Microservices Made Attack-Resilient Using Unsupervised Service Fissioning

Ataollah Fatahi Baarzi
azf82@psu.edu
The Pennsylvania State University

George Kesidis
gik2@psu.edu
The Pennsylvania State University

Daniel Fleck
dfleck@gmu.edu
George Mason University

Angelos Stavrou
astavrou@gmu.edu
George Mason University

Abstract

Application-layer DoS attacks are increasing as the number of cloud-deployed microservice applications is increasing. The attacker tries to exhaust computing resources and brings the nominal applications down by exploiting application-layer vulnerabilities. As traditional solutions for volumetric DoS attacks will not be able to handle these attacks, new approaches are required to detect and respond to application-layer attacks. In this work, we propose an unsupervised, non-intrusive and application-agnostic detection approach and fissioning-based response mechanism. We built our prototype on kubernetes, the state of the art container orchestrator for microservices, and show its effectiveness through experimental evaluation. Our preliminary results show that using our detection and defense mechanism, we are able to a) efficiently identify the attacks and b) reduce the effect of the attack on legitimate users by 3× compared to a case where there is no detection/defense in place.

CCS Concepts

• **Security and privacy → Distributed systems security**; • **Computer systems organization → Cloud computing**.

Keywords

DDoS attack, systems security, microservices, cloud computing

ACM Reference Format:

Ataollah Fatahi Baarzi, George Kesidis, Daniel Fleck, and Angelos Stavrou. 2020. Microservices Made Attack-Resilient Using Unsupervised Service Fissioning . In *13th European Workshop on Systems Security (EuroSec '20), April 27, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3380786.3391395

1 Introduction

Generally, Distributed Denial of Service (DDoS) attacks, particularly widely publicized volumetric ones, continue to grow in scale [8] and cost [16]. In response to these kind of attacks, many solutions have been proposed both in academia and industry. Solutions for volumetric attacks such as Akamai Prolexic and AWS Cloudflare do not address low-volume attacks targeting the application-layer (asymmetric DoS attacks) which exploit application-layer vulnerabilities to exhaust computing resources.

Examples of these low-volume application-layer attacks include Slowloris [18] and BlackNurse [7, 11]. Slowloris employs numerous partially opened HTTP requests, thereby exhausting a server's concurrent-connections store, and so thereafter denying connections to legitimate requesting clients. BlackNurse is a ping-flood attack variant using (unsolicited) ICMP Type 3 (destination unreachable) Code 3 (port unreachable) reply packets whose processing overloads CPU resources of common server firewalls. Other types of application-level attacks include regular expression DoS (ReDoS) attack [19] where the attacker tries to exploit the regular-expression parsing libraries by passing bad regular expressions causing the program to enter an infinite loop and hence exhaust CPU and memory resources [1]. In the Billion Laughs (XML bomb) attack [2], the attacker passes a small-size nested XML file to cause the XML-file parser to run out of memory. A recent example of this attack exploited a vulnerability in kubernetes' YAML file parser which enables an attacker to bring down a whole cluster using a single request containing a bad YAML file [20]. For another example, the Event-handler Poisoning (EHP) attack [3], the attacker causes the event handling thread to become so busy that other requests don't get a chance to be serviced, hence greatly increasing their response times. Recently, a suite of eight DoS attacks targeting HTTP/2 servers was published by Netflix [9]; these attacks involve variations of known exploits. Though NGINX has released patches for most of the attacks targeting HTTP/2 mentioned above, new ones will likely arise.

Even more subtle attacks involve "heavy hitter" clients consistently issuing abnormally heavy workloads, where the individual workloads in isolation are innocuous.

Microservices are one of the targets of the application-layer attacks. Here, the attacker can target one specific service to exhaust resources available to it and hence that service is unavailable to other services and legitimate users [15, 20].

In response to the application-layer attacks against microservices, this paper describes an **unsupervised**, **non-intrusive** and **application agnostic** approach that will be able to combat new attacks by directly monitoring the resource usages of the services under attack and isolating the attacker(s) so it cannot affect the availability of the service for the legitimate users.

