HAL
archives-ouvertes.fr

Challenges to automating security configuration checklists in manufacturing environments

Joshua Lubell, Timothy Zimmerman

▶ **To cite this version:**

Joshua Lubell, Timothy Zimmerman. Challenges to automating security configuration checklists in manufacturing environments. 11th International Conference on Critical Infrastructure Protection (ICCIP), Mar 2017, Arlington, VA, United States. pp.225-241, 10.1007/978-3-319-70395-4_12 . hal-01819144

HAL Id: hal-01819144
https://hal.inria.fr/hal-01819144

Submitted on 20 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

Chapter 12

CHALLENGES TO AUTOMATING SECURITY CONFIGURATION CHECKLISTS IN MANUFACTURING ENVIRONMENTS

Joshua Lubell and Timothy Zimmerman

Abstract Information technology is essential to today's manufacturing systems, but it makes them more vulnerable to cyber security threats than ever before. This chapter discusses the challenges to developing automatable configuration checklists for manufacturing environments using the Security Content Automation Protocol (SCAP) family of standards. Increased use of SCAP in manufacturing environments could reduce security vulnerabilities and the likelihood of damaging cyber attacks on manufacturing systems. However, complex relationships and dependencies within and between checklist rules, checking instructions and software result in platform fragmentation. Platform fragmentation makes it difficult to reuse or repurpose existing SCAP-expressed checklist content. Recent research and technological developments can be leveraged to yield potentially promising approaches for mitigating platform fragmentation and improving reuse.

Keywords: Manufacturing environments, control systems, security, checklists

1. Introduction

Information technology is essential to today's manufacturing systems. Microprocessors, software, data repositories, networking protocols and the Internet improve product quality, increase throughput and reduce production costs. But the increased reliance on information technology makes manufacturing environments more vulnerable to cyber security threats than ever before. A cyber attack can cause a loss of confidentiality, data corruption and costly downtime. An additional consequence for manufacturing systems – as with Stuxnet [3] and the German steel mill cyber attacks of 2014 [16] – is extensive physical damage to equipment and the surrounding environment. Physical and environmental

damage can, in turn, result in personal injury or death as well as large financial losses.

Proper security configurations reduce the likelihood of a cyber attack compromising a system, with the added benefit of protecting against incidents caused by (non-malicious) human errors. Configuration settings such as password and remote access policies, authorizing only the minimum access needed by users or processes to accomplish assigned tasks (least privilege principle) and restricting communications between subsystems and external systems (boundary protection principle) help improve the security posture [6]. The least privilege and boundary protection principles are included in the hundreds of security controls specified in National Institute of Standards and Technology (NIST) Special Publication 800-53, Recommended Security Controls for Federal Information Systems and Organizations [22].

The U.S. Department of Homeland Security 2016 Industrial Control Systems Assessment Summary Report [21] underscores the importance of proper configurations of control systems in manufacturing and industrial environments. According to the report, among the most common areas of weakness are the lack of adherence to the least privilege and boundary protection principles, poor authentication mechanisms and weak passwords. The right configuration settings can mitigate these vulnerabilities.

The U.S. National Checklist Program [31] defines a security configuration checklist – also referred to as a hardening guide or security benchmark – as "a series of instructions or procedures for configuring an information technology product to a particular operational environment, for verifying that the product has been configured properly and/or for identifying unauthorized changes to the product." Because such checklists provide guidance that is both concrete and actionable, they are especially useful to smaller companies and organizations that lack in-house cyber security expertise. A checklist may be automatable so that it can be used as digital input to a software tool that reports how well a system is configured with respect to the checklist's guidance. Automated security configuration checking is highly desirable because it reduces the costs of maintaining security and documenting the extent of compliance with a security policy.

To facilitate automation, a checklist should contain structured, computer-interpretable digital data. Furthermore, the tool that interprets the checklist should implement the data models used in the checklist. In other words, in order for a tool to automate a checklist, the tool should parse and process the data structures used in the checklist's digital representation. In this manner, the structured information in an automatable checklist serves as a "digital thread" [10] for cyber security [17] that integrates configuration guidance with system-specific settings. Additionally, the digital thread can associate a configuration setting with a specific security control or requirement, thereby linking the setting to its rationale. An automatable checklist with data structures that provide computer-interpretable configuration instructions and traceability

to security requirements is extremely useful. Such a checklist enables a longer and broader cyber security digital thread than a checklist that lacks these items.

This chapter discusses the challenges to developing automatable checklists for manufacturing environments using the Security Content Automation Protocol (SCAP – pronounced "ess-cap") family of standards [30, 32]. The standards specify a means for representing checklist information in a manner that an SCAP-conforming software tool can determine how well the system configuration complies with the checklist. SCAP is widely used in government [24] as well as by private sector enterprises to manage configuration compliance of common operating systems and software applications. Greater use of SCAP in manufacturing environments can reduce security vulnerabilities and the likelihood of damaging cyber attacks on manufacturing systems.

