Computers & Security 116 (2022) 102638

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

# Egida: Automated security configuration deployment systems with early error detection

Antonio Paya*, Alba Cotarelo, Jose Manuel Redondo

*Department of Computer Science, University of Oviedo, Science Faculty, Office 240, C/Federico Garcia Lorca S/N, Oviedo 33007, Spain*

ARTICLE INFO

*Article history:*
Received 12 June 2021
Revised 23 November 2021
Accepted 1 February 2022
Available online 3 February 2022

*Keywords:*
Security control
Automation
Defense in depth
Error detection
CIS benchmarks

ABSTRACT

Automated deployment of validated security controls is very important to implement defense-in-depth strategies to secure machine infrastructures. This paper describes a technique to perform these deployments in a more controlled way, using a DSL. According to the target machine configuration or characteristics, the DSL provides ways of deploying automated security controls that are less prone to negatively impact legitimate running services, and allows detecting several deployment errors earlier. This helps to better capture system administrator expert knowledge, and share automated security scripts that obtain better hardening results. The initial results of this technique are promising enough to apply them on educational environments, and can be further developed to be applied on production infrastructures.



## 1. Introduction

The current proliferation of attacks shows that securing IT systems requires the implementation of adequate and trusted security controls. This requires the adoption of a defense-in-depth strategy and implementing international standards (*Information Security Management Systems*, ISMS), such as ISO 27001 (Software and systems engineering, 2020). These standards contain suitable security control lists adapted to different contexts.

Automation may facilitate ISMS implementation. SCAP (*Security Content Automation Protocol*) is an international reference protocol to implement it. Properly automated ISMS implementations also require an adequate source of security controls. These must be international, validated, and supported by their creators. There are several adequate SCAP-compatible sources, such as *STIGs* (Cyberx-Mw and Knowlton, 2020) or the *CIS Benchmarks*. The latter incorporates the *CIS Controls* abstraction layer, that facilitates to map concrete actions to security controls present in frameworks like *ISO 27001* (CIS, 2020c), facilitating their implementation.

However, unsupervised application of any list of security controls on a software product to be secureds may impair the performance or disable services that are running on a system. Security controls could maximize security at the expense of the functionality of legitimate software running into the system. This way, we can have a highly secure but non functional system, requiring security control customization to prevent these problems. These interactions can be also very hard to predict, and some sources of security controls highlight potentially problematic ones (i.e. the *CIS Benchmarks* label them as *Level 2*, see Section 2.2.3). Therefore, blindly applying a list of security controls without giving administrators adequate tools to decide how, or if, certain controls have to be applied in concrete systems is a problem that requires attention.

The research presented in this paper aims to increase the amount of control that system administrators have when applying an existing list of security controls. This will be done through a *Domain Specific Language* (DSL). This DSL statements will use runtime information about each target system to enrich the automation process. This information can be used for example to decide if deploying or not certain parts of a security control list to prevent breaking legitimate functionality. It can be also used to prevent deployment errors earlier, as it can improve the detection of cases in which a security control deployment will be unsuccessful. The DSL allows to incorporate decisions and other programming elements into the automation process, so a single automation file can implement multiple target system needs. This also allows to better capture the expert knowledge of an administrator to avoid control deployment problems.

The structure of this paper is the following: next section will outline the related work of our research, while Section 3 describes how the prototype of our research was de-

* Corresponding author.
*E-mail addresses:* antonio.paya@thenextpangea.com (A. Paya), alba.cotarelo@thenextpangea.com (A. Cotarelo), redondojose@uniovi.es (J.M. Redondo).

https://doi.org/10.1016/j.cose.2022.102638

signed. Section 4 presents the behavior of our prototype in several use cases that cover our requirements, and, finally, the Section 5 presents the conclusions and future work.

## 2. Related work

### 2.1. Implementation of defense in depth strategies

The *Defense in Depth* philosophy is a concept inherited from military defense that applies to every computer in an infrastructure (Stytz, 2004). A well-defined and well-implemented defense-in-depth strategy can prevent and avoid a wide variety of risks, and provide monitoring and alert tools to identify unauthorized access (Cleghorn, 2013). In this strategy, different security measures, divided into layers, are developed to minimize them (Kuipers and Fabro, 2006). These layers overlap to cover potential deficiencies each other (Weaver and Farwood, 2013). Typical defensive mechanisms implemented are: network traffic analysis, behavioral analysis, anti-malware software, data integrity analysis software, restrictions embedded in software code, user and system restrictions, and computer infrastructure hardening.

To properly implement computer infrastructure hardening, reliable and verified security guidelines must be followed. These are composed by a list of security controls with the following characteristics:

- They are *complete*, so a desired security level for a specific OS or product can be achieved. This means that every part of the secured systems suitable to be compromised has to be covered by the controls.
- Each security control must be *justified* (why each control is needed). This also helps to solve potential interferences with the legitimate system activities or functionalities. This is the knowledge we aim to model with *Egida*.
- Each security control includes *how to check* if it is already applied on a target.
- Each security control includes *how to apply* (implement) it when not present on a target.

There are guidelines that comply these characteristics. We can mention the military-oriented *Security Technical Implementation Guides* (STIGs) (Cyberx-Mw and Knowlton, 2020) and the *CIS Benchmarks* (CIS, 2020a). The last one was chosen in this research due to its free availability as PDF documents, its worldwide support by major companies, the frequent maintenance offered by the *Center for Internet Security*, and the ability to facilitate ISO 27001 (Broderick, 2006) implementation (see Section 2.2.3).

### 2.2. Security controls and standards

ISMS implementation can be facilitated via technologies that allow automation, such as the SCAP (*Security Content Automation Protocol*) (Computer Security Division, 2020) protocol and the TCG (*Trusted Computing Group*) (Berger, 2005) framework.

#### 2.2.1. SCAP

It is a set of NIST security automation standards for assessing compliance with security policies. It is also able to detect vulnerable versions of software and system configurations (Waltermire et al., 2018). This protocol provides standard formats and nomenclatures for defining and exchanging information between end users and tools, and has many components that can be classified into the following categories according to their purpose:

- Enumeration (*CVE, CCE*, and *CPE*)
- Metrics (*CVSS* and *CCSS*)
- Languages (*XCCDF, OVAL*, and *OCIL*)
- Report Format (*ARF* and *AI*)
- Integrity (*TMSAD* and *SWID*)

*Enumeration* components include CVE (*Common Vulnerability and Exposures*), a system for naming and documenting known vulnerabilities referenced with a unique identifier. Each CVE contains information such as description, which versions of the software are affected, the possible solutions (if any), and how to mitigate the vulnerability. CCE (*Common Configuration Enumeration*) and CPE (*Common Platform Enumeration*) are very similar to CVE but their objectives are to document system configurations and to collect and identify technology systems, software, and packages respectively.

