Bee: End to End Distributed Tracing System for Source Code Security Analysis

Li Qiu [1, a], Xuyan Song [2, b], Jun Yang [1, c], Baojiang Cui [2, d]

[1] School of Computer Science, Beijing University of Posts and Telecommunications, Beijing, China.

[2] School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, China.

[a] echoqiuli@bupt.edu.cn, [b] sungxy@bupt.edu.cn, [c] junyang@bupt.edu.cn, [d] cuibj@bupt.edu.cn

Abstract. As distributed services become more and more complex, their security is becoming an increasingly important issue. End-to-end tracing has emerged recently as a valuable tool to improve the dependability of distributed systems, by performing dynamic verification and diagnosing correctness and performance problems. However, several shortcomings of the end-to-end tracing system`s security analysis are uncovered during the development. First of all, the density of probes is not enough, and also the descriptions of different operations are not consistent, which does not allow effective access to intermediate information of end-to-end services and brings about data analysis problems. Second, the implantation of probes is complex and many operations are highly coupled with the monitored program. The third point is that the sampling algorithm in the existing end-to-end distributed tracing system is too simple to effectively improve the performance of the high-density probe tracing system used for security analysis. In this paper, we address the above problem and successfully build Bee. To demonstrate the Bee's helpfulness for those problems in Security analysis, we test its performance and prove some privacy leaks vulnerabilities and access control vulnerabilities of OpenStack with Bee. The experimental results show that with the high-density probes. Bee can capture the detailed request process and quickly locate abnormal operations.

Keywords: Distributed Tracing, Vulnerability, OpenStack, Security Analysis.

1. Introduction

Following modern software engineering trends, systems are becoming larger and more decentralized, requiring new solutions and new development models. One approach that has emerged in recent years is the decoupling of large monolithic components into highly cohesive and low-coupling components. These components are called microservices and have become a staple in the enterprise software development industry. However, it also brings complexity to the software performance diagnosis and error location in the distributed microservice system [1].

To address these issues, the industry has adopted several observability techniques to increase system visibility. One of the important technology branches is the end-to-end tracing technologies [2-7]. And based on these end-to-end tracing systems, it is possible to go further for run-time anomaly request detection [3, 8, 9, 10]. At the same time, industry and academia have built some standards for distributed tracing such as OpenTracing [11], OpenTelemtry [12], W3C header [13], to ensure that software from various vendors can achieve converged end-to-end tracing. Ultimately, the core goal of end-to-end tracing tools is to log services across different components, layers, machines, and even administrative domains, enabling developers and operators to effectively understand how their systems are performing and analyze the causes of system anomalies. But using an end-to-end distributed tracking system for security analysis still faces some problems.

First, traditional end-to-end tracing frameworks collect limited data and the number of operations captured by probes is insufficient.

Secondly, the probe implantation of distributed tracing systems is often highly coupled with the system's architecture and has many manual tasks. For example, Dapper [14] makes deep use of Google’s unified RPC integration framework in implementing the tracing system. but many systems

often do not have such a unified messaging framework, and when we need to perform more fine-grained probe implantation, it often tests the engineer's understanding of the whole system and how to use a unified language of events to describe the behavior of the entire system. This is where many researchers and deployers of distributed tracing frameworks find it most time-consuming and difficult to deploy [2, 15, 16, 17]

Third, many distributed tracing systems need to consider the impact on the native system load, and it is difficult to accept a tracing system where the performance of the monitored system is significantly degraded by the distributed tracing system. Sampling is a common way to solve the performance problem. In Dapper [14], the sampling probability of some services is set as 0.001 to reduce the performance impact, but this probability setting is not straightforward and requires different strategies for different scenarios, taking into account the different characteristics of different services and the validity of the sampled data.

