Mitigating Internal, Stealthy DoS Attacks in Microservice Networks

Amr Osman[1(✉)], Jeannine Born[2], and Thorsten Strufe[1]

[1] KIT Karlsruhe, Karlsruhe, Germany
amr.osman@kit.edu
[2] TU Dresden, Dresden, Germany

Abstract. The advent of Microservice (MS) architectures has led to increasingly complex communication patterns between distributed web applications in the cloud. In order to process an incoming request, each MS must invoke multiple remote API calls to the MSes that it is connected to along a service dependency graph. This allows attackers to exploit long-running remote API calls along the performance-critical path to cause application DoS, and potentially amplify subsequent inter-MS communication. This paper focuses on mitigating a class of stealthy, low-volume DDoS attacks that are launched internally from within and exploit this. The attacker uses the MSes under its control to disguise then send and resource-heavy requests to target MSes in a way that is indistinguishable from benign requests. We propose a probabilistic algorithm to proactively identify MSes involved in DDoS, and mitigate the attack in real-time.

1 Introduction

In today's inter-connected microservices, a single incoming request could virtually trigger a chain of hundreds of expensive remote API calls between the involved MSes, forming a complex dependency chain and consuming a lot of CPU and I/O resources. Even a single slow-performing MS may act as a bottleneck to other MSes along the path that is traversed by inter-MS network requests. As a consequence, a high end-to-end delay is experienced.

This opens up new attack vectors to overwhelm MSes with expensive requests in the form of *Stealthy, Internal, Low-volume* DDoS attacks [6] targeted at slowly-performing MSes (SILVDDoS). In such attacks, the adversary disguises resource-consuming requests at low rates using patterns below the detection thresholds of DDoS countermeasures; making them difficult to detect and mitigate as they evade signatures and anomalies observed by traditional Intrusion detection systems [3]. Also as the attacks originate internally and masquerade benign traffic, most countermeasures that rely on perimeter defense [4] and autoscaling [2] are not effective against them.

This work identifies the MSes that participate in SILVDDoS and mitigates the attack on a granular level. Our main contributions are the following: **(1)** A risk metric to evaluate the likelihood of a MS becoming either a target or

© Springer Nature Switzerland AG 2021
C. Johnen et al. (Eds.): SSS 2021, LNCS 13046, pp. 500–504, 2021.
https://doi.org/10.1007/978-3-030-91081-5_37

a source for a SILVDDoS attack. **(2)** A probabilistic algorithm to approximate the identity of the sources SILVDDoS and mitigate it. **(3)** We mount a stealthy, SILVDDoS attack and evaluate the effectiveness of our approach using a popular container-based open-source MS application.

2 Assumptions and Threat Model

We consider a MSes deployment in a container-based cloud environment, e.g. a docker cluster. Each MS is isolated from other MSes via a separate Linux container and communicates with other MSes through a network-exposed API such as REST over HTTPS, or secure RPC. The MSes are exposed to the outside network, i.e. Internet, via an external load-balancing gateway that receives the requests from the end users. MSes may be elastically scaled by internal load balancers. This model is aligned with the vast array of MS deployments and kubernetes clusters today.

Adversary. The main goals of SILVDDoS is to exhaust the CPU and/or I/O resources on target MSes such that the network requests traversing the paths leading to them experience a high end-to-end delay leading to *unavailability*. A side-goal is *financial DoS* as cloud schedulers elastically scale the attacked MSes and cloud customers are charged for the newly provisioned resources, e.g. Yo-yo attack [2]. The adversary remotely controls multiple MSes internally and uses them to initiate its SILVDDoS attack on other target MSes from within. By controlling a MS, the attacker has access to all its resources such as mounted volumes (e.g. Databases), network name-spaces, processes, and user groups. Thus, it also has access to secret keys, and may authenticate itself to the network and other MSes. It may *passively* observe traffic and learn about its neighbours and measure their request-response times, or *actively* replay legitimate user traffic. It aims to remain stealthy by following the same paths and patterns that are followed by benign requests [3,5].