Regarding *Metrics*, the CVSS (*Common Vulnerability Score System*) is a scoring system that allows estimating the impact of vulnerabilities based on their characteristics. CCSS (*Common Configuration Score System*) (Mell and Scarfone, 2010) is a CVSS derivation that measures the impact of security flaws that depend on certain product configurations.

*Reporting* and *Integrity* components of the SCAP protocol are outside the scope of this research. Opposite, its most important part are their XML-based *Languages*:

- OVAL (*Open Vulnerability and Assessment Language*) (Corporation, 2015) allows to standardize the reporting and assessment of system configuration status. It consists on three XML schemas: the *OVAL System Characteristics schema*, to represent system information, an *OVAL Definition schema*, for expressing a specific machine state, and an *OVAL Results schema*, for reporting the results of an assessment. This language represents truth values associated with system components (i.e.: "*the version of the* `/bin/programfile` *is 2.4*", "*the value of a registry key is 0*", or "*the user Alba is a member of the Phd group*").
- XCCDF (*Extensible Configuration Checklist Description Format*) (NIST, 2021) allows to write security checklists, benchmarks, and related documents. These documents are a structured collection of security configuration rules for several target systems. XCCDF specification supports information interchange, document generation, organizational and situational tailoring, automated compliance testing, compliance scoring, and a data model and format for storing results of benchmark compliance testing. XCCDF documents can refer to OVAL documents to fully specify their elements.
- OCIL (*Open Checklist Interactive Language*) specifies the necessary guidelines to define a series of less automatable checklists that require more human intervention.

These languages facilitate the exchange of information between cybersecurity professionals, software vendors, and auditors, as well as enable the development of automated tools and solutions. Files written in these languages can be used by SCAP-compatible tools to automatically verify or enforce security controls on a substantial number of OS and/or software products. *CIS Benchmarks* and *STIGs* can be also found in SCAP- compatible format, although at a cost or not publicly available, respectively. In fact, freely available STIGs only support manual checking and remediation of their security controls.

However, although the SCAP protocol is a very important step in the right direction, it has a series of issues. Specifying security control lists is verbose and require substantial knowledge about the structure and schemas of these files. Even the simplest examples require the definition of multiple elements to achieve correct results (Bergmann, 2017). Additionally, manual specification of security control lists may easily miss critical security controls, as it requires deep systems, software, and security knowledge. Being XML-based, implementation mistakes committed in their defini-

tion will detected at runtime. Because of these reasons, we consider that most users will use validated, international, and maintained third-party control lists they can trust instead of defining their own ones. *CIS Benchmarks* or *STIGs* are suitable candidates, but their limited availability or the substantial cost of their official implementations may also prevent several business to adopt them.

Even if cost is not a factor (unofficial and open implementations are used), the lack of advanced security control application options is still a problem to resolve. Security control lists are provided as a comprehensive list of operations to perform in sequence. They may be categorized by machine role, potential "dangerousness" to legitimate functionalities of a system, or other criteria. Tools that manage them may enable administrators to apply operations bound to one or multiple categories, but do not provide the ability of individually consider the actions of each individual security control upon its application on certain systems. For example, if an administrator decides to deploy a security control that minimize the amount of installed packages, it will uninstall an OS GUI to achieve their goal. However, if some machines have a `Training` user group whose users require a GUI, these users will lose convenient functionality. If the administrator could easily check prior to apply the control if a machine has this group defined, it could decide not to apply it on these machines while implementing it on the others. Providing this kind of flexibility, so security implementation does not go against functionality, is one of the main goals of our research.

The final problem that SCAP-based implementations have is that errors that happen during control application are reported at runtime. This means that if a control breaks the execution, the rest of the controls will not be applied, and the system might be left in an inconsistent state depending on the controls that have been applied. The user is forced then to debug the problem and to repeat the process of control application again, as no way of early detecting potential problems exist. Finding ways of decreasing this possibility is the second main goal of our research.

#### 2.2.2. TCG (Trusted Computing Group)

The *TCG (Trusted Computing Group)* (Berger, 2005) is a consortium of more than 200 companies. They develop, define, and promote a set of open source hardware specifications to protect systems from attacks that cannot be protected by only software solutions to obtain a reliable platform. To this end, the *TCG* defines three characteristics that these systems must meet: (i) Measuring their own integrity, logging and reporting, (ii) Protected capabilities, and (iii) Ensuring the accuracy of a component's status information.

However, a major obstacle in the use of *TCG-based* solutions is the absence of a public repository (published by the software vendor) of hash values of all software components. In addition, the static and rigid whitelisting approach is not well suited to large-scale distributed environments.

#### 2.2.3. Center for Internet Security (CIS) security resources

The *Center for Internet Security Critical Security Controls for Effective Cyber Defense*, also called CIS-CSC (*Center of Internet Security - Critical Security Controls*) (CIS, 2020b) is a collection of best practice guidelines that organizations can implement to reduce their cyber-attack surface significantly. These guidelines are formulated by a group of information technology experts using information obtained from real attacks and their effective defenses.

These guides are composed by 20 (rev. 7, 2018) or 18 (rev. 8, 2021) activities called *Critical Security Controls* (CSC). They are divided into three categories called *Implementation Groups* (IG). Each IG defines which security controls an organization should implement depending on the resources it has at its disposal and its level of risk. Organizations are responsible of classifying themselves into one of the three IGs depending on their security budget. This allows the controls to be used by almost any type of organization regardless of size or resources (Shamma et al., 2018; Winarno et al., 2020). Each IGs include the previous one. This way, IG 1 includes basic security controls, while IG 3 includes the most advanced ones regarding software security, incident response, management, or penetration testing.

*CIS Controls* can work either as standalone resources or as companions to other frameworks. As we said, CIS provides a mapping (CIS, 2020c) of their security controls to other frameworks such as NIST or *ISO/IEC 27001*.

*CIS Benchmarks* are documented industry best practices for securely configuring IT systems, software, and networks. There are currently over 100 benchmarks, that cover many vendor product families. They contain a detailed list of security configuration tasks (security controls) applicable to that product. The *CIS Benchmarks* provide a mapping of each of these tasks to the corresponding *CIS Controls*. Task "dangerousness" to the performance or availability of product features are indicated by levels. *Level 1* tasks can be implemented fairly quickly, and are designed not to have a major impact on product performance. *Level 2* tasks are considered part of a defense-in-depth strategy, and are intended for critical environments. These tasks can affect the proper functioning if not applied properly. *CIS Benchmarks* can be automated following the *SCAP* protocol guidelines, but at a cost.