In this paper, we have successfully solved the above-mentioned problem of applying an end-to-end distributed tracing system to security analysis. Based on the OpenTracing [11] standard, we designed and implemented an end-to-end distributed tracing system for security analysis: Bee, and successfully embedded this system into the source code of the core service of OpenStack [18], and successfully verified the good performance and vulnerability mining capability of Bee through experiments. The main contributions of this paper are as follows:

A unified event logging model that is compatible with the principles of the OpenTracing standard. To ensure that the collected data can be securely analyzed, this paper proposes a quadruplet for recording operational events (identity, operation, input, output). This unified event logging model extends the traditional distributed tracing system to record information. It also effectively solves the problem of inconsistent data institutions at the data collection end, subsequent processing and analysis of the data [19]. In terms of the choice of where to implant the code probe into the application, this paper adopts a hierarchy of different granularities for the monitored interfaces. The coarse granularity includes service level HTTP requests, RPC requests, database accesses, and driver accesses. The fine-grained probes include function and class levels. In solving the complex problem of fine-grained probe implantation, this paper implements an automatic code probe implantation method based on a code abstract syntax tree. The function and class code probes are automatically implanted into the target point by matching them by rules.

In the remainder of this paper, we first present the related work in Chapter 2. Then, the design and implementation of Bee in Chapter 3, and experiments and performance evaluations of Bee in Chapters 4. Finally, the work is presented and summarized in Chapter 5.

2. Related work

As mentioned above, it is important to add visibility to distributed services. The earliest studies that did data collection from the log level [20-22], based on the logs generated by the system, performed in-depth feature analysis, but this independent log analysis, in the increasingly complex distributed systems, appears to be not comprehensive enough to effectively characterize the complex relationships among various services, and thus cannot locate problems quickly. To the now dominant end-to-end distributed tracing systems that focus on recording the causality of events [2, 3, 4, 5, 7, 14]. Including Bee, the core of these end-to-end tracing systems is to pass contexts between different processes, services, and use uniform identifiers to build logical, and temporal relationships between system services. But constructing logical relationships between services does not necessarily require messaging with the help of the context of a distributed tracing system. An alternative is to cluster or derive the logical and temporal relationships of services in the system from the existing output information of the program, such as logs. and using some uniform identification information, such as the calling process ID, IP address, etc. The analysis can be done using statistical and machine learning methods for inferring [21, 23, 24, 25, 26]. Compared to end-to-end distributed tracing systems, the advantage is that it does not require a lot of probe deployment work to get the service logic

relationship of the distributed system. However, these logical reasoning systems can have some obvious disadvantages the causality derived from reasoning is unreliable, the reasoning process is too complex, and it consumes too much time and resources, etc.

At the application level, the end-to-end distributed tracing system is widely used for performance diagnosis and problem analysis [3, 8, 14]. In [14], showing in detail how to use the Canopy tool to effectively detect the services that cause performance bottlenecks in Facebook.

The distributed tracing system is also widely used for system security protection [10, 27, 28, 33], these security protection systems, based on the timing and logical logging capabilities of the end-to-end distributed trace system, use machine learning in the collected trace data to compare request execution paths to analyze and identify the attack requests to which the system is exposed. In [33], they demonstrate an approach to detect dependent and concurrent events using the ability of the model to reconstruct the execution path.