Our approach leverages the microservice architecture to effectively detect and defend the attack. Microserivce-based applications are deployed on a multi-node cluster which is managed by a cluster

EuroSec '20, April 27, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-7523-8/20/04…$15.00
https://doi.org/10.1145/3380786.3391395

manager and container orchestrator such as kubernetes (cf. Figure 1). Different services are loosely coupled to each other. Each service consists of one or more containers ("pod" in kubernetes terminology) that run the service. For higher availability and scalability, there are more than one service replicas. Since each service is separated from other services, we can detect and defend the attack against individual services (and their replicas) and hence prevent availability or performance degradation of the other service replicas or the entire application (consisting of all of the services, of course).

The basic idea of our approach is to monitor the resource usage of each service (pod) in last (configurable) T seconds and predict its usage in next T seconds. By using the knowledge from the historical data, we identify whether the usage is abnormal (in respect to known past behavior) or not. If so, the services is deemed to be under attack. (Thus, our approach is unsupervised because we do not rely on any model of the new attack behavior for purposes of anomaly detection.) We then quarantine the service on a reserved quarantine node in the cluster and perform fissioning to better isolate and identify the attacker (reduce false positives in detection). Because our approach directly detects anomalously high resource utilization (the goal of all such low-volumetric application-layer attacks), it is both non-intrusive and application agnostic (and unsupervised).

To show the the effectiveness of our approach, we built a prototype (see 4) on kubernetes, a widely used and the state-of-the-art cluster manager and container-orchestration platform for microservices. Through experimental evaluation of our prototype we show its effectiveness, e.g., in reducing the latency by 3× for legitimate users of an application that is under the attack compared to a situation where there is no attack detection and defense tool deployed on the cluster.

In summary, we make the following contributions:

- We propose a method to effectively detect application-layer attacks against microservices. The method is agnostic to attacks (unsupervised detection), nominal application/workload type, and service type.
- We propose a service fissioning mechanism to effectively mitigate the effect of attack on legitimate users by isolating the attacker.
- We built a prototype on the kubernetes cluster manager. Performance results using our prototype show the effectiveness of our proposed methods for attack detection and response.

The rest of this paper is organized as follows: In Section 2, we discuss related prior work on defense against application-layer DoS attacks. In Section 3, we motivate and formulate our problem. In Section 4, We explain our design and prototype implementation on kubernetes. In Section 5, we give some preliminary experimental results using our prototype on AWS. Finally, we conclude in Section 6 with a discussion of future work that we plan to extend our prototype.

2 Related Work

DoS attack detection and defense has been studied extensively in the past. In this section we only focus on prior work on application-layer DoS attacks.

Finelame [5] uses (*supervised*) K-means clustering to characterize resource utilization behavior. In order to monitor the resource utilizations, they use eBPF [6] to get fine-grain monitoring at run time (OS calls). Though now eBPF is supported by cluster managers, it opens-up attack surfaces if it's not developed and deployed carefully and hence deployment in production is typically avoided. They also require programmers annotation of the application source-code regarding required resources to insert the probes to collect the utilizations statistics, which is not always possible for a deployed application in a production environment. In contrast, we use a completely **non-application-intrusive** and unsupervised approach for a multi-node cluster (VMs or containers/pods running on them) and do not rely on information from tools such as eBPF. Rather, we use the container-level statistics which are available on any cluster manager, which does not need programmers' involvement. Also, Finelame is limited to only the detection phase. That is, the system only detects the attacker requests and it does not do defense beyond that. In our proposed work, however, we also address attack response. Finelame's scope is also limited to a traditional single-node deployed application, while the scope of our work targets **multi-node clusters** where a **distributed microservice** application may be deployed.

Other recent efforts on application-layer DoS attack detection and defense include Rampart [4] which is focused on DoS attacks against PHP applications and the target resource is CPU. While effective for CPU targeted attacks, Rampart is not able to detect and respond to attacks such as Billion Laughs which the target resource is memory rather than CPU. In contrast, our approach is application agnostic, i.e., is not limited to PHP applications, and is also able to detect and respond to attacks against multiple IT resources (herein, jointly CPU and memory is illustrated).