### 2.3. Automated tools for security configuration and verification

Guidelines and standards, such as *SCAP* or *TCG*, are usually used with automatic hardening tools that try to comply with most of their sections. Examples of these tools are:

- *VM2* (Spichkova et al., 2020), that automatically generates virtual machines applying some of the *CIS Benchmarks* to them.
- The configuration script for a *Linux* web server proposed by Michal Olencin (Olenčin and Perháč, 2019).
- *JShielder* (Soto, 2019), which automates the hardening of *LAMP (Linux, Apache, MySQL/MariaDB, and PHP)* and *LEMP (Linux, Nginx, MySQL/MariaDB, and PHP)*
- Research projects such as *ISCP* (Al-Safwani et al., 2018), or the one proposed by Durkota et al. (2019). They use models to determine vulnerable controls and provide clear guidelines on how to perform control analysis and to represent the possible actions of an attacker using attack graphs respectively.
- The most popular implementation of the *SCAP* protocol, the *OpenSCAP Project* (OpenSCAP, 2020) and its GUI *SCAP Workbench*. It is a collection of open source tools for implementing and enforcing the *SCAP* standard, which are certified up to *SCAP* version 1.2. These tools comprise a multi-purpose specification framework that supports automated configuration, vulnerability checking and patching, technical control compliance activities, and security measurements. *OpenSCAP* supports files written in the *OVAL* and *XCCDF* languages, as well as allowing system audit reports based on already defined policies such as those of the *CIS* or *STIGs*. *OpenSCAP* is typically used with the also freely available SCAP-compatible files of the `scap-security-guide` package (ComplianceAsCode, 2021). However, although this combination is popular, it has several important shortcomings, such as not covering typical products, not supporting *Windows* systems, lack of automatic verification and/or remediation of several security controls, or incomplete implementations when compared with official counterparts.
- Other tools that use these guidelines and standards to perform system audits and tests such as *Lynis, Chef Inspec* or *Prowler*. The result of these tools is a report with a checklist of the security controls that the system has passed or failed.

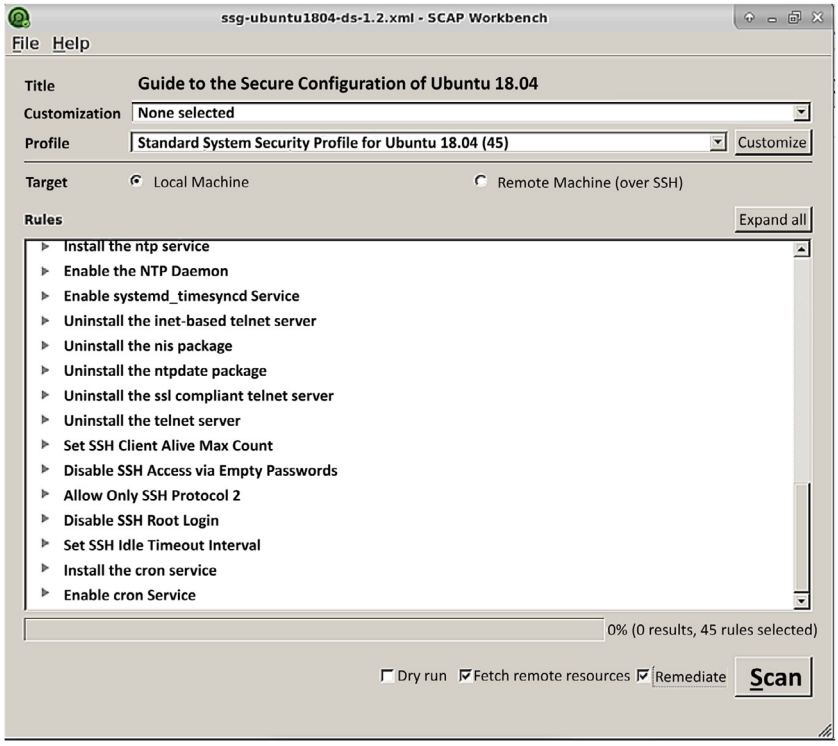


**Fig. 1.** *SCAP Workbench* running a list of security controls for *Ubuntu 18.04.*

These tools have the problem we outlined at the end of the Section 2.2.1: they apply the security control lists in a sequential manner (see Fig. 1), not giving the administrator enough control over their actions to prevent breaking legitimate functionality. Potential errors while control application are reported at runtime, forcing the user to restart the process when these errors are fixed. *Egida* aims to enable the same type of automation that these tools alleviating these problems.

## 3. Egida

This research project proposes an automated security control orchestration system called *Egida*. It allows to deploy security configurations (validated security control checklists) on an infrastructure of machines giving more deployment flexibility to the administrators. These security configurations can protect these machines by applying security measures depending on their usage profile, decreasing the risk of breaking legitimate functionality if compared with other approaches.

### 3.1. Security controls deployment

As we said in Section 2.2.1, developing adequate custom security controls is a very complex task for most use cases, so tried and tested third-party ones are usually preferred. For this reason, *Egida* uses the *CIS Benchmarks* guides and their security controls.

*CIS Benchmarks* officially support the *SCAP* protocol, but at a cost (Sager, 2021) and with a custom SCAP-compatible tool (*CIS CAT Pro*). Very recently, a *Lite* version of this tool can be obtained free of charge. However, due to the shortcomings of using the SCAP protocol in this scenario we outlined in Section 2.2.1, we chose the highly popular *Ansible* configuration deployment tool to distribute the security controls verification and implementation procedures.

Using *Ansible* has three main advantages: the ability to work over any machine accessible via SSH without installing additional software, its ability to be used in large-scale deployments, and its built-in ability of not repeating operations that are detected as already performed. This means that restarting an *Ansible* script due to runtime errors do not perform already done operations. This greatly facilitates security control testing and enforcement propagation with a great degree of efficiency. *Ansible* is indeed used by several of the products listed in Section 2.3 as a complement, to automatically verify and/or remediate security controls. For example, `scap-security-guide` SCAP files are typically complemented by *Ansible* or *Bash* remediation scripts, that may automatically remediate problems in cases these procedures are not available in the companion SCAP files yet.

