Software for recording vulnerabilities in a computer network

Martin Laštoviÿka, Jakub Bartolomej Košuth, Daniel Filakovský,
Antonín Dufka, Martin Husák

2020

Abstract

This document describes the output of the project of the same name ***"Research of tools for assessing the cyber situation and supporting decision-making of CSIRT teams in the protection of critical infrastructure"*** (VI20172020070) addressed in ***the Security Research program of the Czech Republic*** in the years 2017-2020 at Masaryk University. The document contains a description of the result, installation instructions, user documentation and programming documentation.

Contents

Introduction

Detecting vulnerabilities in an organization's infrastructure is an important part of securing it against cyber attacks. This document describes the result No. 1 **Software for the identification of vulnerabilities in the computer network** of the research project ***Research of tools for assessing the cyber situation and supporting the decisions of CSIRT teams in the protection of critical infrastructure.*** This output in the form of software is used to search for information about vulnerabilities and record vulnerable elements of the communication infrastructure. The software approaches vulnerability detection comprehensively on several layers: active and passive network monitoring, processing publicly available information about vulnerabilities, processing data about vulnerable machines from non-profit organizations, and through platforms for sharing events in the security community.

Passive network monitoring based on network flow technology (NetFlow, IPFIX) makes it possible to analyze network traffic passing through measurement points and determine its components without interfering with the network or its elements. An important part of this analysis is the device fingerprinting method, which will provide an inventory of active elements in the network, including their software. In this way, the software is able to determine the operating system of the device, the antivirus installed and the network services provided. In the case of web services, it can determine the CMS (Content Management System) of the web server and the web browser of the client. All this information is complemented by data from periodic active network scanning. Active scans generally have a significantly higher accuracy and level of identification detail, but their coverage (percentage of detected machines and services) is strongly limited by the current state of the monitored devices and firewall settings, preventing their full use in certain types of networks or certain segments of these networks.

By combining an active and passive approach, the software achieves a comprehensive view of the current state of the network and thus enables the search for vulnerable elements.

Information about vulnerabilities is obtained by the software from publicly available sources. These resources can be divided into two groups. The first group are general vulnerability databases, they contain information about vulnerabilities regardless of the manufacturer of the product or the person and company that discovered the vulnerability. Examples of such databases are ***the National Vulnerability Database*** (NVD), ***the Open Sourced Vulnerability Database*** (OSVDB) or ***the Exploit Database .*** [1] [2] [3]
The second group are the databases of information system and application manufacturers who provide information about vulnerabilities in their own products and in third-party products whose functionality their products use. Both groups differ both in the scope of the information provided, as well as in the level of detail and topicality. The software downloads and stores all information about vulnerabilities in a local database, where it is possible to connect vulnerabilities to identified network elements based on the compatibility of the vulnerable software with the software on the given device.

Collaborating organizations are the last significant source of information about network vulnerabilities. The main source of such information is the non-profit organization Shadowserver,

[1] https://nvd.nist.gov/
[2] http://www.osvdb.org/
[3] https://www.exploit-db.com/

which performs periodic vulnerability scans in the entire Internet address space. It then offers the scan outputs free of charge to affected organizations so that they can secure vulnerable devices.
The second important source of information is the platform for sharing events developed in the project ***VI20162019029 Security information sharing and analysis.*** Using this platform, participating organizations share information about current cybersecurity events and discovered vulnerabilities. The software stores information from both sources in a central database for recording and searching for vulnerable devices.

The output of the software is a list of active elements in the network with the identification of vulnerable and potentially vulnerable elements. These outputs will generally be usable as a separate system for recording vulnerabilities in the computer network and it is possible to access them programmatically through the central database or using the graphical interface of this database with the possibility of searching for elements and browsing the network status. As part of the project, the outputs of the created software No. 1 serve as input for the software for the visualization of the security situation in the network (Result No. 2).

Technical parameters of the result The
software for recording vulnerabilities in a computer network connects various data sources providing information about vulnerabilities, and by using them, it determines which KII elements are threatened by a given vulnerability. KII administrators will not have to actively monitor vulnerability reports and manually search for vulnerable elements of the infrastructure, but will receive an automatically generated list of active elements in the network with the identification of vulnerable or potentially vulnerable elements. The advantage of this approach is the ability to detect system vulnerabilities that are not directly under the control of the security team, but the team has the ability to monitor the network traffic of a subordinate or own organization.

The software is distributed as open-source under the MIT license, the owner of the result is Masaryk University, IÿO 00216224: Contact person:

RNDr. Martin Laštoviÿka Department of Computing Masaryk
University Šumavská 416/15, Brno 602 00 e-mail:
lastovicka@ics.muni.cz phone: +420
549 49 6477

Economic parameters of the result The market
segment is represented by organizations that operate or build their own CERT/CSIRT, Security Operation Centre, or mediate cyber security as a service.
The result allows users and operators of networks and services to get a quick overview of the current state of the protected network, especially the occurrence of vulnerabilities in the network. The software enables users to navigate a security incident more quickly and provides the necessary data for deciding on the prioritization of incidents and their resolution. The result thus brings a saving in the time of solving cyber security problems, which reduces the demands on human resources, or allows users to handle more tasks at the same time. The result is thus a shorter response time to a cyber security incident, thereby contributing to a general increase in the level of cyber security in the organization. Use of the result is licensed free of charge.

Description of the result

The architecture of the system for decision support in the protection of critical information infrastructures is based on the concept of the OODA cycle. It was originally designed to support the decision-making of military pilots, but the cycle has seen application in many other areas as well.
A variation of the OODA cycle for decision support in the protection of computer networks can be seen in Figure 1. In the individual phases of the OODA cycle, we can see the phase of gathering information (Observe), visualization (Orient), choosing a defensive action depending on the circumstances (Decide) and executing a defensive action (Act). Essential to the OODA cycle is that each phase provides feedback to the Observe phase so that the information is always up to date.

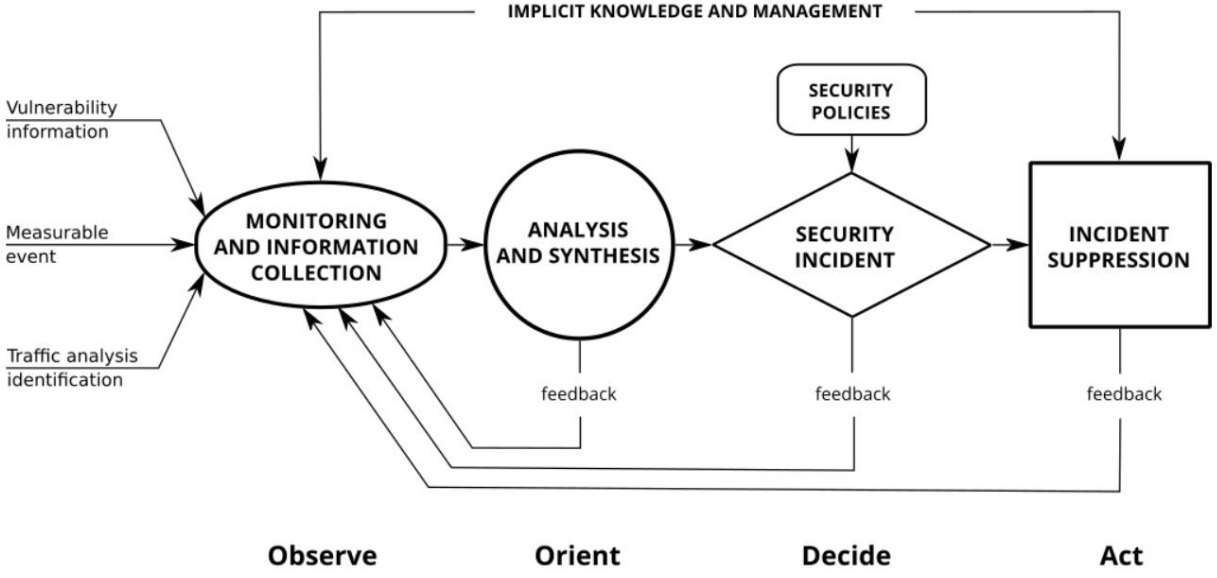

Figure 1: The OODA cycle.

In accordance with the principles of decision-making using OODA and division into individual phases, the system is also divided into four phases, Observe, Orient, Decide and Act. The subsystems of each phase then provide the necessary application logic implementing the described decision-making phases. In the context of the project's software outputs, each software corresponds to one part of the OODA cycle. In Figure No. 2, we can see the overall architecture of the system divided into four phases by colored backgrounds (the image is available in full resolution as an attachment to this document). Shaded in blue are the systems of the Observe phase, which collect data and store them in the database. Shaded in green is the system implementing the Orient phase, especially for the visualization of the acquired data. In yellow is the decision support system in the Decide phase, and in red is the active defense control system falling under the Act phase.

[4] KOTT, Alexander; WANG, Cliff; ERBACHER, Robert F. (ed.). ***Cyber defense and situational awareness.*** Springer, 2015.

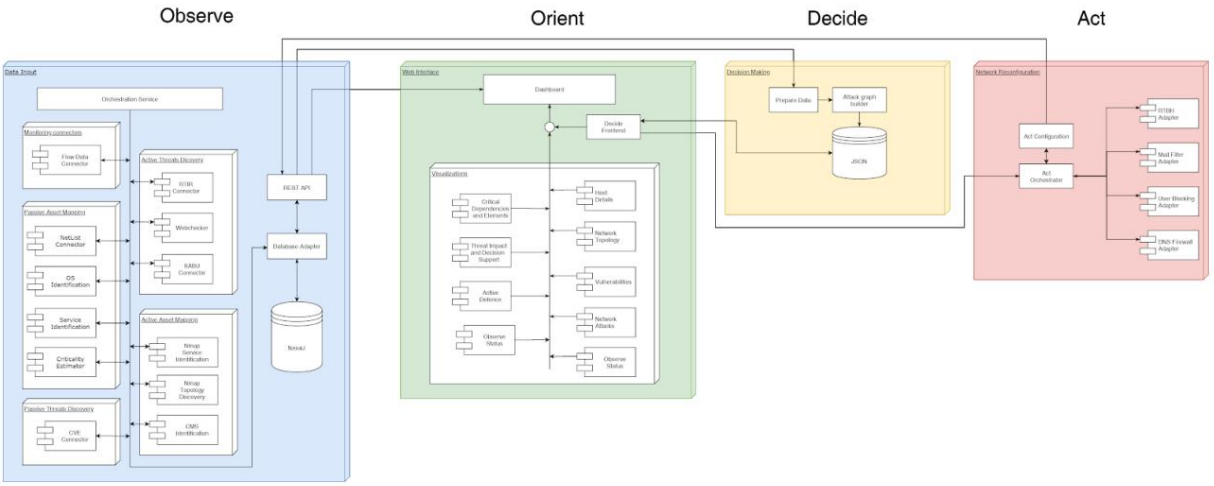

Figure 2: System architecture.

