Low-Level Exploitation Mitigation by Diverse Microservices

Christian Otterstad, Tetiana Yarygina

Low-level Exploitation Mitigation by Diverse Microservices

Christian Otterstad and Tetiana Yarygina

Department of Informatics, University of Bergen, Norway
christian.otterstad@uib.no
tetiana.yarygina@uib.no

Abstract. This paper discusses a combination of isolatable microservices and software diversity as a mitigation technique against low-level exploitation; the effectiveness and benefits of such an architecture are substantiated. We argue that the core security benefit of microservices with diversity is increased control flow isolation. Additionally, a new microservices mitigation technique leveraging a security monitor service is introduced to further exploit the architectural benefits inherent to microservice architectures.

Keywords: security, software diversity, design patterns, robustness

1 Introduction

Microservices is a recent trend in software design. A microservice architecture simplifies the development of complex horizontally scalable systems that are highly flexible, modular, and language-agnostic. We define a microservice as a small specialized autonomous service communicating over a network boundary. By extension, a microservice system is a distributed software system consisting of a set of microservices communicating to perform some computation as an aggregated result of their collective operation. For further information, we refer the reader to the comprehensive study of microservice principles by Zimmermann [1] who identified commonalities in the popular microservice definitions and concluded that microservices represent a development- and deployment-level variant of the service-oriented architecture (SOA).

Although microservice architectures constitute an important trend in software design with major implications in software engineering, surveys such as the one conducted by Dragoni et.al. [2] have highlighted a general lack of research in the area of microservice security. In Newman's book [3] on microservice design, a subset of security traits for improving the security of microservice networks is discussed. The idea of combining microservices with secure containers and compiler extensions to build critical software has been investigated in a recent study by Fetzer [4]. The paper by Lysne et.al. [5] briefly introduces the notion of microservice networks to mitigate vendor-malware and other forms of attacks, without any further elaboration or working examples.

Herein, we expand and elaborate on the generalized notion of mitigating low-level exploitation. To our knowledge, we are the first to demonstrate the benefits of using a microservice architecture to defend against remote low-level exploitation. Unlike a deployment monolith, a microservice architecture facilitates strong process isolation partly because the services run on different physical machines. The paper also introduces a security monitor service that further leverages the architectural benefits of a microservice network, including added software diversity, to enable anti-fragility to low-level exploitation.

2 Microservice Architecture and Its Security Merits

2.1 Model Overview

In general, an attacker wants to gain access to an asset controlled by a defender, extending up to full access to the targeted system. It is assumed that the external attacker is able to carry out the following types of exploits: an initial exploit ($\mathrm{E}_{\mathrm{init}}$), a virtual machine or sandbox escape exploit (E_{VM}), and a lateral exploit ($\mathrm{E}_{\mathrm{lat}}$). $\mathrm{E}_{\mathrm{init}}$ is used to gain a shell on a microservice node, E_{VM} enables the attacker to escape from a sandbox, while $\mathrm{E}_{\mathrm{lat}}$ is an exploit type that abuses the trusted relationship between microservice nodes in cases where additional attack surface is needed and $\mathrm{E}_{\mathrm{init}}$ is not sufficient.

Figure 1 illustrates a generic attack on the system model. The attacker initially obtains access using $\mathrm{E}_{\mathrm{init}}$ and then proceeds to escape the sandbox using E_{VM}. Once the attacker has executed the latter exploit, full control over all nodes hosted by the same hypervisor is obtained. However, the attacker does not control the whole network. To extend the control further, the process must basically be repeated. However, the same exploit $\mathrm{E}_{\mathrm{init}_1}$ may not work against VM_{n_1}—a node hosted by a different machine n, which cannot be reached through the hypervisor. Therefore, the attacker will have to resort to either using a different exploit $\mathrm{E}_{\mathrm{init}_2}$, or, depending on the available attack surface and overall exploitability, a lateral exploit $\mathrm{E}_{\mathrm{lat}_1}$ to utilize the now exposed trusted relationship between the nodes.

2.2 Security Considerations

There are two distinct types of microservices in the context of interaction: microservices that allow both external and internal interaction and microservices that only allow internal interaction. Internal interaction is communication between two microservices within the system boundary. External interaction is interaction between an external host and a microservice that is part of the system. A microservice that only allows external interaction is effectively defined as a monolithic program.