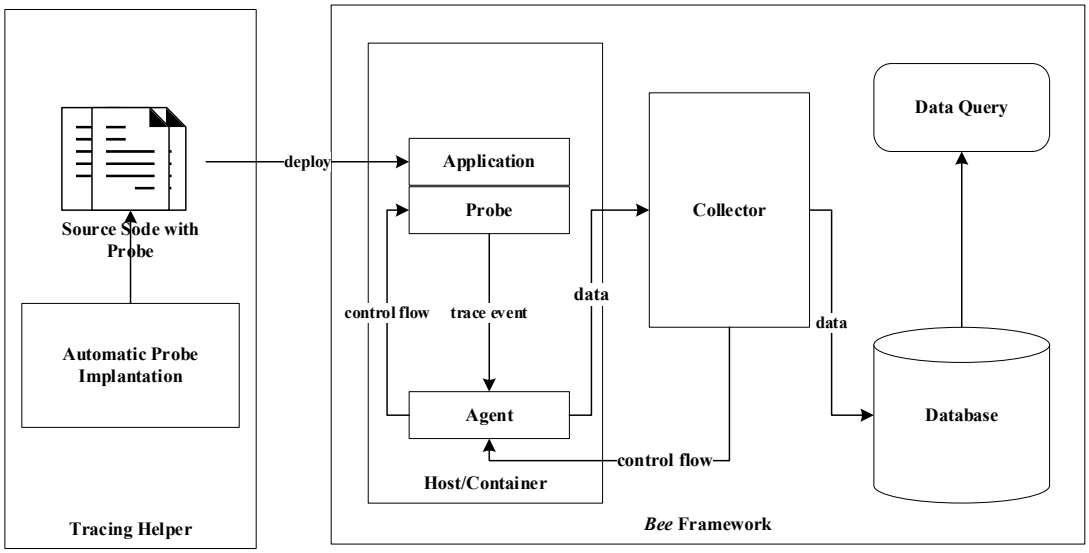

Figure 1. Architecture of Bee

3. Bee in action

This chapter is divided into four parts. The first part will introduce the Bee architecture in general, then the second and third chapters will introduce the data model of the Bee and how to implement semi-automated code probe insertion, and the fourth chapter will introduce the Bee as well as the use of basic head sampling to reduce the performance loss of the system while ensuring maximum information entropy and sampling fairness of the sampling system.

3.1 How bee work

Bee is divided into two general modules, a pre-analysis module and a distributed tracing module, the detailed architecture of which is shown in Fig. 1. The tracing helper uses static code analysis to automatically implant functionally granular decorators into the target code. Then, compile and deploy the new source code into the production environment. At runtime, a probe inserted into the application service collects execution data from the application and reports it to the tracing agent component, which sends the data reports in bulk to the data collector. Finally, the collector stores the trace data in the storage system after basic analysis such as aggregation operations.

3.2 Tracing helper

Bee' tracing assistance tool performs automatic probe implantation at target staking points through static code analysis, based on the rules of the configuration file. Since OpenStack, the service used for the analysis experiments in this article is based on the Python language. the automatic probe

insertion module uses Python's AST module. The process of parsing source code in python is to first parse the syntax to get a parse tree, then generate an abstract syntax tree, analyze it to get a control flow graph, and finally generate bytecode. The Python AST module generates an abstract syntax tree of the intermediate representation of a program from the source code. And provides an interface for accessing and modifying the abstract syntax tree, and for deserializing the modified abstract syntax tree into Python source code. The principle of the probe implementation is based on the Python language's decorator syntax. The principle is that when the Python interpreter runs Python code, the target function becomes an argument to the decorator function, so the decorator function can execute the decorator function's statements before and after the target function.

Based on these syntax features, Bee uses the Python AST module to generate an abstract syntax tree of the source code. A depth-first algorithm is then used to traverse the syntax tree, adding probe statements to the matching function definition nodes. Finally, the new syntax tree is deserialized into a source code file. The probes embedded in the syntax tree will obtain the identity of the manipulation, input, output, record operation events, and duration of the target function.

Tab. 1 and Fig. 2 show the process of implementing probe implantation by the Bee trace helper. First Bee' trace helper takes the source code of the program and generates the abstract syntax tree, the part of Fig. 2 that does not contain the orange node. Then a traversal program is used to add probe statements to the functions specified in the configuration. Function nodes that do not match the aim function were ignored. After traversing all the syntax tree nodes, a new abstract syntax tree is generated in Fig.2, and the orange part is the probe implanted. Finally, deserialize the abstract syntax tree to get the new source code.

3.3 Data model

In terms of the data model, Bee uses the OpenTraing standard as the basis for its model. OpenTracing standard defined three main types of data, Logging, Metrics, and Tracing. Logging is used to record discrete operational events, Metrics is used to record data that can be aggregated, such as the number of interface accesses per second, the number of operational events, etc. Tracing is used to record information within the scope of the request and to build logical call relationships and temporal relationships between services. Bee uses Logging module events to log access events for functions and components including HTTP, RPC, database access, driver access. And construct a unified event description model. The unified event description model includes access identity, lease duration, request, response results, and another service intermediate information relevant for security analysis.

