{V}\Big)^2 ln\Big(P_{static}(T)\Big) \quad (2)$$

2) Impact of Attacker Success on Survival Probabilities: We assume that not all visited hosts by attackers will misbehave and only some related application containers will breakdown. In other words, we propose that a fraction of $S(t)$ will exist with exponential probability $1 - e^{-t_d}$ where t_d is the required time of detecting attacker's visits to a host.

Accordingly, as t_d increases, this means that attackers can malfunction application containers in visited hosts without detection, i.e. for $t_d = 0$, all attackers? visits are detected (no successful attacks). t_d might depend on the operating context and hardware of related hosts. Attackers might visit many hosts, however, no fixed number of containers exist at each visited host. So, we consider average t_d for all hosts in our conducted simulation scenarios.

Equation 3 shows the mean number of visited hosts by mobile attackers in case that the attackers, successfully, are able to malfunction targeted application containers within the same host without detection.

$$S(t)_{success} = (1 - e^{-t_d})S(t) \quad (3)$$

3) Models of Attack Growth: We consider two different growth models of attacks which are exponential and logistic growth as discussed in [10]. Those models discuss different rates of growth which refer to the spreading rate of attack in a networking context and how ESCAPE can succeed in mitigating such growth.

- *Rapid increase (exponential) of hacked hosts*

$$N(t) = N(0)e^{kt}$$

  where $N(0)$ is the initial number of hacked hosts at time 0 and k is a constant related to the increasing rate at any time t

- *Slow increase (logistic) of hacked hosts*

$$N(t) = \frac{\mu N(0)e^k t}{N(0)e^k t + \mu - N(0)}$$

  where μ is the carrying capacity which controls the increasing rate of N that when N approaches μ, the increasing rate approaches zero. Also, $N(t)$ approaches $N(0)$ when $t \to 0$, and approaches μ when $t \to \infty$.

B. Evaluation Study

We conducted a simple simulation scenario of application containers cloud established by a set of connected hosts, as shown in Figure 3, considering the following assumptions:

- All hosts are Internet-enabled devices
- Neighboring hosts to a hacked host are susceptible to the same attack
- We have one prey (one host machine with a targeted application container) and N hacked hosts and V normal and hacked hosts (where $N \leq V$)
- The application container can be migrated from a host to another one. This will be shown in scenarios with mobility feature.

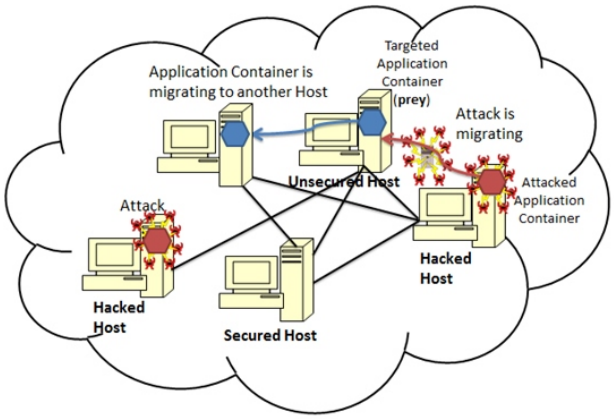

Fig. 3. Simulated Scenario of an Application Container Cloud

As mentioned before, the attacker goal is to malfunction the targeted application containers implemented at hosts. The N attackers are targeting the host with the valuable application container. When the attacker reaches a certain host, it spends some time on it, then, all application containers working on that host will fail. Our scenario considers the following cases

- In case of mobile application container scenario, containers can be migrated from a host to another one
- In case of static application container scenario, containers are settled in one host

We study the effect of having small and large number of hacked hosts (N) on the survival probabilities of a static and mobile targeted application container. Our simulation scenarios consider the following two cases during runs:

1) Static N
   - N is fixed during the simulation (i.e., $\frac{dN(t)}{dt} = 0$)
   - During the simulation time, the attackers migrate their related containers (attacks) from one Host to

another preserving the total number of hacked Hosts (N)

2) Dynamic N
   - We repeat the scenarios when N has exponential and logistic increasing rates and how N affects the survival probabilities of targeted containers
   - N varies with the simulation time (i.e., $\frac{dN(t)}{dt} \neq 0$)
   - During the simulation, attackers migrate to other hosts and attack new hosts (i.e., number of hacked hosts increases with the simulation time)

Then, we repeat the previous scenarios when applying the concept of attacker success on survival probabilities of targeted application containers.

