Software Development Activities for Secure Microservices

Peter Nkomo(✉) and Marijke Coetzee(✉)

University of Johannesburg, Auckland Park 2006, South Africa
ptnkomo@gmail.com, marijkec@uj.ac.za

Abstract. The decomposition of an application into a set of distributed and collaborating microservices using microservices architecture principles, increases an application's attack surface. A preliminary risk analysis can provide an understanding of security threats from a hypothetical attacker's point of view. Identified security threats equip software engineers of microservices compositions with knowledge of assets most likely to be targeted, the most likely attack vectors, and the potential attacker's profile. The knowledge is useful to ensure that microservices compositions are designed to avoid vulnerabilities and to withstand any attack, and in the event of an attack to ensure that adverse consequences of an attack are minimized. In this regard, this paper aims to identify security threats that could arise as a result of flaws in the design of microservices compositions and harm that may arise from misuse of a microservices composition by malicious users. The preliminary risk analysis leads to a list of security requirements to be met by this research to be able to develop secure microservices compositions. The contribution of this review is a list of development activities for secure microservices.

Keywords: Microservices architecture · Security threats · Security requirements · Software development activities

1 Introduction

In the quest to compete in a fast-paced business environment, the complexity of maintaining and enhancing traditional SOA applications has presented significant challenges to many enterprises [1]. These challenges have led to the emergence of a new architectural style called the microservices architecture [1, 2]. The microservices architecture uses a collection of small, loosely coupled software components called microservices that collaborate to automate business functionality, and are ideally developed within fast software release cycles [3]. The microservices architecture promises to provide agility by allowing each microservice to be quickly built, modified, tested and deployed in isolation [4].

Over the past few years, a new trend called DevOps has emerged that aims to unify agile software development, continuous integration, continuous delivery, continuous deployment with software operations [5]. DevOps aims to shorten software development cycles, increase the frequency of deployments and create more dependable software releases that are closely aligned with business objectives. The goals of

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11623, pp. 573–585, 2019.
https://doi.org/10.1007/978-3-030-24308-1_46

DevOps are achieved using automation at all steps of software development from integration, testing, releasing software to production and also the management of servers. A common set of DevOps and continuous delivery ideologies at companies such as Amazon [6], Netflix [7], SoundCloud [8], Facebook [9] and several others has been instrumental in the adoption of the microservices architecture. Microservices architecture is seen as a natural fit to enable continuous delivery and has become a prelude of a new form of concrete implementation of SOA. The adoption of microservices architecture as part of DevOps practices introduces complications when implementing security controls. As development teams continue to deliver software in short and agile sprints cycles, usually one to two weeks in length, often little attention is given to the security of the application.

The research question to be addressed by this paper is *How can software engineers build microservices in a systematic way so that security is an integral part of the entire microservices lifecycle?* This requires an understanding of the underlying architecture that software engineers should address as there may exist new security weaknesses [10]. Once potential security challenges are identified, appropriate security guidelines can be identified to assist software engineers to avoid subtle architecture-level security weaknesses.

The main contribution of this paper is a list of software development activities to be applied in agile approaches when developing secure microservices. Such a list of guidelines can be useful and convenient for software engineers who are generally not trained in software security. To address this research question, Sect. 2 identifies the architectural design features of the microservices architectural style using an example application. Section 3 describes five security challenges of the microservices architectural style. Section 4 pinpoints the security threats due to the identified security challenges. Section 5 gives relevant security requirements that are identified from the list of security threats. Finally, a list of software development activities is given that can guide software engineers when developing secure microservices and finally the paper is concluded.

2 The Microservices Architectural Style

A *microservice* is defined as a self-contained, autonomous, lightweight unit of logic running in its own process [3]. Microservices communicate using lightweight mechanisms over hypertext transfer protocol [2], using the RESTful architectural style as a means of communication. A *microservice* provides a narrowly-focused standardized application programming interfaces to its consumers. The *microservices architectural style* is an approach that structures an application as a set of loosely coupled collaborating microservices. There are no set of rules when choosing between various frameworks or protocols to use in a *microservices architecture*. However, the protocol should be lightweight, keeping in mind that the microservices architecture relies heavily on messaging between collaborating microservices. Using this architecture style, an enterprise can structure development teams as a collection of small autonomous teams, usually at most nine members, who focus on one or more microservices [11].