3.4 Sampling

In an end-to-end distributed tracing system, sampling is one of the very important modules. This is because the computational power of implementing a data collection module will always be smaller than that of the system it is monitoring. And because of its end-to-end nature, it is not possible to use the traditional method of setting probabilities by subcomponents in a logging system, but rather a consistent sampling probability needs to be used on each trace. The importance of a request-based sampling module in a tracing system is mentioned in Dapper [14], and more recently in Facebook's introduction of Canopy [3].

The sampling strategy is divided into two categories. the first one, header-based sampling, where the sampling strategy is determined when the request is created. Dapper uses this strategy to achieve a rate limit for Trace data reporting. This effectively minimizes the performance impact of the Dapper system on the monitored system. Another strategy is tail-based sampling, where tail-based sampling collects data on said requests. The decision to process and store the collected Trace data is then made at the data collection end. Canopy [3], Sifter [29], Pedro Lass-Casas et al [30] are all based on tail sampling. After collecting all the end-to-end tracing data, use methods including usage of pipelines, or clustering, to decide whether the data needs to be stored or not. This ensures that the information collected by the tracing system is the most valuable.

Table 1. Example code before and after implant probe

#before implant probe
def add(arg1, arg2):
return arg1 + arg2
 	
def sub(arg1, arg2):
return arg1 - arg2
	#after implant probe
	@bees.profile.trace('add')
	def add(arg1, arg2)
	return arg1 + arg2
	
	def sub(arg1, arg2):
	return arg1 - arg2

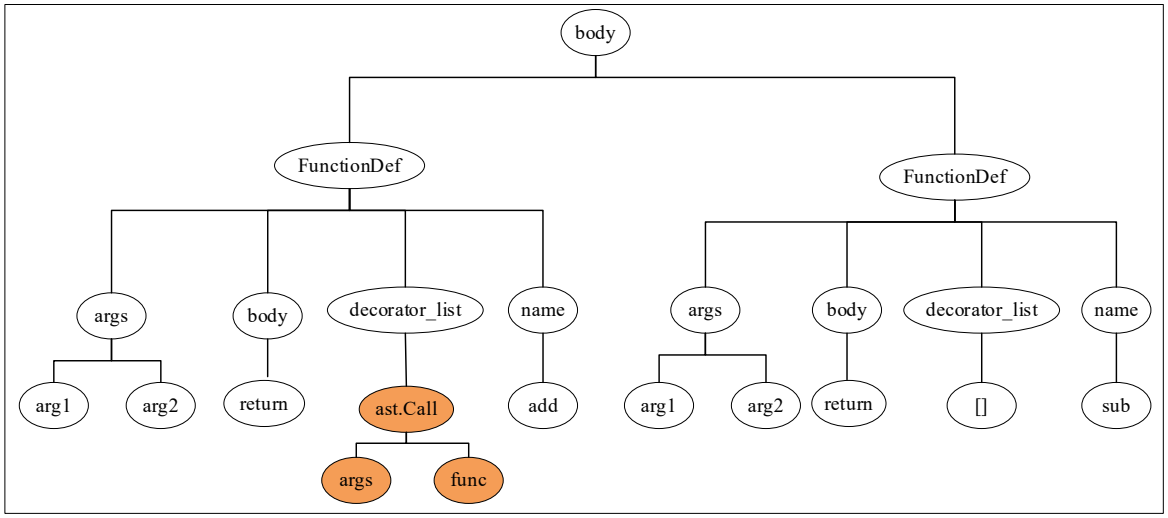

Figure 2. The AST tree of example code, the orange node represents the probe implanted.

Regardless of the strategy used, three elements should be considered when designing the sampling algorithm. The first point is to ensure the performance of the monitored system, a monitoring system that consumes too much performance will affect the developer's willingness to introduce a tracing system. The second point is to ensure fairness in the collection of information, or the validity of the data. In a large distributed system, different services have different traffic sizes, and the same service has completely different data volumes at different times, so it is unacceptable to simply set a fixed sampling probability for all services. The third point is the speed of decision-making. The decision of sampling strategy needs to be fast and efficient, because the traffic changes in the distributed system are dynamic, resulting in a strong time-sensitive sampling strategy. Among the two major existing sampling strategies, the header-based sampling strategy can effectively guarantee the performance of the probes, and have a relatively simple decision-making process when creating. However, it often leads to an unrepresentative collection of sampled data due to the lack of consideration of all features of all data. In the tail-based sampling decision, because the decision where keep the trace data is made at the data collection end. There will be more assurance of the value of the stored data. But it will be at the expense of the performance of the probe.