2. Manufacturing Environments

Modern manufacturing environments use industrial control systems to safely and reliably operate industrial processes. At the front line of these operations are programmable logic controllers (PLCs), which are computers developed specifically to monitor and control industrial processes [19]. Early generations of programmable logic controllers were intended to operate in isolated environments with custom operating systems and proprietary communications protocols. In contrast, modern programmable logic controllers incorporate information technologies such as TCP/IP and Ethernet for communications, USB ports for file transfers and commodity operating systems for efficiency, flexibility and convenience. The technologies enable greater visibility of manufacturing processes, the incorporation of real-time manufacturing data in corporate business systems, interoperability with existing networking infrastructures and remote monitoring capabilities [33]. However, the technologies also expose programmable logic controllers to greater cyber security risks.

Programmable logic controllers must be able to audit their configurations in order to detect potential vulnerabilities before they are exploited as attack vectors. Many attack vectors are introduced by enabled-by-default programmable logic controller features and services (e.g., embedded web servers, terminal servers and remote access servers). These features and services may only be detected after a thorough review of cyber-security-hardening guidance from vendors or discovery through active-scanning techniques employed by penetration testers.

The integration of information technologies directly in industrial control systems and manufacturing environments is now a normal occurrence. This is exemplified in the NIST ICS Cybersecurity Testbed [4, 41]. The testbed contains two robotic arms that emulate a material handling application; the robots are integrated into simulated manufacturing machines to perform repetitive tasks normally performed by a human operator (e.g., insert raw parts, remove finished parts, operate machine guarding and start and stop the machining cycle). The manufacturing machines communicate with the robot controllers via TCP/IP

Figure 1. Robotic portion of the NIST ICS Cybersecurity Testbed.

and Ethernet to coordinate the loading and unloading of parts. Figure 1 shows an image of the robotic portion of the NIST testbed.

Software executing in dedicated external controllers controls each robot. The controllers, which are high-performance servers typically found in information technology environments, run the Robot Operating System (ROS) on top of Ubuntu Linux, enabling the robots to collaborate in performing machine tending tasks. Ubuntu is the most common deployment environment for ROS, a software framework that is widely used in robotics research projects and increasingly in commercial robotic applications [7]. ROS-Industrial, a variant of ROS tailored for commercial applications, accelerates ROS deployment by augmenting its advanced manipulation capabilities with better support for reliability and safety [20, 34].

A motivating example is used to demonstrate the relationships between rules, security controls and a high-level security objective for an Ubuntu Linux server running ROS. Figure 2 shows two rules: one rule stipulates that a firewall should be set by default to deny all traffic (with all exceptions explicitly provided) and the other states that an AppArmor Linux kernel enhancement [1] should be enabled to restrict program access to system resources. AppArmor is commonly used in Ubuntu systems with stringent security requirements and

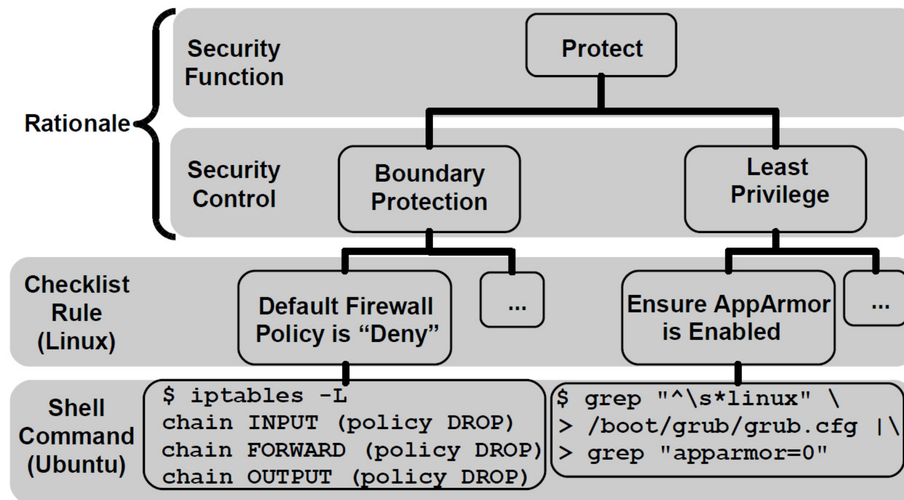

Figure 2. Checklist example.

its namespaced access control mechanism is a good match for the ROS data model [39]. The first rule supports boundary protection while the second rule supports least privilege. Both these security controls, in turn, support the high-level "protect" security function specified in the Cybersecurity Framework [23] – a methodology for managing cyber risk, describing an organization's current security posture and target state, and communicating and assessing progress toward meeting goals. Below each rule are shell commands for checking compliance.

Each level in Figure 2 is less general and more implementation-specific than the level above it. The top two levels describe security objectives and are independent of system-specific and implementation details. They provide the rationale for the rules in the checklist rule level. The checklist rule level applies to Linux systems, but it does not assume a particular Linux distribution. The shell command level applies only to Linux systems with the `iptables` package (pre-installed by default in Ubuntu distributions); `iptables` is an application that can be used to configure a Linux kernel firewall.

