Checklist Usage in Secure Software Development

Zhongwei Teng, Jacob Tate, William Nock, Carlos Olea, Jules White

Vanderbilt University

Abstract

Checklists have been used to increase safety in aviation and help prevent mistakes in surgeries. However, despite the success of checklists in many domains, checklists have not been universally successful in improving safety. A large volume of checklists is being published online for helping software developers produce more secure code and avoid mistakes that lead to cyber-security vulnerabilities. It is not clear if these secure development checklists are an effective method of teaching developers to avoid cyber-security mistakes and reducing coding errors that introduce vulnerabilities. This paper presents in-process research looking at the secure coding checklists available online, how they map to well-known checklist formats investigated in prior human factors research, and unique pitfalls that some secure development checklists exhibit related to decidability, abstraction, and reuse.

Keywords

Checklists, Cyber Security, Software Development

1. Introduction

Checklists have become common in industries, such as aviation, where human errors can lead to significant safety issues. For example, NASA has published detailed design guidance on creating and using checklists for aircraft operation [1]. The World Health Organization (WHO) has begun encouraging hospitals to adopt surgical-safety checklists [2], based on the results of a research study conducted by surgical staff in 8 hospitals from 2007 to 2008 which showed checklists reduced complications and mortality rates [3]. By decomposing a complex system or workflow into a set of deterministic items, checklists can help to avoid common errors and free up mental capacity for important cognitive tasks.

The promising achievements of checklists in other fields have motivated software developers to publish their own checklists for secure software development, which can be easily found online. A Google search for "secure software development checklist" produces thousands of results. These checklists typically consist of items listing things that software developers should do to produce more secure software.

However, the security community needs to determine whether the benefits that checklists show in other domains carry over to secure software development, and if so, in what scenarios [4]. Moreover, the overall quality of the checklists that software engineers and their managers find online has not been assessed. Because of the sense of "completeness" that checking off items on a list can instill, it is critical that checklists be of high quality. For example, a mismatch between a checklist's target domain and an individual application's domain can make developers assume that they have "checked off all the critical security items," when in fact, they are ignoring critical risks in their codebase that emerge from unique aspects of their application domain. Checklists have a connotation of "complete-

Format	{Topic Keyword(s)} {Secondary Keyword(s)} security checklists		
Topic Keyword	**Secondary Keyword(s)**	**#Unique Checklists**	**#Tags in StackOverflow**
Web Development	—	9	492('web-applications'+'security') 708('web'+'security')
Python	Flask	5	248('flask-security') 79('flask'+'security')
Python	Django	7	367('django'+'security')
Python	Secure Coding	4	972('python'+'security')
Java	Spring	4	21773('spring-security')
Java	Android	4	430('android-security') 1084('android'+'security')
Java	Secure Coding	6	311('java-security') 5572('java'+'security')

Table 1: Search Terms for the Secure Coding Checklist Survey

ness" as shown in the Merriam Webster Dictionary definition of checklists:

```
a list of things to be checked or done
a pilot's checklist before takeoff
also : a comprehensive list
```

Online resources, such as StackOverflow, have become a commonly used resource for software developers to learn and gather information about best practices. Especially for small development teams that may not have substantial support from cybersecurity groups within their organization, checklists that developers or their managers find online may create a false sense of security. For example, when a checklist reminds web developers to avoid SQL injection attacks with a few examples, such as "Use prepared statements.", it is not guaranteed that readers of the checklist will fully understand the concept of prepared statements or how similar issues may arise when using object relational mapping systems. In this case, even though developers can show that their application passes a cybersecurity checklist, their underlying code may yet be highly insecure. Further, to outside non-experts, the knowledge that a checklist was followed (e.g., a secure process) may generate a more powerful belief in the security of the project than other artifacts.

In this paper, we explore whether or not checklists are truly effective in helping educate developers on how to produce more secure software. We explore the challenges of adopting security checklists in secure software development. In particular, we show that the scope of tasks where security checklists are effective in software development is much narrower than expected. This narrow scope is a result of significant differences in common tasks in software development and the types of tasks and contexts where checklists are known to be efficacious.