Result No. 1 **The software for recording vulnerabilities in a computer network** described in this document corresponds to the Observe phase of the OODA cycle. The software is designed in a modular way, so that the individual components are as independent as possible, which increases the robustness of the system in the event of a service failure and enables the software functionality to be adjusted according to the requirements of the organization in which it is deployed. The system enables the disconnection of components that the organization does not use, as well as easy expansion with new organization-specific components. Figure 3 provides a detailed view of the software architecture.

Software components are divided into two basic categories - supporting technologies and data connectors. Supporting technologies are based on software used for synchronization and unification of connector communication and for persistent storage of their results in a database.
To integrate the software into the context of other project outputs, or other external systems, it provides a REST API over the central database. Data connectors are separate units.

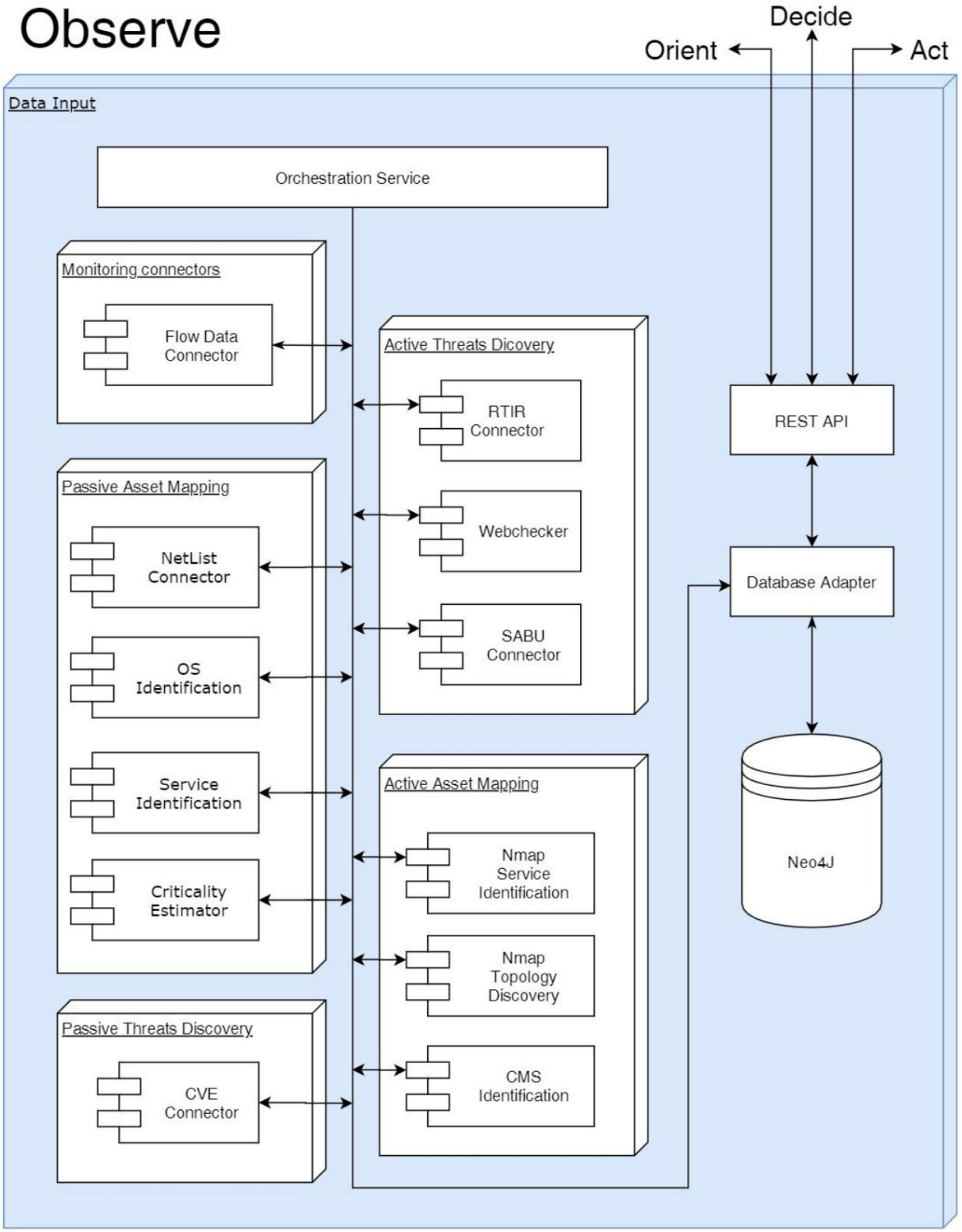

Figure 3: Detail of the architecture of Result #1

Assistive Technology

Supporting technologies are the basic building block of software. The software supporting technologies include the following components. The orchestration service ensures the management of the computing cluster and the regular startup of all data connectors. A central database provides persistent storage of connector outputs. A REST API ensuring external communication is exposed above the database. The database adapter then provides programmatic access to the database and reduces dependence on a specific technological implementation of the database itself.
All assistive technology components are described in detail in this chapter.

Orchestration service

The orchestration service is responsible for the correct execution of the individual components within the software for recording vulnerabilities, which are referred to as tasks. Each task has a precisely defined role in the system, i.e. when and with what parameters it should be started. The records of the execution of each task are recorded in the monitoring system, and it is thus possible to monitor whether the task ended without errors, the duration of the task, its output and possibly a list of errors. In addition, each task has its own logger that records the progress of the component launch in the file ***/var/log/crusoe/<component-name>.log*** for easier detection of possible errors.
The orchestration service also ensures that the results of individual tasks are stored in the database. The orchestration service itself does not contain any logic executed within individual components, it uses their public interface and mediates their mutual communication, therefore it is dependent on all CRUSOE components.

In addition to the basic control of the running of software components, the orchestration service can also be used to manage a computing cluster when deployed in a distributed environment. This is achieved using a common master-worker architecture, i.e. one main machine coordinating the activity and one or more computing nodes that perform specified tasks. When deployed on only one machine, that machine fulfills both roles. For some software components, however, distributed deployment is advantageous and allows them to gain higher visibility in the network, e.g. a component for active network scanning can thus use more monitoring points in the network and its results are thus less distorted by firewalls between individual network segments.

Task list

The list of tasks (tasks) is determined by the configuration file ***celery_config.py*** and it is possible to modify and adapt it according to one's own requirements, or it is possible to add new tasks.
In the basic configuration, the task list contains all software components including their configuration. These components are described in detail in the following chapters of this report.

Technologies used

The orchestration service is largely dependent on other technologies that run within it. Particulary speaking about:

- Celery – å separate scheduler that executes individual tasks according to certain rules, using a specified period or a precisely defined time.

[5] http://www.celeryproject.org/

• Flower - monitoring system designed specifically for Celery. Keeps a history of all tasks since the last restart of the Orchestration service. It contains information such as task name, run time, duration, the result that the component returned, or error messages with the full path where the error occurred. • Redis - performs the function of a database and stores information about the current status and outputs processed tasks.

REST API

The REST API component serves to facilitate communication with other software developed within the project and enables the software to be connected to any system supporting communication via REST. Basic API endpoints are designed to provide a set of CRUD (Create, Read, Update, Delete), i.e. creation, reading, updating and deletion of objects in the central database. These objects correspond to the data model of the project, which was described in detail as a publication output of the project. The REST API implementation is based on the Django open source web application framework, which provides API endpoint exposure and request processing

Database adapter

This component is an implementation of the adapter design pattern, which facilitates the interconnection of various system components. Thanks to the database adapter, the software data connectors are not dependent on a specific database technology, which makes it possible to simply replace the database system according to the needs of the organization operating this software. The adapter itself then contains program calls for individual connectors, which are converted into queries in the language of the given database system.

Central database

The database serves as a central data repository for the entire decision support system, but from an architectural point of view it falls under the Observe phase, which provides data collection. Systems implementing the other phases of the OODA cycle access the database as a data source, the tools of the Observe phase are therefore responsible for the up-to-dateness and consistency of the data in the database.

The Neo4j database, which belongs to the category of so-called graph databases, was chosen for the implementation of the system. Unlike classic relational databases, graph databases make it easier to model entities and relationships in computer networks with regard to extensibility. Thanks to the graph database, it is possible to easily add new types of entities and relationships between them, which simplifies both experimental development and deployment in practice, when it is possible to continuously react to the possibility of supplying new types of input data. It is thus possible to add new types of nodes (entities) and edges (sessions) to the database on the fly without the need to intervene in the existing data model or stored data, as is normally the case with relational databases.

---

[6] https://flower.readthedocs.io/en/latest/

[7] https://redis.io/

8 KOMÁRKOVÁ, Jana, Martin HUSÁK, Martin LAŠTOVICKA and Daniel TOVARŷÁK. CRUSOE: A Data Model for Cyber Situation Awareness. In Proceedings of the 13th International Conference on Availability, Reliability and Security. Hamburg: ACM, 2018. pp. "36:1"-"36:10", 10 pp. ISBN 978-1-4503-6448-5. doi:10.1145/3230833.3232798.

Data connectors Data connectors are the implementation of partial parts of software functionality. Each connector is responsible for obtaining input data, processing it and then saving the results of its activity. Thanks to the architecture design and supporting technologies, connectors are completely independent of the implementation and operation of other connectors. It is thus entirely up to the software operator which connectors it decides to use, or whether it adds its own connector.

Flowmon collector connector

The component allows access to a collector from Flowmon and then obtaining network flows[9] in the form of IPFIX using SSH or the Flowmon REST API. The obtained network streams are later passed to other components, such as OS-parser-component or Service component. The functionality of accessing data using SSH can also be used for any other IPFIX collector, the only condition is the installed set of open-source ***nfdump*** tools on the given collector.[10]

Active network monitoring component

The Active Network Monitoring component is responsible for actively scanning the network to determine the current network topology, active network elements, and exposed network services. The open-source nmap tool is used for network scanning itself.[11]

Nmap scanner is a component that is used for mapping running network services and identifying their software. Service mapping takes place with the help of a scan of the 100 most frequently used ports, and if an open port is found, this information is further processed. In addition to this basic scan, the component uses Nmap's advanced capabilities to identify software.
Additional requests specific to different types of software are sent to the machine's open ports, and based on the responses, more specific requests are sent until the specific software is identified. The resulting identification is converted into the CPE (Common Platform Enumeration) format, which is used for unique identification and is also used in vulnerability detection components.

Nmap topology scanner is a component that performs network topology mapping using an active traceroute network scan. The connector thus obtains information about the current active connections between individual devices in the network at the L3 level (network layer) of the ISO/OSI model.
Connections detected in this way can be used, for example, to identify the machine as an active element in the network or as an end device. The connector performs a TCP traceroute on ports 80 and 443, which are usually enabled on firewalls, and thus the scan covers more of the network than if it used a traditional traceroute using the ICMP protocol. ICMP traceroute is used as a backup option if the target machine is unreachable via the mentioned TCP ports.