Based on the issues raised earlier and the characteristics of Bee as a security analysis framework, using more events and logging, with small probe granularity and high density of implantation. In this paper, we choose a header-based sampling strategy. A header-based adaptive sampling that guarantees maximum-minimum fairness is designed.

First, we can define the Sampling Problem: Give a set of execution traces and a sampling budget s, select a subset of the traces that are maximally diverse (or minimally redundant).

In the Idealize case in which N total traces is made of K services, with $|C_i|$ $1 \leq i \leq K$. as the number of traces in each service, The Sampling budget sN = S traces, and we can make sure budget $sN \geq K$, then every service will be capture at least one of each type of trace. How to allocate these

budgets to all services, A max-min fairness allocation of sampled traces per cluster provides precisely this.

$$\sum_{i=1}^{K} S_i = S \tag{1}$$

$$S_i = \min(|C_i|, \partial) \tag{2}$$

And the ∂ satisfy if $|C_i| \geq S/K \ \forall$ i, then $\partial = S/K$ .In this way, when each service has enough traffic, the trace budgets of all the different services will be allocated fairly, and this max-min allocation principle has been shown to have maximum information entropy [22].

In practice, however, it is difficult to set different sampling targets for services directly. This is because traffic volumes are not the same for different services, and the same service has different traffic volumes at different times of the day. Therefore, we adopt an adaptive approach, as shown in Fig.3, where the counter is a timing task. It is executed at the data collection, storing actual traffic data according to service names for different services, calculating the actual number of Trace in the system, and then comparing the difference between it and the expected number of Trace to dynamically set and regulate the sampling probability in the system, which is passed to the probe client. The header-based policy controls the generation of data for the entire link. Such a header-based sampling approach relieves the probe operation load of the service being inspected while combining the advantages of tail-based sampling, effectively taking into account real-time data characteristics, and ensuring max-min fairness and maximum information entropy of the system sampling while sampling.

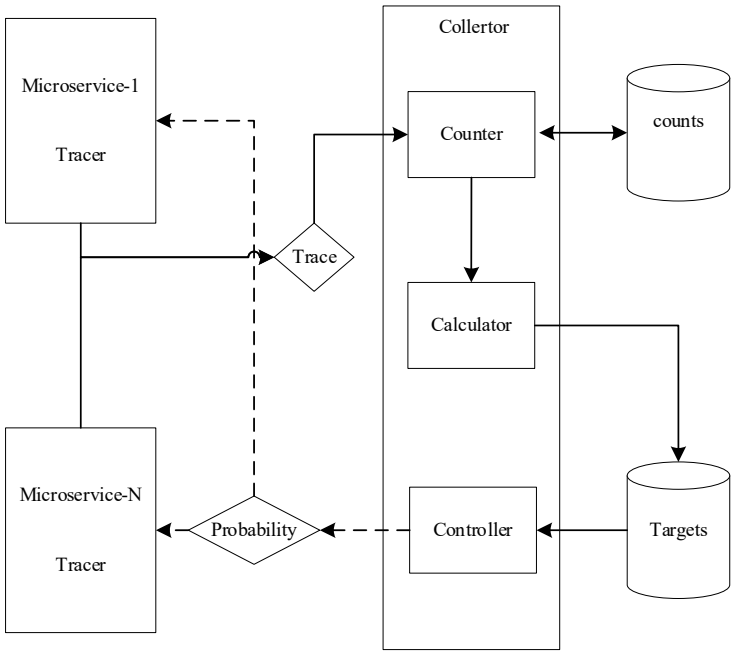

Figure 3. The implement of max-min fairness adaptive sampling

4. Experiment

Our entire experimental environment is deployed on 6 cloud hosts, each with 8 CPUs, 32GB of RAM, 512G disks, and Ubuntu 20.04 operating system. 4 are service deployment hosts for the application OpenStack [18], one host for source code analysis, and one host for data collection for distributed tracing system Bee. The host is used for source code analysis automatically implant the probe into source code, then compile and package the code into a container image, Finally, deploy container image to the target machine via a private image repository. The data collection node analyzes and stores trace data, and stores the data persistently. It is also responsible for executing the max-min fairness adaptive sampling algorithm.