A corollary to the observation regarding levels and generality is that information in the higher levels is more reusable than information in the lower levels. For example, many rules in a wide variety of checklists support the least privilege security control. This is because least privilege is a universal principle that applies to many deployment situations. The AppArmor rule, however, is more specific. It applies only to Linux systems where the security benefit of AppArmor outweighs the convenience of programs having less-restrictive access to system resources. However, several Linux systems have specialized security limited functionality (SSLF) requirements [31], including the robot controller server that runs ROS. These systems have especially stringent security

requirements because of the threats they face and the potential consequences of incidents. Thus, the rule is reusable in many specialized security limited functionality contexts. However, the shell command that implements the rule is less reusable because it assumes the presence of `iptables` instead of an alternative firewall configuration application.

3. SCAP Background

An automated checklist has limited value if it is hardwired to a particular configuration tool or scripting language. Standards for representing rules, system settings, vulnerabilities, platforms and other relevant information enable checklists to be interoperable. Interoperable checklists can be used with any standards-compliant tool. Interoperability standards save checklist developers the trouble of having to learn multiple proprietary formats and lower the barriers to automated configuration checking. To address the need for interoperability, the cyber security research, development and user communities have created several Extensible Markup Language (XML) [40] data representation and exchange standards for software weaknesses and vulnerabilities, naming conventions, system state, configuration checklists, asset identification and severity measurement of software and configuration issues [18]. SCAP provides the recommended practices for using these standards together [30, 32].

SCAP is commonly used to automate security configuration compliance checking, which is the focus of this research. The following SCAP languages and taxonomies are especially relevant to the example in Figure 2 and the upcoming discussion:

- **Extensible Configuration Checklist Description Format (XCCDF):** This language is used to express security checklists, benchmarks and other configuration recommendations. XCCDF can represent the highly structured data needed for automated configuration checking as well as the semi-structured data needed to produce human-readable documentation of a checklist and the results of checking a system configuration.

- **Common Platform Enumeration (CPE):** This naming scheme provides unique identifiers for hardware, operating systems and applications.

- **Common Configuration Enumeration (CCE):** This registry maintains unique identifiers for operating system and software security configurations. SCAP checklists can use mappings from common configuration enumeration values to taxonomies of security principles or business objectives, providing traceability from configuration settings to higher-level requirements.

- **Open Vulnerability and Assessment Language (OVAL):** This language is used to express system configuration information, assess machine state and report assessment results. OVAL is widely used in the cyber security community as part of SCAP as well as with other standards [13].

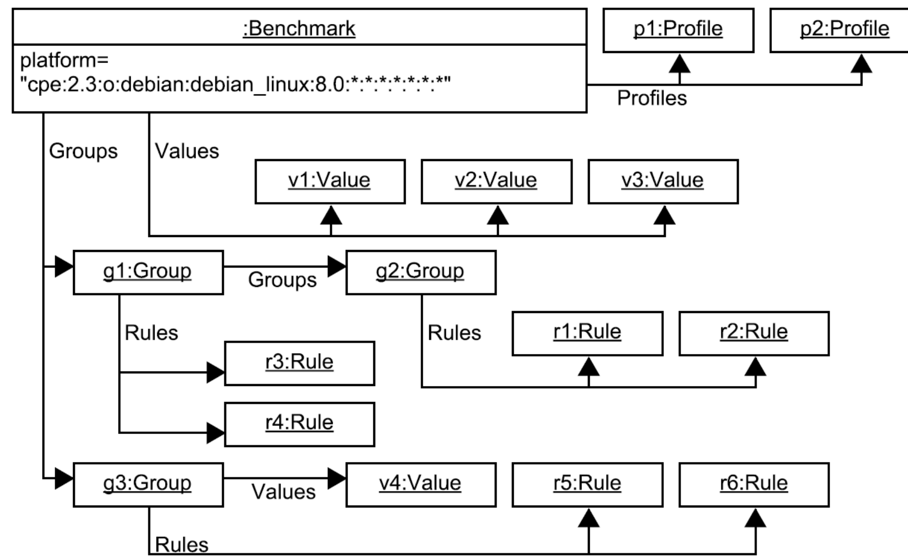

Figure 3. Example of an XCCDF benchmark XML document.

> Many hardware and software vendors produce OVAL content, which they make available to their customers directly and through third-party online repositories. The OVAL data model is versatile and complex; interested readers are referred to [28] for details about the data model

The XCCDF data model [38] represents a checklist document as a `Benchmark` object. A `Benchmark` is a collection of `Rule`, `Value` and `Group` objects. A `Rule` specifies a single item to check, such as the default setting of a firewall. A `Rule` also specifies how the checking should be done, such as with an implementation-specific scripting language or with the OVAL standard. A `Value` represents a named quantity that can be used in a rule and is tailored to a particular configuration scenario. A `Group` collects `Rule`, `Value` and other `Group` objects into an aggregation that is meaningful to a checklist user (e.g., a collection of firewall configuration settings). An XCCDF `Benchmark` also contains one or more `Profile` objects. A `Profile` is a named collection that references `Group`, `Rule` and `Value` objects. For example, an XCCDF Ubuntu checklist may have three profiles: one for single-user desktop systems, another for file servers and a third for specialized security limited functionality systems.