Traceroute network scanning reveals the current routing path between the scanning device and all end devices on the network. However, this is insufficient for network mapping

---

[9] https://www.flowmon.com/cs
[10] https://github.com/phaag/nfdump
[11] https://nmap.org/

topology, since such a procedure will naturally only produce a tree structure that does not correspond to real networks containing redundant paths and thus also cycles in the resulting graph. Therefore, it is necessary to run such a scan repeatedly from multiple monitoring points located in different (logical and physical) locations in the network. This achieves the detection of a larger number of lines in the network and the resulting map more closely matches the real topology.

The component consists of three basic parts – the scanning process itself, parsing, and a module responsible for maintaining data consistency in the database, such as closing out-of-date connections.

Scanner mode

The Scanner module was designed to be easy to use in large networks where a complete scan takes a non-trivial amount of time. Therefore, the basis of the module is the possibility of configuring parallel scanning from multiple machines and dividing the entire scanned space into smaller units that can be processed separately and then combine the results into an overall view.

Parser Module

The task of the Parser module is to load the results of the performed scan and store them in a graph database according to the specification of the data model. Since scans are performed repeatedly from multiple different locations, this module must ensure both the addition of newly detected connections and services and the updating of previously detected connections.

Cleaner mode

The Cleaner module solves the problem of scanning in discrete time intervals. The fact that a previously revealed network connection or service was not detected during the scan has two basic explanations. The first is the inactivity of the target device on which the scan is in progress, and thus the missing detection of the last traceroute hop, or services. The second explanation is the actual termination of the given connection or termination of the provision of the network service.

Passive Network Monitoring component

The goal of this component is to analyze network traffic records based on IPFIX technology, in order to detect and identify services running on devices in the network. The output of the component is a display assigning identifiers of detected services to IP addresses.
The component contains subcomponents, each of which is used to detect a different type of service and can combine multiple identification methods within its functionality. Subcomponents are run separately and their outputs are merged into the resulting output of the component.
The functionality of the component is independent of the specific environment. Interactions with the environment take place through a pair of files - input and output. Paths to these files are passed to the component and it will perform its work independently. The input file should contain the IPFIX records to be parsed, and the output file will be written to the component's output in JSON format after processing. The component contains three modules: run, core, rules. Their design and functionality are detailed below.

The anti-virus software identification subcomponent performs identification based on communication with specific domains. Values (domain names) are observed from individual IPFIX records, which are compared with a pre-defined list of rules. When a value matches a rule, the corresponding type of antivirus software is included in the auxiliary data structure to the IP address that produced the given network flow. After processing all input IPFIX records, the auxiliary structure is transformed into the output of the subcomponent so that the most likely antivirus (based on frequency) is selected for each IP address.

The web browser identification subcomponent contains two identification methods.
The first method is communication with specific domains, where values (domain names) are observed from individual IPFIX records and compared with a pre-defined list of rules. The second method is identification from the ***HTTP User-agent entry,*** which contains a text string identifying the web browser.

The last part is ***a sub-component for identifying services in the network*** using machine learning methods. In the learning phase, it is first trained on network traffic containing NBAR2 tags. Using this, a model is created that enables the classification of a network service from communication parameters such as volume characteristics, packet sizes, time distribution or TCP/IP header settings. The method is thus able to determine from normal communication which service caused this communication and to identify the provider of this service.

Module run

This module contains the entry point of the component - the run method. Calling it with a pair (input file, output file) results in the following sequence of operations:

1. loading of input data from the input file, 2. initialization of
individual subcomponents, 3. running of subcomponents
over the input data, 4. output of results to the output file.

Core mode

The core module is the core of a component that uses individual subcomponents to represent its results and enables easy interoperability. The main part of this module is the Result class, which, together with the hierarchy of classes corresponding to the individual used subsections of the CPE chain (vendor:product:version), provides an interface for simple manipulation of findings. The structure of this module is shown in the class diagram below the paragraph.

Rules mode

The rules module can be used for subcomponents or their detection methods that do not use algorithmically more sophisticated techniques. The rules module implements the Rules class, which can process network flows based on predefined rules. These rules can be defined using a configuration file in JSON format.

The rule configuration file has a dictionary structure between the rule names and the rule definition itself. The rule definition is again a dictionary that contains "information keys" and "rule keys". Information keys are used to specify information if a match is found.
Rule keys are optional and define what conditions a flow must meet to match a given rule. Items that are used to find a match using a string are represented by a regular expression, or even a list of them. Items that search for a match using a numerical value can be represented in the format "first..last", possibly also specifying the size of the increment "first, second..last".

Fingerprinting Component of Operating Systems

The issue of fingerprinting of operating systems is currently so interesting for research that its implementation was separated from the general component for passive monitoring of the computer network. The operating system identification component aims to detect and identify the operating system of devices located on the network based on the data contained in IPFIX network flows. The functionality is based on the three methods described in the article ***Passive OS Fingerprinting Methods in the Jungle of Wireless Networks .*** Specifically, these are[12] specific domains, the User-Agent of the web browser, and the default values of the TCP/IP packets sent. The outputs of the individual methods are combined and the finding with the highest frequency for the given IP address is chosen as the result.

In addition, the method using TCP/IP parameters has been extended by the use of machine learning algorithms. A decision tree model was used, which according to the article ***Machine Learning Fingerprinting Methods in Cyber Security Domain: Which one to Use?*** [13] seems suitable for solving this problem.

Module Specific domains

Based on the observation of the automatic behavior of the operating systems (checking internet connection, checking for updates, sending telemetry data), a mapping was created between some specific domain names and the operating systems that normally contact these domains during the execution of their routines. These are, for example, the addresses of update servers, services for checking connectivity, and the like.

Modul HTTP User-Agent

User-Agent is an optional HTTP protocol header into which web browsers insert information about the client software. This header often also contains the name of the operating system or its version, which can then be parsed and used to identify the running OS.

---

[12] Martin Laštoviÿka, Tomáš Jirsík, Pavel ÿeleda, Stanislav Špaÿek and Daniel Filakovský. Passive OS Fingerprinting Methods in the Jungle of Wireless Networks. In NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium. Taipei, Taiwan. 2018. ISBN 978-1-5386-3416-5.

[13] Martin Laštoviÿka, Antonín Dufka a Jana Komárková. Machine Learning Fingerprinting Methods in Cyber Security Domain: Which one to Use? In IEEE. Proceedings of the 14th International Wireless Communications and Mobile Computing Conference. Limassol, Cyprus. 2018. ISBN 978-1-5386-2069-4.

TCP/IP parameters module

Identification based on TCP/IP parameters is based on the dependence of some default values of TCP/IP packets on the implementation of the specific operating system that produced them. The following three parameters were used in the component:

- IP TTL – Time to Live IP field value rounded to the nearest higher power twos,
- TCP SYN Size – the size of the packet initiating the TCP connection in bytes,
- TCP Win Size – the value of the TCP Window field in the TCP header.

CMS detection component

The purpose of the CMS detection component is to obtain information about web servers and content management systems (from the English CMS - Content Management System) of web applications. The component primarily focuses on the identification of CMS systems that are user-friendly and easy to manage, and therefore very widespread. However, due to their diverse functionality and number of plugins, they are prone to vulnerabilities.
Using data from the component, administrators can get a better picture of web applications and their vulnerabilities. Thanks to this, it is possible to more easily detect weak points in the security of the infrastructure.

Webchecker component

The component is used to identify the organization's active websites and to check the validity of the certificates that these websites present to users. To detect active websites, the component uses passive traffic monitoring, where it looks for user accesses to HTTP(S) pages in the captured data. For each page discovered in this way, the component performs data validation and provides information about the hosting of the page on the respective machine with additional information about whether the page is accessible via HTTP or HTTPS. In addition, for all detected HTTPS pages, the component performs an active verification of the validity of the certificate, and any detected problems are also an output of the component, thus providing administrators not only with a global map of the organization's pages, but also helps to identify, for example, unmaintained pages with an expired certificate, which may indicate other security risks.

Vulnerability information acquisition component

The main functionality of the component consists in obtaining information about vulnerable services, services exposed from the internal network to the world, and machines infected by botnets. This information is obtained by the component from the data provided by ShadowServer. [14]
ShadowServer is a non-profit organization that performs periodic vulnerability scans of the entire Internet address space. It then offers the scan outputs free of charge to affected organizations so that they can secure vulnerable devices. To gain access to the data, you need to register and go through a verification process to see if the information can be shared.

[14] https://www.shadowserver.org/wiki/

Currently, ShadowServer monitors 42 types of different vulnerabilities, which include detection of misconfigured services that can be exploited for reflection and amplification of DDoS attacks. It also monitors selected application vulnerabilities, especially web services.
The last category of vulnerabilities is remote access detection. This in itself does not represent a risk, but in many cases remote access to the device is turned on from the factory without the knowledge of the administrator (e.g. network printers, cameras, etc.) or uses outdated technology (e.g. telnet protocol). For these reasons, it is advisable to include this information in the overview of the organization's status. A special case of ShadowServer's detections is the detection of malware infected machines. ShadowServer cooperates with security forces in Europe, and in the case of discovery of the control center of the botnet, this center is eliminated and a detector is placed in its place. Infected devices thus try to connect to this detector and their activity is captured.

The component automatically downloads the above-described data relevant to the given organization and stores it in a central database for further processing.

CVE Connector

The CVE connector is used to obtain information about vulnerabilities in the form of CVE. It consists of two parts. The first of them ensures the acquisition of data from the US national NVD database, and the second obtains information from individual software manufacturers. All received descriptions of vulnerabilities are further classified according to the taxonomy created in the project.

NVD module

The NVD module obtains data from so-called Data Feeds, which NVD publishes on its website (in human-readable form). The component itself downloads data from https://nvd.nist.gov/feeds/json/cve/1.1/nvdcve-1.1-YYYY.json.zip, where the string YYYY represents a specific year number, e.g. 2018. The downloaded files are then saved and decompressed.

CVE Vendor Mode

The CVE connector also receives information about vulnerabilities from eight software manufacturers - Adobe, Android, Apple, Cisco, Lenovo, Microsoft, Oracle and RedHat. The information obtained in this way is used to supplement data from the NVD database, which cannot be obtained from this database. It is thus possible to add the following data to the already created CVE node in the graph database:

- a description of the vulnerability from the manufacturer, which is often much more detailed than in NVD, • information about the availability of a patch or software updates addressing the given vulnerability,
- the date of publication of the vulnerability by the manufacturer, if it differs from the date of publication in the NVD.

As some vulnerabilities are published in the NVD only after they are published by the manufacturer, the vulnerabilities from the manufacturers are obtained 14 days back in order to map the data from the manufacturers to the vulnerabilities from the NVD.

