МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2018
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2018
Proceedings of the Forty-seventh Spring Conference
of the Union of Bulgarian Mathematicians
Borovets, April 2–6, 2018

SECURITY PATTERNS FOR MICROSERVICE COMMUNICATION*

Tihomir Tenev, Dimitar Birov

Distribute application, used in many cases as cloud solution, has sufficient complexity toward communication. This intricacy implies security problems as eavesdropping and tampering of sensitive information. Many guides contribute with methods on how to determine vulnerable points, and part of them aim to help developers by advising with proper resolution. However, such guides are either rather common or describe different security problems. Based on these findings, we decided to make a step forward in constructing thorough classification of security patterns, which strongly relates to microservice architecture. Our classification consists of four sections as "Communication", "Deployment and host management", "Data Management" and "Accounts and Identity". Here we represent the first section "Communication", which, for more clarity, we split in two subsets: "Inter-process communication" and "Messages". Moreover, we describe briefly each of the selected patterns toward microservice architecture style and then relate them to relevant abbreviation of the STRIDE model and list of security properties. This approach facilitates users in better finding of appropriate pattern depend on their scenario.

1. Introduction. Nowadays security is one of the main concerns in developing distributed system, since it operates, in many cases, with sensitive information. Therefore, working toward mitigating of security gaps, at earlier phase of application constructing, prevents sequential problems. In that context, we made a classification of security patterns [1], which helps in choosing a proper pattern for securing communication between distributed components based on microservice architecture [5]. We split that classification into two subsets, where each of them is categorized so as to simplify the process of choosing the appropriate pattern. The first subset is toward "Inter-process communication" and the second is toward "Messages".

Microservice is an architecture style organized by atomic independent components called *microservices*. Each microservice should be as decoupled and cohesive as possible [5]. These constraints lead to importing at least one connector at each microservice. That architecture style might be used in diverse solutions as server side applications, mobile applications, cloud applications etc., where each of them can be built as a set of components. In this paper, we decided to work toward mitigating security issues with cloud solution.

*2010 Mathematics Subject Classification: 68N01, 68M14, 68M15.
Key words: software architecture, cloud, microservices, security patterns, pattern classification.

The cloud system is a functional part of a distributed system [5] and has the ability to grow with contemporary activities, which rapidly increases its complexity. The National Institute of Standards and Technology (NIST) [3] categorizes the cloud context in the meaning of three service models: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). The significant model, that we consider here, is the Platform as a Service. It provides a platform for deploying customer application, and excludes the support of any hardware assets.

Here, we shall consider only connectors, which bind microservices and to exclude the rest of the connectors, which bind "API Gateway" [5] to microservices. Depending on communication styles, each connector can be either synchronous or asynchronous. We describe them in section 2 and give examples with different message types, which flow through a connector.

Essentially, security patterns [1] are described as a solution for a problem, which occurs frequently in a certain case. Furthermore, each security pattern should take into account certain software constraints to implement its resolution accordingly.

There are many classifications [2], [11], but we could not find any toward microservice architecture. For that reason, we decided to make a step forward to building a security pattern classification based on microservice architecture style. The whole classification consists of "Communication", "Deployment and host management", "Data Management" and "Accounts and Identity". In this paper, we present the first section, which concerns the secure "Communication".

For better understanding, we describe each of the selected patterns toward microservices and then relate them to a relevant threat category of the so called STRIDE model [6]. Here STRIDE stands for *Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege*. We describe these threat categories in Section 3. In addition, to increase the comprehensibility, we relate each of the STRIDE threat categories to corresponding security property as *Authentication*, *Data Integrity*, *Non-repudiation*, *Confidentiality*, *Availability* and *Authorization*. We describe them in details in Section 3.

The paper is structured as follows: Section 2 describes various communications and message types between microservices. Section 3 explains in details how we filter the proper security patterns and how we construct our classification. Section 4 shows two subsets of security patterns, where the first relates to "Inter-process communication", and the second relates to "Messages". Section 5 shows other papers in that matter. Section 6 derives conclusion and place the next steps for future work.

2. Communication between microservices. A significant problem with microservice architecture concerns a large number of connectors within an application. Unlike most of software architectures, where application objects communicate in-memory and share data through pointers or object references, in microservice architecture each microservice runs under a different process. In that context, the communication occurs only via external calls. This in turn creates prerequisites for disclosing of sensitive information.