The mathematical model equations were built using MATLAB 2013. The simulation runs were executed on a Windows 10 operating system machine. Table I shows the simulation parameters.

TABLE I
SIMULATION PARAMETERS

Parameter	Value
Number of hosts with the targeted application container (prey) (fixed)	1
Number of hacked hosts (N) (variable)	1,2,3,4,5,7
Total number of victim hosts (V) (variable)	20,30,40,50
Exponential increasing rate constant (k)	1,2,3,4
Initial number of hacked hosts at time 0	1
Log increasing rate constant (k)	1,2,3,4
Log carrying capacity constant (μ)	1,2,3,4
Average attack detection time (t_d) (variable)	0,0.2,0.5,1,10
Simulation time (time unit)	100

Figure 4 shows the impact of changing the number of hacked hosts (N) on a network of 20 hosts where the targeted contained located in one host. For each studied case, the value of N does not change over the simulation time. The figure shows that we can have higher survival probabilities in case of having mobile targeted application container capability. Also, as we increase the number of hacked hosts, the survival probability gets worse.

Figure 5 shows that at various detection times and 7 hacked hosts, we can get better survival probabilities at smaller detection times. If we compared the obtained results in figure 5 with the simulated survival probability value at case $N = 7$ at figure 4, we can notice the improvement in the results at small detection times. So, in case that ESCAPE employs efficient intrusion detection systems, it can move the targeted container from attacked hosts and get high survival probabilities.

On the other hand and compared with the obtained results in figure 5, we got small survival probabilities at various attack detection times in case of having static application containers as depicted in figure 6.

Figures 7 and 8 show the impact of having logistic growth of attacks on the survival probability of a static/mobile application containers in case of having various increasing rates and attack detection times, respectively.

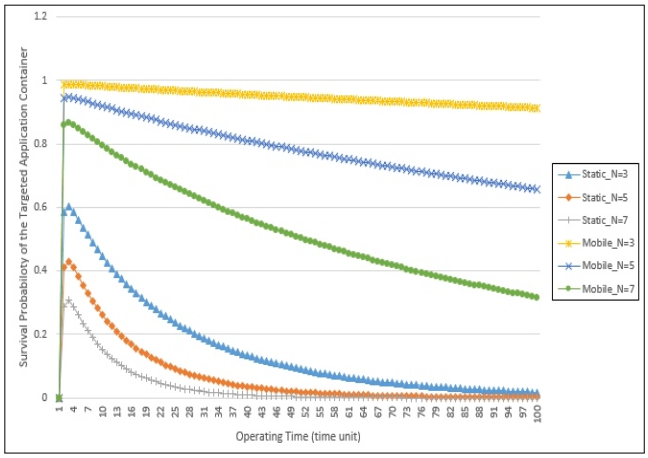

Fig. 4. Survival probability of a static/mobile container at different number of hacked hosts (N)

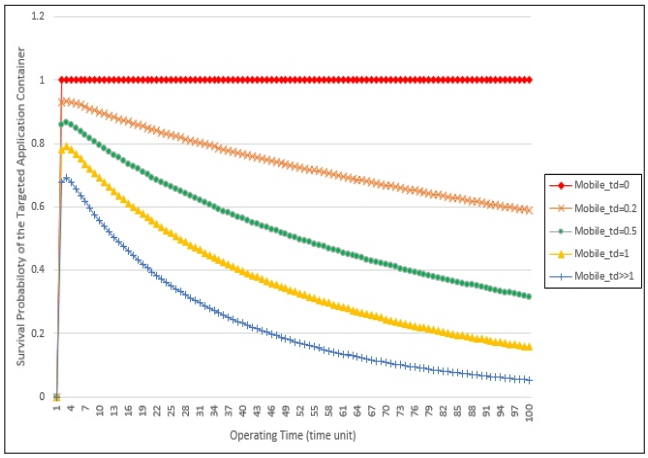

Fig. 5. Survival probability of a targeted mobile container at 7 hacked hosts and various attack detection times