Another advantage of using *Ansible* is that we can follow a more granular approach. Therefore, with *Egida* we can achieve a more precise and flexible management of the *CIS Benchmarks* security controls to be deployed. To do that, we can create smaller or more specialized security control *profiles* that can be combined thanks to *Ansible* features. This facilitates deploying just what it is needed, favoring more flexible machine specialization. Aggregation of security control profiles allows better security control micromanagement, which in turn may decrease the probability of breaking legitimate functionality.

Even tough *CIS Benchmarks* security controls are a very strong foundation to obtain secure systems, there are also additional security software that may complement them to further enhance their security. Reverse proxies, perimetral firewalls, host-based firewalls, *NIDS, HIDS, WAFs*, etc. strengthen certain points of an infrastructure, better implementing the defense-in-depth approach. Installation of this type of advanced software is not typically covered by *CIS Benchmarks*. The *Egida* project also want to apply its automated and flexible approach to these higher-level security elements, so they are easier to apply over specific machines. The installation of these products is also facilitated with *Ansible*, as there are documented ways of automating its deployment using it. It is important to remark that other solutions studied in 2.3 typically lack this functionality.

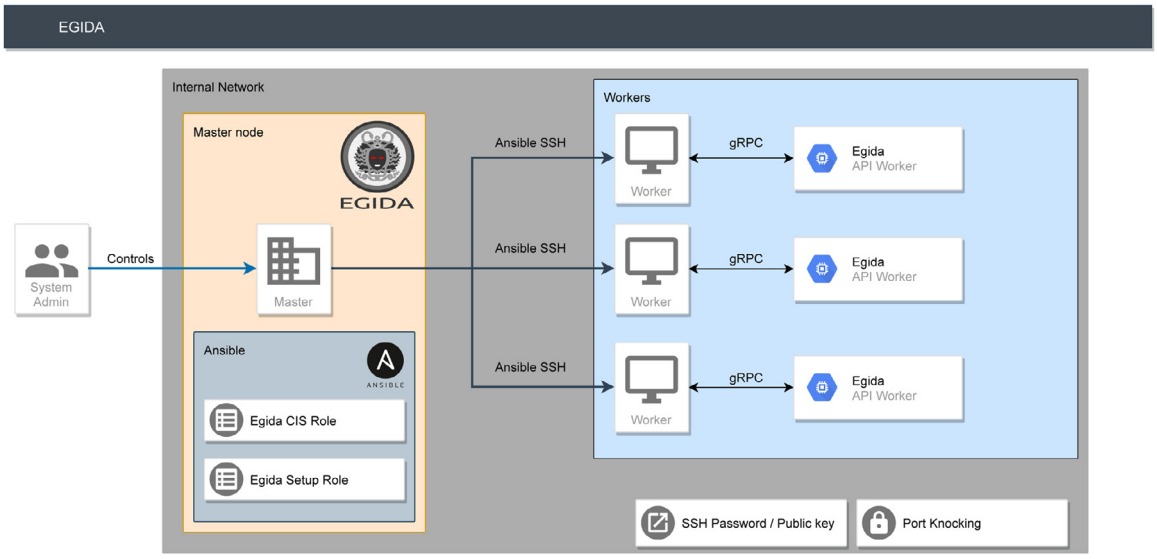


**Fig. 2.** Egida structure.

Finally, in order to achieve *Egida* goals we require a method to obtain runtime information of the state and configuration of each target system. This type of information will be used to customize the application of the CIS security controls thanks to the *Egida* DSL, that will be described in the following section.

### 3.2. Domain Specific Language (DSL)

As we said, *XML* markup-based solutions like *OVAL* or *XCCDF* provide specifications to precisely detail and deploy security controls, but they do not allow to incorporate what we can call *semantic checks*. *Ansible* and SCAP tools deployment errors are discovered at runtime. The final goal of the *Egida* research project is to develop a DSL that allows administrators to deploy security control groups from the *CIS Benchmarks*, but performing early validations of the infrastructure machines and/or contents prior to this deployment. Things like inconsistencies with the machines that are expected to be found, open ports, available software, necessary directories, required users, etc. could prevent a deployment to be successful or break legitimate functionality. *Egida* aims to explore the limits of what can be validated prior to perform a deployment, so the probability of finding a runtime failure or break a service is significantly lower. A precise early error control may also enable administrators to solve problems in substantially less time, as the deployment procedure need not to be run subsequently until no error is found. We successfully tried this philosophy with *Nmap* scans in Redondo and Cuesta (2019).

The purpose is to define *DSL* programs containing the hardening configuration of machines fulfilling a specific role of a company. This way, these programs contain all the hardening configuration (security controls, extra security software, plus their associated variable values) of their *HTTP* servers, *SGBD* servers, proxies, or other machine types. These files could also be distributed and reused all through the company infrastructure. If configuration changes, or must be refined, the *DSL* could be able to detect inconsistencies produced in the new versions prior to its deployment.

### 3.3. Structure

*Egida* works with a *Master-Worker* design pattern (see Fig. 2) where the system administrator has control of the *Master* machine with the *Egida* software installed. From this *Master* machine we can manage the security configurations of the *Worker* machines and how it will be implemented.

*Egida* is composed of four independent modules: *Egida Core, Egida Role CIS, Egida Role Setup* and *Egida API Worker*.

#### 3.3.1. Egida Core

*Egida Core* is the main program, and also the interface that the system administrators use to interact with the machines they want to configure. *Egida* allows two usage modes: an interactive menu, and a *Domain Specific Language (DSL)* called *Aspida*.

Some of the tasks and operations can require variable data that depend on the type of installation or configuration that you want to apply to a machine (e.g. user names, passwords, etc.). To solve this, *Egida* allows using variables that can be modified by the system administrator, both in the interactive menu execution mode (through a *YAML* configuration file) and in the *Domain Specific Language* (within the *variables* block).

*Menu* An interactive console menu that allows to select hardening options to perform on a machine or set of machines. It allows the customization of the machine's security configuration by selecting the *CIS Controls, sections*, or specific points of the *CIS Benchmarks* to be applied.

*Domain Specific Language Aspida* allows the development of predefined configuration scripts that act differently depending on the state of the target machine. The language is aimed at facilitating the implementation of security controls linked to machine profiles, also allowing users to check different states, conditions, or variable values so that they can prevent or react to deployment errors before configuration deployment occurs and also prevent breaking legitimate functionality.

The language also aims to implement a semantic error prevention module so that a given program cannot be deployed if the language processor detects any kind of problem with the deployment infrastructure, incompatibilities between security controls, wrong variable values, and any other condition that may cause a runtime error that can be prevented at compile time (see 3.2).