In a microservice architecture, choreography is preferred when creating a microservices composition, unlike traditional SOA that employs orchestration. No central microservice is thus used as a composition controller to control communication with other microservices [12]. Microservices communicate using point-to-point exchanges or by listening to events on their environment [13]. The inter-communication mechanisms can either be synchronous or asynchronous.

Consider an imaginary on-demand taxi application such as Uber [14] that is implemented as an SOA-based application as shown in Fig. 1.

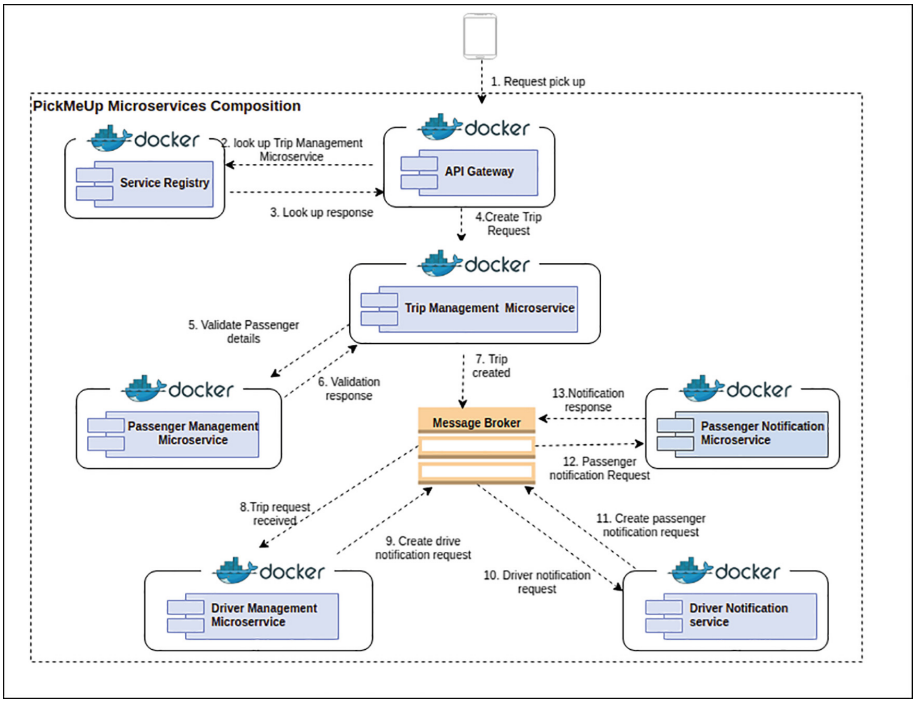

Fig. 1. PickMeUp application

The application is referred to as the PickMeUp application. Registered passengers request taxi rides using mobiles phone or desktop computers. As soon as the request is made, a notification about location and passenger details is sent to the nearest driver. The driver either accepts or rejects a request for a ride. In case the ride is rejected, a notification is sent to drivers in the area. If the driver accepts the ride, driver details are sent to the customer along with the estimated arrival time. The passenger can track the drivers and drivers can track the exact location of the passenger to reach their exact location. The payment procedure between the passenger and the driver is either cash or credit card. The various architectural components of the microservices composition for the PickMeUp application are shown in Fig. 1 namely the API gateway, a service registry, and message broker. The deployment strategy adopted for the application is to

deploy components of the application on separate Docker containers that run on a single host. Access to the composition is given via the API gateway. The API gateway locates the instance of a trip management microservices using the service registry. The trip management microservices communicate either directly by calling other microservices' REST interface or by sending a message to the message broker. Each microservice is deployed in its Docker container.