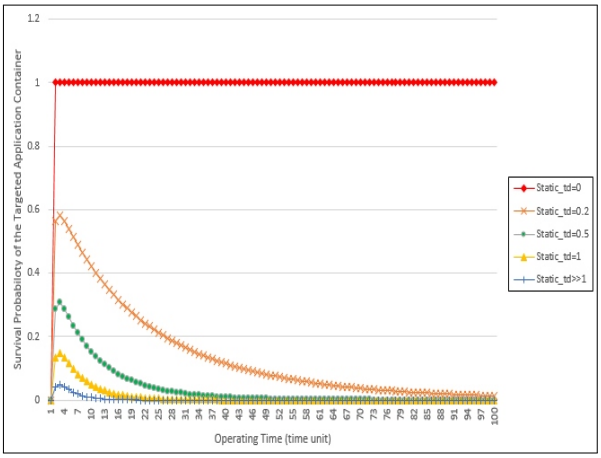

Fig. 6. Survival probability of a targeted static container at 7 hacked hosts and various attack detection times

For figure 7, as we increase the growth rate of attacks and their spread in many hosts in the network, this leads to lower survival probabilities of the targeted container. In figure 8, we assume a certain operating parameters for the logistic growth where $N(0) = \mu = k = 4, V = 20$. As we have small attack detection times, we can mitigate the high growth rate of attacks and their spread in many hosts. Consequently, this leads to higher survival probabilities of the targeted container compared with the same case obtained in figure 7.

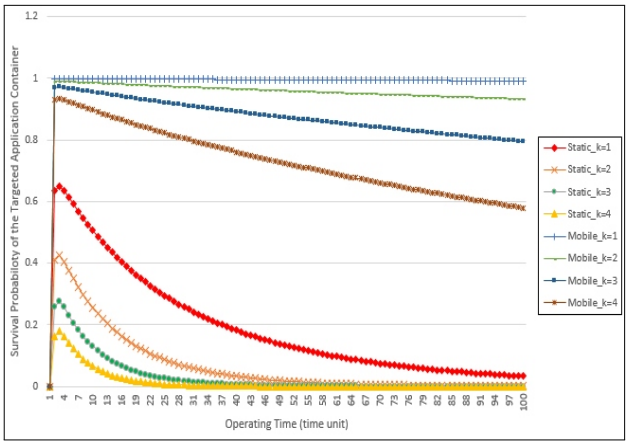

Fig. 7. Survival probability of a static/mobile targeted container at various increasing rates of logistic attack growth

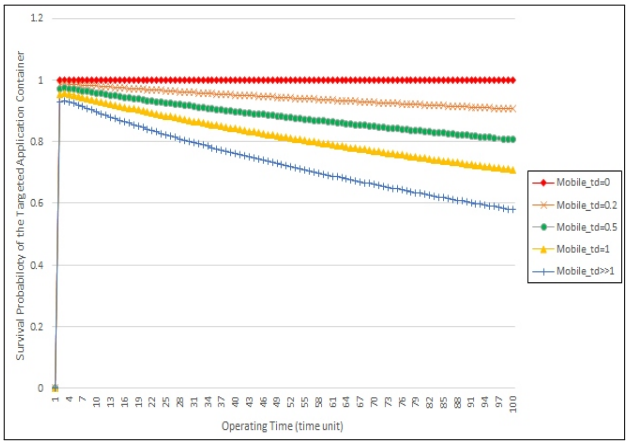

Fig. 8. Survival probability of a mobile targeted container at logistic attack growth with various attack detection time

Figure 9 shows that survival probabilities of a mobile targeted container improve as the number of victim hosts increases. This is because, in case of large number of victim hosts, there is a high probability of ESCAPE to move safely the container to a secured host.

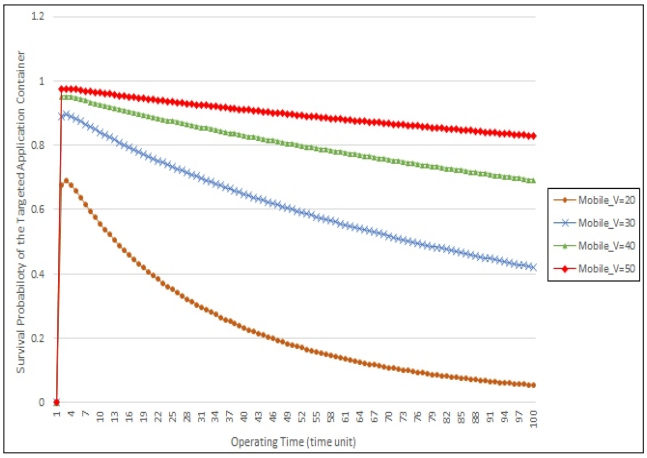

Fig. 9. Survival probability of a mobile targeted data container with different numbers of victim hosts (V)

V. Related Work

Alarifi and Wolthusen used system calls for implementing a host-based intrusion detection for virtual machines residing in a multi-tenancy Infrastructure-as-a-service (IaaS) environment. They dealt with the VM as a single process, despite the numerous processes running inside it, and monitored system calls between the VM and the host operating system [21] [22].