There are many fashions for building connectors, however, here we represent only two of them: the Representational State Transfer (REST) [7] and Message Bus [4]. The REST is used to bind two microservices in synchronous way with one-to-one connector. Whereas, Message Bus can be used to bind at least two microservices in asynchronous way with one-to-many connector. Both connectors can exchange two data types: human

readable as JSON and XML, and machine readable as binary. We provide more details in the next subsections.

2.1. REST and synchronous communication. The architecture of REST is revealed by Roy Fielding in his PhD dissertation [7]. It describes a resource-oriented architecture style for network systems, which encapsulates a set of constraints for a synchronous communication.

Essentially, synchronous communication has a request/response-based mechanism [8]. A client sends a request and waits for immediate responds afterwards. Moreover, such a communication occurs only between two components. Some authors [9] entitle this communication as Point-to-Point.

Depending on the scenario, some of the connectors can be used to transmit sensitive information. They require special attention and should be treated in different fashion. The Secure Channels [1] pattern helps in solving such a scenario by securing only these connectors, which transmit sensitive information.

2.2. Asynchronous communication. The asynchronous communication mechanism is message-based [8], [9] and represents a broadcasting method with single caller and many listeners. Respectively, a caller does not expect to receive any answer in time. If a listener wants to establish connection with a caller, he firstly should subscribe himself to that event. Once a caller has at least one listener he then can proceed with message broadcasting. There are several open source providers for asynchronous tools, which can be used in combination with microservice: RabbitMQ [25], Apache Kafka [26], and Apache ActiveMQ [27].

The problem with asynchronous connectors comes from the opportunity multiple listeners to bind with one message bus. Thus, some listeners can break confidentiality and disclose sensitive information. The Multiple Secure Observers [18] pattern works against that problem.

2.3. Data types. There are two common data types, used by the microservices to exchange data between one another [10]: human readable (XML and JSON) and machine readable (binary).

XML is a markup language, which consists of set of rules for reading both by human and machine. JSON is similar to XML, but is represented in a lightweight and simplified manner.

The binary flow can be used to transport a data such as files and data broadcasting. Many of the most famous languages embed libraries to allow developers in implementing of these standards easily.

All these data types have metadata, which reveals information about the owner. This in many cases might be used to find the exact location of certain microservice. Hidden Metadata [22] pattern helps to protect metadata against interception.

Knowing more about communication methods implies proper adoption of security patterns. “Inter-Process communication” table (Table 1) shows patterns, which can be used either with Synchronous or with Asynchronous communication, and “Messages” table (Table 2) shows patterns for each of the message types represented above. We describe both tables in Section 3.

3. Classification. By design, Security patterns originate from plenteous experience of developers, engineers and researchers. All policies and methodologies that they offer are used as a template in modelling of a secure system. Moreover, such patterns can be

enforced with microservices in building completed cloud solution.

Each security pattern has its own structure. In many cases, it consists of several sections – *Intent, Context, Problem, Solution, Structure, Implementation, Consequences* and *Known uses*. However, we decided to narrow them to only three points, viz. *Context*, *Solution* and *Known uses*. They mostly reveal the essential of each pattern and contribute to better selecting. *Context* describes the nature of a situation, which includes domain assumption and expectation of a system environment. *Solution* guides a consumer how to solve a problem by providing a decision. *Known uses* gives example on where a certain pattern is already implemented.

After estimating an appropriate reading method, we had to filter patterns suitable from communication perspective. Based on that, we collected two sets of security patterns, which we present in Table 1 and Table 2. The first table deals with "Inter-Process Communication", and the second – with "Messages".

The next step, after filtering the security patterns, is classifying them in threat models approach. Such a combination gives more clarity to a security pattern, which in turns helps in better understanding.

In fact, threat modelling facilitates security experts by estimating vulnerable points. In this regards, binding vulnerable points with the most suitable pattern is a good way for building a secure application without eavesdropping or tampering sensitive information.

After thorough research, we decided to use threat modelling approach based on STRIDE [6]. This approach advises readers to split a software into several components for identifying the types of attacks that they may encounter.

Here is a more detailed description of the six threat categories:

• *Spoofing* is a type of fraud where a violator tries to gain access to a user's system or information by pretending to be the user. For example, when a certain user waits for an event, and the event is triggered by another user with malicious purpose, usually this leads to irrelevant proceeding. The best way to prevent such a threat is to build guard method. Furthermore, that method closely relates to *Authentication* and implies using of security patterns in this regard.