Security challenges arise from the manner in which microservices are deployed as the microservices architecture provides an attacker with more avenues to attack the microservices compositions. In general, a set of distributed microservices are designed to trust each other completely where a compromise of a single microservice could affect an entire application. Furthermore, many of the most popular tools used for ensuring continuous integration, continuous delivery and continuous deployment and DevOps are often new to the market or are open-sourced. The relative immaturity leads to concerns about the degree to which secure development standards are being adhered to.

The manner in which microservices security can be implemented is described next.

3 Security Challenges of the Microservices Architecture

The development of secure microservices applications is not a problem that has been solved yet. However, the core security principles that apply to SOAP-based, and RESTful web services hold for microservices as well. WS-Security [15] specifies the way integrity and confidentiality can be enforced, and security tokens used for authentication. Middleware is used to enforce distributed security via components such as interceptors [16]. In contrast, the RESTful style of web services does not provide any formal guidance on how security mechanisms should be applied and leaves their implementation to the discretion of software engineers. Such services mainly rely on ad-hoc security mechanisms or transport layer security.

With microservices, security becomes more of a challenge because no middleware component is available to manage security-based functionality. Instead, each service is required to manage security on its own, or in other cases, an API gateway is given the responsibility of managing the security of the application. REST web services lack a specific security model, unlike SOAP-based services which rely on the WS-Security standard. Most REST web services rely on transport-layer security and custom message protection mechanism. Transport-layer security offers secure point-to-point communication channels. REST web services can use JSON Web Tokens (JWT) [17] as the format for security tokens for authentication and ensuring message integrity. API keys and JWT does not solve the problem of confidentiality therefore custom mechanisms need to be used to ensure data confidentiality and non-repudiation.

Microservices are changing the assumptions about how SOA-based applications are created and consequently how SOA applications should be secured. The adoption of microservices architecture presents the following five security challenges.

(a) Increased surface attack: When microservices are considered from a networking perspective, the instance of a microservice is a unique network endpoint with an open network port exposing an application programming interface. When a

new instance of microservice is created, a new application programming interface is exposed. An attack on the microservices-based application can be made directly on each microservice [2]. This gives the attacker an increased attack surface due to the spread of microservices instances exposed across the network. Security of microservices consequentially become a distributed security challenge.

(b) Indefinable security perimeters: Many microservices are deployed in containers [18]. The challenge of deploying microservices on containers is that containers can be set up quickly from anywhere within the network without any consideration for the traditional notion of demilitarised security perimeters. Containers allow port mapping functionality to masquerade standard microservices application programming service ports to dynamically allocated ones. The use of dynamic addressing and scaling of microservices makes it a challenge to statically configure internet protocol addresses or steer network traffic to traditional perimeter security appliances.

(c) Security monitoring is complex: Containers present a security monitoring challenge. Containers on a host machine can use network address translation (NAT) which makes them invisible to the outside world. Network address translation is the process where a network device, usually a firewall, assigns a public address to a computer inside a private network. Network traffic from containers using NAT is challenging to identify. When containers use NAT, a definition of security policies becomes complicated because it becomes difficult to know which microservices is running in each container. Containers may also bundle applications with a lot of software libraries and files that software engineers may not be aware of. This may increase the security risk due to vulnerabilities that may be hidden inside the software libraries.

(d) Authentication is centralized: Microservices deployed in containers interact remotely, mostly over HTTP. The challenge of this approach is how users of microservices are authenticated and how user credentials are passed between microservices in a symmetric manner. Another challenge when ensuring inter-microservices communication between a large number of microservices is that when microservices use transport layer security, certificate revocation becomes a harder problem. The microservices that initiate the handshake may get a list of revoked certificates from the corresponding certificate authority which can grow bigger.

(e) Threat modeling and risk assessment is localized: The emphasis on team autonomy makes it challenging to ensure that threat modeling, and risk assessment is done before new versions of microservices are released. Continuous delivery can mean that new vulnerabilities are delivered with every new microservices deployment.