However, regardless of the type of microservice and of the granularity at which microservices are implemented, every microservice must contain functionality for network interaction. The code the user can externally interact with is

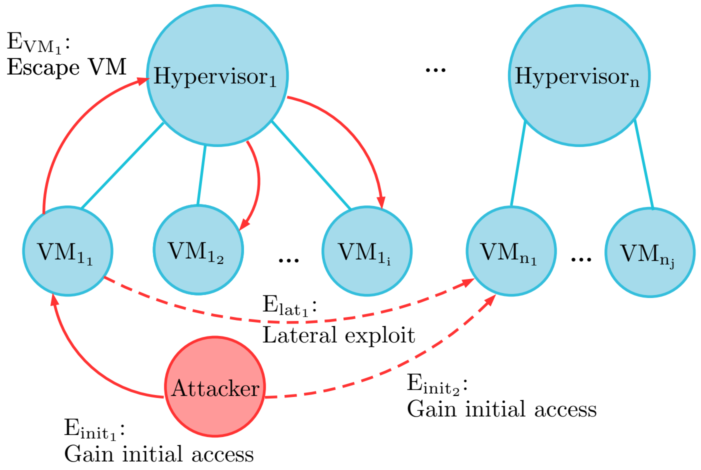

Fig. 1. Attacking a microservice architecture with diverse microservices running in a virtualized environments on networked machines.

the most obvious attack vector. The microservices must assume that any input encountered is hostile. Not only are the microservices communicating over an insecure network, but some of the nodes in the network may be compromised. Therefore, even properly authenticated nodes should not trust the subsequent input to be sane or properly formatted by its peer(s).

Microservice systems employ several design patterns [6, 7] to facilitate the basic operation of the overall system—some of which affect the security of the microservice network. The *API Gateway* pattern is the entry point for all clients. A system without an API Gateway or equivalent would need to expose the required services to external users—hence increasing the initial attack surface. *Circuit breaker* prevents cascading failures by changing the component behavior based on the number of failed calls made. *Service Discovery* is a centralized scheme allowing services to discover other services. An attacker can exploit the service discovery to determine the internal structure and communication patterns between services.

A *robust system* is basically what is commonly referred to as a hardened system. Robustness is a property we use to denote how much effort is required to successfully perform a low-level exploit against the system. The following discussion covers some security considerations specific to enhancing the robustness of microservice networks and moving towards anti-fragility [8].

Maximizing API security. Exposed network interfaces must be minimal, have strong input validation, and be of the highest type in the Chomsky hierarchy [9]. These are well-known design traits for a secure system, and they apply equally to both monolithic designs and microservice designs. If there is any way to accomplish the same functionality while exposing the server to less computa-

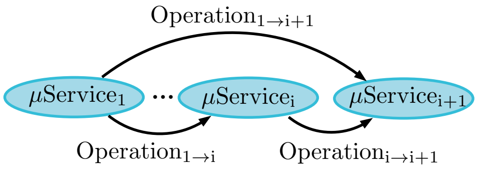
Operation$_{1\rightarrow i+1}$
μService$_1$
μService$_i$
μService$_{i+1}$
Operation$_{1\rightarrow i}$
Operation$_{i\rightarrow i+1}$

Fig. 2. Depiction of an unnecessary edge, exposing additional attack surface.

tion on external input, this is advisable. The defender should strive to minimize the set and depth of possible control flow paths that the attacker can influence at any step.

Avoiding unnecessary node relationships. The defender must employ an architecture that prevents unnecessary node relationships. Consider Figure 2. If μService$_1$ can reach μService$_{i+1}$ through μService$_i$, then there should not be any edge between μService$_1$ and μService$_{i+1}$. Adding the extra edge may increase the attack surface for the involved nodes. While taking a shortcut of this type to obtain information or perform functions directly might result in better performance and less complexity, doing so would violate the trade-off of increased security for less performance and higher complexity. If a microservice network forms a dense graph, then most likely the design of such a system and/or its decomposition into microservices is incorrect.

Asymmetric node strength. To optimize the robustness of the network to low-level exploitation, the more secure nodes should be placed at critical network segments, such as entry points and nodes guarding the more valuable assets, as shown in Figure 3. A more priced asset could be functionality that allows making a transaction as compared to merely viewing the list of already performed transactions. The payment functionality could use most of the budget for hardening whereas viewing an account is considered less severe and should not be as prioritized. Examples of hardening are given in the next section. High diversity as a mechanism for hardening microservices is also discussed in the next section. Such changes can be done a priori, in contrast to tactical choices based on real world statistics.

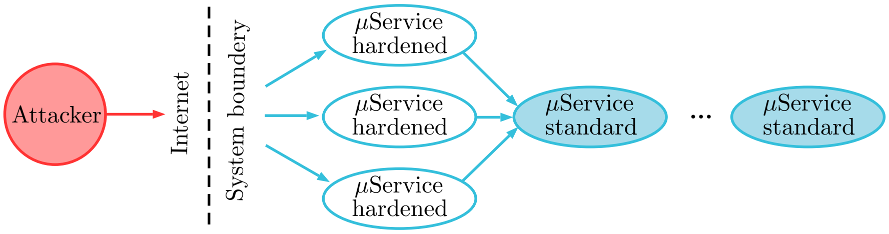

Fig. 3. The use of asymmetric node strength to defend against low-level attacks.