Finally, our preliminary work [17] was a supervised detection framework which focused on the theoretical performance of multiple fissioning steps. We use the insights from that work to design and implement our unsupervised service fissioning approach in this work.

3 Problem Formulation

In this section we present our assumptions and problem formulation. Our approach to attack response (service fissioning) naturally allows for remediation of false positives. In the following, we do not consider prior *known* attacks for purposes of detection of new ones, but such behavior can easily be included in our detection framework. Also, we assume standby pods for attack response on the (small) quarantine node (VM).

First, we assume that the users each provide a workload consisting of a stream of requests.[1] While all the requests submitted to a service are all assumed to be of the same type, but for a given set of resources, the execution times of the requests may vary depending on their "size". The size of the requests could in some cases be captured simply by the size of the input dataset to be processed, or the size of the content to be retrieved. Though we assume the workload (and hence the service) is generic, we also assume that requests are continually streamed and hence very repetitive and recurring [10]. As a result, it is reasonable to conclude that we can well characterize *nominal* individual streaming workloads in terms of the cloud resource utilizations and needs in order to meet the

[1] This includes the possibility that "users" are defined as segments of IP address space (or IP + port-number to account for NAT). That is, a "user" is the superposition of all clients from its address segment.

Service-Level Objectives (SLOs) of individual requests as a function of their size. An individual request could correspond to a different client associated with the user.

To differentiate between overloads arising from high demands and overload arising from application-layer attacks, user demand can be controlled by token bucket mechanisms in terms of request arrivals, but can only be controlled in terms of request sizes if the sizes are well characterized (so that request sizes map clearly to execution times given a set of available resources). For more complex workloads, notwithstanding such controls, unexpectedly high demand may result. However, in a security setting and in presence of an application-layer attack, malicious requests or malware may be piggybacked on requests, in particular zero days which are not detected by defenses such as firewalls. Thus, we motivate the need for run-time monitoring of *utilized* resources to detect overloaded services and individual users that are heavy hitters or attackers. Since the attacks under consideration wish to exhaust such resources, detection based on utilized resource statistics will be robust to polymorphism or metamorphism of malicious requests. Detection is based on "null" models of known (generally including both known nominal and known attack) user behavior. Null models may be reinforced using run-time statistics that have been correctly "labelled" by a security administrator.

Another basic assumption of our system is that user sessions are *not* nominally placed in *statically resource-provisioned* pods[2] within servers so as to promote efficient utilization of the servers' resources (by statistical multiplexing) and reduce overhead.

Our approach to defense is based on overload detection at the pod (containers) level. Though each service might consist of multiple pods, when a service is under attack, not all the pods are involved so it suffices to only quarantine the attacked pod rather than the entire service (i.e. all the pods supporting the service). For simplicity, in this work, we assume that a service consists of one pod and our ideas are applicable to services with multiple pods as well. For purposes of overload detection, user workload can be measured in terms of plural resources jointly utilized, e.g., CPU, memory, network IO, disk IO, and capacity for open network connections. Congesting one or more of these resources may be the aim of the DDoS attacker(s), where it's possible that just one such attacker can take down a pod.

4 Defense Design and Implementation

In this section, we explain our design and implementation of prototype on Kubernetes, a state-of-the-art container orchestrator and cluster manager. Our defense design consists of two parts. First the detection phase where we aim to detect an application-layer attack, and second the response phase where we aim to defend the attack either by preventing the attacker to send further requests or isolating the attacker to mitigate the influence of resource exhaustion on the legitimate users.

4.1 Attack Detection Design

It's well-known that many workloads that cloud-deployed applications receive over the time follow a pattern that is recurring and similar which can be inferred using historical data (i.e. what has been seen in past days, weeks, or month) [10]. We leverage this opportunity to collect resource (e.g. CPU and memory) utilizations of the deployed application at different load levels for legitimate users over time. By analyzing the historical data we calculate the average and variance (and covariance) of both CPU and Memory utilization at different load levels.