The five security challenges above provide an answer to the research question formulated in the introduction of this paper. Considering the five challenges listed above, the adoption of microservices, therefore, require new ways of ensuring security. To this end, the next section discusses a preliminary risk analysis of the microservices architecture to provide an understanding of security threats. This knowledge is useful to ensure that microservices-based applications are designed to avoid vulnerabilities and to withstand any attack.

4 Microservices Composition Threat Modeling

Various security weaknesses or vulnerabilities may exist on microservices and their runtime infrastructure that an attacker can exploit. To be able to understand the security of microservices compositions, it needs to be analysed from the perspective of a potential attacker. There are various approaches to threat modeling, where the architecture-centric threat modeling approach is found to be more suitable as it provides a means to step through the components of a microservices composition to identify potential types of attacks against each component. The architecture-centric threat modeling approach, also called system-centric, or design-centric approach focuses on the design of a system and attempts to identify the potential types of attacks against each component of the system [19]. The architecture-centric threat modeling steps from the Microsoft threat modeling process is followed, as shown in Fig. 2 [20]. Each of the four steps are followed by analyzing the PickMeup application.

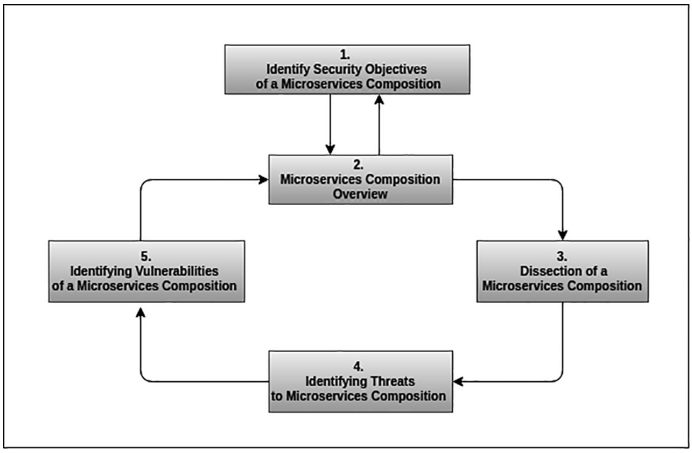

Fig. 2. Architecture-centric threat modeling

Step 1: *Identify microservices composition security objectives*. Each microservice is an open system similar to any web service, and therefore needs to address security services as prescribed by ISO 7498-2 [21] namely authentication, access control, data confidentiality, data integrity and non-repudiation.

Step 2: *Microservices composition overview*. The essential components that any microservices architecture application generally consist of is as follows:

- *The API gateway* - a lightweight entry point into an application.
- *A set of microservices* - components that automate business functionality.
- *Service registry* - a database of instances and locations of all active microservices in a microservices composition.

- *Message broker* - used by microservices in composition to publish and receive messages.
- *Containers or virtual machine* - provide the runtime environment to microservices.

Step 3 and 4: *Decomposition of a microservices composition:* Steps 1 to 13 in Fig. 1 shows the flow of information from when a request for a trip is received from a passenger using a mobile device until the final response is sent. For the sake of brevity, information flow for payment is not shown. The deployment strategy adopted for the application is to deploy components of the application on separate Docker containers. The containers run on a single host. The following architectural components provide potential entry points to maliciously access and compromise the PickMeUp microservices composition:

- *The API gateway and the microservices API* - the attacker may use the gateway and microservices API to perform various types of injection attacks.
- *The service registry* - the attacker may control the service registry to compromise the microservices composition or to shut the microservices composition down by ensuring that collaborating microservices cannot locate one another.
- *Message broker* - the attacker may gain access to messages exchanged by microservices or to bring the message broker down so that the composition cease to function.
- *Container or virtual machine* - the attacker may gain control of the runtime environment where the application is running and control or shut down the microservices composition.

