A Human Error Based Approach to Understanding Programmer-Induced Software Vulnerabilities

Vaibhav Anu[#]
Department of Computer Science
Montclair State University, NJ, USA
anuv@montclair.edu

Kazi Zakia Sultana[#]
Department of Computer Science
Montclair State University, NJ, USA
sultanak@montclair.edu

Bharath K. Samanthula
Department of Computer Science
Montclair State University, NJ, USA
samanthulab@montclair.edu

***Abstract*—Many security incidents can be traced back to software vulnerabilities, which can be described as security-related defects/bugs in the code that can potentially be exploited by the attackers to perform unauthorized actions. An analysis of vulnerability data disseminated by organizations such as NIST's National Vulnerability (NVD) and SANS Institute shows that a majority of vulnerabilities can be traced back to a relatively small set of root causes mostly related to the repeated mistakes by the programmers. That is, programmers exhibit a pattern of erroneous coding practices or behavior which lead to vulnerable code. Cognitive Psychologists have long been studying these erroneous behavior patterns and have termed them as human cognition failures or simply, human errors. The primary goal of this paper is to propose a classification for the most frequently observed human errors committed by the programmers (the commitment of a human error can lead to injection of one or more security defects/bugs). Such a classification can be useful for software development organizations as they can train developers on the human errors so that developers can avoid committing the human errors themselves, thereby reducing the chances of vulnerability injection in their code.**

***Index Terms*—human error; secure software development; vulnerability; cyber security; survey; software engineering**

I. Introduction

It has been well-established that at the root of most security incidents are one or more software vulnerabilities. Vulnerabilities are those aspects of a computer system that allow breaches in the system's security policies. Vulnerabilities may be typically caused in two ways: (1) due to the absence of security controls such as lack of firewalls or lack of access controls, or (2) due to missing precautions in the software development life cycle such as insufficient input validation or unchecked buffer. The focus of this research effort is on the vulnerabilities that are introduced by the programmers/coders while developing various software components. We focus specifically on the security bugs (i.e., vulnerabilities) that result from programmers' mistakes during the implementation stage of software development lifecycle (SDLC).

During implementation stage, programmers transform the requirements and design specifications (from previous SDLC stages) into a working software solution. The implementation stage is mostly human-centric and creative as it involves a team or teams of developers working together to carefully translate the specifications into code, which requires them to continuously make informed choices to create an efficient and reliable software solution. Given the human-centricity of implementation stage, it is not surprising that a lot of software bugs can be traced back to human fallibility [1]. Cognitive Psychologists often refer to human fallibility as *human cognitive failures* or *human errors*.

Authors note that the word "error" can be easily confused with "system error". In this study, the term "error" refers to human error, a strictly mental or psychological event. Furthermore, the IEEE Standard 24765 [2] defines an error as the failing of human cognition in the process of problem solving, planning, or execution. As mentioned earlier, programming is a creative activity that involves both decision making and planning for solving a problem. The cognitive failures that occur during programming can lead to various types of software bugs including security-related bugs (i.e., vulnerabilities) being unintentionally inserted into the code by the developers. Cognitive Psychologists have long been studying cognitive failures. By exploring how human cognition fails in different situations, human error research has successfully supported error prevention in fields ranging from aviation to healthcare to software engineering [3]–[5].

Furthermore, the impact of human errors on software security has also been noted by security researchers. Daugherty [6] reported that the most common causes of Data Security Incident are phishing and malware (31% of incidents) followed by employee actions and errors as the second most common cause (24%). Oliveira et al. found some cognitive issues related to software developers that can lead to vulnerable code [7]. They found that security is not the primary concern of the software development teams. That is, security is not part of developers' mindset while coding and developers assume only common cases for their code [7] rather than modeling all potential security conditions (a phenomenon described as bounded rationality by cognitive psychologists).

In the current research, we propose that the ability to understand the underlying human errors that are committed by the developers during the implementation stage of SDLC can be beneficial from a defect/bug prevention standpoint in software projects. That is, awareness of commonly committed errors can help developers to be more careful and avoid those errors, thereby reducing the chances of bug injection. More

[#]Authors Vaibhav Anu and Kazi Zakia Sultana are co-first authors and have contributed equally to this work.

978-1-7281-7735-9/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSREW51248.2020.00036

specifically, from a security standpoint, we focus on those human errors that cause insertion of security bugs in the code. In order to make human error information tractable and easy-to-understand, cognitive psychologists build taxonomies to categorize the specific types of human errors that occur in each domain (example domains include aviation and medicine) [3]. While the underlying theoretical foundation remains same across domains, the specific types of human errors differ. Although existing research focused on relating software vulnerabilities to cognitive issues, it is important to frame the issues using a theoretical Cognitive Psychology model. This will not only assist to identify the underlying causes of the vulnerabilities, but also will work as a beacon to take proper countermeasures. To that end, ***the high-level goal*** of the current research is:

To build a framework of developer-committed human errors that cause them (i.e., developers) to write insecure code.

