Supplementary Material

For lecture on 9.1. and labs on 11./12.1. + 18./19.1.

The next two lab sessions will be part of a scientific experiment in which we want to analyse if you're able to perform software security assessment tasks more effectively if you are provided dataflow diagrams of the examined applications. The lecture on Jan. 9th will prepare you for the tasks. This supplementary material covers the content from the lecture that is directly relevant for the labs. You can use it to prepare for the lecture and the experiment and also during the experiment, to look up background information about the tasks you will perform.

The topics of this document are:

- Microservice architecture
- Dataflow diagrams (DFDs)
- Manual code analysis on GitHub

Microservice architecture

The microservice architecture is an architectural pattern that splits an application's codebase into multiple microservices (or simply *services*). Such microservice applications (or simply *microservices*) have many benefits in the modern, often cloud-based software environments. Development, deployment, and maintenance benefit from this architecture. The opposing software architecture is the monolithic architecture, where the code may be divided in methods, classes, etc. but is shipped as one for the whole software system.

In order to fulfil the same business logic as an equivalent monolithical application, the services in a microservice application have to communicate with each other and share information. For example, if in a social media platform, a single service is responsible for managing the comments each user has written, then it has to provide an interface for other services, where the comments for a specific user can be retrieved. For example, the service displaying the frontend will then retrieve these comments by querying the comments-service.

The communication between services is realized via direct API calls between the services using Restful HTTP, or by using a message broker for asynchronous communication, usually over an untrusted network. The communication needs to be secured. Mechanisms of choice are encryption and authorization/authentication. These are in fact the most pressing concerns when securing microservice applications.

There are a number of typical infrastructural services which are part of most microservice applications:

- Service discovery: a functionality specific for microservices. Due to the separation into microservices and possibility to deploy new instances of single services, the architecture at any given point in time is not necessarily determined at time of implementation. The services have to be designed such that they adopt to the state

of deployed services. Newly deployed instances always register with the service discovery (or *service registry*), which keeps track of deployed instances and provides this information to services that need to adopt to the current state.
- Authorization server: in a microservice application, the services should not trust incoming requests without authenticating them. The authorization server is a central point issuing authorization tokens, which get sent with authenticated requests.
- API Gateway: is a service that serves as single entry-point to the system. The user interacts with the application via the gateway, typically with a website as front-end.
- Monitoring dashboard: as another consequence of the dynamic architecture, a monitoring dashboard is needed to ease maintenance of the running system. It shows for example the experienced load of all services to determine whether new instances should be deployed due to a high load.
- Logging server: in distributed applications, the logs should be collected at a central point. Otherwise, making sense of multiple different logs would be complicated. The logging server receives and collects periodic log messages from all other services.
- Message broker: serves as intermediary between communicating services. Messages get stored in queues, so that receiving services can retrieve them at their convenience (asynchronous communication). Most technologies also offer distribution policies which allow to simplify some communications (e.g. when multiple receivers listen to the same queue instead of the sender sending the same message to each of them separately).

Dataflow Diagrams

Dataflow diagrams (DFDs) are directed graphs that model a software system with nodes and edges. As the name implies, the focus lies on the flow of information that gets shared between system components. Microservices lend themselves to be depicted with DFDs, as they naturally provide a separation into system components and they make extensive use of sharing of information (see above).

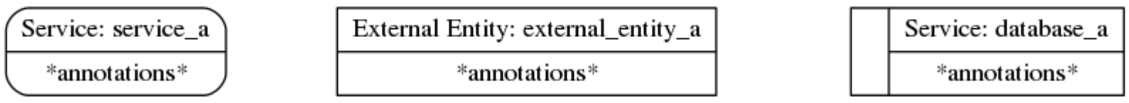
In our DFDs, there are five different item groups:

- Services (rectangles with rounded corners)
- External entities (rectangles)
- Databases (rectangles with additional vertical line on the left)
- Information flows (arrows between two nodes)
- Annotations (*stereotypes* and *tagged values* on nodes and edges)

The three kinds of nodes (services, databases, and external entities) depict the building blocks an application is made of. *Services* are the microservices of a microservice application. Their code is found in the analysed codebase and they serve a single business functionality. *External entities* are different in that their implementation is not found in the analysed code. They are deployed separately from the application but used by it. Examples are a mail server or external databases. *Databases* are services that are used to store data. They are typically marked differently from other nodes in DFDs.