The basic idea is to monitor the resource usage of each service (pod) in last (small and configurable) T seconds and predict its usage in next T seconds. By using the knowledge from the historical data (i.e. offline data from offline profiling), we identify if the usage is abnormal (in respect to known past behavior) or not. In particular, at run-time, we use first-order auto-regressive to predict the CPU and Memory utilization of the pod at current load level. Suppose that the CPU utilization and load at time t (from last T seconds) are denoted as $C(t)$ and $L(t)$ respectively. As explained above, for different load levels L, the mean μ and variance v are known and we use the normalized CPU utilization $\hat{C}(t) = C(t)/L(t)$ at load level $L(t)$. We use a first-order auto-regressive $\xi(t)$ with a fade factor α such that $\xi(t) = \alpha\hat{C}(t)+(1-\alpha)\xi(t-1)$ to predict the CPU utilization of the container. Let F be the cumulative distribution function of 2 parameter Gamma distribution (i.e., two degrees of freedom) with mean μ and variance v that are measured and mentioned above. Note that these quantities are also get slowly updated over time in a reinforcement learning way. We check if the p-value $1 - F(\xi(t)) > \varepsilon$ for small positive threshold $0 < \varepsilon \ll 1$, then deem the pod attacked (overloaded). We use the same process for Memory utilization as well. If at least one of the resources (e.g. CPU or Memory) is overloaded we deem the pod as attacked.[3]

4.2 Attack-Response Design

In this section we explain our attack-response mechanism (i.e. fissioning). Note that the goal is to identify the attacker user and isolate it so it won't have an influence on the legitimate users. Once a replica pod is detected as attacked, we deschedule the pod and launch and schedule two replica pods (i.e. fission) of the service that was running on the attacked pod on the quarantine node (see Section 4.4) in the cluster. We then redirect all the users of that replica pod to the new service replicas on the quarantine node and assign them evenly to the two replica pods. Note that by quarantining the attacked replica pod and its users, we completely mitigate the effect of the attack on the other replica pods that are sharing the same VM. We then repeat the same detection process (as described in section 4.1). If after a time period T, a quarantined pod is not deemed as attacked, we reschedule the pod and redirect its users back to the normal (not quarantined) replica pods on the normal nodes in the cluster. However, if a pod is deemed as attacked, we repeat the same process and launch one more pod (i.e. fission) and assign half of its users to the new pod and the same process goes on so that finally the attacker user is identified and all the nominal users are moved back to the normal nodes.

Once the attacker is identified, a "business" decision can be made regarding how to treat the attacker. For example, one can block that user and drop any request that the attacker sends. In order to reduce the effect of false positives, we take a more conservative approach and instead of dropping all the requests from attackers,

[2]Because we will be taking the context of kubernetes, we will use the term pod instead of container.

[3]Note that, generally, the CPU and memory covariance could also be measured and, to reduce false positives, a p-value based on the joint CPU-memory utilization could be computed. Reducing false positives in detection means that the attack-response phase will need to "rehabilitate" fewer quarantined legitimate users.

we maintain the pods that are serving them on the quarantine node but strictly limit their resource quotas so they won't exhaust the quarantine node's resources (i.e., there is no inter-pod statistical multiplexing on the quarantine node).

Note that the two initial pods can be increased as well in order to to reduce the number of steps to identify the attacker. We perform a sensitivity analysis in our evaluation (see 5).

4.3 Fissioning and Quarantine

To fission the attacked service replica and its users (as described above), we reserve one node in the cluster as a *quarantine* node (VM) so that we can schedule a quarantined pod replica of the service on this node. To do so, we leverage the Kubernetes feature known as **taint and toleration** [13]. This feature allows us to taint one of the nodes in the cluster as *quarantine* node. Once a node in the cluster has a taint, the Kubernetes scheduler won't be able to schedule any pod on that node unless the pod has a *tolerance* of that taint. When we want to launch a quarantined replica of the attacked service, we add the *quarantine* tolerance to the pod so it will be able to be scheduled on the quarantine node. To prevent scheduling the quarantined pods on the normal nodes, we leverage the Kubernetes **node affinity** feature [12]. This enables us to assign the normal pods to the normal nodes in the cluster. Using these two features, we are able to to do both fissioning and quarantine.