Paper organization. The remainder of this paper is organized as follows: Section introduces the survey that we conducted of online checklists. Section presents four key issues that we found when assessing a set of online secure development checklists. Section discusses the results of the survey and highlights key issues that were commonly found. Section introduces related work on checklists in other industries. Section provides concluding remarks and future work.

2. Survey of Cybersecurity Checklists found Online

Software developers looking to increase the security of their applications will often turn to the web for available resources and risks of seeking advice from web search are revealed by a survey [5]. Checklists can exist on programming-specific sites, blogs, or other sites. If a programmer is inspired to use a checklist to audit their work, these are the resources they will find. On a pragmatic level, this is the current state of cybersecurity checklists.

This survey provides initial data on the prevalence and potential failings of checklists related to secure coding and software implementation that a developer may find online. This analysis seeks to explore the extent to which checklists found online accommodate, or fall victim to, the proposed theoretical pitfalls described later in Section . The data should be considered a work in progress.

Using a variety of search keywords, which developers may use to address security concerns, secure software development checklists were located using the Google search engine. Searches were conducted using the incognito mode on Google Chrome, so that past searches did not influence results. The search queries for the survey were generated by concatenating a **Topic Keyword** & **Secondary Keywords**, with the term "Security Checklists". For example, as shown in Table 1, "Python Django Security Checklists" was one such query. For each query, only links that were on the first page (excluding advertisements) were evaluated, since it is estimated that 95% of search engine visitors click through to a link on the first page of Google search results [6].

Based on choices of programming applications and trends on Stack Overflow, we chose two topic keywords, "Python" and "Java", as well as corresponding programming frameworks as secondary keywords. The number of unique results, excluding ads, is shown in Table 1.

As shown, we analyzed 39 secure coding checklists in total. The application domain which we focused on was web application development. Selections of central keywords and auxiliary keywords are based on statistics of StackOverflow Trends. The checklists covered general security topics (web development security), two languages, including Python and Java, and four frameworks, such as DJango and Spring. General secure coding checklists for each topic keyword were also evaluated (e.g. "Java Security Checklist").

3. Challenges of Translating Checklists to Secure Software Development

It's an open question whether the benefits seen in other domains, such as from the WHO's surgery checklist, can be realized in cybersecurity to better educate software engineers. Despite this open question, there are a wide variety of cybersecurity checklists for software engineers available.

The most significant work on human factors considerations when designing checklists has come from the aviation and space domains [1, 7]. In general, checklist items are expected to fall into either a "read-do" format [7] or a "read-respond" (also known as "challenge-response") [1] format. Read-do items dictate actions that should be taken by the reader immediately after reading the item (e.g., circle incision site). Read-respond items are designed to be confirmed after reading (e.g., Is the incision site circled?).

We conducted research to understand the characteristics of the cybersecurity checklists for software engineers and understand how their items and scope compare to the basic best practices listed in prior research [1, 7]. Throughout the remainder of this section, we describe five key qualitative issues that many checklists suffer from and make creating clean "read-do" or "read-respond" items challenging in the secure coding domain. Read-do and read-respond items are the fundamental expected structure of checklist items. As described in human factors research on checklists in other domains, secure coding tasks that can't fit into a read-do or read-respond format aren't suited for a checklist. Secure software engineering checklists composed of items that fail to fit into the read-do or read-respond format should be approached with caution.

Based on observations of checklists from the survey conducted in Section , we identified repeated anti-patterns / challenges in secure coding checklists. Although they may appear sound, checklists that exhibit these patterns fundamentally violate past human factors research in best practices for creating usable checklists [1, 7]. We will discuss 4 main challenges for secure software development checklists throughout the remainder of this section.