[15] https://cve.mitre.org/
[16] https://nvd.nist.gov/
[17] https://nvd.nist.gov/vuln/data-feeds

CVE classifier

Based on the properties of the given CVE (specifically according to the CVSS, CPE and description), the module finds out what impact exploiting the vulnerability can have on the system. As a result, the classifier returns impact according to the following taxonomy:

• arbitrary code execution as root/administrator/system, • gain root/system/ administrator privileges on system, • privilege escalation on system, • gain user privileges on system, • arbitrary code execution as user of application, • gain privileges on application, • system integrity/availability/confidentiality loss, • application integrity/availability/confidentiality loss, • communication integrity/availability/confidentiality loss.

These categories are not exclusive, and a single vulnerability can have multiple impacts on a system.
The categories were created to match the NVD categories, but more granularity was added to get a better picture of real-world impacts. Each category corresponds to the attacker gaining certain privileges, or the attacker's ability to execute commands on the system or damage the system. The classification of vulnerabilities and a detailed description of the categories was published in the article ***Community Based Platform for Vulnerability Categorization .*** [18]

SOAP connector

The SABU connector is used to receive information from the SABU platform. SABU[19] is a platform for sharing security events operated by the CESNET association, which mainly involves academic networks in the Czech Republic, but also other partners. The partners share not only information about security events, but also information about the occurrence of vulnerabilities, obtained, for example, by scanning networks by one of the partners or obtaining data from third parties. Technically, the SABU connector is the receiving connector of the Warden system of the central node of the SABU platform. [20]

The connector receives messages about the occurrence of vulnerabilities in the protected network, then processes the messages and stores the data in the graphic database of the CRUSOE system.

RTIR Connector

The main functionality of the RTIR connector is the acquisition of data on security incidents.
RTIR is[21] an open-source ticketing system widely used by the security community for incident management. The connector accesses this system using a REST API to retrieve incident data, often in the form of unstructured text. From this primary data, it subsequently extracts information of interest such as devices affected by the incident, time stamps,

---

[18] Jana Komárková, Lukáš Sadlek and Martin Laštoviÿka. Community Based Platform for Vulnerability Categorization. In NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium. Taipei, Taiwan. 2018. ISBN 978-1-5386-3416-5.

[19] https://sabu.cesnet.cz/cs/start

[20] https://warden.cesnet.cz/cs/index

[21] https://bestpractical.com /rtir

the classification of the incident or the person and organization involved in the incident. The processed data is then stored in a central database.

Critical node search component

The component also referred to as ***Criticality estimator*** is responsible for finding the most critical nodes in the network based on their centrality. The term "most critical node" refers to nodes that are important in the graph because they have a great influence on the flow of information in the graph. Such nodes typically serve as connections between different parts of the graph, and in case of their removal, the connection of the graph is violated or the flow of information is significantly restricted.
The connector obtains information by means of algorithms searching for the shortest paths in the graph and by the number of incident edges of individual nodes.

The component works primarily on the organization's network topology, which is maintained in the database using the active network monitoring component. Thanks to the information calculated by this component, we can identify important nodes in the network, get detailed information about the consequences of a potential node failure and prepare a recovery plan.

Algorithm for determining the influence of nodes on the flow of information in a graph

The algorithm is built on the implementation of ***algo.betweenness*** available in the graph algorithm library of the Neo4j database[22]. The principle of the algorithm is to calculate the shortest path between every two nodes of the graph using a breadth-first search. Subsequently, each node is evaluated with a number that corresponds to the number of shortest paths passing through this node. Nodes through which more shortest paths pass thus have a higher score and are considered more important from the point of view of their influence on the flow of information in the graph.

Algorithm for finding the number of incident edges of a node (node popularity)

The algorithm is available in the graph algorithm library of the Neo4j database under the name ***algo.degree .*** The algorithm's iterative process[23] consists of traversing all nodes defined by a Cypher query and simply counting all input and output edges of a given type.
The type of edges to calculate must also be defined in the Cypher query. For each node, the output is a number indicating the number of edges of that type.

Netlist connector

The main task of the Netlist connector is the mapping of IP addresses in the graph database to the relevant network segment. In addition, when mapping IP addresses, it determines the smallest segment of those to which the address can be assigned, thereby enabling the most specific contact for a given address to be found. The final task of the connector is to supply domain name information for each IP address (reverse DNS record) that falls within the organization's network. The Netlist connector is based on the organization's asset management and will thus be domain-specific for each organization. The Netlist connector is therefore more of a sample component illustrating the deployment of the software in a specific organization, where asset information is exported to a machine-readable CSV file.

---

[22] https://neo4j.com/docs/graph-algorithms/current/labs-algorithms/betweenness-centrality/
[23] https://neo4j.com/docs/graph-algorithms/current/labs-algorithms/degree-centrality/

Installation instructions

There are two ways to install Vulnerability Detection Software in a computer network. The first of them is the manual installation of individual tools based on the attached instructions. The second option is to use a pre-prepared Ansible script that automatically deploys the entire software.

Manual installation It is advisable to choose this option in case of installation of a specific tool for recording vulnerabilities in the computer network. For a complete installation, it is recommended to use a ready-made one Ansible package.

Each tool contains, in addition to the source code, a README file that summarizes its use cases, contains information about the dependencies required for the successful operation of the tool, as well as instructions for manual installation of the tool.

Automated installation

Prerequisite

Before starting the actual installation, it is necessary to install support systems that guarantee a trouble-free installation of the Software for the identification of vulnerabilities in the computer network. Below you can find specific versions of support systems that are guaranteed to run smoothly. • Ansible 2.9.10

• Vagrant 2.2.9 • VirtualBox 6.0.24 r139119

It's worth noting that other versions of the support systems mentioned above aren't necessarily problematic, and there's a good chance they won't be.

Preparation

The installation package is available in the **ansible folder.** Before the actual launch, it is necessary to add the tools to configuration data for the individual configuration file **ansible/group_vars/all/vars.** in Specifically, the following is required:

• flowmon_key_path - local path to the SSH key that has access to the FlowMon collector. • flowmon_ssh_passphrase - password to the above SSH key. • warden_ca_path, warden_cert_path, warden_key_path - local paths to the files needed for Warden to run correctly. These files can be obtained at https://warden.cesnet.cz/cs/participation • neo4j_password - Your chosen password for the graph database. • flower_password - Your chosen password for the graphical interface of the orchestration service. • rtir_user, rtir_password - Login data to the RTIR ticketing system. • shadowserver_user, shadowserver_password - Login data to the service Shadowserver.

- shodan_api_key - api key with access to the Shodan api.

Installation

To install the entire package, it is sufficient to enter the **vagrant up** command from the **ansible folder.**

Installation check

By installing, you can verify the availability of the following services in the created environment:
- Neo4j graph database - available on port **TCP/7474**
- Orchestration service - available on port **TCP/5555**
    - In the interface of the orchestration service, it is possible to see tasks that are currently in progress and that have already been completed.

User documentation

Network vulnerability detection software is largely designed to run independently without user interaction. Individual components autonomously collect and process data and store the results of their activity in a central database. Result No. 2 - Web application for visualizing the security situation in a computer network is primarily intended for user interaction and data presentation. However, Result #1 can be run independently and get all its functionality.

Application management

The Orchestration service is used to monitor the running of the Software and its sub-components. Its graphical interface in the form of the Flower system is available on the server with Result #1 on port TCP/5555. Flower provides the user with an overview of the status of individual parts of the computing cluster, statistics of successfully and unsuccessfully completed tasks and the average load of computing nodes. An example of a long-term deployed computing node with the tasks of Result No. 1 can be seen in Figure No. 4.

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@crusoe	Online	6	123088	1198	120686	1200	2.44, 2.1, 1.81

Figure 4: Overview of running and completed jobs on a compute node

The web interface also provides a detailed overview of all executed tasks, including their input parameters and the current status of the task. For completed tasks, information about the output of the task and the time required for its processing are subsequently added. Figures 5 and 6 show examples of successfully completed tasks of the Operating Systems Fingerprinting Component and the Webchecker Component.

Worker	Name	State	args	Result	Started	Runtime
celery@crusoe	crusoe.OS_parse	SUCCESS	(('Processed scan flag: out, number of flows: 793594', '/data/flow/out_202011181140.json'),)	'New: 0, unchanged: 10501, changed: 1666, inactive: 17099 sessions Measurement: python = 26.317745447158813 neo = 62.96205520629883'	2020-11-18 10:45:54.259	89.280

Figure 5: Successfully completed task Operating Systems Fingerprinting Components

Worker	Name	State	args	Result	Started	Runtime
celery@crusoe	crusoe.detect_domains	SUCCESS	(('Processed scan flag: in, number of flows: 882', '/data/flow/in_202011181145.json'),)	'658 domains detected. Measurement: python = 3.030447006225586 neo = 37.21648836135864'	2020-11-18 10:50:07.891	40.248

Figure 6: Successfully completed Webchecker Component task

An example of a failed job can be seen in Figure 7. The interface for such jobs provides a complete overview of the error that led to the failure of the job and ***a stack trace*** to facilitate error tracing. In this sample case, an unhandled exception occurred where the input IPFIX data contained an invalid IP address, probably due to an exporter error.

crusoe.service 166ef72c-4c5b-48b1-ad9b-fe0b44aceaf8

Basic task options

Name	crusoe.service
UUID	166ef72c-4c5b-48b1-ad9b-fe0b44aceaf8
State	FAILURE
args	(('Processed scan flag: out, number of flows: 757041', '/data/flow/out_202011181505.json'),)
kwargs	{}
Result	None

Advanced task options

Received	2020-11-18 15:10:45.045426
Started	2020-11-18 15:10:45.046492
Failed	2020-11-18 15:11:43.604172
Retries	0
Worker	celery@crusoe
Exception	ValueError("'141.60.202.1.23' does not appear to be an IPv4 or IPv6 address")
Timestamp	2020-11-18 15:11:43.604172
Traceback	Traceback (most recent call last): File "/usr/local/lib/python3.7/dist-packages/celery/app/trace.py", line 412, in trace_task

Figure 7: The Passive Network Monitoring Component task failed

Access to software outputs

All Software outputs are stored in a central graphic database, whose web interface is available on TCP/7474 port in the basic settings. This interface allows the user to browse stored data and enter commands in the Cypher query language to perform advanced data analysis and gain a comprehensive view of the data. [24]

To illustrate the possibilities of manual data analysis, we will use the first five use cases specified in the technical report ***Design of the system architecture for the protection of KII based on OODA*** issued in the first year of the project. These use cases correspond to the outputs of Result #1 and can be directly answered with properly formulated questions:

1. Identification of services mapped to specific machines, e.g. ***server A provides a service "mail" a "web"***
   - Cypher dotaz: match (i:IP)-[]-(n:Node)-[]-(h:Host)-[]-(s:NetworkService) where i.address = "256.10.49.121" return i,n,h,s

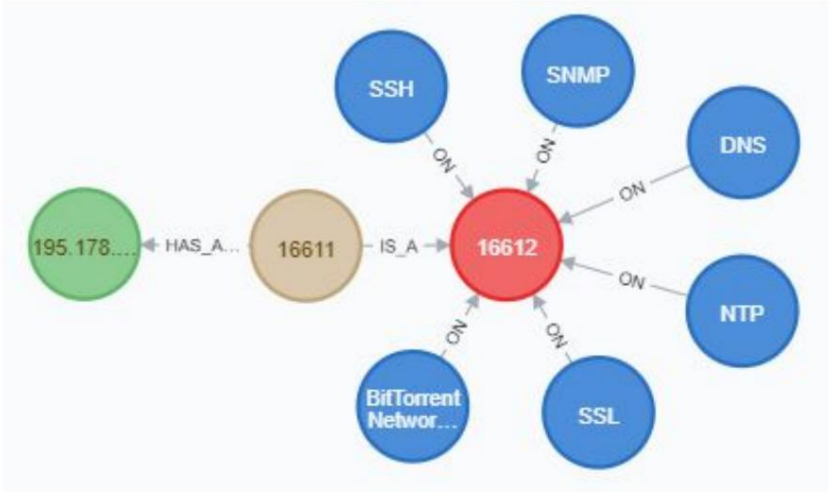

Figure 8: Identification of services running on machine 2.