4.4 Implementation

In this section, we described the detailed implementation of our prototype. Our prototype consists of modules which run as pods on Kubernetes and are implemented using GoLang as it is the Kubernetes' native language. Figure 1 depicts a high-level overview of our prototype and its main components. In the following, we describe each component and how it is used for attack detection and response phases.

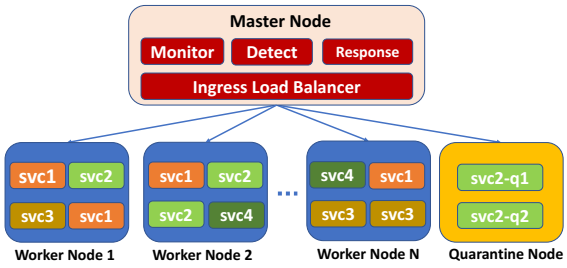

Figure 1: Overview of a deployed microservice on Kubernetes along with our prototype (service2 is attacked and hence quarantined)

4.4.1 Monitor: The *monitor* module runs as a pod on the master node and periodically collects the system statistics and resource utilizations of all the pods. Whenever the *detection* module (discussed below) asks for the statistics of a pod, it responds by providing the mean and the variance of the utilization of each resource over the past period of (configurable) time T.

4.4.2 Detection: The *detection* module stores the mean and variance of utilizations for (rough) load levels in a hash tables for each service. It then periodically polls the resource utilizations of each pod from the *monitor* module in a round-robin fashion. As described in sec 4.1, it calculates the p-value for the service and if that pod is deemed as attacked (p-value is too small), it invokes the *response* module).

4.4.3 Response: The *response* module receives defense requests from the *detection* module. The request consists of the service name and the name (or ID) of the pod replica that is running that service. Once a defense request is received, as described in Section 4.2, the *response* module takes the following steps to perform the defense:

- It deschedules the attacked pod.
- It launches two (or more) replicas of that service (named *svc_name_q*1, *svc_name_q*2, etc.) and schedules them on the *quarnatine* node.
- It redirects the users of the attacked pods to new quarantined pods by updating the routing tables in the *ingress* controller (discussed below) so any further request from those users will be routed to the quarantined pods.

After these initial steps are taken, the last two steps are repeated until the attacker is identified. Once the attacker is identified, the resource quota of the pod that is serving the attacker are limited in order to not exhaust the resources on the quarantine node. As an example, once we have two quaranatined pods for a service named *svc_name_q*1 and *svc_name_q*2, and the detection module finds that the pod *svc_name_q*1 is not under attack, it deschedules it and returns back its users to the normal pods by updating the ingress controller routing tables. And if the pod *svc_name_q*2 is under attack it launch one (or more than one) pod named *svc_name_q*3 and updates the ingress controller routing table to evenly distribute the pod *svc_name_q*2 users among the both pods to further identify the attacker in the next iteration.

4.4.4 Ingress Controller: In Kubernetes, the ingress controller acts as an internal load balancer to (evenly) distribute the requests among the target service replicas (i.e. pods).[4] We leverage the ingress controller to assign the users to the normal pods (reside on normal nodes) and extend it to have a routing table to route the users of the quarantined pods to the quarantine node. As mentioned above, the routing table of the ingress controller is updated each time that a quarantine pod is scheduled on the quarantine node or a pod is removed from that node and its users need to be returned back to the normal pods/nodes.

5 Experimental Results

In this section we demonstrate some of our preliminary results of our prototype. To gather the per load-level statistics, we run our workload with legitimate users (i.e. no attacker) at different load-levels against our deployed application on a kubernetes cluster. As explained before, the *detection* module later uses these statistics to detect whether an attack is happening or not.