Table 2. The test operations list

Test Tasks	Resources	Abbreviation
authenticate_user_and_validate_token	Identity	AUAVT
create_delete_user	Identity	CDU
create_user_update_password	Identity	CUUP
list_get_hypervisors	Compute	LGH
boot_delete_server	compute	BDS
create_delete_image	Image	CDI
create_image_boot_instances	Image and compute	CIBI
create_delete_network	Network	CDN
create_and_delete_security_groups	Network	CADSG

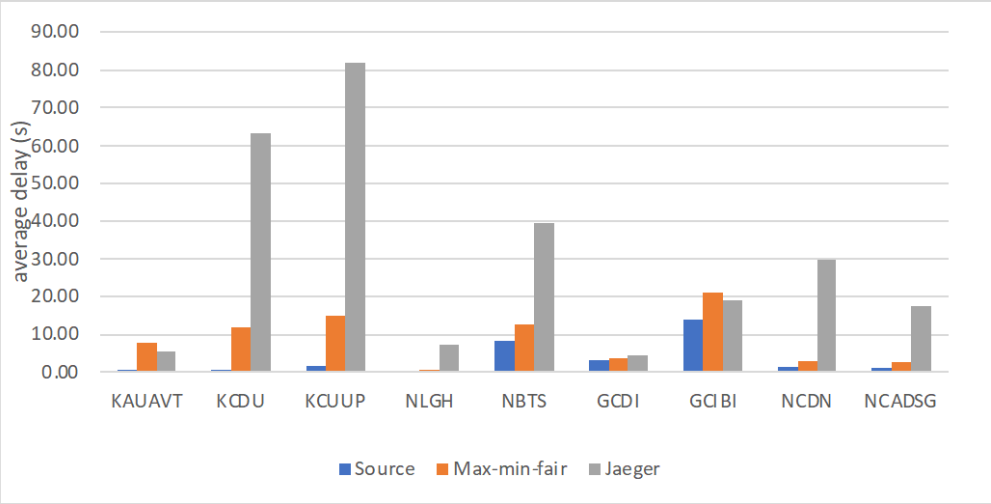

Figure 4. Average latency comparison chart. The vertical axis in the bar chart is the average time delay (Seconds), the blue bar represents the source code performance, and the gray bar means using the sampling strategy of Jaeger, and the orange bar the Max-min-fair is our new strategy.

4.1 Performance evaluation

In order to effectively probe the performance load of the Bee framework. we used the test framework of OpenStack Rally. Rally tools allow developers to customize test tasks and execute the task. we have selected some representative selection of read and write operations, as present in Tab. 2 Every test task in the table was executed 500 times.

In the meantime, we have deployed and run three systems separately to compare the performance of the Bee framework using the max-min fairness sampling strategy. These are the native system without probes, the system with code probes, the system with sampling strategy using Jaeger [7], and the system with the max-min fairness adaptive sampling strategy. The average time taken for requests in percentage is shown in Figure Fig.4.

As can be seen in Fig.4, with the use of Jaeger's native sampling strategy, the request-response rate of the system drops significantly after the implantation of probes. In the blue bar graph representing the request latency of the native system, there are operations with a latency of less than 1 second, while the highest load growth is nearly a hundred times with Jaeger's sampling strategy. This indicates that after using a high density of probes and using detailed event logging to describe operations at all levels, the probes have a very serious impact on the performance of the system, illustrating the need to use the new sampling strategy. In contrast, after using max-min fairness header-based adaptive sampling, the performance impact decreases significantly, with an average latency reduction of 70.00 %. This proves the effectiveness of our sampling algorithm. The comparison also allows us to conclude that operations with low latency are sensitive to probe implantation, while operations with

high latency are not sensitive to the performance impact of probe implantation, providing a guideline for how to reduce the performance impact when doing probe implantation.