Machine identification

- determining the OS at the Windows/Mac OS X/Linux level, ideally also with the version

[24] https://neo4j.com/developer/cypher/

- Cypher dotaz: match (i:IP)-[]-(n:Node)-[]-(h:Host)-[]-(s:SoftwareVersion) where i.address = "256.10.49.122" return i,n,h,s

Figure 9: Identification of the device's operating system

3. List of machine vulnerabilities

- Vulnerability detection can be divided into two groups according to their method disclosure. The first group are detections revealed by an active scan aimed directly at the given vulnerability. These include detection from from ShadowServer and from Webchecker Components.
- We can obtain a global overview of the occurrence of newly detected vulnerabilities e.g. with this Cypher query: MATCH (i:IP)-[]-(t:SecurityEvent) where t.detection_time > datetime("2021-01-01T00:00:00Z") RETURN t.type, COUNT(t) order by count(t) desc limit 5

"t.type"	"COUNT(t)"
"cert"	17732
"portmapper"	1477
["Vulnerable.Config","Test"]	1440
"rdp"	160
"mdns"	104

Table 1: Overview of the most common vulnerabilities in the network

- The second group of vulnerabilities includes those detected by the CVE component connector that associates known vulnerabilities with devices based on compliance of vulnerable software with software detected on the device by components passive and active network monitoring.
- We can find out the vulnerabilities of a specific device by asking: MATCH (c:CVE)-[]-(v:Vulnerability)-[]-(s:SoftwareVersion)-[]-(h:Host)-[]-(n:Node)-[]-(i:IP) where i.address = "256.10.49.123" return c,v,s,h,n,i

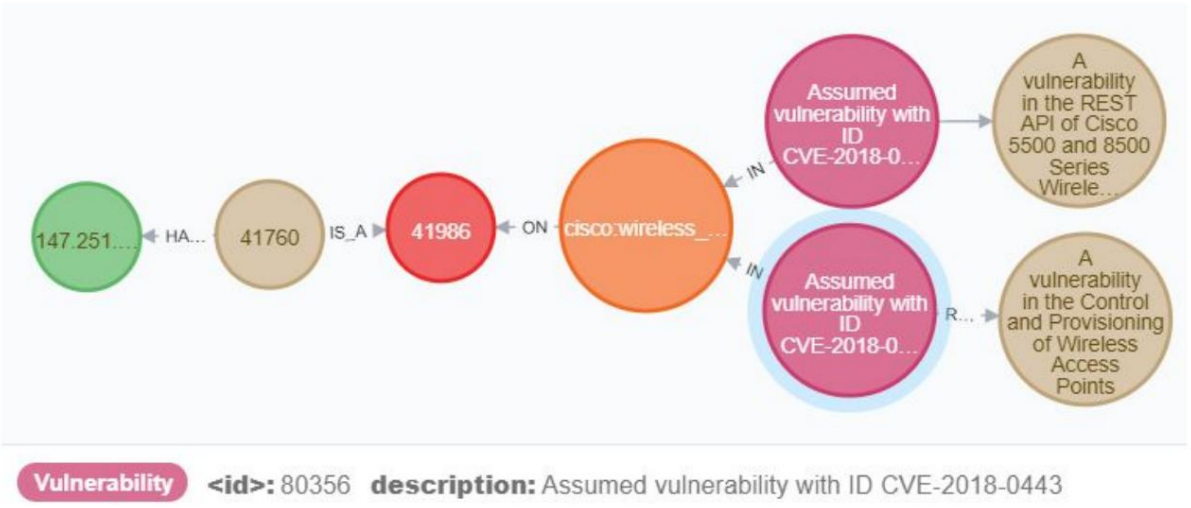

Figure 10: Identification of device vulnerabilities

4. Machine dependencies

• Machine dependencies are identified based on provisioning and usage of services detected by passive network monitoring. In this case, the orientation of the edges is important, indicating which machine is dependent on which. We can get an overview of the dependencies of a specific machine by asking Cypher: match (i:IP)-[]-(n1:Node)-[dep:IS_DEPENDENT_ON]->(n2:Node)-[]-(i2:IP) where i.address = "256.10.49.124" return i,n1,n2,i2

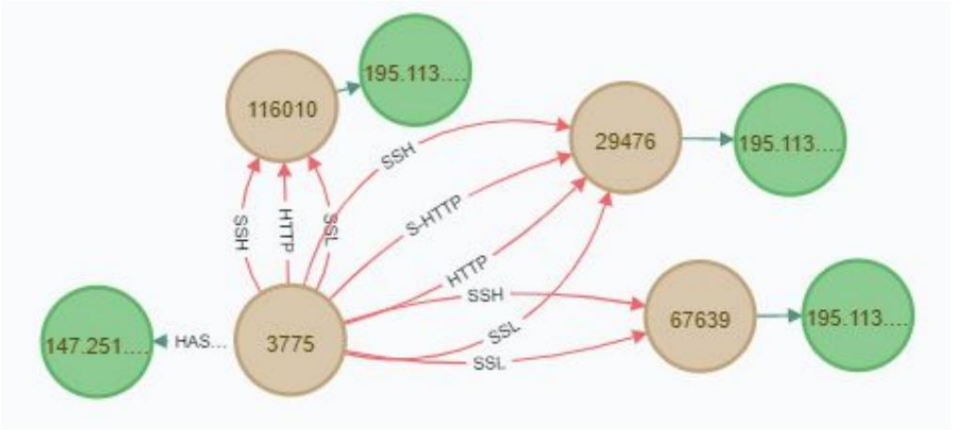

Figure 11: Identifying device dependencies

5. Network Topology •

Displaying the complete topology of a large network is highly impractical. Therefore, it is better to choose a view only of network elements important for the correct operation of the network. Such elements are identified from the overall topology by the Critical Node Search Component, which calculates their significance metrics for each network node. The values of these metrics depend on the size and layout of the network, so the exact values for a Cypher query will vary for each network. • Display the most important elements of the network with the query: MATCH (n1:Node)-[r:IS_CONNECTED_TO]->(n2:Node) where n1.topology_betweenness > 500 and n2.topology_betweenness > 500 and r.hops =1 RETURN n1,r , n2 LIMIT 100

Figure 12: Topology of the most important network elements

Programming documentation

The software source codes are supplemented with documentation of the individual methods, which covers their use and parameters in detail. This document describes the structure of individual components, details of their implementation and use. The structure of the description is the same as the description of the result, i.e. it corresponds to the components according to the proposed software architecture.

Assistive Technology

Orchestration service

The Orchestration Service is organized into six files—namely, two Python files containing the logic of the Orchestration Service, two configuration files specifying its behavior, and two auxiliary files containing documentation and functional dependencies.

filler-orchestration-service

ÿÿÿ celery_config.py ÿÿÿ
celery_logger.py ÿÿÿ
config ÿÿÿ
conf.ini ÿÿÿ
crusoe.py ÿÿÿ
README.md ÿÿÿ
requirements.txt

Within crusoe.py and celery_config.py, a series of tasks are created for each component that have predefined basic parameters with which the component is used.

Orchestrator is implemented in the following files:

- crusoe.py - contains the main orchestrator logic. Its task is to call functions and process their results.
- celery_logger.py - this is the main logger for the orchestration service and for the components running in it. Logger can be set using parameters with required properties for each component separately. This approach allows for centralized logger modifications. Individual components do not need to implement their own logger, as the orchestration service already offers this functionality. In the background, the logger is implemented using the structlog system.
- celery_config.py - the main configuration file of the orchestration service, which defines with which period, or at which specific time, the given tasks should be started. It also contains global orchestration service settings.
- config.ini - this is a configuration file intended for individual components, which have their own specific section in it. It contains information such as authentication, file path and other necessary parameters for the given component.
- requirements.txt - contains a list of required Python packages used in the orchestration service.

Below is a list of canned tasks in the crusoe.py file that can be scheduled to run using a configuration file:

• rtir_connector - task for starting the RTIR connector component • scan_init -
auxiliary task for starting Active Network Monitoring Components • topology_scan - auxiliary task for starting
Active Monitoring Components

networks

• vertical_scan - auxiliary task for starting Active Network Monitoring Components • topology_scan_save -
auxiliary task for starting Active Network Monitoring Components
network monitoring

• vertical_scan_save - helper task for starting the Active component
network monitoring

• attack_graphs - helper task for Software No. 3 • shadowserver -
helper task for running Components of obtaining information about
vulnerabilities

• shodan - helper task to run Components of obtaining information about
vulnerabilities

• cleaner - a task for periodically deleting historical data from the database • sabu - a task for
starting the SABU connector component • act_overseer - an auxiliary task
for Software No. 4 • netlist - a task for starting the Netlist
connector component • nvd_CVEs - an auxiliary task for starting the CVE
component connector • vendor_CVEs - auxiliary task for launching CVE components
connector • OS_parse - task for launching operating system fingerprinting components • service
- task for launching passive network monitoring components • check_certs - task for launching Webchecker
components • detect_domains - auxiliary task for launching the Netlist connector component
• compute_criticality - task to run the Critical Node Search component • flowmon
- task to run the Flowmon Collector Connector component • flowmon_chain - helper task to run the
passive monitoring component

networks

• cms_scan - task for starting the CMS detection component

Usage

First of all, you need to make sure that a broker, such as redis-server, is available, otherwise starting the orchestration service will fail. This can be tested with the command: # redis-cli ping

If we also want to use the Flower monitoring system, we use the following command to start it: # celery flower --
broker = redis: // localhost: 6379/0
Orchestrator (Celery) itself is started with:
celery worker -A /path/to/crusoe -l = INFO -B

REST API

The REST API is structured according to the used Django framework and fully takes over its directory structure.
Individual files and methods also follow the common implementation of this framework.

```
neo4j-rest
    ÿÿÿ django
        ÿÿÿ manage.py
        ÿÿÿ db.sqlite3
        ÿÿÿ static
            ÿÿÿ “default Django file structure”
        ÿÿÿ crusoe_django
            ÿÿÿ admin.py
            ÿÿÿ apps.py
            ÿÿÿ conf.this
            ÿÿÿ models.py
            ÿÿÿ settings.py
            ÿÿÿ tests.py
            ÿÿÿ urls.py
            ÿÿÿ views.py
            ÿÿÿ wsgi.py
            ÿÿÿ middleware
                ÿÿÿ whitelist.py
    ÿÿÿ README.md
```

API endpoints are defined in the views.py file, their complete listing and documentation is available in the appendix of this document.