3.1. Challenge: Non-deterministic Read-respond Items

Read-respond items in a checklist are meant to be deterministic in nature. Different users reading a "read-respond" item should be expected to always arrive at the same answer in the same situation. Read-respond items are constructed such that diverse users of the checklist interpret the item in the same manner, precisely and efficiently. For example, an aircraft maintenance checklist item that requires verifying that "the XYZ electrical lead wire does not exceed 30 metres in length", can be answered as "yes" or "no" in a consistent fashion. In contrast, if a checklist contains ambiguous items, such as "no electrical leads are too long", it's not precise enough in definition to be deterministically evaluated. One user may know the maximum length to be 30 meters, while another may be familiar with a different standard.

In many circumstances, however, security checklists in software development contain non-deterministic items, often statements warning developers to avoid a specific type of vulnerability, included with limited examples. For example, "Sensitive transactions require re authentication" [8] is a non-deterministic read-respond item. What is a "sensitive transaction"? Alternatively, the read-respond item "Session cookies are encrypted and have a length of at least 128 bits, are complex", provides concrete deterministic guidance on the encryption and length, but provides non-deterministic guidance on "complex". Two developers may arrive at different conclusions on cookie complexity.

When secure coding checklist items are non-deterministic, a number of problems arise. First, an item may be checked off simply because the developer doesn't really understand the original intent of the question. Second, different developers on the same team may apply varying security standards to their source code analysis / production, leading to inconsistent security. Finally, since checklists instill a sense of security / completeness, checking off items that can't consistently be answered / applied is potentially dangerous.

3.2. Challenge: Undecidable Read-respond Items

The Halting Problem and Rice's Theorem [9, 10] are well known theoretical proofs that inform the limits of what we can know about an arbitrary software application. Rice's Theorem shows that we cannot decide most semantic properties of arbitrary software systems, such as "their security." A fundamental problem in many secure coding checklists is that they use read-respond items that would violate Rice's Theorem or the Halting Problem if they could be answered for arbitrary software applications.

For example, a web development security checklist item warns developers "Do not leak session IDs" [11]. Given an arbitrary web application that uses session IDs, this question is likely undecidable. A similar read-respond item would be "do not create inputs that prevent the application from halting", which is likely not helpful to a software engineer.

Although these types of items appear as helpful reminders to developers, they are unhelpful when delivered in a checklist format. A web development security guide that explains what leaking a session ID is and provides educational examples is a better format for delivering this material. However, this type of material does not fit into a read-respond or read-do checklist item. Unfortunately, declarative statements that express what they want accomplished, but not how, tend to be undecidable in security checklists. To fit into a *decidable* checklist item, the specific steps on how to do what is stated need to be expressed and each step decidable in nature. Checklist items that state "don't introduce security vulnerability X" are generally undecidable due to Rice's Theorem.

3.3. Challenge: Failed Generalization for Reuse in Read-respond Questions

Checklists are built on the idea of summarizing only essential critical activities or information into a short and actionable list to make sure that people consistently and correctly perform these tasks. However, if we transfer an effective checklist from one context to a new context, due to differences in architecture, software stacks, etc., there is a significant risk of introducing non-determinism or irrelevant items. For example, checking for double freeing of memory may make sense in one programming language, but not make sense in another language, leading to the introduction of unnecessary items that distract the user.

In software development, however, it is uncommon to create a checklist for exactly one product – checklists are almost always abstracted and generalized like software to facilitate reuse. Checklists are usually designed for a domain, a framework, or a programming language, such as mobile application/Android/Java. These checklists are then implemented by developers appropriately within the new product's software architecture. The issue then arises: generalizing the use of these domain- or architecture-specific checklists relies on the assumption that their content will be transferable to other software domains.

There is a tension in developing an item for a cybersecurity checklist between the reusability of the checklist and the specificity of the items. Ideally, a checklist can be carefully crafted and reused, just like we do with software components. This desire for reuse leads checklist authors into the trap of creating non-deterministic read-respond items. The greater the number of non-deterministic read-respond items, the more likely that the checklist will be used incorrectly **AND** confer incorrect trust in the security of the system.