4.2 Proof of concept

To validate the effectiveness of our proposed granular implantation probes and our unified event description model, and the success of our automated probe implantation. We selected known vulnerabilities of different components of OpenStack and performed vulnerability validation based on the Bee framework. The list of successfully recovered and validated vulnerabilities is shown in Tab.3 below.

The verification process for these vulnerabilities based on the Bee framework is as follows: first, the vulnerable software version is selected, probed, and then compiled and deployed. After preparing the experimental environment, we simulate a mixture of abnormal and normal requests, and finally analyze and compare the results of the different requests captured by Bees to obtain the data of the abnormal requests, thus proving the triggering of the vulnerability.

For example, CVE-2019-19687 [23] In keystone versions 15.0.0 ~ 16.0.0, when configuration enforce_scope is False, a user under a project can use the list credentials interface to obtain the credentials of all users under that project, resulting in an information leak on the system.

Table 3. The CVE list proven by the Bee framework

Common Vulnerablities and Exposures Number	type	Component
CVE-2018-20170	Privacy leaks	Keystone
CVE-2019-19687	Privacy leaks	Keystone
CVE-2013-4292	Permission leaks	Keystone
CVE-2014-2237	Permission leaks	Keystone
CVE-2013-7130	Privacy leaks	Nova
CVE-2014-0167	Access Control	Nova
CVE-2019-10876	Access Control	Neutron

When using the Bee to verify the vulnerability, first create two experimental users, A and B, under a project. and user A invokes the system's list credentials, A can obtain the identity of user B. User A then uses user B's Credentials to request resources. Here, in our verification, we perform both normal and anomalous requests, using Bee to capture service calls, and we find that Bee can uncover 116 operations in the trace data generated by a single list credentials request. The first anomaly is that when user A invokes the system list credentials, the database access operation returns more user credentials than normal. The second anomaly is that when user A uses the illegally obtained credentials of user B to make a resource request, data that does not exist in the normal request will appear, but the comparison reveals the same resource information as the victim user B. B's resource information is consistent. Finally, the existence of system privacy leakage was confirmed.

Through this experiment, we can effectively demonstrate the success of our OpenTracing data model extensions and the proposed unified event description model. And the captured trace data shows that our multi-granularity probe implantation, and the use of automated function probe implantation, ensure that we can obtain detailed execution paths in a single request. Finally, the analysis comparison yields anomalies that effectively demonstrate the effectiveness of Bee in security analysis tasks.

5. Conclusion

In this paper, we introduce Bee, an end-to-end distributed tracing tool based on the OpenTracing standard. And in the process of implementing the system, First, we chose different positions to implant the probe, and extend the data probe collect. Then to better solve the problem of complex work of implanting probes in non-VM languages, we propose the introduction of static code analysis,

combined with abstract syntax trees, to automate implant the probe of the Bee. Also, to optimize the impact of probes on the overall performance of the system, a maximum-minimum-fair sampling strategy is adopted for the system to ensure performance while maximizing the information entropy of the collected data. Finally, the vulnerability mining capability of the system is verified using proof of concept.

Through this paper, we validate Bee's ability to perform vulnerability mining and explore a new idea for probe implantation: a framework-independent static code analysis approach for implantation. Based on this system, we can effectively integrate the security protection system and performance diagnosis system in the future, and finally realize the runtime protection and vulnerability mining of the system. Of course, there are still some shortcomings in this system, such as the automatic probe module, which is still limited to the implantation of probes of function types, and for HTTP-type requests, it still relies on the manual addition of custom middleware in the configuration file.

In the future, based on the framework of this system, we will combine the experience of the existing vulnerability verification recurrence, propose an automatic detection model, perform targeted test requests to the system, and automatically analyze and compare the captured data to mine more complex cross-service logic vulnerabilities of distributed systems.