In [21], they used the BoSC technique in combination with the sliding window technique for anomaly detection. In their technique, they read the input trace epoch by epoch. For each epoch, a sliding window of size k moves over the system calls of each epoch, adding bags of system calls to the normal behavior database. The normal behavior database holds frequencies of bags of system calls. After building the normal-behavior database, i.e. training their classifier, an epoch is declared anomalous if the change in BoSC frequencies during that epoch exceeds certain threshold. For a sliding window of size 10, their technique gave 100% accuracy, with 100% detection rate, and 0% false positive rate.

In [22], Alarifi and Wolthusen applied HMM for learning sequences of system calls for short-lived virtual machines. They based their decision on the conclusion from [23] that "HMM almost always provides a high detection rate and a low minimum false positives but with high computational demand". Their HMM-based technique gave lower detection rates, yet required lower number of training samples. By using $780,000$ system calls for training, the resulting detection rate was 97%.

A number of intrusion detection systems used sequences of system calls to train a Hidden Markov Model (HMM) classifier [23] [24] [25] [26]. However, each system differs in the technique used for raising anomaly signal. Wang et al. [24], for example, raise anomaly signal when the probability of the whole sequence is below certain threshold. Warrender et al. [23], on the other hand, declares a sequence as anomalous when the probability of one system call within a sequence

is below the threshold. Cho and Park [25] used HMM for modeling normal root privilege operations only. Hoang et al. [26] introduced a multi-layer detection technique that combines both outcomes from applying the Sliding Window approach and the HMM approach.

Warrender et. al compared STIDE [27], RIPPER [28], and HMM-based methods in [23]. They concluded that all methods performed adequately, while HMM gave the best accuracy on average. However, it required higher computational resources and storage space, since it makes multiple passes through the training data, and stores significant amount of intermediate data, which is computationally expensive, especially for large traces.

The *Kernel State Modeling* (KSM) technique represents traces of system calls as states of Kernel modules [29]. The technique observes three critical states, namely Kernel (KL), Memory Management (MM), and File System (FS) states. The technique then detects anomaly by calculating the probability of occurrences of the three observed states in each trace of system calls, and comparing the calculated probabilities against the probabilities of normal traces. Applied to Linux-based programs of the UNM dataset, the KSM technique shows higher detection rates and lower false positive rates, compared to STIDE and HMM-based techniques.

VI. Conclusion

With the modern evolution of container clouds, there is a desperate need for security solutions to mitigate attacks and threats. In this paper we presented a novel nature inspired attack detection and avoidance mechanism, ESCAPE. Escape aims at avoiding attacks with no prior knowledge of their existence. ESCAPE is guided by an IDS system monitors container behavior for potential attack indications. based on these indications, ESCAPE runs a risk assessment following prey-vs-predator model that will decide whether to move and where to move the running container to keep the attacker target away from the attacker reach. We introduced a novel interrelated container-live-migration and real-time container behavioral-monitoring mechanisms. We also proposed a mathematical model based on evolutionary search games to guide ESCAPE. we applied extensive simulations for ESCAPE's operation to evaluate its effectiveness. Results showed that ESCAPE was able to efficiently detect and avoid mobile continually-growing attacks achieving high survival probabilities of legitimate application containers. Our future work includes smarter manipulations of the operational characteristics of the working containers, comprehensive evaluation of the system on our testbed, and enabling live-migration of containers between heterogeneously-configured hosts to mitigate more complex persistent attacks.

References

[1] (2014) Linux containers: Why they're in your future and what has to happen first. [Online]. Available: https://www.redhat.com/en/resources/future-of-linux-containers

[2] O. Laadan and S. E. Hallyn, "Linux-cr: Transparent application checkpoint-restart in linux," in *Linux Symposium*. Citeseer, 2010, pp. 159–172.