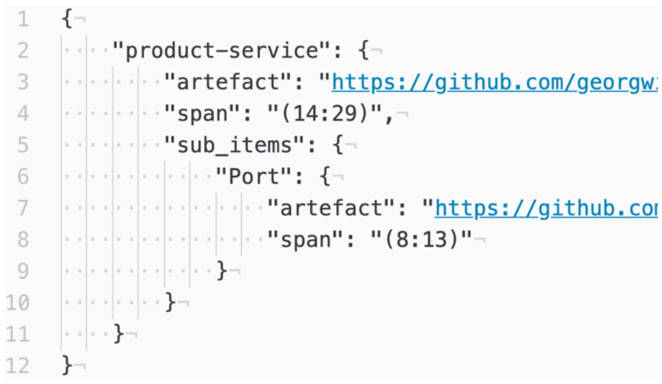
Information flows represent connections between two nodes over which information is sent from one to another. They have an arrow point on one end, depicting which is the sending node (flow is outgoing from this node) and which the receiver (flow is incoming to this node).

Nodes and edges can further have annotations in the form of *stereotypes* and *tagged values*. Stereotypes reveal some property or the functional purpose of an item, for example the business functionality that a service fulfils. Tagged values are key/value-pairs that give further details which cannot be represented with a single term as a stereotype. For example, the port number of a service is given as tagged value.



For all items in the DFD, there is traceability information provided which shows the place in the code where the evidence for the item was found. These are artifacts like a command, a property in a config file, combinations of multiple artifacts, or other.

For the study, you'll receive the traceability information in a separate file similar to the one shown in the screenshot below. The items are ordered by nodes and information flows. For example, if you want to find the traceability of the *Port* of the *product-service*, you find it in line 6 (in the screenshot below). The following link in line 7 will lead you directly to GitHub and highlight the correct line of code, the span in line 8 tells you, which is the exact portion of the line that proves the existence of the model item.



```
 1  {
 2      "product-service": {
 3          "artefact": "https://github.com/georgw
 4          "span": "(14:29)",
 5          "sub_items": {
 6              "Port": {
 7                  "artefact": "https://github.co
 8                  "span": "(8:13)"
 9              }
10          }
11      }
12  }
```

Manual Code Analysis on GitHub

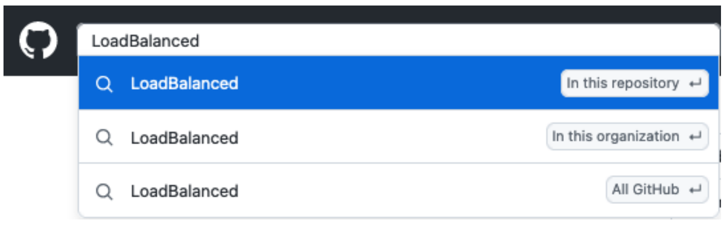
To reason about a software system's security, the underlying code is often analysed manually. Assessors examine the code to determine whether certain threats are mitigated with appropriate security mechanisms and if these are implemented correctly.

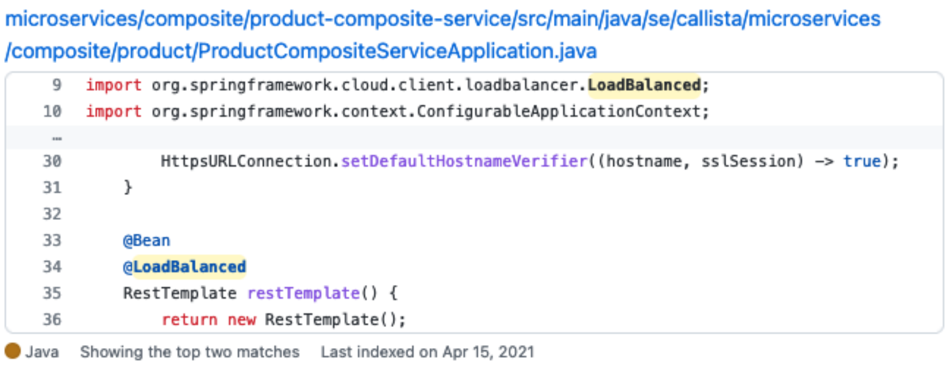
GitHub is a widely popular social coding platform where users can share their code and collaboratively work on it. It is used to distribute open-source software, to manage code in corporations, and for personal coding alike.

Users can create *repositories* for their projects, similar to directories in a file system. A repository is structured as a file system, i.e. all kinds of files can be added and organized with folders. Each repository typically holds a single software system or component of it.

To analyse the code contained in a repository, you can click through the files and inspect the content directly in the browser (at least for the file types relevant for our experiment). After you opened a file, you can use the browser's search functionality (strg + F / command + F) to search for specific keywords in the code.

For repository-wide searches, GitHub offers its own search functionality. In the top left of the website, there's a search bar where you can enter your search term. If you search "in this repository", you'll get all occurrences of that term in the repository you are currently in (see screenshots).




microservices/composite/product-composite-service/src/main/java/se/callista/microservices/composite/product/ProductCompositeServiceApplication.java

```
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.ConfigurableApplicationContext;
…
        HttpsURLConnection.setDefaultHostnameVerifier((hostname, sslSession) -> true);
    }

    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
```

● Java   Showing the top two matches   Last indexed on Apr 15, 2021