Figure 3 shows an example of an XCCDF checklist, which is based on an example from the XCCDF specification [38]. The Unified Modeling Language (UML) [26] object notation is employed. The checklist applies to the Debian Linux distribution version 8.0, as indicated by the common platform enumeration value assigned to the `platform` attribute of the `Benchmark` object. The checklist has two profiles, p1 and p2, each of which references a subset (omitted from Figure 3 for simplicity) of groups g1, g2 and g3 and values v1, v2 and v3.

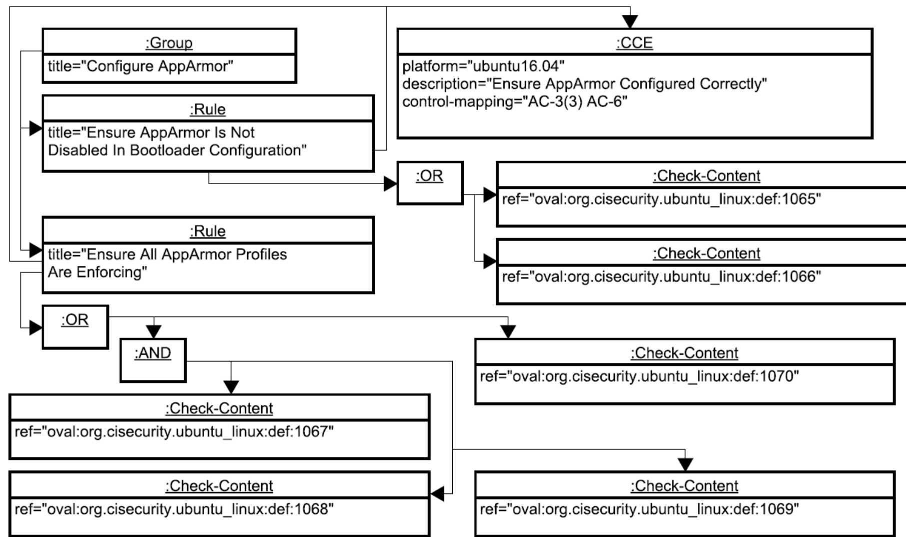

Figure 4. Example AppArmor rule in XCCDF.

In the figure, g1 aggregates rules r3 and r4; g2 aggregates rules r1 and r2; and g3 aggregates value v4 and rules r5 and r6.

The UML object model in Figure 4 shows how XCCDF is used to express the rules for checking that AppArmor is configured correctly in the Ubuntu scenario shown in Figure 2. Since the actual XCCDF rule representation would have more objects and would be harder to understand, Figure 4 shows a simplified version of the rule representation. Interested readers are referred to [5, 27] for details about an actual XCCDF XML representation. The `CCE` object in the upper-right corner of Figure 4 maps to two NIST Special Publication 800-53 security controls from the Access Control (AC) family, AC-3(3) and AC-6, both of which support the least privilege principle. The two `Rule` objects that support the `CCE` each reference Boolean expressions involving `Check-Content` objects. Each `Check-Content` object references a specific Ubuntu Linux OVAL definition in an external location. SCAP-conforming configuration scanner tools must support OVAL as a checking system. Support for other checking systems, such as those based on Linux shell command languages, is optional. To facilitate interoperability, the use of OVAL in checklists is preferable to non-SCAP checking systems.

The example in Figure 4 also illustrates the complexity of SCAP content. Each rule requires multiple OVAL definitions. Each OVAL definition, in turn, requires additional OVAL objects that are not shown in Figure 4.

4. SCAP Reuse in Manufacturing Environments

A major benefit is that an SCAP-expressed checklist can be used with multiple software tools and can be presented in multiple ways to users depending on

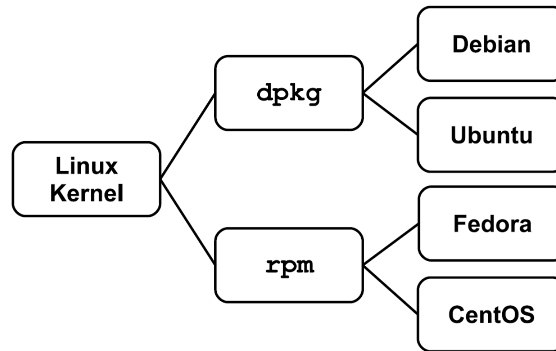

Figure 5. Shared components of Linux distributions.

their desire for details or need to know. Furthermore, XCCDF enables checklist developers to modularize content into groups and profiles, and enables checklist users to create tailoring files based on profiles. These capabilities, along with various online XCCDF and OVAL repositories, benefit SCAP checklist developers and users.