[3] M. Azab, B. M. Mokhtar, A. S. Abed, and M. Eltoweissy, "Toward smart moving target defense for linux container resiliency," in *41st Annual IEEE Conference on Local Computer Networks (LCN 2016)*, Nov 2016.

[4] M. Azab and M. Eltoweissy, "Chameleonsoft: Software behavior encryption for moving target defense," *Mobile Networks and Applications*, vol. 18, no. 2, pp. 271–292, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11036-012-0392-0

[5] ——, "Migrate: Towards a lightweight moving-target defense against cloud side-channels," in *international symposium on security and privacy workshops, BIOSTAR*. IEEE, 2016.

[6] ——, "Defense as a service cloud for cyber-physical systems," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011 7th International Conference on*. IEEE, 2011, pp. 392–401.

[7] M. Azab, R. Hassan, and M. Eltoweissy, "Chameleonsoft: a moving target defense system," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011 7th International Conference on*. IEEE, 2011, pp. 241–250.

[8] A. S. Abed, C. Clancy, and D. S. Levy, "Applying bag of system calls for anomalous behavior detection of applications in linux containers," in *IEEE GlobeCom 2015 Workshop on Cloud Computing Systems, Networks, and Applications - 4th International (GC'15 - Workshop - CCSNA)*, Dec. 2015.

[9] ——, "Intrusion detection system for applications using linux containers," in *Security and Trust Management*, ser. Lecture Notes in Computer Science, vol. 9331, 2015, pp. 123–135.

[10] S. Alpern and S. Gal, *The theory of search games and rendezvous*. Springer Science & Business Media, 2006, vol. 55.

[11] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[12] M. Helsley, "LXC: Linux container tools," *IBM devloperWorks Technical Library*, 2009.

[13] (2015) OpenVZ Virtuozzo Containers. [Online]. Available: https://openvz.org

[14] (2015) CRIU - Checkpoint/Restore In Userspace. [Online]. Available: http://criu.org

[15] D. Fuller and V. Honavar, "Learning classifiers for misuse and anomaly detection using a bag of system calls representation," in *Proceedings of the Sixth Annual IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop*. IEEE, 2005, pp. 118–125.

[16] (2015) Sysdig. [Online]. Available: http://www.sysdig.org

[17] (2015) runC - a lightweight universal runtime container. [Online]. Available: http://runc.io

[18] J. Hofbauer and K. Sigmund, *Evolutionary games and population dynamics*. Cambridge university press, 1998.

[19] M. Chapman, G. Tyson, P. McBurney, M. Luck, and S. Parsons, "Playing hide-and-seek: an abstract game for cyber security," in *Proceedings of the 1st International Workshop on Agents and CyberSecurity*. ACM, 2014, p. 3.

[20] G. Oshanin, O. Vasilyev, P. Krapivsky, and J. Klafter, "Survival of an evasive prey," *Proceedings of the National Academy of Sciences*, vol. 106, no. 33, pp. 13 696–13 701, 2009.

[21] S. Alarifi and S. Wolthusen, "Detecting anomalies in IaaS environments through virtual machine host system call analysis," in *International Conference for Internet Technology And Secured Transactions*. IEEE, 2012, pp. 211–218.

[22] ——, "Anomaly detection for ephemeral cloud IaaS virtual machines," in *Network and system security*. Springer, 2013, pp. 321–335.

[23] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: alternative data models," in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, 1999, pp. 133–145.

[24] W. Wang, X.-H. Guan, and X.-L. Zhang, "Modeling program behaviors by hidden markov models for intrusion detection," in *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, vol. 5. IEEE, 2004, pp. 2830–2835.

[25] S.-B. Cho and H.-J. Park, "Efficient anomaly detection by modeling privilege flows using hidden markov model," *Computers and Security*, vol. 22, no. 1, pp. 45 – 55, 2003.

[26] X. D. Hoang, J. Hu, and P. Bertok, "A multi-layer model for anomaly intrusion detection using program sequences of system calls," in *Proc. 11th IEEE Int'l Conf. Networks*, 2003, pp. 531–536.

[27] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A sense of self for unix processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996, pp. 120–128.

[28] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *Usenix Security*, 1998.

[29] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj, and M. Couture, "A host-based anomaly detection approach by representing system calls as states of kernel modules," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 431–440.