*Aspida* allows the use of conditional structures in order to provide the user with the ability to specify different types of actions depending on the information obtained from the target machine. The main elements of the structure of the grammar are shown in Listing 1.

```
program : main hosts tasks variables?;

// Blocks
main : MAIN_KW ':' '{' main_content '}';
hosts : HOSTS_KW ':' STRING NS;
tasks : TASKS_KW  ':' '{' tasks_content '}';
variables : VARS_KW  ':' '{' vars_content '}';
...
// ** MAIN Block **
main_content : main_prop (main_prop)*;
main_prop: name | connection | description
...
// ** TASKS Block **
tasks_content : tasks_prop (tasks_prop)* |
    ifStat (elifStat)* elseStat;
tasks_prop : sections | points | controls |
    exclusions | tags;
...
// ** VARS Block **
vars_content : vars_prop (vars_prop)*;
vars_prop : STRING ':' value | STRING ':' '{'
    vars_content '}';
...
```

**Listing 1.** Aspida grammar.

*MAIN* This block provides information about the script, such as the name or a description or the type of connection to the target machine (`Local` or `SSH`).

*HOSTS* Name of the host(s) on which the script will be executed.

*TASKS* This block is the most important one, as it contains all the information about the tasks to be performed. It allows you to define which sections or specific points of the *CIS Benchmarks* will be executed or excluded, as well as to select *CIS Controls* (see Section 2.2.3), or to execute all the tasks that correspond to a label (for example all the tasks that are related to SSH). In this block, `If-ElsIf-Else` conditional statements can be used along with values obtained from the target machine state to control the script flow and to apply the CIS elements.

*VARS* The *variables* block allows you to give a value to each of the variables that will be used during script execution.

Some examples of programs written in *Aspida* are shown in Fig. 3.

#### 3.3.2. Egida Role CIS

The *Egida Role CIS module* is an *Ansible Role* that defines all security configuration operations based on the *CIS Benchmarks* that *Egida* can perform. *Ansible Roles* allows to automatically load related vars, files, tasks, handlers, and other *Ansible* artifacts based on a known file structure. This way, if all the hardening content is grouped in roles, they can be easily reused and shared with other users, or even the public *Ansible* repository (*Ansible Galaxy*) to be available worldwide.

This *Ansible Role* contains one task for each point corresponding to the *CIS Benchmarks*. Each of these tasks contains tags that indicate the control, point and section of the *CIS Benchmarks* to which it belongs, as well as some extra tags that provide extra information such as software or elements affected by that task.

In the evaluation of the prototype of this benchmark, most of the *Ubuntu 18.04 CIS Benchmarks* security controls are currently implemented, except for some of them that require manual action and make automation difficult. In the future, the remaining security controls are expected to be developed.

#### 3.3.3. Egida Role Setup

Is an *Ansible Role* that is responsible for installing the *Egida API Worker* service and the tools or options necessary to ensure the correct behavior of *Egida* in the system.

#### 3.3.4. Egida API Worker

The *Egida API Worker* is a *gRPC* API installed on a *Worker* machine through the *Master* node. It works as a service to provide the *Egida Core* module with information about each machine we mentioned, such as data about running or stopped services, machine information, installed packages or results of security audits using evaluation tools.

## 4. Evaluation

In this section we will evaluate our *Egida* proposal with some use cases. Subsequently, we will perform an analysis and discussion of the results obtained. All measurements and tests have been performed using a 64-bit *Ubuntu 18.04 LTS* and its corresponding *CIS Benchmark*. More *CIS Benchmarks*, covering other OS and/or products, will be incorporated in the future.

### 4.1. Use cases

In order to evaluate the *Egida* system, we have defined 3 use cases with several functionalities that should be fulfilled by the system.

#### 4.1.1. Automated and customized system hardening profiles

The system should be able to automatically run all the implemented security controls of the mentioned *CIS Benchmark*. It must also allow users to customize the hardening to be performed. To check this, we have defined four tests:

1. *Automatic execution of all available security controls*. To test how this improves *Ubuntu* security, we will use several audit tools of Section 2.3 to obtain an overall security score. These are: *Lynis*, *Chef InSpec*, and *OpenSCAP*, comparing the score obtained before and after executing all CIS security controls implemented in *Egida*. This use case was chosen because *Egida* could not lose functionality over other similar tools that use *CIS Benchmarks* to improve the security of an *Ubuntu* OS.
2. *Execution of all tasks that belong to a single CIS Control*. To test this we will run all the benchmarks tied to an specific *CIS Control* and check if all of them have been run or there are missing ones. This facilitates implementing CIS *Implementation Groups* (see Section 2.2.3) with *Egida* and, thanks to the existing mapping of these controls (CIS, 2020c), also international frameworks like ISO 27001.
3. *Use of tags to run all security controls related to a topic* (e.g. `cron` or `sshd`). This facilitates applying only the security options of certain key services at the administrator discretion, letting the rest unmodified. This increased granularity facilitates not breaking legitimate functionalities with *Egida*.
4. *Change default values using variables*.

#### 4.1.2. Security operation customization according to the characteristics of the target machine

As we said in Section 3.2, our DSL allow the administrator to decide which security controls are applied on a machine based on its current state and characteristics. In this way, we can better model the experience of a trained system or security administrator according to these characteristics. To evaluate this, we have defined two different tests that need information from the target machine.

```
MAIN: {
        name: "Conditionals";
        connection: SSH
        description: "Aspida Example";
}

HOST: "192.168.56.1";

TASKS: {
        IF "services.ufw" == "stopped" {
                sections: ["1.1", "1.2"];
                exclusions: ["1.1.1.3"];
        }
        ELIF "hardscores.lynis" <= 50 {
                points: ["1.1.1.5"];
        }
        ELIF "services.apache" == "STOPPED" {
                points: ["1.1.1.4"];
        }
        ELSE {
                controls: ["9.4"];
        }
}
```

```
MAIN: {
        name: "SSH Config";
        connection: LOCAL
        description: "Aspida Test";
}
HOST: "localhost";
TASKS: {
        IF "machine.open_port" == 22 {
          sections: ["5.1", "5.3", "5.4"];
        }
        ELSE {
          sections: ["5.1", "5.2", "5.3", "5.4"];
        }
}
VARS: {
        "user_ssh": "Antonio";
        "password": {
                "max_days": 365;
                "min_days": 7;
                "warn_age": 7;
                "inactive": 30;
        };
        "nameservers": ["8.8.8.8", "8.8.4.4"];
}
```