However, SCAP is not as helpful at facilitating the reuse of content applicable to multiple platforms. Consider, for example, the firewall rule in Figure 2, which uses `iptables` to check that the default policy is configured properly. The rule assumes the presence of `iptables` and, therefore, requires another rule to verify that `iptables` is installed. For Ubuntu Linux, the shell command `dpkg -s iptables` could implement such a rule. The `dpkg` package manager `-s` command option determines the status of a package (i.e., whether or not it is installed). However, `dpkg` is not universal across Linux distributions. Linux distributions based on the Debian Linux distribution, such as Ubuntu, use `dpkg`. Other Linux distributions use different package managers. For example, Fedora and CentOS use `rpm` for package management. Figure 5 shows the relationships between the Debian, Ubuntu, Fedora and CentOS Linux distributions in terms of their shared kernel and package manager components.

This example illustrates the problem of platform fragmentation. Platform fragmentation occurs when the same operating system, software application and/or hardware component are bundled by multiple entities, with each bundler providing different customizations [35]. A real-world example that demonstrates how Linux platform fragmentation complicates SCAP checklist development is the SCAP Security Guide (SSG) Project [27]. The project goal is to produce SCAP content (security guides) for a variety of Linux platforms. The SCAP Security Guide developers deal with platform fragmentation by splitting their source code into pieces that can be shared by multiple security guides versus pieces that are specific to a security guide. Building security guides from the source requires running scripts that perform XML transformations, macro substitutions and merging of source files into larger files. The build process is complicated and requires the SCAP Security Guide developers to

understand not only SCAP, but also the one-off manner in which the source files are organized and structured [29].

The SCAP platform fragmentation problem is not just limited to Linux. For example, consider the effort needed to develop a checklist for a programmable logic controller. A programmable logic controller may include an operating system (developed by a third party but customized by the vendor) in addition to automation software. Each of these components may be vulnerable to cyber attacks. The NIST ICS Cybersecurity Testbed has a Beckhoff CX9020 programmable logic controller [2] that runs the Windows Embedded Compact operating system (which Beckhoff licenses from Microsoft and has customized) and Beckhoff's TwinCAT automation software. It is actually less challenging to automate the security configuration of a Beckhoff programmable logic controller than that of many other programmable logic controllers with hardware-optimized, vendor-accessible-only operating systems. For these programmable logic controllers, there is no way for even an advanced user to deploy an automated configuration compliance checker. Indeed, only the vendor can alter the operating-system-level configuration settings.

A security professional tasked with developing an SCAP checklist for a Beckhoff CX9020 programmable logic controller might begin by reviewing the ICS-CERT Advisory ICSA-16-278-02 [12]. This document provides guidance for mitigating the vulnerabilities associated with the Windows Embedded Compact operating system and TwinCAT components of the programmable logic controller. Since it is easier to reuse existing SCAP XML content than to create new content from scratch, the security professional could search for XCCDF and OVAL content that is applicable to Windows Embedded Compact and TwinCAT. Unfortunately for the security professional, no registered common platform enumeration identifiers exist for Windows Embedded Compact and TwinCAT. However, SCAP content may exist for a third-party software library used by TwinCAT and Windows Embedded Compact has a number of components in common with Windows 7 (for which a great deal of SCAP content exists). In such a situation, it would be beneficial if the security professional could determine which, if any, of the existing SCAP content could be easily repurposed to automate the ICSA-16-278-02 guidance. Unfortunately, the existing SCAP content lacks the metadata needed to make such a determination.

5. Relevant Research and Standards

In addition to the SCAP Security Guide Project, other recent and ongoing research, implementation and standardization efforts have proposed solutions for coping with fragmentation and promoting reuse. The solutions can be categorized as employing: (i) information modeling; (ii) document-focused; (iii) centralized; or (iv) content-focused approaches.

Fitzgerald and Foley [8] have analyzed SCAP content from several repositories, classified the types of inconsistencies that create ambiguity and impede reuse and provided examples of implicit relationships. Their analysis focused on

the OVAL language and common platform enumeration and common configuration enumeration taxonomies, but did not include XCCDF content. They then developed an SCAP ontology employing semantic threat graphs and demonstrated how the SCAP ontology addresses inconsistency challenges by making implicit relationships explicit.

Other researchers have pushed the envelope of SCAP deployment by developing checklists for platforms beyond the usual SCAP realm of desktop operating systems, Internet browsers and office applications. Hlyne et al. [11] have developed a configuration checklist for Cisco routers, which uses OVAL content developed by Cisco and is deployed using the jOval SCAP configuration scanning tool. Kuo and Yang [15] have created an SCAP configuration checklist for Android devices and a configuration management tool using the jOval scanning engine; the tool runs on a server and performs remote scanning and remediation of misconfigurations. Kuo and Yang have used as their information source prose text (without SCAP content) benchmark documentation from the Center for Internet Security (CIS) [9]. CIS benchmarks [5] are available for a variety of operating systems, software environments and network devices. Although many CIS benchmarks are in prose form and not expressed using SCAP, an increasing number are now available to CIS members in the XCCDF and OVAL formats.