• *Tampering* means making illegal changes of data flow when a user should not. In many cases, tampering leads to adjusting the content of a file, memory unit or data, transferred over network. Nobody wants to receive or read information, which persists with incorrect data. To prevent such a threat, keeping data in consistent state with correct content is a must. Therefore, *Tampering* closely relates to security property called *Data Integrity*.

• *Repudiation* is in meaning of rejecting to accept something. An example of such a situation is when a user did something, but claims he did not touch it. This in many cases can leads to less responsibilities. Such threat can be handled with logging of each activity went over a system. Enforcing *Non-repudiation* related patterns imply preventing of repudiation claims.

• *Information Disclosure* is in situation when an unauthorized user can see information, which is forbidden for disclosing. The source of data might come from a running process, storage or data flow. The best way to mitigate that risk is to enhance the *Confidentiality*.

• *Denial of Service* mostly relates to exhausting of a certain part within a system. This kind of attack works against memory, CPU or data store and mostly leads to software

inaccessibility. Moreover, it may fill up network bandwidth and inflict high degree of time responding. The option to mitigate *Denial of Service* is to look how to increase *Availability.*

• *Elevation of Privilege* means allowing user to execute a command without required access rights for that. Nobody, except Administrator user, should operate with major processes. Otherwise, somebody may bring a system in corruption state. *Authorization* is the property, which gives someone permission to do or to own something and supports preventing *Elevation of Privilege.*

As can be seen, we extend each STRADE category by linking it with a certain security property, which mostly fits in that context. The properties are *Authentication, Data Integrity, Non-repudiation, Confidentiality, Availability* and *Authorization.* Such an approach implies injection of greater perceptiveness in selecting appropriate security pattern and leads to more intuitive classification.

Table 1. Inter-Process Communication

Patterns	Spoofing / Authentication	Tampering / Integrity	Repudiation / Non-repudation	Information Disclosure / Confidentiality	Denial of Service / Availability	Elevation of Privilege / Authorization
Input Guard [19]		x				
Multiple Secure Observers [18]				x		
Output Guard [19]		x				
Secure Channels [1]		x		x		
Secure Communication [17]				x		
Security Association [17]	x	x				
XML Firewall [21]	x	x				
Authoritative Source of Data [23]		x				
Content–independent Processing [24]		x		x		

Table 2. Messages

Patterns	Spoofing / Authentication	Tampering / Integrity	Repudiation / Non-repudation	Information Disclosure / Confidentiality	Denial of Service / Availability	Elevation of Privilege / Authorization
Anonymity Set [22]				x		
Cryptographic Protocol [SR01]				x		
DATA INTEGRITY IN P2P-SYSTEMS [SY10]		x		x		
Hidden Metadata [22]				x		
Layered Encryption [22]				x		
Morphed Representation [22]				x		
XML Encryption Syntax and Processing [21]		x		x		

Part of the patterns run into more than one STRIDE category, however, this can only facilitate the right choice. For example, Secure Channels [1] and Content–independent Processing [24] protect microservice architecture against *Tampering* and *Information Disclosure.*

4. Enforcing Security Patterns. As part of distribution architecture, microservice has several drawbacks: tough deployment process with many microservices, complexity of testing process, external communication between each microservice, etc. We decided to

focus on communication between all microservices. It happens via external calls rather than internal invoking of an object or a method. This is at least a prerequisite for eavesdropping of sensitive information. To solve such threats, we suggest using security patterns.

In next several paragraphs we explain each pattern from Table 1 and Table 2 in regard to securing microservices. Therefore, the ones that come from Table 1 are:

If only an input microservice communication needs to be guarded, the Input Guard security pattern [19] will handle it. Otherwise, the Output Guard [19] does the same, but with exiting information. Both work in way of keeping the *Integrity*

the Secure Channel [1] represents secure manner in data exchanging between two microservices (Microservice 1 and Microservice 2). Only sensitive data should be secured by channel encrypting. Otherwise, in case of transferring non-sensitive data or reducing some overhead, such a channel does not need to be encrypted. Therefore, a Secure Channel is written in regard to preventing two STRIDE aspects *Tampering* and *Information Disclosure*. The Structure perspective is shown in Figure 3.