  The four entry points listed above can in general form the attack surface of any microservices composition. Considering the technical design and implementation choices made during the development of the API gateway, service registry, message broker, containers or virtual machines the following five security threats are derived from the four entry points listed above:

  1. Insecure application programming interfaces
  2. Unauthorized access
  3. Insecure microservice discovery
  4. Insecure runtime infrastructure
  5. Insecure message broker

Step 5*: Identifying vulnerabilities for a microservices compositions:* Conceptually, threat modeling is performed by applying a methodology. For this research, STRIDE [22] is chosen as it offers a very systematic approach to analyse threats against each of the microservices architectural components. STRIDE supports a comprehensive review of security services such as authentication, authorization, confidentiality, integrity, nonrepudiation, and availability. STRIDE is an acronym that stands for spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege.

Due to space considerations, only the first of the five threats is discussed as an example of how such vulnerabilities are identified.

Insecure Application Programming Interfaces: A weak set of APIs exposes microservices to a variety of security attacks that may result in tampering with data, information disclosure, denial of service and elevation of privileges. Table 1 below list the applicable attack methods and weaknesses on the composition that can make the attack possible.

Table 1. STRIDE analysis of insecure application programming interfaces

Security threats	Attack methods	Exploitable weaknesses or known vulnerabilities
Tampering with data	• Intercept and modify messages sent to a microservices API when communication channels used is not secured • Exploit vulnerability in mechanisms used for transport-layer security • Perform all forms of injection attacks on the API	• Insecure communication channel • Lack of mechanisms to protect against injections of all forms on the APIs • Weak access control schemes on the microservices API • Vulnerability CVE-2014-3566 [23] that allows an attacker to obtain clear text when Secure Socket Layer (SSL) v3.0 is used
Information disclosure	• Perform all forms of injection attacks on the microservices API • Exploit weak access control schemes used to protects APIs	• Lack of mechanisms to protect against injections of all forms on the APIs • Weak access control schemes on the API. For example, the United States of America Internal Revenue Service (IRS) exposed over three hundred thousand (300 000) customer records using a vulnerable web API [24] • Vulnerability CVE-2017-9805 in the REST plugin of a web application framework called Struts. The vulnerability resulted in the Equifax data breach were an attacker gained access to consumer credit reports of about one hundred and forty-three (143) million United States citizens [25]
Denial of service	• Craft a request to API gateway that fans out into multiple computationally expensive requests to microservices behind the gateway so that microservices slow down and impact all legitimate users [26]	• Failure to prioritize authenticated traffic over unauthenticated one traffic • Lack of reasonable microservices requests time-outs • Lack of fallback options when microservices does not respond on time • Lack of fault isolation mechanisms
Elevation of privileges.	• Exploit a parser of messages used on the API that allow deserialization of hostile or tampered objects by changing the serialized object to gain administrative privileges	• Insecure message deserialization

5 Microservices Composition Security Requirements

Security requirements for a microservices composition describe more concretely the conditions and capabilities that must be met or be possessed by the microservice composition to assure the security of assets. Security requirements of interest for this research are secure functional requirements, functional security requirements and secure development requirements. Using the five security threats listed in the tables above, the security requirements for a secure microservices composition are now documented. Satisfying the security requirements should lead to more secure microservices compositions (Table 2).

Table 2. Security requirements

Security threats	Security requirements	Suggested protection measures
Insecure application programming interfaces	• Only authenticated users should access the API • Keys, tokens, and password should be rotated periodically • The API should validate all requests • The communication channel between microservices should be secure	• Use keys or security tokens or passwords to protect API • Perform input validations on the microservices API • The API should white-list permitted HTTP methods • Ensure secure management of keys, password, and tokens • Use transport-layer security • Monitor the microservices API at all times
Unauthorized access	• Access to microservices should be denied by default • The microservices composition should use multi-factor authentication at all entry points • Any credentials used in the microservices composition should be rotated periodically	• Use keys, security tokens, and password to protect API • Use transport-layer security • Automate management of keys, password, and tokens
Insecure microservice discovery	• The service registry should authenticate all requests for registration • Communication between microservices and service registry should use a secure channel • Messages for registration and de-registration should be protected for integrity	• Ensure the host on which the registry run is securely configured • The service registry should use certificates and certificates should be distributed securely • Use the transport-layer security • Monitor the service registry at all times