Use

A server providing a REST API can be started with a command

```
# python3 django/manage.py runserver
```

Database adapter

The database adapter is divided according to the individual components of the software architecture, which provides an interface for manipulating the central database. The component consists of 15 modules providing basic functionality and one supporting module. Support module ***AbstractClient*** is the starting point of the component. Its main function is to create a connection with database, which is subsequently used within the specific modules described below. Except it also contains a set of queries in the native Neo4j query language of the Cypher Query database Language (hereinafter referred to as Cypher), which is used to create restrictions ensuring data integrity in database. Thanks to this, the database itself prevents an attempt to insert erroneous data. Other modules components then supplement ***AbstractClient*** with functionality specific to connectors, which creates an interface to access the database. The resulting structure of the component looks like as follows:

```
neo4j-client
    ÿÿÿ neo4jclient
        ÿÿÿ AbsClient.py
        ÿÿÿ AttackGraphClient.py
        ÿÿÿ CMSClient.py
        ÿÿÿ Cleaner.py
```

ÿÿÿ CriticalityClient.py ÿÿÿ
CveConnectorClient.py ÿÿÿ
MissionAndComponentClient.py ÿÿÿ
NETlistClient.py ÿÿÿ
NmapClient.py ÿÿÿ
OSClient.py ÿÿÿ
RESTClient.py ÿÿÿ
RTIRClient.py ÿÿÿ
SabuConnectorClient.py ÿÿÿ
ServicesClient.py ÿÿÿ
VulnerabilityCompClient.py ÿÿÿ
WebCheckerClient.py ÿÿÿ
README.md ÿÿÿ test/

To verify the functionality of the Neo4j-client component, a set of tests was also written using the pytest framework. By[25] inserting so-called mock data into the test instance of the database, these tests verify that the individual functions of the modules insert data correctly and persistently into the database. The tests for this component are located in the test folder and can be called from the component's root directory.

Use

The Neo4j-client component is primarily used within the Observe phase connectors. To use it in the connector, you need to: 1. Import the required module:

```
# from neo4jclient import <ClientYouWantToUse>
```

2. The second step is to initialize the database connection using the following command:

```
# client =
<ClientYouWantToUse>.<ClientYouWantToUse>(password="Pass")
```

3. The last step is to call the desired method itself. This is possible using of the following command:

```
# client.<MethodYouWantToUse>
```

Central database

The Neo4j database, which belongs to the category of so-called graph databases, was chosen for the implementation of the system. For the software, there is no need to modify the database in any way, just set the firewall and listeners correctly so that the database is accessible to the software components.

Usage

After installing the Neo4j database (e.g. using the supplied Ansible scripts), the graphical interface of the database is available at:

```
# http://localhost:7474
```

[25] https://docs.pytest.org/en/latest/

Data connectors

Flowmon collector connector

The component consists of a single file flowmon_connector.py and provides two methods, ***download_ssh()*** and ***download_rest().*** Both methods fulfill the same functionality. After connecting to the collector, the acquired network data is stored in a predefined file in JSON format.

flowmon-connector

ÿÿÿ flowmon_m/ ÿÿÿ
flowmon_connector.py ÿÿÿ
README.md ÿÿÿ
setup.py ÿÿÿ
requirements.txt

Usage

When downloading less than 10,000 records from a collector, it is recommended to use the REST API, which is provided by the method:

```
>>> flowmon_connector.download_rest()
```

The function mediates communication with the Flowmon REST API, from where the required data is obtained. Through the arguments passed to the method, it is possible to authenticate, set the time window of the obtained network data, set the domain on which the Flowmon REST API is accessible, as well as the data filter. In the case of downloading a larger volume of data, i.e. downloading more than 10,000 records, it is necessary to use the SSH service. The method serves this purpose:

```
>>> flowmon_connector.download_ssh()
```

The SSH service is not primarily intended for obtaining network streams, and therefore, in addition to the parameters mentioned above (filter, authentication, time window), other parameters need to be defined in the REST API method. Specifically, it is the path to the nfdump application and its parameters, which are used to analyze data directly on the collector and to convert it into a readable form. The structure of the files in which the network streams are located and the probes from which the data is to be processed is specified.

Active Network Monitoring component

The component consists of three basic parts – the scanning process itself, parsing, and a module responsible for maintaining data consistency in the database, such as closing out-of-date connections. The project directory has this structure:

nmap_topology_scanner

ÿÿÿ nmap_topology_scanner/ ÿÿÿ
scanner.py ÿÿÿ
README.md ÿÿÿ
setup.py

Scanner contains a ***scan()*** function that handles the scanning process, taking a dictionary of configuration data as an argument. Among other things, this specifies the size and number of subnets to be scanned in one call of this process, or the credentials for the machines to be used for scanning. After the scan of the selected subnets is finished, the results are written to an XML file (in the format of the nmap tool, which is used for the scan itself) on the machine that performed the scan. The scanner module downloads this file to the machine that launched the module and passes it on for further processing.

The results of the performed scan are stored in a graphic database. As scans are performed repeatedly from multiple different locations, this module must ensure both the addition of newly detected connections and the updating of previously detected connections. The component therefore includes functionality for closing non-updated connections.

Use

The scanner can be started as follows:

```
>>> from nmap_topology_scanner import scan >>> scan(subnets, 'nmap
args', logger_instance)
```

The subnets parameter in this case is a list containing the address ranges to be scanned.

Passive Network Monitoring component

The component consists of the following items: services-
component ÿÿÿ src

```
ÿÿÿ core.py ÿÿÿ
rules.py ÿÿÿ
run.py ÿÿÿ
utils.py ÿÿÿ data
ÿÿÿ av.json
    ÿÿÿ si_nbar.json
    ÿÿÿ services ÿÿÿ
antivirus.py ÿÿÿ
    browser.py ÿÿÿ
    service_identifier.py
    ÿÿÿ README.md ÿÿÿ setup.py
```

The ***src/ data*** directory contains additional data files required by the component or subcomponents. ***run.py*** executes the individual subcomponents of the detection service, combines their results, and creates a new result file. ***core.py*** contains an implementation of the ***Result object,*** which contains ***a Hierarchy*** for each monitored device (IP). Object ***Hierarchy*** subpart of CPE identification (Vendor: Product: Version). ***The hierarchy*** contains a tree whose levels consist of ***Vendor, Product ,*** and ***Version objects.*** This structure is designed for

results of individual detection methods and further processing. Each subcomponent should use these classes to simplify interaction with other parts of the component. ***The Result*** class contains the mapping between the software detected on a given IP and the associated ***Hierarchy object. rules.py*** is designed to simplify the identification of usage of network flow logging services based on rules. Provides a rules object that is initialized with a set of preconfigured rules that are then used to compare against flows.

Use

The component's entry point is the run method, which is exported directly to the top level.
First, you need to import the component, and then you can run it directly: >>> import services_component >>> services_component.run(input_file, output_file)

Fingerprinting Component of Operating Systems

The component consists of three modules for basic functionality and three modules that implement individual identification methods. The implementation is located in the ***osrest*** and ***osrest/ method folders.***

OS-parser-component ÿÿÿ
osrest ÿÿÿ
method ÿÿÿ
domain.py
ÿÿÿ tcpml.py
ÿÿÿ useragent.py ÿÿÿ
data ÿÿÿ
os.json ÿÿÿ
num2os.json ÿÿÿ config.ini
ÿÿÿ fingers_map.csv
ÿÿÿ train.csv ÿÿÿ OS_parser.py
ÿÿÿ run.py
ÿÿÿ test
ÿÿÿ test_os.py ÿÿÿ
README.md ÿÿÿ
setup.py

The ***run*** module is the entry point of the component. It contains a ***parse*** method that accepts two arguments - the path to the network stream file to process and the path to the output file.
The ***parse*** method reads network streams from a file, passes them to the ***OS_parser*** module for processing, and then writes the results to an output file.

The ***OS_parser*** module is the main part of the component. It contains free functions that can initialize individual detection methods, run them over network streams, merge their results, and prepare them for output.

Individual detection methods are designed as separate objects that take over the necessary individual settings in the constructor and can then function completely independently using a unified interface. This interface requires the existence of a single ***run*** method that takes the loaded network streams as a parameter and returns its result in the form of a dictionary between IP addresses and mapping OS names to numbers that represent the level of trust in that OS.

The component in the database creates a node of type OS, which carries information about the version of a specific operating system. This node is connected to an IP type node by the HAS_OS edge. It carries attributes affecting the variability of the
environment: • ***start_time*** – timestamp of the detection of the operating system for
the given IP, • ***end_time*** – timestamp of the last detection of the OS for the given IP. If there is no OS change at the IP address, this timestamp is only incremented with each detection. In the case of dynamic IP address allocation, together with start_time, the time period in which the address was used by the device with the given operating system is clearly defined.

The described OS node does not exactly correspond to the data model, and therefore a method of mapping the fingerprinting outputs to clearly defined CPEs and thus corresponding to ***the Software Resource*** entity of the data model will be developed in the following year.

Use

The component's entry point is the parse method of the ***run module.*** First you need to import the component and then call the ***parse()*** method with the input and output file paths:

```
>>> from osrest import run >>>
run.parse(flow_path, output_path)
```

CMS detection component

The entire logic of the component is contained in one ***scanner.py*** script and an auxiliary text file in JSON format. This file contains information about the Common Platform Enumeration (CPE) and their versions so that the component is compatible with the data model and other components. The structure of the component can be represented by the following diagram:

```
CMS-component
    ÿÿÿ cmsscan
        ÿÿÿ data
            ÿÿÿ cms.json ÿÿÿ
        scanner
            ÿÿÿ scanner.py
    ÿÿÿ README.md
    ÿÿÿ setup.py
```

run() is the base method for running the entire module. Its task is to load information about the CPE from the configuration file and then identify the CMS on devices in the network. This is achieved by , which by running the WhatWeb tool, it [26] actively scans the network and returns the results. ***The parse()*** method se takes care of the processing of the scan outputs and the mapping of individual names and CMS versions to

[26] https://github.com/urbanadventurer/WhatWeb

standard CPE format. The output of the method is structured information about individual CMSs in the network in CPE format.