Vecchiato et al. [35] have studied configuration assessment data from more than 500 Android smartphones and have found several recurring misconfigurations, the most common being weak passwords and overly-permissive network settings. Android device configuration compliance is an interesting SCAP use case because its platform fragmentation challenges parallel those encountered in manufacturing environments. In comparing Android with common desktop and laptop operating systems, Vecchiato and colleagues characterize Android smartphones as being more personalized to consumer preferences and having lesser capabilities due to their reduced size and other physical constraints. These characteristics are similar to those in manufacturing environments, where hardware and information technology capabilities have to meet requirements unique to the production scenarios and environmental conditions. Furthermore, as in the case of the programmable logic controller example in Section 4, smartphone vendors customize Android devices with their own applications and carrier-specific settings. As a result, Android smartphone vulnerabilities and configuration issues can be hardware-vendor-dependent.

Vecchiato et al. [35] have also proposed an intriguing approach for reducing the fragmentation problem – that vendors and researchers work together to transition some Android device capabilities currently available only through vendors to services available through Google Play. The anticipated result of this approach is that vendors could leverage Google Play's automatic update mechanism, enabling third-party software developers and end users to receive new Android features and patches without having to wait for less-frequent Android updates from their carriers. The approach of Vecchiato and colleagues appears to be appealing in manufacturing environments because it addresses a

significant obstacle to automating configuration checking of industrial control devices such as programmable logic controllers – the lack of software interfaces for third-party access to the underlying operating system information. Of course, vendor-provided tools employed to program the devices may enable users to issue queries required to support the auditing of programmable logic controllers, but it is currently not possible to use these tools to conduct automated audits.

An adaptation of the idea of Vecchiato and colleagues could help mitigate the fragmentation problem in manufacturing environments. Specifically, vendors of programmable logic controllers, switches and routers could make their firmware, operating system and other software updates available through centralized trusted digital distribution services. However, unlike Android smartphones, automatically updating industrial control devices raises significant concerns. The availability of the devices and the safety of their operations are paramount. Enabling a third-party developer to update a device automatically without end-user intervention could have catastrophic consequences, especially if the update occurs during production or software bugs are present in the update. However, with the right modifications to address industrial control system availability and safety requirements, an adapted approach might be acceptable. Indeed, the increasing use of commodity operating systems in programmable logic controllers and other devices provides new opportunities for developers to implement automatic software updates as part of centralized services.

Software identification (SWID) tags [37], an international standard for describing software products, offer another potential solution to platform fragmentation. SWID tags use an XML format that allows for better-structured and information-richer content than provided by a common platform enumeration identifier. The common platform enumeration naming scheme is syntactic and does not provide explicit semantics, which leads to ambiguity [8]. In contrast, SWID tags can explicitly represent the unique identifier of a product as well as information about its versioning scheme, patch level, relationships to other software and a host of other useful metadata. Because they enable better software inventory management, SWID tags can also help detect system misconfigurations. Indeed, current SCAP recommended practices provide guidance for replacing common platform enumeration identifiers in SCAP content with SWID tags [36].

Another standard that could promote the reuse of SCAP content is the Darwin Information Typing Architecture (DITA) [25]. As the SCAP Security Guide Project's complicated build system illustrates, producing an SCAP checklist from a collection of components poses XML publishing and content management problems. However, as discussed in Section 4, the source code version control systems that projects such as the SCAP Security Guide typically use are file-based and inadequate for managing relationships between objects within a source XML file or relationships between files. Multiple publications (checklists) can share the same XCCDF or OVAL component. Additionally, a publication can use the same component in more than one context. Kimber [14]

Table 1. Research and development approaches, objectives and proposed solutions.

Approach	Objective	Solutions
Information Modeling	Mitigate platform fragmentation by improving the ability of SCAP to represent platform dependencies and relationships within and between XCCDF rules and OVAL definitions.	Fitzgerald and Foley [8]; SWID tags [37].
Document-Focused	Use XML-based methods to make SCAP authoring and content reuse easier.	DITA [14, 25]; SSG *ad hoc* build system [29].
Centralized	Reduce the likelihood of misconfigurations by centralizing software distribution and updates. Addresses the root cause of platform fragmentation instead of putting the onus on checklist authors. Requires coordination and trust between stakeholders and tailoring to meet the stringent operational and safety requirements of manufacturing environments.	Vecchiato et al. [35].
Content-Focused	Develop more XCCDF checklists, particularly for platforms where SCAP content is in short supply.	SSG [27]; Hlyne et al. [11]; Kuo and Yang [15]; CIS benchmarks [5].

has implemented a content management and publishing system that combines the capabilities of the Darwin Information Typing Architecture with the Git version control system to generate publications from a collection of interrelated version-controlled XML components. Such a standards-based strategy, if feasible for the SCAP Security Guide Project, would be less *ad hoc* and brittle than the current approach, which uses a mix of scripts, makefiles and manual searches within files [29].