3 Mitigating SILVDDoS

Existing DDoS countermeasures assume that attack traffic is distinguishabe, which is not true in a SILVDDoS [4]. Cluster resource management protects the infrastructure from overutilization but cannot be used in the presence of SILVDDoS where the bottleneck are the MSes themselves [5] [3]. Chaos Engineering [1] is only used for resilience testing and does not assume malicious intent. Critical-Path-Analysis unfortunately requires knowledge about the application, low-level instrumentation and does not assume adversarial presence [7].

Our approach. We propose a risk metric that assesses the susceptibility of MSes and costly APIs of being used in a SILVDDoS attack based on key network performance and graph properties, and then later use this risk metric in an iterative probabilistic estimation algorithm that improves its quality

every iteration to identify the attack sources and mitigate the attack. Unlike a greedy Critical-Path-Analysis (CPA), our algorithm also considers cases when the adversary uses MSes that may not lie on the critical path to mount the attack.

we formulate a risk metric that is computed for each API endpoint a for each MS m to determine DDoS sinks as follows:

$$\overset{<}{h}_{m,a} = \frac{w_1(1-T) + w_2(1-R) + w_3E + w_4L + w_5D_{in} + w_6A + w_0}{7} \tag{1}$$

Similarly, we formulate a DDoS source risk metric for each API end point per node as:

$$\overset{>}{h}_{m,a} = \frac{w_1(T) + w_2(R) + w_3(1-E) + w_4(1-L) + w_5D_{out} + w_6A + w_0}{7} \tag{2}$$

where $w_{0..6} \in [0,1] \subset \mathbb{Q}$ are selected weights for each metric, and T, R, E, L, D, A are the arithmetic means in their normalized form in the range $[0,1]$ and are calculated for each API $a \in m$ for each MS m.

These properties used are: *Transfer rate (T)*, *Request rate (R)*, *Error rate (E)*, *Latency (L)*, *Node degree (D)*: The number of possible logical connections each MS has to other MSes in the topology. We distinguish between incoming (D_{in}) and outgoing connections (D_{out}), *Amplification factor (A)*: The ratio between the number of external requests sent to other MSes and a given incoming request. The algorithm to mitigate SILVDDoS on a MS s can be summarized in the following steps:

1. **Pick** $a \in s$ with $j+1^{th}$ highest $\overset{<}{h}_{s,a}$ and $m \in M$ with $i+1^{th}$ highest $\overset{>}{h}_{m,a'}$, where $m \overset{a}{\rightarrow} s$
2. **Apply** either rate-limiting or container-restart to m with respect to a and s
3. **Measure** the performance metrics, i.e. health, of s.
4. With probabilies $p1$ and $p2$, **increment** j and i to the next API and microservice respectively.
5. With a probability $p3$, **undo** all rate-limiting and container-restart $\forall_m \forall_a$
6. If the health of s has improved, **go to** step 1. Otherwise, **undo** step 2 and **go to** step 1.
7. After a health threshold for s or num. of iterations is reached, **terminate** and **return** all (m, a) that were used in step 2.

4 Preliminary Evaluation

We deployed an open-source heterogeneous MSes-based application [8] with the topology in Fig. 1. We then used FastNetmon to detect high volume DoS, and Zeek to detect traffic anomalies and low-volume DoS. After that, we initiated a SILVDDoS attack on 'carts' from 'frontend' and 'orders' following the same benign user traffic paths. (See Eq. 1 and 2). *Neither FastNetmon nor zeek with*

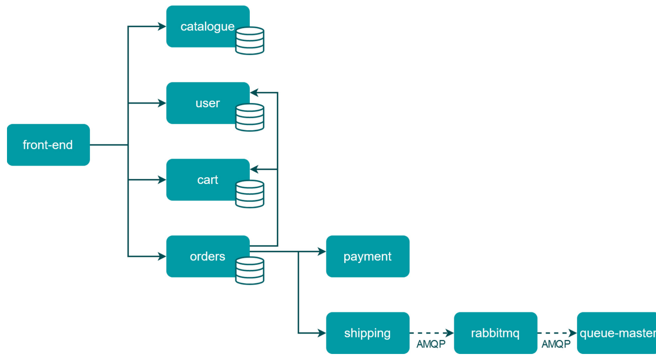
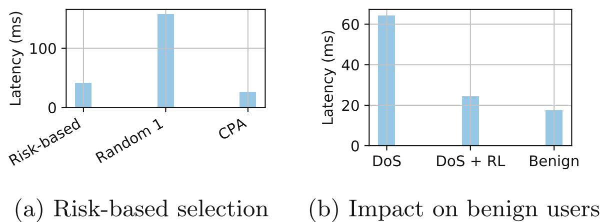


Fig. 1. Sockshop MSes topology

the latest up to date rules were able to detect SILVDDoS and zero alarms were triggered, confirming the stealthiness of our attack.