The cms.json file contains information about the services that the component detects. The file contains the text name of the service and its version, which are then used to map input data to output data.

Requirements

To successfully run the component, you need to have the WhatWeb tool installed. It is freely available at https://github.com/urbanadventurer/WhatWeb.

Use

The component can be run using the method: >>>

```
cmsscan.run(whatweb_path, hosts, extra_params="-q", out_path, cpe_path="",
logger=structlog.get_logger(), is_file=True)
```

The method accepts the following parameters as input:

- whatweb_path - path to the WhatWeb tool. • hosts - a list of websites that the component should scan. • extra_params - optional parameters of the WhatWeb tool. • out_path - path to the file in which the output of the component will be written. • cpe_path - path to the file with data about the services that the component detects. At if the parameter is not filled in, the default cms.json file will be used .
- logger - logger instance for recording the state of the component and its running. • is_file - if the list of sites to be scanned is loaded from a file, this parameter is set to ***True*** and ***the hosts*** parameter contains the path to this file.

Webchecker component

The structure of the component is very simple. It contains one main script with all the functionality. The structure looks like this:

```
webchecker-component ÿÿÿ
    src
        ÿÿÿ core.py
    ÿÿÿ README.md ÿÿÿ
    setup.py
```

The core module consists of a single ***Webchecker*** class and contains the following methods:

- ***__init__*** - Class constructor. After creating a new object, this reserved method takes care of initializing logging and loading configuration.
- ***run_detect()*** - Method to start detection of active sites. The only parameter is the path to the file containing the current network streams supplied by the Observe phase support tools. The method loads the current data and gradually builds a list of all websites, on

accessed by at least one user. In order to ensure data consistency, the method also checks the format of IP addresses and the correctness of the domain using DNS translation. The output of the method is a list of all visited websites with the time of the last activity. • ***get_ips()*** - auxiliary method for validating the correctness of the link between the IP address and domain name.

• ***run_certs()*** - method to start checking certificates. It serves as a wrapper for ***the check_cert method,*** which sequentially passes all websites for checking and returns the results of the certificate check.

• ***check_cert()*** - method for checking the validity of the certificate itself. As input, it receives the domain name to check and then starts an active check. It does this by actively querying the given server

The output of the component checking certificates is information about problems with them. Specifically, the output is a JSON structure that contains the following information. • ***time*** - information about the detection time. • ***data*** - a list containing items with individual problems. Each item contains: • ***type*** - specification of item type, always contains the value "cert". The output of the component is further processed as part of the Observe phase and this information is used to differentiate detections of certificate problems from other detected problems in the network.

• ***hostname*** - hostname of the machine where the problem was detected. • ***description*** - detailed information about the problem. It always contains one from the following options:

ÿ "Expired certificate." ÿ "Self-signed certificate." ÿ "Certificate revoked."

ÿ "Certificate hostname mismatch."

ÿ "Certificate will expire on {date}." ÿ "Unspecified certificate error." • ***confirmed*** - information on whether the problem was confirmed by manual control. This is another piece of data that is pre-prepared for automating work with the output within the Observe phase. Its value is initially always ***False,*** because manual checking can take place only after automatic detection.

The second output of the component is information about active websites. As with the certificate validity check, the output is a JSON structure. In this case, it has the following format:

• ***time*** - information about the detection time. • ***data*** - list of items with the following information: • ***ip*** - address of the website to which user accesses were detected according to the information from passive monitoring of network traffic. • ***hostname*** - hostname corresponding to the above IP address.

Use

Since the component performs two different tasks, its execution is also divided into two separate units. In both cases, however, for the component to run correctly, it is necessary to create an instance of the ***Webchecker*** class as follows:

```
>>> from webchecker_component import Webchecker >>> wc =
Webchecker(config, logger_object)
```

The class constructor takes two parameters as an input: •

config - a dictionary whose role is to specify the IP ranges with which the component will work. A dictionary can have one of the following structures:

- ***empty dictionary*** - the case when we do not want to limit the range in any way processed IP ranges.
- ***a dictionary containing the following items:***
  - ÿ ***"ignore"*** - list of IP ranges that will be ignored during the check certificates.
  - ÿ ***"target_network"*** - list of IP ranges where the component will be detect active sites.

• ***logger_object*** - logger instance that will be used for the current running of the component. If this parameter is left blank, an instance of the standard ***structlog will be used.***

After instantiating a class, the user has the following launch options:

1. Checking certificates

```
>>> wc.run_certs(hostnames)
```

Where ***the hostnames*** parameter is a list of pairs (IP, domain name) or just a list of domain names. The check will be performed only on the entries that are defined in the list ***of hostnames.***

2. Detection of active sites

```
>>> wc.run_detect(flows)
```

Where the ***flows*** parameter is the path to the file that contains the network flows over which the detection itself should take place.

Vulnerability Information Acquisition component

The component consists of two parts. The first is the shadowserver subcomponent, consisting of four files written in Python, located in the ***shadowserver_module folder.*** The second part contains 2 files in the ***shodan_module folder.*** In addition, tests are available in the tests subfolder:

```
vulnerability_component/ ÿÿÿ
    shadowserver_module/ ÿÿÿ
            Download.py ÿÿÿ
            Remove.py ÿÿÿ
            Neo4j.py
        ÿÿÿ Shadowserver.py ÿÿÿ
    shodan_module/ ÿÿÿ Shodan.py
        ÿÿÿ Shodan_config.json
        ÿÿÿ tests/ ÿÿÿ data/ ÿÿÿ example-

    masaryk_university-
        ip.csv
```

ÿÿÿ rooibos-masaryk_university-isp.csv ÿÿÿ threat-
masaryk_university-ip.csv ÿÿÿ testCSV/ ÿÿÿ
2018-10-22-
botnet_drone-masaryk_university-ip.csv ÿÿÿ 2018-10-23-scan_redis-
masaryk_university-ip.csv ÿÿÿ 2020-10-22-scan_error_csv1-masaryk_university-
ip.csv ÿÿÿ 2020-10-22-scan_valid_csv-masaryk_university-ip.csv ÿÿÿ 2020-10-22-
valid_csvs-masaryk_university-ip.csv ÿÿÿ invalid_csv1.csv ÿÿÿ invalid_csv2.csv

ÿÿÿ neo4j_test.py ÿÿÿ
remove_test.py
ÿÿÿ README.md
ÿÿÿ setup.py

- Download.py – This module of the ShadowServer subcomponent takes care of downloading data from the servers of the ShadowServer organization. The module first establishes a connection with an external server, performs authentication, and then downloads data for the last day from the server.
- Remove.py – The task of this module is to remove incorrect or outdated data from the Download module. Specifically, this method module ensures: • processing of only files younger than 12 hours, • removal of all downloaded files that do not contain correct data, • removal of all downloaded files with already processed data.
- Neo4j.py – The main functionality of this module is the processing of downloaded files and saving the obtained data to the Neo4j database. The module first gets the names of all files from the Download module. Subsequently, it parses the individual files and thus obtains the information contained in them. It converts these into JSON format and stores them in the database.
- Shadowserver.py – The control module that is in charge of running individual methods. After its completion, it will write statistics about the run and found vulnerabilities as an output to the Orchestration service.
- Shodan.py - The task of this module is to get information about potential problems from the Shodan service.
- Shodan_config.json - Configuration file containing information about issues that should be searched for in the Shodan service interface.
- tests/ – This folder contains the test methods for the component.

The data from the ShadowServer servers is in the form of csv files, which contain context-specific data for each type of vulnerability detection. In addition, all detections have common data, which are:

- timestamp – time of detection,
- ip – IP address of the machine on which the problem was detected,
- protocol – protocol that is associated with the detected problem,
- port – port that is associated with the detected problem.

The name of the detection itself is then available in the name of the CSV file, which has a fixed format consisting of a time stamp, the type of detection and the name of the scanned organization, e.g. ***2018-12-17-scan_ssl_poodle-masaryk_university-ip.csv.***

Before saving to the database, the data needs to be rearranged into the required format. It has the following structure: • data on the
detection system. • list of detections
for the last period: • IP – IP address of the
machine where the problem was detected, • timestamp –
detection time, • vulnerability –
name of the detected problem, • description – detailed
information about the detection.

The component in the database creates the following types
of nodes: • IP - contains information
about the IP address, • DetectionSystem - contains information about the detection source, in this case it is Shadowserver organization name,
• SecurityEvent - contains information about a security incident. Specifically, this is the time of detection, the type of problem, the description of the problem and whether the existence of this problem has been confirmed.
The component also creates two types of edges:
• SOURCE_OF - an edge between an IP address and a security incident, which indicates that the machine with a given IP address is the source of a given security incident, •
RAISED_BY - an edge between a detection system and a security incident, which indicates that a given security incident was detected by a given detection system.

Use

The component can be run after the method is run: >>>
from vulnerability_component import Shadowserver >>>
Shadowserver.process_vulnerabilities("neo4jPass",
"ShadowLogin", "ShadowPass") which
takes the following parameters as input:
• neo4jPass - password to the Neo4j database instance,
• ShadowLogin - login name to the Shadowserver server, • ShadowPass
- login password to the Shadowserver server.

CVE Connector

cve-connector/

ÿÿÿcve_connector/
ÿÿÿnvd_cve/
ÿÿÿ categorization/
ÿÿÿ cia_loss.py
ÿÿÿ classifier.py
ÿÿÿ code_execution.py ÿÿÿ
gain_privileges.py ÿÿÿ
helpers.py ÿÿÿ
test/ ÿÿÿ
test-data/

ÿÿÿ classifier_data.json ÿÿÿ
nvdcve-1.0-2017.json ÿÿÿ test-
data1.json ÿÿÿ test-
data2.json ÿÿÿ
classifier_tests.py ÿÿÿ
neo4j_test_client.py ÿÿÿ
nvd_test.py ÿÿÿ
cve_parser.py ÿÿÿ
toneo4j.py ÿÿÿ
utility.py
ÿÿÿvendor_cve/ ÿÿÿ
implementation/ ÿÿÿ data/
conf.ini ÿÿÿ parsers/
ÿÿÿ storing_to_db/
neo4j_storing.py ÿÿÿ utilities/ ÿÿÿ

vendors_storage_structures/ ÿÿÿ
vulnerability_metrics/ ÿÿÿ main.py
ÿÿÿ README.md
ÿÿÿ setup.py

The component consists of two partial sub-components: nvd_cve and vendor_cve.

The nvd_cve subcomponent is responsible for downloading CVE data feeds from NVD, analyzing them and assigning a predicted impact and saving them to a database. • The categorization folder contains the implementation of the split categorization algorithm between several modules.

• The cve_parser.py module contains a parser for CVE data provided by NVD. • The toneo4j.py module contains functions that add analyzed data to the Neo4j database according to the CPEs present in the database.