Table 1 summarizes the objectives and the solutions of the research and development approaches discussed above.

6. Conclusions

This chapter has investigated the challenges to reusing existing SCAP content for checking configuration compliance of information technology components in manufacturing environments. An illustrative example using an Ubuntu Linux configuration checklist scenario demonstrates that platform fragmentation complicates the reuse of SCAP content. The same platform fragmentation exists in the industrial control system and Android device domains as well. A review and classification of related research and development approaches and the relevant standards reveal promising solutions that ameliorate platform fragmentation and encourage more SCAP deployments.

The need for cyber security solutions in manufacturing environments is growing. Although platform fragmentation is a barrier to SCAP deployment in the manufacturing sector, other areas – most notably Android mobile devices – share a number of the same issues. Recent and ongoing research and development results could lead to concrete gains in SCAP adoption in the manufacturing sector and in other areas, especially if the stakeholder communities work together and learn from each other.

Disclaimer

This chapter is a contribution of the National Institute of Standards and Technology (NIST). Certain commercial and third-party products and services are identified in this chapter to enhance understanding. Such identification does not imply any recommendation or endorsement by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

References

[1] M. Bauer, Paranoid Penguin: AppArmor in Ubuntu 9, *Linux Journal*, issue 185, September 1, 2009.

[2] Beckhoff Automation, Manual CX9020, Embedded PC, Version 1.8, Verl, Germany (`download.beckhoff.com/download/document/ipc/embedded-pc/embedded-pc-cx/cx9020_hwen.pdf`), 2017.

[3] E. Byres, A. Ginter and J. Langill, How Stuxnet Spreads – A Study of Infection Paths in Best Practice Systems, Version 1.0, Tofino Security, Lantzville, Canada, 2011.

[4] R. Candell, T. Zimmerman and K. Stouffer, An Industrial Control System Cybersecurity Performance Testbed, NISTIR 8089, National Institute of Standards and Technology, Gaithersburg, Maryland, 2015.

[5] Center for Internet Security, CIS Benchmarks, East Greenbush, New York (`benchmarks.cisecurity.org`), 2017.

[6] A. Creery and E. Byres, Industrial cybersecurity for power system and SCADA networks, *Proceedings of the Industry Applications Society Fifty-Second Annual Petroleum and Chemical Industry Conference*, pp. 303–309, 2005.

[7] C. Fairchild and T. Harman, *ROS Robotics by Example*, Packt Publishing, Birmingham, United Kingdom, 2016.

[8] W. Fitzgerald and S. Foley, Avoiding inconsistencies in the Security Content Automation Protocol, *Proceedings of the IEEE Conference on Communications and Network Security*, pp. 454–461, 2013.

[9] R. Fritz, CIS Google Android, Android 4 Benchmark V1.0.0, Center for Internet Security, East Greenbush, New York (`learn.cisecurity.org/benchmarks`), 2012.

[10] T. Hedberg, J. Lubell, L. Fischer, L. Maggiano and A. Feeney, Testing the digital thread in support of model-based manufacturing and inspection, *Journal of Computing and Information Science in Engineering*, vol. 16(2), 2016.

[11] C. Hlyne, P. Zavarsky and S. Butakov, SCAP benchmark for Cisco router security configuration compliance, *Proceedings of the Tenth International Conference on Internet Technology and Secured Transactions*, pp. 270–276, 2015.

[12] Industrial Control Systems Cyber Emergency Response Team (ICS-CERT), Advisory (ICSA-16-278-02), Beckhoff Embedded PC Images and TwinCAT Components Vulnerabilities, U.S. Department of Homeland Security, Washington, DC (`ics-cert.us-cert.gov/advisories/ICSA-16-278-02`), January 5, 2014.

[13] P. Kampanakis, Security automation and threat information-sharing options, *IEEE Security and Privacy*, vol. 12(5), pp. 42–51, 2014.

[14] E. Kimber, Hyperdocument authoring link management using Git and XQuery in service of an abstract hyperdocument management model applied to DITA hyperdocuments, *Proceedings of Balisage: The Markup Conference*, vol. 15, 2015.

[15] C. Kuo and C. Yang, Security design for configuration management of Android devices, *Proceedings of the Thirty-Ninth Annual Computer Software and Applications Conference*, vol. 3, pp. 249–254, 2015.

[16] R. Lee, M. Assante and T. Conway, German Steel Mill Cyber Attack, ICS CP/PE (Cyber-Physical or Process Effects), Case Study Paper, SANS Institute, Bethesda, Maryland (`ics.sans.org/media/ICS-CPPE-case-Study-2-German-Steelworks_Facility.pdf`), 2014.

[17] J. Lubell, Extending the cybersecurity digital thread with XForms, *Proceedings of Balisage: The Markup Conference*, vol. 15, 2015.