(*continued*)

Table 2. (*continued*)

Security threats	Security requirements	Suggested protection measures
Insecure runtime infrastructure	• Containers and virtual machines should only use verified operating system platforms or container-specific operating system • The outbound network traffic sent by container should be monitored and controlled • The configuration of containers and virtual machines should comply with the configuration standards	• Create secure configurations of infrastructure • Validate the configurations of infrastructure • Scan container images before deployment • Grouped containers by relative sensitivity and only run containers of a single sensitivity level on a single host • Monitor infrastructure at all times
Insecure message broker	• The message broker should authenticate all requests • No client should be able to subscribe to data on the broker • A secure channel should be used for communications • The client should protect the message it sends for integrity • A redundancy mechanism should be configured to guarantee the delivery of the messages	• User transport-layer security • Use authentication plugins or write a custom filter to authenticate a message • Set up read and write permissions on the message broker • Monitor the service registry at all times

The list of security requirements and the protection measures above points to the need to integrate security in different phases of the software development lifecycle such as requirements gathering, design, implementation, and testing. To achieve this, various security-focused activities are required during the different development phases of microservices to assure that the microservice composition is not subject to security vulnerabilities. The Open Security Alliance refer to such security-focused activities as secure development activities [27].

6 Software Development Activities for Secure Microservices Compositions

Using the security requirements and suggested protection measures, a list of six software development activities is now derived that can be used to ensure that microservices are adequately protected.

1. *Document security requirements of microservices compositions*. Documentation is vital to provide a security strategy to secure various assets of the microservices.
2. *Adopt secure programming best practices*. A vital activity is to ensure that engineers adopt identified guidelines to ensure that microservices are designed and

implemented to avoid vulnerabilities. Examples of such documented instructions include validating inputs, ensuring that the application executes with the least set of privileges required for the job and sanitizing any data sent to other application to avoid injection attacks.

3. *Validate security requirements and secure programming best practices*. Validating the implementation of a microservices composition against a set of security requirements and security coding standards at various stages of the microservices' build, test and deployment is vital to ensure that end-to-end security.
4. *Secure configuration of runtime infrastructure*. A vital security activity is to ensure that containers and the virtual machine are securely configured in an automated manner. Such configuration eliminated human errors that may result in misconfiguration.
5. *Continuously monitor the behavior of components of the microservices composition*. Ensuring continuous security of microservices composition requires engineers to have a view of the behavior of the various components at runtime. Given the potential for harm that can arise from persistent attacks by hostile entities at runtime, there is a need to monitor microservices and their runtime environments. Monitoring allows identification, detection, and even ability to foresee critical events and situations that occur during runtime.
6. *Securely respond to attacks using adaptation mechanisms.* A microservices composition should be built to withstand failures of individual components. Mechanisms that ensure that the application responds adequately to changes at runtime to maintain an appropriate security posture are referred to as secure adaptation mechanisms.

7 Conclusion

Building secure microservices compositions is a complicated exercise. This paper has laid the foundation towards understanding the microservices architecture's security threats using an attacker's point of view and identified security requirements common to most implementation of the microservice architecture. Suitable protection measures have also been identified. Even when security requirements and protection measures are identified early during the development process, there is still the challenge of validating if the implementation of various components of the architecture is safe. Furthermore, there is a need to ensure that the runtime environment does not provide attackers with the means to control or compromise the microservices composition. To this end, this paper identified security-focused activities that ensure that the suitable protection measures are implemented. The next research will identify various tools, techniques, and methods that can be used to support software engineers in incorporating the six secure development activities to become part of their daily development task.