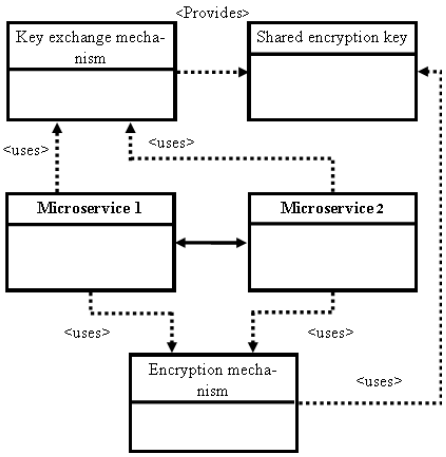

Fig. 3. "Secure Channel" security pattern

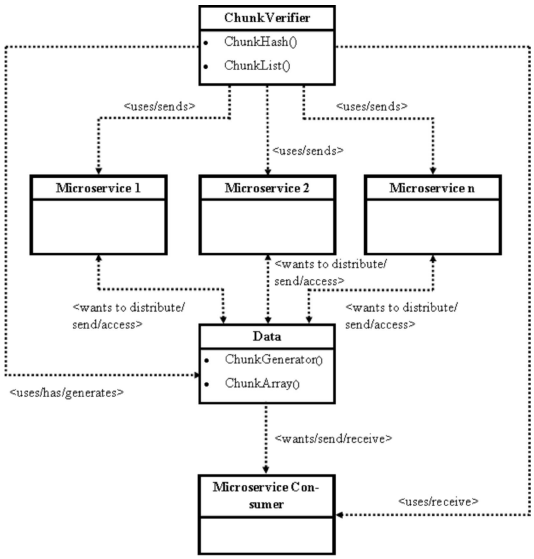

Figure 4 "DATA INTEGRITY IN P2P-SYSTEMS" security pattern

The Secure Communication [17] pattern describes a manner of securing the traffic between two sets of microservices. This pattern does not depend on whether the parties are already secured or not. The main purpose is to increase the *Confidentiality*.

The Security Association [17] might be combined with Secure Communication [17] or another pattern, which has as main task to do encryption of a connector either between two microservices or between two sets of microservices. Such a combination accelerates overloaded communication by excluding re-negotiation process when the encrypted connection is already established. Such a pattern operates toward *Spoofing* and *Tampering*.

The XML Firewall [21] can be applied on a connector, which is used in transferring XML messages. This Firewall intercepts messages and understands whether a content

is applicable for a certain microservice or a rejection process should be invoked. This pattern works against *Spoofing* and *Tampering*.

Content–independent Processing [24] has context of splitting transferred data into at least two channels, where each of them handles only one type of data. It works against *Tampering* and *Information Disclosure.*

The Authoritative Source of Data [15] protects a microservice against receiving unexpected messages. This may come in many cases right after deploying a new version of a microservice, which has additional parameters at its message content. Adjusting a content may lead to misunderstanding on the part of the receiver, due to a lack of readiness to consume it. This pattern increases the *Integrity* part of Security properties.

The Multiple Secure Observers [18] pattern works against the problem with asynchronous connector and the ability multiple listeners to bind with one message bus. Some of the listeners can break *Confidentiality* and *Disclose Information*.

The rest patterns concern the securing Messages (Table 2). Mostly they relate to *Information Disclosure* part of STRIDE model, which is toward *Confidentiality*:

The Cryptographic Protocol [20] applies cryptographic algorithms and mechanisms to protect messages transferred between microservices. It concerns the *Confidentiality* increasing.

Encrypting the entire message is not the only thing that facilitates messages. The Hidden Metadata [22] ensures anonymity of data content written in packet headers. For instance, sender and receiver metadata may consist of microservice IP addresses and some identity entries. That information should be consumed only by both parties. Otherwise, the *Confidentiality* might be broken by a malicious user.

DATA INTEGRITY IN P2P-SYSTEMS [23] holds another scenario, where data content is split into several microservices (Microservice 1 to Microservice "n") with limited resources. Applying this security pattern allows for retrieving a data from at least two holders in secure way by preventing *Tampering* attacks and *Information Disclosing*. The structure perspective can be seen in Figure 4.

The XML Encryption Syntax and Processing specification [13] represents a standard, where only sensitive information within a XML should be encrypted, but the rest stays untouched. Encrypting only certain information increases *Integrity* and *Confidentiality*.

The Anonymity Set [22] describes a manner for data protecting by inserting sender's and receiver's metadata within a set of another set of data. This kind of arranging works against *Information Disclosure* by confusing malicious users in finding the right location of a microservice instance.

The Layered Encryption [22] works with several microservices connected in chain. They encrypt and decrypt information in each pair. However, the consumers are only the innermost and the outermost microservices. The rest of them serve as intermediate to increase the *Confidentiality* by eliminating the direct link.