• The utility.py module contains functions that download and remove the CVE data source from NVD.

The module parses the obtained files in JSON format and stores the obtained data in an auxiliary structure. From each file, it is possible to obtain general information about the file, e.g. the number of CVEs and information about each CVE: • CVE ID - unique CVE identifier, • CPE - product identifier in the form of manufacturer, name, product version, • CWE ID - weakness identifier, • URL of the website where the vulnerability was published, • description of the vulnerability, • CVSSv2 and CVSSv3 - evaluation of the severity of the vulnerability,

• date and time of publication in the NVD database, • date and time of the last modification of the record.

The structure of the CPE chain and the information in the CVSS metrics are described in detail in the technical report Automated Vulnerability Information Acquisition.

When deciding which CVEs to add to the database, we decide as follows:

1. If there is at least one software version in the database that is affected by the vulnerability and the CVE was created or changed within the last 24 hours, the CVE is added to the database. Otherwise, it is not added. Subsequently, a node of type Vulnerability is created and the CVE is linked to it by the REFERS_TO relationship. The software version is then connected to the Vulnerability node by the IN edge.

2. If the CVE is already in the database, but one of its properties has changed, it is Updated CVE to match current state in NVD.

The vendor_cve subcomponent is responsible for downloading CVE data from software vendor websites (supported vendors are Adobe, Android, Apple, Cisco, Lenovo, Microsoft, Oracle, RedHat).

• The data folder contains a configuration file with URLs for manufacturer websites. • The parsers folder contains: • a general parser and parsers for specific file types (HTML, JSON and XML)

• analyzers for specific suppliers - each folder contains analyzers of data provided by the manufacturer. There is usually a parser for the main page (ending in main_page_parser.py) that links to specific web pages for individual vulnerabilities (ending in vulnerability_parser.py). • The storing_to_database folder contains the neo4j_storing.py module responsible for adding CVEs to the database.

• The utilities folder contains several auxiliary functions. • The vendors_storage_structure folder contains a class for each vendor that stores its information. • The vulnerability_metrics folder contains functions related to CVSSv2 and CVSSv3. • The main.py module contains an entry point function for processing data from suppliers.

Use To get data from NVD, you need to import the necessary functions using the command on the first line. The function on the fourth line then performs the functionality of the module itself and needs three parameters: • time - vulnerabilities that were created or modified after this date are added to the graphic database. An example of the calculation of this parameter is shown on the second and third lines of the sample code, • the password to the Neo4j database, • the path to the folder where the files downloaded from NVD will be stored.

```
>>> from cve_connector.nvd_cve import toneo4j >>> from datetime import
datetime, timedelta >>> specified_time = (datetime.now()-
timedelta(days=1)).isoformat() >>>
 toneo4j.move_cve_data_to_neo4j(specified_time, "neo4jpassword",
```

"/tmp/cve-data")

The Vendor CVE module works similarly and also needs a file storage folder and database password.

```
>>> from cve_connector.vendor_cve.implementation.main import add_vendor_CVEs

>>> add_vendor_CVEs("/tmp/ms_downloads", "neo4jpassword")
```

SABU Connector

The connector consists of a single file written in python that contains all the functionality.

```
sabu-connector/
  ÿÿÿsabu/
      ÿÿÿJsonParsing.py ÿÿÿ
  README.md ÿÿÿ
  setup.py
```

The JsonParsing.py script consists of four methods:

- ***warden_is_running*** - method verifies if Warden is running and if not, tries to start it. If the execution attempt fails, the method throws an exception.
- ***dump_data_to_json_if_ref_key*** - the method saves all data obtained from messages in which the key "Ref" containing the value "cve id" was found to the file /var/www/neo4j/import/sabu_ref_key.json.
- ***dump_data_to_json_if_not_ref_key*** - similar to the previous method, except that it stores data from messages in which the "Ref" key did not occur. It stores data in /var/www/neo4j/import/sabu_not_ref.json.
- ***parse*** - the method parses the messages received from the warden. It selects only those in the correct format and subsequently calls the methods for saving data in json described above with this data. Finally, these json files will be processed by the neo4j client and saved to the graph database.

Messages received from the Warden system are in JSON form. These messages contain, among other things, keys of interest for further processing:
- DetectTime - detection time,
- Target.IP4 - IPv4 address of the machine on which the vulnerability occurs,
- Impact - description of the vulnerability,
- Category - type of vulnerability.

In the case of a message containing the Ref key:
- DetectTime - detection time,
- Target.IP4 - IPv4 address of the attack target,
- Ref - CVE identification number.

Use

The connector is started by running the method:

```
>>> from sabu import JsonParsing >>>
JsonParsing.parse(directory, passwd, regex,
    path_to_warden_filer_receiver, path_to_neo4j)
```

with the following parameters: • directory

- directory containing received messages from the warden, • passwd - password to the Neo4j database instance,
• regex - regular expression for searching for an IP address from the MU range in a message from the warden, • path_to_warden_filer_receiver - path to the program warden_filer_receiver,
• path_to_neo4j - path to the directory where the resulting jsons will be stored.

RTIR Connector

All functionality of the component is provided by one ***rtir module.*** In addition, the module contains a set of tests. The whole structure looks like this:

```
RTIR-connector/
    ÿÿÿ rtir_connector/
        ÿÿÿ rtir.py
        ÿÿÿ tests/ ÿÿÿ
            rtir_test.py
        ÿÿÿ certs/
            ÿÿÿ cert_file.crt
    ÿÿÿ README.md
    ÿÿÿ setup.py
```

The rtir module contains the ***parse_rt() method,*** which is the starting point of the program. After it starts, the logger is initialized, in which information messages are written. Subsequently, the method arguments are processed and ***the download_data()*** method is called , which establishes a connection with the RTIR server and downloads the list of current security events. If this list contains unprocessed events, the ***parse_ticket() method is called,*** which returns detailed information about the given security event. The last step is to process the data obtained in this way into JSON format and then store it in a graph database.

The JSON file contains data about individual security events, which are then added to the database. Each event in this file contains:

• the category of security events, • the
date of creation in the RTIR system, • the
entity that created the event, • the list
of IP addresses to which the event relates, • the e-
mail addresses for the notifier of the event, • a brief
description of the event.

In a graph database, the connector creates the following types of nodes:

• IP - IP address of the originator of the event, if available, •
DetectionSystem - source of the event, if the notifier is a detection system, • SecurityEvent
- contains data about the security event.

The connector in the database also creates the following edges:

- SOURCE_OF - an edge between an IP address and a security event that indicates that the machine with the given IP address is the originator of the given event,
- RAISED_BY - an edge between the detection system and the security event indicating that this system detected the event.

Use

The connector is started by calling the method:

```
>>> rtir.parse_rt(user, password, output, uri, subnet_filter,
        last_day, logger) with
```

parameters: •

user is the login name for the RTIR system, • ***password*** is the login password for the user account , • ***output*** is the path where the resulting JSON file is saved, • ***uri*** is the URL address of the REST API of the RTIR system, • ***subnet_filter*** is a filter that ensures the connector receives only security events from that subnet,

• ***last_day*** is a parameter specifying that the connector should only download data for the last day, • ***logger*** is the logger instance that the connector will use.

Critical node search component

```
criticality-estimator/ ÿÿÿ
  src/ ÿÿÿ
      core.py ÿÿÿ
  README.md ÿÿÿ
  setup.py
```

The task of this component is to perform calculations in the database with the aim of discovering the most important nodes based on their centrality. The results are also stored in a graph database, so this component does not export any data.

The component consists of one file: • core.py is used to calculate the criticality (importance) of nodes in the database.

The component currently enables the calculation of: • topological betweenness centrality • topological centrality measured by degree centrality • centrality measured by node degree based on the dependencies of the services provided

Usage

The connector is started by calling the method:

```
>>> from criticality_estimator import CriticalityEstimator >>> ce = CriticalityEstimator(bolt="bolt://
localhost:7687", password="test", logger=logger) >>> ce.run( )
```

Netlist connector

NETlist-connector/

```
ÿÿÿ NETlist_connector/ ÿÿÿ
    data/ ÿÿÿ
        subnets_data.txt ÿÿÿ
    contacts.py ÿÿÿ
    domains.py ÿÿÿ
    subnetParser.py ÿÿÿ
README.md ÿÿÿ
setup.py
```

This component is responsible for continuous updates of domain names corresponding to IP addresses. In addition, it updates information about subnets as well as responsible contacts and organizational units. The component consists of the following three files: • contacts.py is responsible for adding contacts for subnets. • The task of dopmains.py is to find domain names for IP addresses. • subnetParser.py finds subnets and communicates with the database instance.

Use

The connector is started by calling the method: >>> from NETlist_connector.subnetParser import NETlist >>> nl = NETlist("neo4_password", "path to neo4j import directory", logger=logger)

>>> nl.update()

Thanks

The work on the software was supported by the project ***Research of tools for assessing the cyber situation and supporting decision-making of CSIRT teams in the protection of critical infrastructure*** (VI20172020070) solved in ***the Security Research of the Czech Republic*** program in the years 2017-2020 at Masaryk University. The authors of the software are Martin Laštoviÿka, Jakub Bartolomej Košuth, Daniel Filakovský, Antonín Dufka and Martin Husák.

Appendix 1: System Architecture

A diagram showing the system architecture using vector graphics is available in the attached file ***Appendix No. 1 - System architecture.pdf .***

Appendix 2: REST API endpoints

Complete available REST API documentation is already address https://app.swaggerhub.com/apis/CSIRT-MU/DASHBOARD/1.0.0. Its offline copy is available in the attached file ***Appendix No. 2 - Crusoe REST API.pdf.*** An overview list of API endpoints is given below:

AccessControlLayer •
GET /org_units • GET /
org_units/{name}/subnets
CompleteViews •
GET /acces_control_layer • GET /
host_layer • GET /
mission_layer • GET /
network_layer • GET /
response_layer • GET /
system_layer • GET /
threat_layer
HostLayer •
GET /services
• GET /services/{name} • GET /
software
• GET /services/{name}/ips • GET /
software/{name}/ips
MissionLayer •
GET /mission/{name}/configuration/{config_id}/hosts • GET /mission/
{name}/configurations • GET /mission/{name}/
hosts • GET /missions

• GET /missions/hosts
• DELETE /missions/{name} • GET /
missions/{name} • POST /
missions
NetworkLayer •
GET /ip/{address}/cve • GET /
ip/{address}/events • GET /ip/
{address}/events/{date} • GET /ip/{address}/
events/latest • GET /ip/{address}

• GET /ip/{address}/services • GET /
ip/{address}/software • GET /ip •
GET /subnets

• GET /subnets/{subnet} • GET /
subnets/{subnet}/ips
ResponseLayer •
GET /events/after/{date} • GET /
events/{date} • GET /events

ThreatLayer •
GET /cve/{cve_id} • GET /
cve/{cve_id}/ips • GET /cve