**Fig. 3.** Aspida DSL examples.

1. *Change configuration based on target machine information* (services, packages, open/closed ports,...). To test this, we propose three possible realistic situations.
   (a) If the `apache` service is running and port 80 is in use, do not run *CIS Benchmark* security control 2.2.10. It may affect the correct operation of this web server, preventing breaking legitimate functionality.
   (b) If a GUI is detected, do not run *CIS Benchmark* security control 2.2.2, as it may affect the performance of the system's GUI. This is the same use case we outlined at the end of Section 2.2.1, also preventing breaking legitimate functionality for some users that require a GUI.
   (c) If `telnet` is being used and *SSH* is enabled, run the *CIS Benchmarks* security control 2.3.4 to disable `telnet`. This replaces an inherently insecure service with a more secure one, using administrator knowledge.
2. *Detect if the target system does not obtain a specific score through an audit of the Lynis tool*. This is useful because having machines complying a base security score is a common requirement for several infrastructures. If any of these machines fall behind the minimum expected security score, this will indicate that some kind of misconfiguration has happened, and it needs to be promptly fixed. Even worse, some kind of malware may be disabling security mechanisms as a previous step to perform a more complex attack, so it is necessary to take decisive action inmediately. Note that these kinds of operations is only possible with the *Egida* approach, and not with the other tools we reviewed. The rationale to include this use case is to show how "machine characteristics" can be more than just values to certain properties, but also the result of executing third-party' programs we decide to use.

#### 4.1.3. Early configuration errors detection

The DSL should detect at compile time possible configuration errors, problems with the deployment infrastructure, wrong variable values, and any other condition that may cause a run-time error. As we said, the difference with other tools we analyzed in Section 2 and *Ansible* is that *Egida* allows early detection of errors. This follows a similar approach and has the same benefits than other research projects with security tools (Redondo and Cuesta, 2019) or languages (Ortin et al., 2015). To evaluate this functionality, we have defined the following tests of each of the error detection procedures that our current DSL prototype implements at this moment:

1. Syntactic and lexical error detection.
2. Display warnings when the default value of a variable is used (its value has not been set and is going to be used).
3. Display Warnings when a *Level 2-Workstation* task is going to be executed.

**Table 1**
*Chef InSpec*, *OpenSCAP*, and *Lynis* results.

| Audit Tool | After/Before | Successful | Failures | Other | Score [%] |
|---|---|---|---|---|---|
| **Chef Inspec** | Before | 85 | 118 | 32 | 36.17 |
| | After | 151 | 53 | 31 | 64.25 |
| **OpenSCAP** | Before | 31 | 39 | 1 | 32.08 |
| | After | 52 | 18 | 1 | 71.46 |
| **Lynis** | Before | 78 | 72 | 0 | 52.00 |
| | After | 126 | 24 | 0 | 84.00 |

### 4.2. Evaluation results

In this section, we will detail the results of the of the defined use cases.

#### 4.2.1. Automated and customized system hardening profiles

To test the first section of this use case, all the security controls available in *Egida* (see see 3.3.2) were run, auditing the OS with the tools we mentioned in Section 4.1.1. These tools use different criteria to evaluate it, so the returned score may depend on the initial configuration of the system, installed services, defaults, or other features. To create a common baseline, we started with a default *Ubuntu* installation with its base configuration.

For the *Chef InSpec* tool we have used its *CIS Distribution Independent Linux Benchmark* (CIS, 2020a) profile, which implements the *CIS Distribution Independent Linux 2.0.0 Benchmark*. For the *OpenSCAP* audit tool we used its *CIS Benchmarks profile for Ubuntu 18.04 LTS* (`xccdf_org.ssgproject.content_profile_cis`). Finally, in the *Lynis* tool, we have taken the executed and not skipped tests and represented the warnings and suggestions as failures. The Table 1 shows these results.

After applying *Egida* to the machine, we have managed to increase the hardening score by 28% in the *Chef InSpec* tool, 39.38% in *OpenSCAP*, and 31% in *Lynis* audit tool. These results show how proper automated hardening of a machine with tried and tested validated controls (*CIS Benchmarks*) can greatly increase its security compared to its initial configuration with little effort, showing that automation is a way to facilitate increasing the overall security of an infrastructure.

**Table 2**
CIS Controls benchmarks expected vs executed.

| CIS Controls v7 security controls | Expected | Executed |
|---|---|---|
| **2.- Inventory and Control of Software Assets** | 8 | 8 |
| **3.-Continuous Vulnerability Management** | 6 | 6 |
| **4.-Controlled Use of Administrative Privileges** | 15 | 15 |
| **5.-Secure Configuration for Hardware and Software on Mobile Devices, Laptops, Workstations and Servers** | 52 | 52 |
| **6.-Maintenance, Monitoring and Analysis of Audit Logs** | 10 | 10 |
| **8.-Malware Defenses** | 3 | 3 |
| **9.-Limitation and Control of Network Ports, Protocols and Services** | 34 | 34 |
| **13.-Data Protection** | 1 | 1 |
| **14.-Controlled Access Based on the Need to Know** | 23 | 23 |
| **16.-Account Monitoring and Control** | 19 | 19 |
| **Total** | **171** | **171** |

```
1  ...
2  TASKS : {
3    tags: ["sshd"];
4  }
5  VARS : {
6    "user_ssh": "antonio";
7    "sshd_access": {
8      "ssh_port": 372
9    };
10 }
```

**Listing 2.** Aspida use Case 1.

To test the second section of this use case, we have executed all the security controls that belong to each of the *CIS Controls* implemented in *Egida* for *Ubuntu 18.04LTS* individually and checked if the amount corresponds to the expected one (see Table 2). All the expected security controls have been executed correctly, so *Egida* is able to successfully deploy all implemented security controls that compose each *CIS Control*, and therefore help to quickly implement the *Implementation Groups* defined by the CIS, as well as standards like ISO 27001 (CIS, 2020c). To check that the controls have been successfully applied, we used *Ansible* output. *Ansible* clearly informs when each automated operation has been successful or has failed, creating a very detailed and easy to read trace of its operations.

To test the last two sections of this use case we have developed a script (see Listing 2) in *Aspida* that performs all `ssh`-related tasks and changes the value of the default variables according to the *CIS Benchmarks* indications. As we see, it is very easy to isolate security controls from different services in separate units, so the administrators can decide which services they secure automatically, and which ones require further study to avoid breaking legitimate functionality, allowing more granularity in control application.