The Morphed Representation [22] has similar logic as the Layered Encryption [22], but it doesn‘t encrypt information and uses only one microservice as mediator just to eliminate direct access. This pattern can be used in low latency scenario to protect against *Information Disclosure*.

5. Related Works. Hafiz et al [2] show a similar method, however, their categorization presents an approach, which follows "one pattern per one STRIDE point". Here it is not the same, because a pattern may participate in more than one STRIDE point.

Furthermore, current collection is split into two directions, which match "Inter-process communication" and "Messages" constrains toward microservices.

In [12], Roman Malisetti reviews a secure method, which relies on REST communication. The patterns he enforces are: Transport level security (TLS/SSL), which provides secure peer-to-peer authentication; OAuth, which enables consumers to access services through UI API, without using service credentials; Token-based authentication, which can be applied with OAuth together and can be used for exposing services over REST or SOAP. However, authentication and authorization methods are out of scope in our paper.

Similar to [12], Anivella Sudhakar [14] represents techniques for REST securing. He describes differences between securing of REST and HTTP. Such mechanisms are: HTTP Authentication Schemes, which consist of Basic Authentication Scheme and Digest Authentication Scheme; Token-Based Authentication, which supports the authenticating of REST services; Transport Layer Security (TLS) and Secure Socket Layer (SSL); OAuth and OpenID. His paper shows patterns for Authentication and Authorization, which don't relate to the current work.

The focus in [15] is mainly against securing of web applications. Its author distinguish security patterns in two directions: procedural and structural. Structural patterns can be applied in an already completed product, while procedures are aimed in phase of planning and writing software. Many of the listed patterns do not match our paper. However, "Authoritative Source of Data" is borrowed to enrich "Inter-process communication" table.

Fern et al [16] consider only three security patterns: Authorization, Role-Based Access Control, and Multilevel Security. They argue that these are the only three basic patterns that can be applied at each level of the entire system. Unfortunately, there are many scenarios, which require specific patterns. For example, the Hidden Metadata [22] pattern ensures anonymity data content written in packet headers.

6. Conclusion. There are many publications, providing approaches to how to enhance security [2], however, the current paper is slightly different. It aims to show how security patterns can help in developing secure cloud system, since that system can be built via microservice architecture style.

We reveal two points of cloud systems, which cover communication between microservices. The first point is "Inter-process communication", which describes two connector types as synchronous and asynchronous. The second is "Messages", which describes the types of transported data within a connector. This division helps in better associating of security patterns.

Our next step is to look for an appropriate way of reading security patterns. Initially, they consist of several sections and we decided to use only three of them, viz. *Context*, *Solution* and *Known uses*.

After estimating the appropriate reading method, we filtered two sets of security patterns, which closely relate to each of both communications points. Additionally, we categorize them toward the threat model named STRIDE. This in turn led to a more intuitive classification.

For simplicity, we associated each category of STRIDE with six security properties: *Authentication, Data Integrity, Non-repudiation, Confidentiality, Availability* and *Authorization*. This implies understandability of the current approach and facilitates developers

in choosing appropriate security pattern in building cloud decision. Applying at least one pattern from each of both tables enhance the security.

Further works will deal with research of other security patterns, which match the rest aspects of the security pattern classification lean on Microservice architecture: "Deployment and host management", "Data Management" and "Accounts and Identity". The entire classification will help in enhancing the security at earlier phase of building a cloud decision based on microservice architecture style.

REFERENCES

[1] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad. Security Patterns Integrating Security and Systems Engineering, 2006.

[2] M. Hafiz, P. Adamczyk, R. Johnson. Growing a pattern language (for security). Onward!, 2012, 139–158.

[3] P. Mell, T. Grance, The NIST Definition of Cloud Computing, Special Publication 800–145.

[4] V. Emeakaroha, P. Healy, K. Fatema, J. Morrison. Cloud Interoperability via Message Bus and Monitoring Integration. Workshop on Dependability and Interoperability in Heterogeneous Clouds (DIHC13), At Aachen, Germany, 2013.

[5] M. Richard. Software Architecture Patterns. O'Reilly Media Inc., 2015, 27–35.

[6] A. Shostack. Threat Modeling: Designing for Security 1st Edition, WILEY, 2014.

[7] R. Fielding. Representational State Transfer (REST). 2000, Chapter 5.

[8] C. Richardson, F. Smith. Microservices: From Design to Deployment. NGINX Inc., 2016.