[18] G. McGuire and E. Reid, The State of Security Automation Standards – 2011, A Survey, MP110439, MITRE Corporation, Bedford, Massachusetts (`www.mitre.org/sites/default/files/pdf/11_3822.pdf`), 2011.

[19] S. McLaughlin, C. Konstantinou, X. Wang, L. Davi, A. Sadeghi, M. Maniatakos and R. Karri, The cybersecurity landscape in industrial control systems, *Proceedings of the IEEE*, vol. 104(5), pp. 1039–1057, 2016.

[20] M. Munaro, C. Lewis, D. Chambers, P. Hvass and E. Menegatti, RGB-D human detection and tracking for industrial environments, *Proceedings of the Thirteenth Conference on Intelligent Autonomous Systems*, pp. 1655–1668, 2014.

[21] National Cybersecurity and Communications Integration Center/Industrial Control Systems Cyber Emergency Response Team, NCCIC/ICS-CERT Industrial Control Systems Assessment Summary Report, FY 2015, U.S. Department of Homeland Security, Washington, DC, 2016.

[22] National Institute of Standards and Technology, Security and Privacy Controls for Federal Information Systems and Organizations, NIST Special Publication 800-53, Revision 4, Gaithersburg, Maryland, 2013.

[23] National Institute of Standards and Technology, Framework for Improving Critical Infrastructure Cybersecurity, Version 1.0, Gaithersburg, Maryland, 2014.

[24] National Institute of Standards and Technology, The United States Government Configuration Baseline (USGCB), Gaithersburg, Maryland (`usgcb.nist.gov`), 2017.

[25] OASIS, Darwin Information Typing Architecture (DITA) Version 1.3 Part 0: Overview (Plus Errata 01), OASIS Standard (Incorporating Approved Errata), Burlington, Massachusetts, 2016.

[26] Object Management Group, OMG Unified Modeling Language (OMG UML), Version 2.5, Document No. Formal/2015-03-01, Needham, Massachusetts, 2015.

[27] OpenSCAP, SCAP Security Guide (`www.open-scap.org/security-policies/scap-security-guide`), 2017.

[28] OVAL, OVAL Documentation (`ovalproject.github.io`), 2017.

[29] M. Preisler, Contributing to SCAP Security Guide – Part 1 (`martin.preisler.me/2016/10/contributing-to-scap-security-guide-part-1`), October 28, 2016.

[30] S. Quinn, K. Scarfone and D. Waltermire, Guide to Adopting and Using the Security Content Automation Protocol (SCAP), Version 1.2 (Draft), NIST Special Publication 800-117, Revision 1, National Institute of Standards and Technology, Gaithersburg, Maryland, 2012.

[31] S. Quinn, M. Souppaya, M. Cook and K. Scarfone, National Checklist Program for IT Products – Guidelines for Checklist Users and Developers, NIST Special Publication 800–70, Revision 3, National Institute of Standards and Technology, Gaithersburg, Maryland, 2015.

[32] S. Radack and R. Kuhn, Managing security: The Security Content Automation Protocol, *IT Professional*, vol. 13(1), pp. 9–11, 2011.

[33] A. Sadeghi, C. Wachsmann and M. Waidner, Security and privacy challenges in the industrial Internet of Things, *Proceedings of the Fifty-Second ACM/EDAC/IEEE Design Automation Conference*, 2015.

[34] SwRI Manufacturing Technologies, ROS-Industrial, San Antonio, Texas (`rosindustrial.org`), 2017.

[35] D. Vecchiato, M. Vieira and E. Martins, The perils of Android security configuration, *IEEE Computer*, vol. 49(6), pp. 15–21, 2016.

[36] D. Waltermire and B. Cheikes, Forming Common Platform Enumeration (CPE) Names from Software Identification (SWID) Tags, NISTIR 8085 (Draft), National Institute of Standards and Technology, Gaithersburg, Maryland, 2015.

[37] D. Waltermire, B. Cheikes, L. Feldman and G. Witte, Guidelines for the Creation of Interoperable Software Identification (SWID) Tags, NISTIR 8060, National Institute of Standards and Technology, Gaithersburg, Maryland, 2016.

[38] D. Waltermire, C. Schmidt, K. Scarfone and N. Ziring, Specification for the Extensible Configuration Checklist Description Format (XCCDF), Version 1.2, NISTIR 7275, Revision 4, National Institute of Standards and Technology, Gaithersburg, Maryland, 2012.

[39] R. White, H. Christensen and M. Quigley, SROS: Securing ROS over the wire, in the graph and through the kernel, presented at the *IEEE-RAS International Conference on Humanoid Robots*, 2016.

[40] World Wide Web Consortium, Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation, Massachusetts Institute of Technology, Cambridge, Massachusetts (`www.w3.org/TR/REC-xml`), 2008.

[41] T. Zimmerman, Metrics and Key Performance Indicators for Robotic Cybersecurity Performance Analysis, NISTIR 8177, National Institute of Standards and Technology, Gaithersburg, Maryland, 2017.