### *4.2.2. Security operation customization according to the characteristics of the target machine*

To test this use case, we have developed a script in *Aspida* that checks the state of the target machine and acts accordingly. As we can see in Listing 3, the language allows us to obtain information such as the current state of a service or the open ports. We use this script to implement all the use cases we outlined in Section 4.1.2. Therefore, we are checking if the *Apache* service is running and port 80 is in use (Line 3). If these two conditions are met, we do not execute its *CIS Benchmark* `2.2.10` (Line 5). In addition, if the target machine has an active GUI, we also add `2.2.2` to the exclusions (Line 9). In case the `telnet` package is installed and the *SSH* service is also running (Lines 11 and 12), we add `2.3.4` to the execution list as it disables `telnet` (Line 13). Finally, we check the

```
1  ...
2  TASKS : {
3    IF "services.apache" == "RUNNING" {
4      IF "machine.open_port" == 80 {
5        exclusions: ["2.2.10"];
6      }
7    }
8    IF "machine.gui" == true {
9      exclusions: ["2.2.2"];
10   }
11   IF "packages.telnet" == "INSTALLED" {
12     IF "services.ssh" == "RUNNING" {
13         points: ["2.3.4"];
14     }
15   }
16   IF "hardscores.lynis" <= 50 {
17     controls: ["2.6", "3.4", "3.5", "4.3",
           "4.4", "4.5", "4.8", "4.9", "5.1",
           "5.5", "8.3", "9.4"];
18   } ELIF "hardscores.lynis" <= 60 {
19     controls: ["5.1", "5.5", "8.3", "9.4"];
20   } ELSE {
21     controls: ["9.4"];
22   }
23 }
24 ...
```

**Listing 3.** Aspida Use Case 2.

score obtained on the target machine with *Lynis*, and we act depending on this score (Lines 16–22). If the score is found to be too low (less or equal to 50 points), we add 12 additional security controls of the *Ubuntu CIS Benchmark* that can improve the overall system security score to a minimum value we consider acceptable. Higher but not great scores (less or equal than 60 points) add 4 additional controls to reach this minimal security score. Finally, we ensure that at least security control 9.4 from the *Ubuntu CIS benchmark* is implemented.

As shown in Listing 3, making decisions based on the characteristics of the target machine gives us flexibility to develop scripts that allow to specialize machine hardening, applying all available security controls except those that interfere with its legitimate operation.

### *4.2.3. Early configuration errors detection*

Finally, to test this use case we have developed some scripts that use undefined variables, have lexical or semantic errors, or try to execute *Level 2-Workstation* tasks. Processing the script pro-

duces lexical and syntactic errors in lines 7 and 9 (missing `HOST` and `TASK` keywords), and also in line 11 (badly constructed `IF` condition). It also produces warnings for not defining the value of variable `ssh_access` and using its default value and, finally, reports a warning for using a Level-2 security control that may break system functionality.

As we said, detecting errors or potential problems at compile time (such as forgetting to change the value of the ports allowed by the firewall or setting the SSH access user) allows us, in addition to develop scripts with potentially less errors in less time, to avoid possible default configurations that may be vulnerable, or to avoid running security controls that may affect system operation.

## 5. Conclusions and future work

The proposed *Egida* system allows automated deployment of security configurations to components of an infrastructure, giving the administrators more flexibility and control. As a base of these configurations we use the tried and tested security controls of the *CIS Benchmarks*. The hardening of target machines can be customized according to their characteristics. This allow to implement security controls minimizing the possibility of interfering with their correct operation. In this way, different security profiles composed by these controls can be created better capturing the expertise of a security administrator.

To better capture this expertise, the *Domain Specific Language* of *Egida* obtains information about the characteristics of the target machine, and allows customizing the security controls to be applied. Additionally, the language processor performs the checks the statements to early detect different types of errors, thus decreasing the possibility of breaking the control application at runtime due to misconfigurations, as happens with other similar solutions. Therefore, with *Egida* we complement the work of projects such as SCAP or the *CIS Benchmarks* by adding customization, flexibility, and error prevention, based on the characteristics of each machine.

Once the first prototype of our research work as expected, we intend to improve the *Egida* API prototype in the future so we can obtain more information from the target machines. Some features that are being considered are: checking the existence of certain files, checking the permissions on directories and files, or checking the configuration values of known services. This enhanced knowledge of the target systems will improve the ability to customize *CIS Benchmarks* application, decreases the probability of finding runtime errors, and improve the ability of translating expert administrator knowledge to them, based on its current configuration.

We also want to improve the *Domain Specific Language* by including elements to facilitate code development. For example, the possibility of being able to print messages (error, info, and warning) or the declaration of variables. All these future changes are aimed to fully deploy this security automation technique over the real infrastructures of a company, capturing the required information of their running systems, and using the *Ansible* features to implement large-scale deployments.

This tool has been incorporated as part of the teaching materials of the *Computer Security* course of the *School of Computer Engineering* in the *University of Oviedo* (Redondo, 2021). We also intend to provide this tool to administrative personnel of the *University of Oviedo* in the future to collect usage feedback and see how they can share and model their knowledge. The ideas supporting this research were Top 10 finalist in the *I INNCYBER AWARDS* (2019). The next year, an evolution of these ideas (the ones presented in this paper) were Top 5 (out of 68 teams worldwide) in the 2nd edition of the same awards. All the source code of the *Egida* project and documentation of its use is available at https://egida-kassandra.github.io/egida.

## Declaration of Competing Interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

We confirm that the manuscript has been read and approved by all named authors and that there are no other persons who satisfied the criteria for authorship but are not listed. We further confirm that the order of authors listed in the manuscript has been approved by all of us.

We confirm that we have given due consideration to the protection of intellectual property associated with this work and that there are no impediments to publication, including the timing of publication, with respect to intellectual property. In so doing we confirm that we have followed the regulations of our institutions concerning intellectual property.

We understand that the Corresponding Author is the sole contact for the Editorial process (including Editorial Manager and direct communications with the office). He/she is responsible for communicating with the other authors about progress, submissions of revisions and final approval of proofs.

## CRediT authorship contribution statement

**Antonio Paya:** Software, Conceptualization, Methodology, Investigation. **Alba Cotarelo:** Writing – original draft, Visualization, Investigation. **Jose Manuel Redondo:** Supervision, Conceptualization, Writing – original draft, Writing – review & editing.