References

1. Zimmermann, O.: Do microservices pass the same old architecture test? Or: SOA is not dead-long live (micro-)services. In: Microservices Workshop at SATURN Conference, SEI (2015)
2. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216 (2017)
3. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media Inc., Newton (2016)
4. Bossert, O.: A two-speed architecture for the digital enterprise. In: El-Sheikh, E., Zimmermann, A., Jain, L.C. (eds.) Emerging Trends in the Evolution of Service-Oriented and Enterprise Architectures. ISRL, vol. 111, pp. 139–150. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40564-3_8
5. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Boston (2015)
6. Bernstein, D.: Is Amazon becoming the new cool software company for developers? IEEE Cloud Comput. **2**(1), 69–71 (2015)
7. Ravichandran, A., Taylor, K., Waterhouse, P.: DevOps foundations. In: DevOps for Digital Leaders, pp. 27–47. Apress, New York (2016)
8. Baresi, L., Filgueira Mendonça, D., Garriga, M.: Empowering low-latency applications through a serverless edge computing architecture. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 196–210. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_15
9. Feitelson, D.G., Frachtenberg, E., Beck, K.L.: Development and deployment at facebook. IEEE Internet Comput. **17**(4), 8–17 (2013)
10. Feng, Q., Kazman, R., Cai, Y., Mo, R., Xiao, L.: Towards an architecture-centric approach to security analysis. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 221–230. IEEE, April 2016
11. Lalsing, V., Kishnah, S., Pudaruth, S.: People factors in agile software development and project management. Int. J. Softw. Eng. Appl. **3**(1), 117 (2012)
12. Butzin, B., Golatowski, F., Timmermann, D.: Microservices approach for the internet of things. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–6. IEEE, September 2016
13. Sheng, Q.Z., Qiao, X., Vasilakos, A.V., Szabo, C., Bourne, S., Xu, X.: Web services composition: a decade's overview. Inf. Sci. **280**, 218–238 (2014)
14. Rogers, B.: The social costs of Uber. U. Chi. L. Rev. Dialogue **82**, 85 (2015)
15. Nadalin, A., Kaler, C., Monzillo, R., Hallam-Baker, P.: Web services security: SOAP message security 1.1 (WS-Security 2004). Oasis Standard, 200401 (2006)
16. Shah, D., Patel, D.: Dynamic and ubiquitous security architecture for global SOA. In: The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, UBICOMM 2008, pp. 482–487. IEEE, September 2008
17. Jones, M., Bradley, J., Sakimura, N.: JSON web token (JWT) (No. RFC 7519) (2015)
18. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
19. Martins, G., Bhatia, S., Koutsoukos, X., Stouffer, K., Tang, C., Candell, R.: Towards a systematic threat modeling approach for cyberphysical systems. In: Resilience Week (RWS), pp. 1–6. IEEE, August 2015
20. Priya, S.S., Arya, S.S.: Threat modeling for a secured software development. Int. J. Adv. Res. Comput. Sci. **7**(1), 40–48 (2016)

21. IEC/ISO 7498-2: Information Processing Systems - Open Systems Interconnection, 7498-2 (1989)
22. Shostack, A.: Threat Modeling: Designing for Security. Wiley, Hoboken (2014)
23. Sheffer, Y., Holz, R., Saint-Andre, P.: Summarizing known attacks on transport layer security (TLS) and datagram TLS (DTLS) (No. RFC 7457) (2015)
24. Borazjani, P.N.: Security issues in cloud computing. In: Au, M.H.A., Castiglione, A., Choo, K.-K.R., Palmieri, F., Li, K.-C. (eds.) GPC 2017. LNCS, vol. 10232, pp. 800–811. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57186-7_58
25. Gressin, S.: The Equifax Data Breach: What to Do (2017)
26. Behrens, S., Heffner J.: The avalanche application DoS in microservice architectures (2017). https://medium.com/signal-sciences-labs/starting-the-avalanche-application-dos-in-microservice-architectures-4f5eb4730a60. Accessed 20 Jan 2019
27. Open Security Alliance: IT Security Requirements (2017). http://www.opensecurityarchitecture.org/cms/definitions/it_security_requirements. Accessed 15 Jan 2018