5.1 Experimental Setup

Our cluster on AWS EC2 consists of four *m5.large* instances each with 2 vCPUs and 8 GB of memory. One of the instances is used for the master node where our prototype module pods (i.e. *monitor*, *detection* and *response*) along with other kubernetes pods such as kubernetes scheduler are deployed. The other three instances are

[4]Generally, the ingress controller could also group into pods the requests that have similar loads (for example, same input data sizes) to facilitate statistical characterization used for attack detection.

our worker nodes where the application pods and also the ingress controller are deployed. Note that, as mentioned before, we *taint* one of the worker nodes as a quarantine node so by default no application pods are scheduled on it and this node is only used for quarantined pods (see Section 4.4).

Application: We use a computer vision application that performs facial landmark detection on an input image. We deploy the application as a service with 4 replicas so each of our two normal nodes will have two of the replicas scheduled on them.

Attack Scenario: The attack consists of malicious requests that attacker sends in order to exhaust the CPU or memory resources. For our application, the malicious requests can be of two types: *a*) A request with a file pretending to be an image file but contains code injected into the application to exhaust memory and/or use-up CPU cycles, *b*) A request with an standard-sized image file that contains no faces but too many face-like objects resulting in anomalously long processing times.

Baselines: We compare the performance (i.e. the total latency or execution/response time that legitimate users experience in presence of attack) of our prototype compared to a baseline where there is no detection and response mechanism in place. Because we always take one of the nodes in the cluster as a quarantine node, there are $n-1$ nodes (where n is the total number of nodes, e.g. $n=3$ in our reported experimental results herein) that serve the legitimate nominal pods, we use two baselines. First, a baseline with $n-1$ nodes where the number of worker nodes is same as ours. Second, to be fair in terms of total cost of worker nodes we compare to a baseline with n worker nodes serving legitimate users.

5.2 Experiment Scenario

We deploy our application on the cluster described above. Twelve distributed users, one of which is the attacker, start sending requests to the cluster each with mean rate of 100 requests per second for a duration of 600 seconds (i.e. 10 minutes). The scheduler assigns 4 users to each replica in the cluster. After 300 seconds, the attacker starts sending malicious requests described above. The attack continues for 300 seconds. The parameters for *detection* module are as follow. We empirically find that the best fade factor for the first-order auto-regressive estimator for both CPU and memory utilization is $\alpha = 0.15$ hence we stick with that throughout the experiments. The value for T the window time in which the *detection* module polls statistics to detect the attack is chosen such that the it spans at most the serving time for two requests, therefore we set this value to $T = 3$ *seconds*. By analysing the data from our offline profiling, we set the two parameters for the Gamma distribution at the heart of the *detection* module to $shape = 390$ and $rate = 5.2$ for CPU and $shape = 10560$ and $rate = 6.6$ for memory. Finally, the p-value to deem a pod attacked is set to 0.15.[5]

Figure 2 depicts the average end to end latency[6] that the 11 non-attacker users experience during the experiment. We compare the performance of our prototype against two baselines (see above). 4 seconds after the attack starts, the *detection* module detects the attack and invokes the *response* module. The *response* module then schedules two replicas of the attacked pod and quarantines them on the quarantine node and redirects its 4 users to the quarantine node and each quarantined pod serves two of the users. In next 4 seconds the *detection* module detects the attacked quarantined pod and invokes the *response* module again. The *response* module then moves back the healthy pod to the normal nodes and makes one more replica of the attacked pod and each pod will serve one of the remained two users of which one of them is the attacker. In next 3 seconds, the *detection* module detects the attacked pod and invokes the *response* module for the last time in this scenario. The *response* module then moves back the normal/not-attacking user to the normal nodes, limits the resources of the attacked pod which is serving the attacker, and keeps it scheduled on the quarantine node for the rest of the experiment. In total, it takes less than 14 seconds for the *detection* and *response* modules to identify the attacker and isolate it. As can be seen in Figure 2, this results in about 3× less latency for legitimate users compared to the baseline with 3 nodes, since fewer users are directly affected by the attack. This performance is reduced for the baseline with 2 nodes where less resources are available and more users are affected by the attack.