For example, the checklist item "Do not use eval() and similar functions" [11], is somewhat deterministic / decidable in the Javascript context, where eval() is a built-in function that source code can be checked for. However, the "similar functions" language is trying to facilitate use of this item in other contexts, possibly outside of Javascript – simply map eval() to the equivalent function in your context – which is not clearly deterministic or decidable. The goal of making the item reusable outside of Javascript with "similar functions" language creates non-determinism. Contrast this item with the checklist item "If I handle XML files, I disable entity and DTD processing" [12]. This checklist item provides more detail on what needs to be accomplished and but is still not necessarily decidable for an arbitrary language and set of frameworks.

3.4. Challenge: Missing Description of the Expected Checklist Context

Compared to daily-use checklists, such as shopping or errand lists, checklists for most non-trivial domains have explicit linkages between the items and domain concepts, such

Topic Keywords	Web development	Python	Java
%Determinism Issues	88.9	75	57.1
%Undecidable	100	87.5	57.1
%Improper Generalization	33.3	100	35.7
% Lacking Context	55.6	50	42.9

Table 2: Results from Secure Coding Checklist Survey

as aircraft wing flaps, ailerons, etc. Moreover, the checklists implicitly expect users to understand key concepts in the application domain, such as Australian aircraft maintenance workers being expected to understand the Australian Standard 2865[13]. In aviation, training programs are necessary for practitioners who use checklists to reduce risk and ensure that pilots know how to deterministically and correctly respond to or do what is asked. For example, pilots are trained on checklists specific to the aircraft they fly in order to ensure they understand how the checklist applies to their aircraft.

Many checklists in the secure coding domain, however, do not have a rigorously defined context where they are applicable nor an explicit set of concepts that the user should be familiar with. Without an explicit context in which the checklist is applicable, similar to a software design pattern's list of context, forces, etc; software engineers may incorrectly assume that the checklist is designed for their application domain / constraints. Developers that use Google to find and apply security checklists to their codebases, without sufficient knowledge of the assumptions made when the checklist was designed, will inevitably produce flawed evaluations of their software's security. When *completed*, these improper evaluations cause developers to boast a false sense of security to the desired users of their products.

In many other domains, particularly aircraft, the exact context that the checklist was designed for is well-understood. Pilots of a Boeing 777 don't have to wonder if the checklist they are using was actually designed for an Airbus a380. Not only is the domain of the checklist's applicability well-understood, but the training requirements to effectively use the checklist are as well. In contrast, there are often few expectations listed in the surrounding documentation of secure coding checklists to guide developers in determining if the checklist is applicable to their context. Was the checklist only designed for web applications built in Javascript with Express and NodeJS or is it equally applicable for .Net or PHP applications? We found many checklists fail to document this information in the materials accompanying the secure coding checklist.

4. Survey Results

Table 2 shows the results from our survey of 39 checklists from the Internet. Results in our survey **DO NOT represent human performance** of using the surveyed checklists. Our survey, however, identified the prevalence of the challenges from Section in the checklists we surveyed. For each checklist, if at least one item had demonstrated the challenge, it was added to the count of checklists demonstrating that issue. For example, if at least one item on the checklist was non-deterministic, the checklist was added to the count of checklists suffering from "determinism issues." The reason we allowed a single issue to add an item to the count is that security is inherently a weakest link game. We provide detailed per-checklist results in Tables II-V. The list of checklists and URLs is provided in the appendix.

[0]https://github.com/zhongwei-teng/security-checklists

Keywords	Determinism Issues	Undecidable	Improper Generalization	Lacking Context	Checklist
Python Django Security Checklist	2	1	1	2	C1
	2	1	1	2	C2
	1	1	1	0	C3
	2	2	1	0	C4
	1	1	1	0	C5
	2	2	1	2	C6
	0	0	0	0	C7
Python Flask Security Checklist	1	1	1	2	C8
	0	0	0	0	C9
	1	1	1	2	C10
	1	1	1	2	C11
	1	1	1	0	C12
Python Secure Coding Checklist	1	1	1	2	C13
	0	0	0	2	C14
	0	0	1	0	C15
	1	1	1	0	C16