3 The Security Monitor Service

3.1 Security through Diversity

The purpose of diversity in this security context is to make an exploit less statistically likely to succeed and to make the attack scale less effectively, thus, providing the defender with time to react to the attack. The most common (as of 2017) examples of diversity in computer systems are the use of different programming languages, hardware architectures, cloud providers, operating systems, hypervisors, compilers or compiler arguments, and ASLR (Address Space Layout Randomization) versions that enable identical programs to possess diversity. It has previously been argued that there are inherent benefits to software diversity in the context of mitigation of attacks [10, 11].

Minimal diversity has previously been defined [12] as "when failure of one of the versions is always accompanied by failure of the other". This definition is also applicable in the context of exploitation. If there is so little diversity that the exact same exploit works equally well on both versions, then the diversity is of no benefit to the defender. However, diversity still serves a purpose in terms of redundancy against other types of failures, but not against targeted attacks.

It should be stressed again that a microservice system has inherent diversity, simply as a consequence of microservices implementing different functionality. Different bugs are assumed to be associated with different functionality. However, this may not be true in all cases—two microservices with different functionality could employ a common library with an exploitable vulnerability.

3.2 Introducing the Security Monitor Service

Normally, a system will only get patched after developers have identified issues and rolled out the changes. Although this improves the system over time it can introduce a large attack window due to the inherent latency of the process. A microservice network may automate some of the issues that arise, specifically by introducing a security monitor system. The security monitor can identify nodes that exhibit unusual behavior, trigger IDS detections, or in the case of an N-version programmed system simply report inconsistent data compared to its siblings. Anomalous behavior may result in the monitor taking explicit, autonomous action, as explained later in this section.

A simple example would be an N-version programmed system with a set of nodes that perform the same task using compiler derived diversity [13]. Similarly to the N-variant system suggested by Cox et.al., we propose a security monitor scheme to exploit the fact that the defender retains part of the control flow of the overall system [11]. If a particular node issues erroneous data, the security monitor can detect it by comparing the output against the healthy nodes. The erroneous node is then isolated and the security monitor notes the compiler arguments that resulted in this defective machine code. The security monitor is not concerned with the root cause of the program error, but the compiler arguments used to derive the code are assumed to be faulty and should not be reused for the particular code in question.

Consider the case of removing an infection as indicated in Figure 4. The security monitor detects invalid data being sent from a service. The security monitor's presence on the host system is more privileged than the service itself. Hence, the security monitor is able to forcibly destroy the environment for the service, permute, and restore it. If the permutation step was skipped, the attacker could simply replay the exploit. The security monitor should proceed to flag the event as an anomaly to allow a human to examine the faulty binary to identify the underlying cause—which is likely only masked by the permutation. The security monitor may choose to no longer trust the hosting machine for the infected service, i.e. informing the assumed clean services to blacklist the malicious nodes as well as wipe and restore the system in an attempt to deal with a rootkit on the hosting machine. In addition, the security monitor can decide to destroy, permute, and restore all immediately adjacent services.

Another option is to start a new node and ignore, but record the I/O of the infected node, as well as monitor it through the host system. The defender would be able to learn information about the attacker—in particular, exploitation attempts—as the attacker is likely to continue to interact with the system. Such a honeypot strategy could be implemented to varying degrees of sophistication, all requests could be ignored, or some could be simulated, such that the attacker would continue to interact with the simulated environment, but not be able to gain any valuable asset or do damage. In the case of multiple infected nodes, a segment of the system could be isolated. Regardless, the defender should then also migrate away any other services running on the same infected host(s). There is always the risk that the attacker could escape the VM and take control over the whole system.

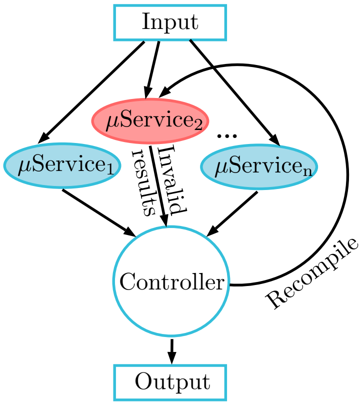

Fig. 4. A security monitor dealing with an infection in an N-version system.