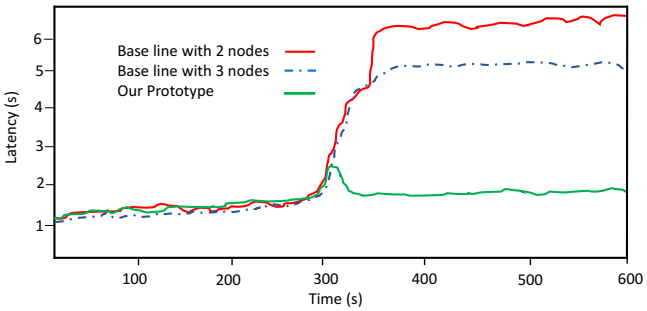

Figure 2: The average latency that legitimate users experience in presence of the attacker compared to in presence of our detection and response mechanism

We also ran the same experiments explained above on a scaled up cluster with 10 nodes, 60 users and 4 attackers. The results follow the same trend as the results we reported. The only difference is that since there are more attackers involved, it takes longer to recover the nominal latency for the legitimate users.

5.3 Sensitivity Analysis

We also perform a sensitivity analysis for the effect of number of quarantined replica pods on the time it takes to identify the attacker. Since each time the *detection* module invokes the *response* module, it performs binary fissioning, the number of quarantine pod replicas in each step affects the time it takes to identify the attacker, because it reduces the number of steps needed to divide the users of the attacked pod to finally identify the attacker. On the other hand, the time to identify the attacker will also depend on the number of other legitimate users along with the attacker that are served by the attacked pod. Figure 3 depicts the effect of number of quarantine pod replicas on the time to identify the attacker for different number of users (denoted as N) assigned to a pod.

As it can be seen in this figure, as the number of users being served by an attacked pod increases, it takes a longer time to identify the attacker(s). Again, this is because of the performed binary fissioning which requires more steps as the number of users are increased.

[5] Generally for unsupervised detection, this p-value threshold can be set to minimize false positives.

[6] The time at which user sends the request until the response is received

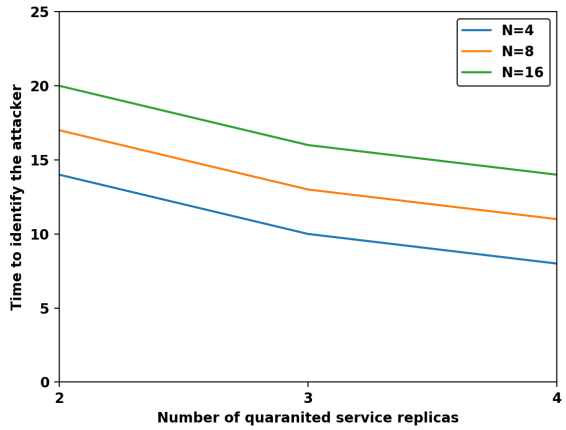

Figure 3: The effect of number of quarantine pod replicas on the time to identify the attacker for different number of users assigned to a pod

As the number of quarantine pod replicas increases, the time to identify the attacker(s) decreases. This is because it basically reduces the number of steps needed for *response* module to identify the attacker. Hence one might think that it is a reasonable direction to choose higher number of quarantine pod replicas to reduce the identification time. However, having too many quarantine pod replicas on one single quarantine node might lead to resource contention among the pods which can affect the latency of legitimate users on the quarantine node as a result. This specifically arises when there are multiple attacks on different services, so we need to keep the number of quarantine pod replicas limited to prevent from resource contention.

6 Discussion of Future Work

In future work, we plan to study the performance of our prototype and improve it for more complicated applications consisting of a greater number of different interdependent microservice types. In such applications, because of dependency that exists between the services, when one service is under the attack, other dependent services will be affected by the attack indirectly. We will extend our work to further explore the dependencies of the microservices in order to protect the un-attacked services from availability and performance degradation arising from attacked services that they depend on.