We then evaluate two main questions: **(1)** How does our risk-based placement of countermeasures compare to critical-path-analysis or a random placement? **(2)** What is the impact of our approach on benign user traffic?

Quality of risk-based selection. We compared the risk-based application of rate-limiting compared to a random placement approach that, based on majority-occurance, rate-limits requests from 'orders' to 'user'. Second, we performed a Critical path analysis with the sink as a root node and applied the rate limiters to the MSes that lie on that critical path. CPA rate-limits requests from 'orders' to 'user' and from 'orders' to 'carts'. The output can be see in Fig. 2a.



(a) Risk-based selection (b) Impact on benign users

Fig. 2. Latency measurements. Subfig. (a) shows the median response time of 'carts' (under attack). Subfig. (b) shows the median response times experienced by endusers at the frontend. Each experiment was evaluated with a confidence level of 80%

We observe that random 1 does not lead to a performance restoration. The sources of the DoS were not correctly identified and restricted. CPA however, correctly identified and rate-limited only one of the sources, namely, 'orders'. It

also rate-limited requests from 'orders' to 'user' which lies on the critical path. That effectively reduced the concurrency level from 2 to 1, and explains why the sink has a slightly lower latency compared to our risk-based selection mechanism. So, while CPA appears to be a better strategy than risk-based selection, it in fact also excessively limits benign user traffic. With reference to Fig. 2b, we observed that the benign user traffic latency was increased to 38.4 ms which is higher than both: the benign setting and the risk-based application of countermeasures.

Impact on benign users. To measure the impact of the risk-based application of rate-limiting on the perceived performance by the end users, we measure the HTTP request rate to the 'frontend'. The output is in Fig. 2b.

During DoS, the user experiences 64.1 ms of HTTP response time instead of 18 ms in the case of benign traffic. The risk-based application of rate-limiting improved the response time and brought the HTTP response time down to 22.7 ms. Hence, the end user temporarily experiences a 26.11 % performance penalty when the risk-based application of rate-limiting is done, instead of a 256.11% performance penalty.

Discussion. Our approach mitigates the attack, but *temporarily* rate-limits some existing benign traffic, until the sources of the attack are identified and replaced with fresh instances. Unlike a greedy CPA, our approach is able to correctly identify attacks that may be off the critical path, and requires neither prior knowledge of the system and application, nor low-level instrumentation. In the future, we would like to optimize the algorithm parameter selection with respect to the properties of multiple MS topologies and compare the effectiveness of different countermeasures other than rate-limiting.

References

1. Blohowiak, A., Basiri, A., Hochstein, L., Rosenthal, C.: A platform for automating chaos experiments. In: IEEE ISSREW (October 2016)
2. Bremler-Barr, A., Brosh, E., Sides, M.: DDoS attack on cloud auto-scaling mechanisms. In: IEEE INFOCOM (May 2017)
3. Ficco, M., Rak, M.: Stealthy denial of service strategy in cloud computing. IEEE TCC **3**(1), 80–94 (2015)
4. Garcia, V.F., et al.: DeMONS: a DDoS mitigation NFV solution. In: IEEE AINA (May 2018)
5. Li, Z., Jin, H., Zou, D., Yuan, B.: Exploring new opportunities to defeat low-rate DDoS attack in container-based cloud environment. IEEE TPDS **31**(3), 695–706 (2020)
6. Payne, B., Behrens, S.: Starting the avalanche: application ddos in microservice architectures (July 2017). https://netflixtechblog.com/starting-the-avalanche-640e69b14a06
7. Qiu, H., et al.: FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In: USENIX OSDI 20, pp. 805–825 (November 2020)
8. Weaveworks: Microservice sockshop (June 2021). https://microservices-demo.github.io/