A simple policy for a security monitor service would be to detect an intrusion, e.g. by using an IDS, kill the service environment, rebuild the environment, and finally restart the service. In this generalized procedure, the defender can either host the security monitor as a normal process with normal user privileges, in a container environment, or in a virtual machine. Regardless, the policy should be the same. It is important to destroy the whole environment, otherwise the risk of the attacker persisting increases dramatically. Even when destroying the environment the risk is only made smaller. If no containers are used, all processes should be removed and ideally the system (and firmware) restored from a trusted image—although even in this case advanced rootkits may persist. If containers or virtual machines are used, the entire container or virtual machine must be rebuilt. The recompilation step ensures that diversity is added, which hopefully removes the issue. Such an approach reduces the overhead in terms of cost and time in terms of enabling the system to react to certain types of attacks. The security monitor scheme essentially allows the system to autonomously discover certain security related issues and react to them. Manual interaction is still required to resolve the root cause of the issue. However, at the same time, the microservice architecture ensures that more effort is required to compromise the overall system, which makes the system more secure.

The security monitor system can be multi-layered. A local security monitor may reside in each execution context for each service. However, an additional external security monitor is also possible. An external security monitor would enable more complex evaluations and actions being taken as a result of the state of the overall system, as compared to merely a single node.

3.3 Evaluating the Security Monitor Service

In terms of the overall system architecture, the security monitor service becomes a part of the infrastructure similarly to logging, monitoring, and discovery services that are needed for any reasonably sized microservice system to function properly. In contrast to these basic services, the security monitor attempts to mitigate attacks autonomously, making the overall system more resilient to low-level exploitation.

A more privileged mode that offers an attack surface is an ideal target. Indeed, the security monitor is such a target itself. IDS systems and anti-malware solutions have previously become a viable attack surface which raises the question whether such systems do more harm than good [14]. An IDS is always a trade-off, to prevent it from exposing the system to more risk rather than protecting it, the security monitor should adhere to the aforementioned principles from Section 2.2 of least privilege, minimal attack surface, and have any grammar be of the highest type in the Chomsky hierarchy [9].

4 Conclusion

We have examined how the increased isolation of microservices coupled with software diversity can mitigate the impact of low-level exploitation. Microservices, when coupled with some method of achieving diversification, appears to offer added robustness over monolithic solutions. Key design rules and examples were presented to substantiate this claim.

We claim that the slow turnaround time for issues to be detected, fixed, and finally deployed by human operators can be made more autonomous and with lower latency if we introduce an automated security monitor to resolve the issues. One of the open questions that still remain is determining to what extent arbitrary programs can benefit from hardening and diversification. It is particularly important to consider the cost as most security enhancing features introduce overhead in terms of performance, compatibility, or usability, the mitigations suggested herein being no different.

References

1. Zimmermann, O.: Microservices tenets: Agile approach to service development and deployment. Computer Science - Research and Development pp. 1–10 (2016)
2. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. CoRR abs/1606.04036 (2016), `http://arxiv.org/abs/1606.04036`
3. Newman, S.: Building Microservices. O'Reilly Media (2015)
4. Fetzer, C.: Building critical applications using microservices. IEEE Security Privacy 14(6), 86–89 (Nov 2016)
5. Lysne, O., Hole, K.J., Otterstad, C., Ytrehus, Ø., Aarseth, R., Tellnes, J.: Vendor malware: Detection limits and mitigation. Computer 49(8), 62–69 (Aug 2016)
6. Richardson, C., Smith, F.: Microservices From Design to Deployment. NGINX, Inc. (2016)
7. Montesi, F., Weber, J.: Circuit Breakers, Discovery, and API Gateways in Microservices. CoRR abs/1609.05830 (2016), `http://arxiv.org/abs/1609.05830`
8. Hole, K.J.: Anti-fragile ICT Systems. Simula SpringerBriefs on Computing, Springer International Publishing (2016), `http://link.springer.com/book/10.1007/978-3-319-30070-2`
9. Sassaman, L., Patterson, M.L., Bratus, S., Shubina, A.: The halting problems of network stack insecurity. ;login: 36(6) (Dec 2011)
10. Homescu, A., Jackson, T., Crane, S., Brunthaler, S., Larsen, P., Franz, M.: Large-scale automated software diversity – program evolution redux (2015)
11. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems a secretless framework for security through diversity (2006)
12. Partridge, D., Krzanowski, W.: Software diversity: practical statistics for its measurement and exploitation. Information and Software Technology 39(10), 707 – 717 (1997)
13. Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., Franz, M.: Compiler-Generated Software Diversity, pp. 77–98. Springer New York, New York, NY (2011)
14. Ormandy, T.: Fireeye exploitation: Project zero's vulnerability of the beast. `https://googleprojectzero.blogspot.no/2015/12/fireeye-exploitation-project-zeros.html` (December 2015), [Online; accessed 7-February-2017]