[9] G. Hohpe, B. Woolf. Enterprise Integration Patterns, 2003.

[10] S. Newman. Buiding Microservices, NGiNX, 2015.

[11] A. Motii, B. Hamid, A. Lanusse, J. Bruel. Guiding the selection of security patterns based on security requirements and pattern classification. ACM Transactions on EuroPLoP 2015, July 2015.

[12] R. Malisetti. Securing RESTful Services with Token-Based Authentication. *CA Technology Exchange*, **1** (2011), 43–48.

[13] W3C, XML Encryption Syntax and Processing, 10 December 2002, `http://www.w3.org/TR/xmlenc-core/`

[14] A. Sudhakar. Techniques for Securing REST. *CA Technology Exchange*, **1** (2011), 32–40.

[15] S. Romanowsky. Security Design Patterns Part 1, Morgan Stanley Online, September 2001.

[16] E. Fern, R. Pan. A pattern language for security models. PLoP 2001 Conference, 2001.

[17] B. Blakley. Heath, and members of The Open Group Security Forum. Security Design Patterns. The Open Group, Apr. 2004.

[18] V. Gondi. Multiple secure observers using j2ee. Proceedings of the Conference on Pattern Languages of Programs, 2010, 1–13.

[19] T. Saridakis. Design patterns for fault containment. Proceedings of the European Conference on Pattern Languages of Programs. UVK – Universitaetsverlag Konstanz, 2003, 493–520.

[20] M. Schumacher, U. Roedig. Security engineering with patterns. Proceedings of the Conference on Pattern Languages of Programs, 2001, 1–17.

[21] S. R. Nelly Delessy-Gassant, E. B. Fernandez, M. M. Larrondo-Petrie. Patterns for application firewalls. Proceedings of the Conference on Pattern Languages of Programs, 2004, 1–19.
[22] M. Hafiz. A collection of privacy design patterns. Proceedings of the Conference on Pattern Languages of Programs, New York, NY, USA, ACM, 2006, 1–13.
[23] B. Schleinzer, N. Yoshioka. A security pattern for data integrity in p2p systems. Proceedings of the Conference on Pattern Languages of Programs, Oct. 2010.
[24] M. Hafiz, R. E. Johnson, R. Af. The security architecture of qmail. Proceedings of the Conference on Pattern Languages of Programs, 2004, 1–9.
[25] https://www.rabbitmq.com/
[26] https://kafka.apache.org/
[27] http://activemq.apache.org/

Tihomir Tenev
e-mail: tenevtih@gmail.com
Dimitar Birov
e-mail: birov@fmi.uni-sofia.bg
Faculty of Mathematics and Informatics
Sofia University "St. Kliment Ohridski"
5, J. Bourchier Blvd
1164 Sofia, Bulgaria

МОДЕЛИ ЗА СИГУРНОСТ ЗА МИКРОСЪРВИСНА КОМУНИКАЦИЯ

Тихомир Тенев, Димитър Биров

Дистрибутивно приложение, което се използва в много от случаите като *облачно решение*, има сложност по отношение на комуникацията. Тази сложност предполага проблеми със сигурността като подслушване и управление на конфиденциална информация. Има много ръководства, които предлагат методи, с които може да се определи застрашена точка и начини за справяне с нея. Въпреки това, такива ръководства са или твърде общи, или описват различни проблеми със сигурността. Осланяйки се на тези открития, ние решихме да направим стъпка напред в направата на подробна класификация от модели за сигурност, която има тясна взаимовръзка с микросървисната архитектура. Плануваната от нас класификация се състои от четири точки: „Комуникация", „Деплойване* и управление на хостове", „Управление на съхранена информация" и „Акаунт и идентичност". В тази статия ние представяме първата точка „Комуникация", която за по-голяма яснота сме разделили на две подточки: „Вътре-процесна комуникация" и „Съобщения". Освен това ние даваме пример за всеки един от подбраните модели за сигурност къде по-точно може да се използва в една микросървисна архитектура. След това го свързваме със съответната категория от метода за моделиране на заплахите STRIDE и допълнителен списък от свойства на сигурността. Този подход подпомага читателите по-лесно да намерят подходящия за тях модел за сигурност.

*В IT-контекст *deployment* обхваща процесите, чрез които нов софтуер или хардуер бива приведен в правилно работещо състояние в дадено обкръжение, включващи инсталация, конфигурация, пробно пускане и правене на необходимите промени (бел. ред.).