Table 3: Secure Coding Checklist Results for Python

Keywords	Determinism Issues	Undecidable	Improper Generalization	Lacking Context	Checklist
Web Development Security Checklist	1	1	2	0	C17
	1	1	2	2	C18
	1	1	2	0	C19
	1	1	2	2	C20
	1	1	1	2	C21
	0	1	1	0	C22
	2	1	2	0	C23
	1	1	1	2	C24
	1	1	2	1	C25

Table 4: Secure Coding Checklist Results for Web Development

A single flaw is all that is needed – or a single misunderstanding of what a checklist item is asking for. If a checklist item is created, it should be deterministic, decidable, have proper generality, and an appropriate context. Otherwise, although it may communicate important information, it should be moved out of a checklist format and into a "guide" or "tutorial" where appropriate explanation can be given and there is no connotation of "security completeness" due to the delivery format.

We observed some interesting variation in the results based on the programming language. Checklists for Java web applications exhibited the lowest percentage (25%) of determinism issues, that only 2 of 8 checklists have part of determinism issues. Their lower non-determinism was primarily attributable to the fact that the majority of the Java checklists in the top ten Google search results specifically targeted the Spring Framework and gave concrete guidance on configuration of Spring beans and the Spring Security Framework that could easily be checked. Secure coding checklist results found with the Spring keyword overall struck a fine balance between specificity, breadth, and relevance. Checklist items were confined to addressable, framework-specific programming prescriptions, and gave

Keywords	Determinism Issues	Undecidable	Improper Generalization	Lacking Context	Checklist
Java Security Checklist	1	1	2	0	C26
	1	1	2	0	C27
	2	1	2	2	C28
	2	2	2	2	C29
	2	1	2	2	C30
	2	2	2	2	C31
Java Spring Security	2	2	2	2	C32
	2	2	2	2	C33
Checklist Java Android Security Checklist	0	0	0	0	C34
	0	0	1	2	C35
	1	0	0	0	C36
	1	2	1	2	C37
	1	2	2	0	C38
	0	0	0	0	C39

Table 5: Secure Coding Checklist Results for Java

extensive context as to when the checklist is pertinent. Additionally, these checklists either provided proper code examples, or proposed existing third-party tools for identifying common mistakes. For example, in the blog [14] from top search results of "Java Spring Security Checklist", each checking item, which is usually a general security suggestion, are elaborated in the context of Java Spring with detailed code snippet, so that readers can easily check if their projects fit this secure coding rule. Besides regular security suggestions, they also recommend users to use OWASP ZAP [15], a security tool, to perform penetration testing in their Spring application.

Results for Android secure coding checklists, overall, scored most poorly out of all Java checklists, that 4 of 6 checklists have bad performance in our evaluation metrics. Though nearly every checklist highlighted the best-known Android vulnerabilities, most lacked any additional context. Furthermore, several checklists provided many non-deterministic items for developers to act on. For instance, a checklist mentions that developers should adopt Firewall to secure the server and API, while it doesn't either explain assessment of secure server or provide an external link as a suggestion. Checklists which were specific to Android often warned developers of broad classes of vulnerabilities to check for, absent of a decidable action for developers to determine if the vulnerability was actually present in their codebase.

Django, which is a Python web framework, provides an official security checklist and an automated checker to apply to your codebase – demonstrating the decidability and determinism of their items. Each item has a concrete instruction to check it. For example, instead of only emphasizing keeping secret keys properly, it provides two ways to keep secret keys(from environment variables or files) [16]. Further, it's notable that in the beginning of the official Django checklist, a link to key information that users need to know in order to understand the related security risks was provided. On the other hand, we found numerous personal blogs that provided Django security checklists that were not as carefully crafted and suffered from frequent determinism, generalization, and decidability issues. For instance, a checklist [17] mentions that outside data is not reliable and should be validated without providing readers with detailed examples. Users may also have trouble in deciding if all outside data in their application has been validated.