In this work, we evaluated our prototype with an attack similar to the billion laughs attack. We will also evaluate our prototype with more attack scenarios as discussed in Section 1.

We will also explore the use of a more sophisticated null model, e.g., multiple-component Gaussian Mixture Models (GMMs) [14], including components corresponding to *known* attacks. Such models will *jointly* consider utilizations of different cloud resource types, i.e., their covariances, when computing p-values.

Finally, to reduce costs associated with the defense subject to latency constraints, we plan to explore the use of "warm start" cloud functions (e.g., AWS Lambda) [21] for timely and cost-effective attack response, instead of VM-based pods, for quarantine.

Acknowledgments

Our research is supported by Defense Advanced Research Projects Agency (DARPA) Extreme DDoS Defense (XD3) contract no. HR0011-16-C-0055 and a Cisco Systems URP gift. Opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or Cisco.

References

[1] cloudflare [n.d.]. CloudFlare regex outage. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.
[2] CVE-2003-1564 [n.d.]. CVE-2003-1564. https://nvd.nist.gov/vuln/detail/CVE-2003-1564.
[3] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 343–359. https://www.usenix.org/conference/usenixsecurity18/presentation/davis
[4] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 343–359. https://www.usenix.org/conference/usenixsecurity18/presentation/davis
[5] H.M. Demoulin, I. Pedisich, N. Vasilakis, V. Liu, B.T. Loo, and L.T.X. Phan. July 2019. Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FINELAME. In *Proc. USENIX ATC*.
[6] ebpf [n.d.]. eBPF. https://github.com/iovisor/bcc.
[7] L. Hansson, P. Hogh, B. Bachmann, K.B. Jorgensen, and D. Rand. 2016. The BlackNurse Attack. http://soc.tdc.dk/blacknurse/blacknurse.pdf.
[8] Help Feb. 11, 2019. Help Net Security. Average DDoS attack volumes grew by 194% in 12 months. https://www.helpnetsecurity.com/2019/02/11/ddos-attack-volumes-grew-by-194-in-12-months/.
[9] HTTP 2019-08-13. HTTP/2 Denial of Service Advisory. https://github.com/Netflix/security-bulletins/blob/master/advisories/third-party/2019-002.md.
[10] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 416–427. https://doi.org/10.1145/3357223.3362726
[11] S. Khandelwal. Nov. 13, 2016. Even A Single Computer Can Take Down Big Servers Using BlackNurse Attack. http://thehackernews.com/2016/11/dos-attack-server-firewall.html.
[12] kubeaffinity [n.d.]. Kube8s node affinity. https://kubernetes.io/docs/concepts/configuration/assign-pod-node/.
[13] kubetaint [n.d.]. Kube8s taint and toleration. https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/.
[14] D.J. Miller, Z. Qiu, and G. Kesidis. Sept. 2018. Parsimonious Cluster-based Anomaly Detection (PCAD). In *Proc. IEEE MLSP*. Aalborg, Denmark.
[15] netflixddosmicro [n.d.]. Netflix DDoS for Microservices. https://www.infoq.com/news/2017/07/application-ddos-microservices.
[16] C. Osborne. May 2, 2017. The average DDoS attack cost for businesses rises to over $2.5 million. https://www.zdnet.com/article/the-average-ddos-attack-cost-for-businesses-rises-to-over-2-5m/.
[17] Y. Shan, G. Kesidis, D. Fleck, and A. Stavrou. Oct. 2017. Preliminary Study of Fission Defenses against Low-Volume DoS Attacks on Proxied Multiserver System. In *Proc. Malware Conference (MALCON)*. Peurto Rico.
[18] slowloris [n.d.]. http-slowloris. https://nmap.org/nsedoc/scripts/http-slowloris.html.
[19] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu
[20] J. Wallen. 9 Oct 2019. Kubernetes 'Billion Laughs' Vulnerability Is No Laughing Matter. https://thenewstack.io/kubernetes-billion-laughs-vulnerability-is-no-laughing-matter/.
[21] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proc. USENIX ATC*. Boston.