References

[1] P. Di Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in 2017 IEEE International Conference on Software Architecture (ICSA). IEEE, 2017, pp. 21–30.

[2] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-trace: A pervasive network tracing framework," in 4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07), 2007.

[3] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, et al., "Canopy: An end-to-end performance tracing and analysis system," in Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 34– 50.

[4] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in Proceedings of the 25th Symposium on Operating Systems Principles, 2015, pp. 378–393.

[5] Twitter. Zipkin. Retrieved October 2021 from http://zipkin.io/.

[6] Apache.Skywalking. Retrieved October 2021 from https://skywalking.apache.org/.

[7] Uber.Jaeger. Retrieved July 2021 from https://www.jaegertracing.io/.

[8] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows." in NSDI, vol. 5, 2011, pp. 1–1.

[9] K. Ostrowski, G. Mann, and M. Sandler, "Diagnosing latency in multi-tier black-box services," 2011.

[10] Y.-Y. M. Chen, Path-based failure and evolution management. University of California, Berkeley, 2004.

[11] Opentracing.Retrieved December 2020 from https://opentracing.io/.

[12] C. N. Foundation, "Opentelemetry," 2021, https://opentelemetry. io/.

[13] Trace Context. Retrieved June 2021 from https://w3c.github.io/trace-context/.

[14] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large scale distributed systems tracing infrastructure," 2010.

[15] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale Internet services," in 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), 2014, pp. 217– 231.

[16] R. Fonseca, M. J. Freedman, and G. Porter, "Experiences with tracing causality in networked services." INM/WREN, vol. 10, no. 10, 2010.

[17] R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger, "Principled workflow-centric tracing of distributed systems," in Proceedings of the Seventh ACM Symposium on Cloud Computing, 2016, pp. 401–414.

[18] "Openstack," 2021. [Online]. Available: https://www.openstack. org/

[19] A. Bento, J. Correia, R. Filipe, F. Araujo, and J. Cardoso, "Automated analysis of distributed tracing: Challenges and research directions," Journal of Grid Computing, vol. 19, no. 1, pp. 1–15, 2021.

[20] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering," in Proceedings of the ACM Symposium on Cloud Computing, 2019, pp. 312–324.

[21] P. Las-Casas, J. Mace, D. Guedes, and R. Fonseca, "Weighted sampling of execution traces: capturing more needles and less hay," in Proceedings of the ACM Symposium on Cloud Computing, 2018, pp. 326–332.

[22] A. Coluccia, A. D'Alconzo, and F. Ricciato, "On the optimality of max–min fairness in resource allocation," annals of telecommunications-annales des télécommunications, vol. 67, no. 1, pp. 15–26, 2012.

[23] "Cve-2019-19687," 2019. [Online]. Available: https://nvd.nist.gov/ vuln/detail/CVE-2019-19687.

[24] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), 2012, pp. 353–366.

[25] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 117–132.

[26] Y. Jiang, L. R. Sivalingam, S. Nath, and R. Govindan, "Webperf: Evaluating what-if scenarios for cloud-hosted web applications," in Proceedings of the 2016 ACM SIGCOMM Conference, 2016, pp. 258–271.

[27] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 468–479.

[28] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1285–1298.

[29] G. Mann, M. Sandler, D. Krushevskaja, S. Guha, and E. Even-Dar, "Modeling the parallel execution of black-box services." in HotCloud, 2011.

[30] A. J. Oliner, A. V. Kulkarni, and A. Aiken, "Using correlated surprise to infer shared influence," in 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN). IEEE, 2010, pp. 191–200.

[31] D. Gorige, E. Al-Masri, S. Kanzhelev, and H. Fattah, "Privacy-risk detection in microservices composition using distributed tracing," in 2020 IEEE Eurasia Conference on IOT, Communication and Engineering (ECICE). IEEE, 2020, pp. 250–253.

[32] S. Jacob, Y. Qiao, and B. A. Lee, "Detecting cyber security attacks against a microservices application using distributed tracing." in ICISSP, 2021, pp. 588–595.

[33] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE, 2019, pp. 179–186.