5. Related Work

Checklists have emerged as a popular decision aid in the last decade. Popular books, such as the best selling novel *The Checklist Manifesto* [18], have promoted their benefits. However, checklists do not always provide benefits and can be potentially detrimental if they create a false sense of cybersecurity. Even in aviation, one of the most successful fields where checklists are used, significant effort is dedicated to carefully designing checklists [1] and it would be unheard of to use a casually designed checklist downloaded from a blog.

Many domains have dedicated significant research to showing the contexts in which checklists provide benefit. For example, research [19] indicates that checklists can improve the efficiency of students and teachers in the classroom in certain scenarios. Using checklists for educational purposes combines the goal of "providing a list of critical tasks" with the goal of "educating users on what tasks they need to know." However, there is conflicting guidance on whether or not checklists are valuable for educational purposes. Many resources, such as the "Checklist for Checklists" published by Project Check [20] explicitly warns against using checklists to teach: "A checklist is NOT a teaching tool or an algorithm".

Application of checklists in healthcare has also not been universally successful. Given the complexity of patient care, checklists were proposed to aid in suppressing the frequency of a number of potentially life-threatening medical errors. For example, in one checklist study, infections fell to zero after a doctor at John Hopkins University implemented a checklist to remind doctors of the necessary steps to prevent infection during a number of procedures. A WHO pilot project to incorporate surgical checklists proved successful in a number of pilot hospitals. However, when the project was scaled up outside of the pilot hospitals, the original benefits were not seen. After evaluating the original checklist, a new 19-item checklist [21] was developed by WHO, which has demonstrated improved performance in ensuring surgical safety [22, 23].

In contrast to many other domains, however, secure software development checklists are not well-studied. The closest research is by Bellovin et al. [4], which investigated potential risks to using arbitrary checklists to enhance software cybersecurity. Much more additional research is necessary to help developers and organizations determine when checklists are appropriate in cybersecurity and the qualities that checklists need to be successful.

The importance of assessing the quality of online materials available for secure software development has been explored in other literature. Results from a survey illustrated that informal but more accessible online resources usually have negative effects on coding security [24], showing the necessity of paying more attention to the quality of online resources with respect to security. Our research is complementary to this prior work and specifically focuses on how well checklists are crafted to convey security information.

6. Conclusion

The domains where checklists have been successful are those where the most effort is put into careful design and testing of the checklists. Aircraft manufacturers put significant effort into designing checklists for take-off and other operations and have dedicated human factors experts evaluating them. In the domain of secure coding, however, there are a lot of checklists focused on helping developers produce secure software. Some of these checklists, such as the official checklist for Python's Django web framework, are well-designed and offer concrete guidance.

At the same time, there are a large number of checklists available that are not nearly as well crafted. These checklists suffer from significant problems in their decidability, determinism, level of abstraction, and context communication. For example, many checklists include items of the form "ensure you are not vulnerable to X", with no specific guidance on how to *decide* if you are vulnerable. Often, determining if a codebase is vulnerable to "X" is fundamentally undecidable.

Our survey of checklists taught us a number of valuable lessons regarding their design and usage in the cybersecurity domain:

- **Framework-specific checklists** provided by the framework developers themselves were of much higher quality than other checklists. As opposed to many other checklists, framework developers would create extremely specific checklist items that were deterministic and decidable. We assume that framework developers aren't tempted to generalize, since they are interested in seeing increased adoption of their framework and not competing frameworks/approaches. Thus, they tend to produce very precise and actionable checklist items.
- **Business marketing content** formed some of the poorest quality checklists in our survey. We found many company blog posts are often structured as "secure coding checklists" but are written to be as general as possible and attract a large number of readers. The checklist items inherently suffer from non-determinism and undecidability because they are too general to be useful.
- **Aiming for reuse** may end up being an anti-pattern in cybersecurity checklist design. The best checklists are highly specialized to the context / language / framework to ensure that items are deterministic, decidable, not overly broad, and clearly communicate what is expected of how they are used. Communicating key vulnerabilities and ideas in a reusable way is important – but should be done outside of the context of a checklist. Instead of using the "checklist" term that connotes "completeness", lists should be used to communicate this type of information.