## References


Al-Safwani, N., Fazea, Y., Ibrahim, H., 2018. ISCP: in-depth model for selecting critical security controls. Comput. Secur. 77, 565–577. doi:10.1016/j.cose.2018.05.009.

Berger, B., 2005. Trusted computing group history. Inf. Secur. Tech. Rep. 10 (2), 59–62. doi:10.1016/j.istr.2005.05.007.

Bergmann, A., 2017. Simple XCCDF/OVAL example. Accessed: 2021-05-17. https://gist.github.com/abergmann/13e1ef5c0ad06a640f90aa8a9897644e.

Broderick, J.S., 2006. ISMS, security standards and security regulations. Inf. Secur. Tech. Rep. 11 (1), 26–31. doi:10.1016/j.istr.2005.12.001.

CIS, 2020a. CIS benchmarks. Accessed: 2021-05-20. https://github.com/dev-sec/cis-dil-benchmark.

CIS, 2020b. CIS Controls. Accessed: 2021-05-20. https://www.cisecurity.org/controls/.

CIS, 2020c. Mapping and compliance. Accessed: 2021-05-20. https://www.cisecurity.org/cybersecurity-tools/mapping-compliance.

Cleghorn, L., 2013. Network defense methodology: a comparison of defense in depth and defense in breadth. J. Inf. Secur. 4, 144–149. doi:10.4236/jis.2013.43017.

ComplianceAsCode, 2021. Compliance as code. Accessed: 2021-05-17. https://github.com/ComplianceAsCode/content.

Computer Security Division, I. T. L., 2020. Security content automation protocol: CSRC. Accessed: 2021-05-25. https://csrc.nist.gov/projects/security-content-automation-protocol.

Corporation, M., 2015. Open vulnerability and assessment language. Accessed: 2021-10-25. https://oval.mitre.org/language/index.html.

Cyberx-Mw, Knowlton, 2020. Security technical implementation guides (STIGs). Accessed: 2021-05-17. https://public.cyber.mil/stigs.

Durkota, K., Lisý, V., Bošanský, B., Kiekintveld, C., Pěchouček, M., 2019. Hardening networks against strategic attackers using attack graph games. Comput. Secur. 87, 101578. doi:10.1016/j.cose.2019.101578.

Kuipers, D., Fabro, M., 2006. Control systems cyber security: defense in depth strategies. Technical Report. Idaho National Laboratory (INL).

Mell, P., Scarfone, K., 2010. The Common Configuration Scoring System (CCSS): metrics for software security configuration vulnerabilities. doi:10.6028/NIST.IR.7502.

NIST, 2021. eXtensible Configuration Checklist Description Format (XCCDF). Accessed: 2021-10-25. https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/xccdf.

Olenčin, M., Perháč, J., 2019. Automated configuration of a Linux web server security. In: Proceedings of the IEEE 15th International Scientific Conference on Informatics, pp. 000491–000496. doi:10.1109/Informatics47936.2019.9119272.

OpenSCAP, 2020. OpenSCAP portal. Accessed: 2021-05-20. http://www.open-scap.org/.

Ortin, F., Schofield, J., Redondo, J.M., 2015. Towards a static type checker for Python. In: Proceedings of the ECOOP 2015. STOP 2015 Workshop.

Redondo, J.M., 2021. Improving concept learning through specialized digital fanzines. In: Proceedings of the IEEE/ACM 43rd International Conference on

Software Engineering: Software Engineering Education and Training (ICSE-SEET), pp. 134–143. doi:10.1109/ICSE-SEET52601.2021.00023.
Redondo, J.M., Cuesta, D., 2019. Towards improving productivity in Nmap security audits. J. Web Eng. 18 (7), 539–578. doi:10.13052/jwe1540-9589.1871.
Sager, T., 2021. Secure configurations and the power of SCAP. Accessed: 2021-05-17. https://www.cisecurity.org/blog/secure-configurations-and-the-power-of-scap/.
Shamma, B., et al., 2018. Implementing CIS critical security controls for organizations on a low-budget. University of Houston. Ph.D. thesis.
Software, I. J. S., systems engineering, 2020. ISO/IEC 19770-8:2020. Accessed: 2021-05-12. https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/25/72588.html.
Soto, J., 2019. JShielder. Accessed: 2021-05-26. https://github.com/Jsitech/JShielder.
Spichkova, M., Li, B., Porter, L., Mason, L., Lyu, Y., Weng, Y., 2020. VM2: automated security configuration and testing of virtual machine images. Proc. Comput. Sci. 176, 3610–3617. doi:10.1016/j.procs.2020.09.025.
Stytz, M., 2004. Considering defense in depth for software applications. IEEE Secur. Privacy 2 (1), 72–75. doi:10.1109/MSECP.2004.1264860.
Waltermire, D., Quinn, S., Booth, H., Scarfone, K., Prisaca, D., 2018. The technical specification for the Security Content Automation Protocol (SCAP) version 1.3. doi:10.6028/NIST.SP.800-126r3.
Weaver, R., Weaver, D., Farwood, D., 2013. Guide to Network Defense and Countermeasures. Cengage Learning.
Winarno, H., Yasin, F., Prasetyo, M.A., Rohman, F., Shihab, M.R., Ranti, B., 2020. IT infrastructure security risk assessment using the Center for Internet Security Critical Security Control framework: a case study at insurance company. In: Proceedings of the 3rd International Conference on Computer and Informatics Engineering (IC2IE), pp. 404–409. doi:10.1109/IC2IE50715.2020.9274594.

**Antonio Payá González** has a Web Engineering Master Degree in the University of Oviedo, and works at ArcelorMittal R&D Asturias and ICUBE SL in the development of artificial intelligence solutions in the Additive Manufacturing field. He is also working on the implementation of secure network architectures for an additive manufacturing pilot plant and implementing new ways to automate the deployment of security configurations. He has currently started his Ph.D. studies.

**Alba Cotarelo** has a Web Engineering Master Degree in the University of Oviedo, and works with Machine Learning algorithms at Ingenica STS and ArcelorMittal R&D Asturias. She also worked in Artificial Intelligence at Model-Driven Engineering Research Group in Universidad de Oviedo. She has currently started her PhD studies.

**Jose Manuel Redondo Lopez** is a Tenured Associate Professor in the University of Oviedo, Spain. Received his B.Sc., M.Sc., and Ph.D. degrees in computer engineering from the same university in 2000, 2002, and 2007, respectively. He participated in various research projects funded by Microsoft Research and the Spanish Department of Science and Innovation. He has authored three books and over 20 articles. His research interests include computer security, dynamic languages and computational reflection.