7. References

[1] A. Degani and E. L. Wiener, "Human factors of flight-deck checklists: the normal checklist," 1991.

[2] WHO, "Who surgical safety checklist," *https://www.who.int/teams/integrated-health-services/patient-safety/research/safe-surgery/tool-and-resources*, 2020.

[3] A. B. Haynes, T. G. Weiser, W. R. Berry, S. R. Lipsitz, A.-H. S. Breizat, E. P. Dellinger, T. Herbosa, S. Joseph, P. L. Kibatala, M. C. M. Lapitan *et al.*, "A surgical safety checklist to reduce morbidity and mortality in a global population," *New England Journal of Medicine*, vol. 360, no. 5, pp. 491–499, 2009.

[4] S. Bellovin, "Security by checklist," *IEEE Security & Privacy*, vol. 6, no. 2, pp. 88–88, 2008.

[5] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl, "Developers need support, too: A survey of security advice for software developers," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 22–26.

[6] M. Malik, "Search engine optimization seo: Business digital marketing success," 2018.

[7] A. Degani and E. L. Wiener, "Cockpit checklists: Concepts, design, and use," *Human factors*, vol. 35, no. 2, pp. 345–359, 1993.

[8] S. Secured, *Secure Code Review Checklist*, December 5, 2018. [Online]. Available: https://www.softwaresecured.com/secure-code-review-checklist/

[9] H. G. Rice, “Classes of recursively enumerable sets and their decision problems,” *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[10] L. Burkholder, “The halting problem,” *ACM SIGACT News*, vol. 18, no. 3, pp. 48–60, 1987.

[11] WikiBooks, *Web Application Security Guide/Checklist*, November 26, 2011. [Online]. Available: https://en.wikibooks.org/wiki/Web_Application_Security_Guide/Checklist

[12] T. Mendo, “Web application security checklist,” *https://blog.probely.com/web-application-security-checklist-ee0479bf60c6*, December 6, 2018.

[13] A. B. Licence and I. S. (ABLIS), “Australian standard as 2865-2009: Confined spaces - western australia,” *https://ablis.business.gov.au/service/wa/australian-standard-as-2865-2009-confined-spaces/29626*, 2020.

[14] M. Raible, “10 excellent ways to secure your spring boot application,” *https://developer.okta.com/blog/2018/07/30/10-ways-to-secure-spring-boot*, 2018.

[15] OWASP, “Zap,” *https://www.zaproxy.org/.*

[16] D. S. Foundation, “Django deployment checklist,” *https://docs.djangoproject.com/en/3.0/howto/deployment/checklist/*, December 6, 2018.

[17] O’Reilly, “A handy security checklist,” *https://www.oreilly.com/library/view/django-web-development/9781787121386/ch33s02.html.*

[18] A. Gawande, *Checklist manifesto, the (HB).* Penguin Books India, 2010.

[19] K. D. Rowlands, “Check it out! using checklists to support student learning,” *English Journal*, pp. 61–66, 2007.

[20] P. Check, *A Checklist for Checklists*, January 14, 2010. [Online]. Available: https://www.projectcheck.org/uploads/1/0/9/0/1090835/checklist_for_checklists_final_10.3.pdf

[21] W. H. O. (WHO), “Who surgical safety checklist,” *https://www.who.int/patientsafety/ safesurgery/checklist/en/*, 2017.

[22] T. G. Weiser, A. B. Haynes, G. Dziekan, W. R. Berry, S. R. Lipsitz, A. A. Gawande *et al.*, “Effect of a 19-item surgical safety checklist during urgent operations in a global patient population,” *Annals of surgery*, vol. 251, no. 5, pp. 976–980, 2010.

[23] A. Fudickar, K. Hörle, J. Wiltfang, and B. Bein, “The effect of the who surgical safety checklist on complication rate and communication,” *Deutsches Ärzteblatt International*, vol. 109, no. 42, p. 695, 2012.

[24] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, “You get where you’re looking for: The impact of information sources on code security,” in *2016 IEEE Symposium on Security and Privacy (SP).* IEEE, 2016, pp. 289–305.