We have built our framework based on a well-respected theoretical human error model proposed by Cognitive Psychologist, Dr. James Reason in his seminal work titled, Human Error [8]. Section II provides a detailed overview of James Reason's human error model, followed by Section III where we describe the human error framework that we have developed to help software developers in understanding and applying a human error based software quality improvement approach. We believe that our proposed taxonomy (described in Section III) is interesting for the following reasons:

- It has an educational value. For example, being familiar with the frequently observed root causes of security bugs will make the programmers conscious to avoid similar traps.
- Such a taxonomy can serve as a type of "checklist" for software testers or third-party inspectors as it can allow them to do a more directed search for security bugs/issues in the software under test.
- For each human error identified in the taxonomy (shown in Table I), focused countermeasures can be developed. Developing these countermeasures (or preventive measures) is an ongoing effort by the authors.

II. Reason's Human Error Classification System

In this section, we provide a description of James Reason's human error theory [8]. Reason's theory has been applied (and continues to be applied) in accident/incident analysis in domains such as medicine, road/rail transport, aviation, and software development to understand the underlying human errors that occur and lead to accidents [3], [4], [9]–[11]. In a sense, we are proposing that security bugs are analogous to accidents and occurrence of a security bug (i.e., accident) simply provides an opportunity to investigate the underlying weaknesses (the weaknesses in our case are cognitive failures of programmers). Reason's theory can help us classify and understand the cognitive failures of the programmers.

Reason provides a framework to understand the specific breakdowns (i.e., human errors) in human information processing. Reason proposed that in any problem-solving situation, human errors are induced during two primary cognitive activities of the humans: *planning* and *execution* [8]. Reason further associated the human errors that occur during planning and execution to commonly observed erroneous human behaviors. The human errors during execution are related to inattentiveness, carelessness, and forgetfulness. Human errors related to inattentiveness and carelessness are called ***slips***, and those associated with forgetfulness are referred as ***lapses***. ***Mistakes*** are the human errors that occur during planning due to the lack of adequate knowledge in planning.

Slips, which result from inattention while executing routine tasks, can be exemplified in day-to-day activities like typing incorrectly or "fat-fingering" due to carelessness or inattention. *Lapses* happen when executing routine tasks, but they are failures of memory. For example, having planned to replace/repair a broken machine-part, but forgetting it due to an interruption (e.g., taking a coffee break) is a common lapse. From a programming perspective, slips and lapses typically occur during mundane/routine activities like typing, reading, taking notes, and filing. Cognitive breakdowns related to slips and lapses are generally a failure to pay attention to performing routine actions. For instance, consider a programmer finishing the end of a "for loop" header when she receives a phone call. When she returns to the "for loop" after this interruption, she fails to complete the increment statement, which in turn will cause a runtime error [1].

Mistakes are planning failures and occur while planning a solution for an unfamiliar problem. For instance, a doctor misdiagnosing a patient either due to not studying the patient's symptoms properly or not having any experience whatsoever with the symptoms exhibited by the specific patient. Mistakes usually result from the lack of adequate knowledge while working for a novel or unprecedented situation. In that case, individuals attempt to use the rules/procedures that have successfully worked for them in similar situations in the past. Mistakes are particularly applicable to programming and software development as development is a creative activity where software teams are trying to build a solution for solving a new problem (e.g., a new user need).

III. The Developer Human Error Taxonomy

Our Developer Human Error Taxonomy (Dev-HET) is a two-level hierarchy, summarized in Table I. The first level of Dev-HET is comprised of *Error Categories* (Slip-Lapse-Mistake) and the second level consists of *Human Error Classes* that form each error category. In this section, we describe 15 human error classes (or simply, human errors) that form the Dev-HET. Section IV provides examples of how a human error committed by a programmer can lead to a security bug/defect.

Taxonomy Development Process: The Dev-HET was built with inputs from multiple sources: a literature survey of other existing studies describing root causes of security bugs [12], [13], [15]–[17], lists of known vulnerabilities (CWE [14], NVD [18]), and authors' experience in secure software devel-

TABLE I
THE DEV-HET: DEVELOPER HUMAN ERROR TAXONOMY

Error Category (from Reason's Human Error Model)	Human Error Classes	Source
Slip	S1. Overlooking the design documentation	Revnivykh et al. [12]
	S2. Overlooking or failure to read the API documentation before using the API	Green et al. [13]
Lapse	L1. Forgetting to remove debug log files when software is transitioned from a debug state to production.	MITRE [14]
	L2. Forgetting to remove the test code before deploying their applications.	Green et al. [13]
	L3. Forgetting to fix those issues that are bookmarked in the earlier system versions (i.e., bookmarked to be fixed in later versions)	Revnivykh et al. [12]
	L4. Forgetting to fix "self-injected" backdoors in the system.	Revnivykh et al. [12]
	L5. Forgetting to check every access to every object because security relevant code is distributed between functional code.	Piessens et al. [15]
Mistake	M1. Bounded Rationality when choosing libraries.	Wurster et al. [16]
	M2. Incorrect assumptions or lack of knowledge about the type of environment in which his program would be running.	Piessens et al. [15]
	M3. Lack of knowledge about handling exceptional conditions.	Piessens et al. [15]
	M4. Wrong assumptions about the potential program inputs.	Piessens et al. [15]
	M5. Wrong assumptions about user authorization.	MITRE [14]
	M6. Blind trust on code from reputable sources (e.g. API code).	Oliveira et al. [7]
	M7. Incorrect assumption that blindly following the specifications generated during the design stage of SDLC guarantees security.	Assal et al. [17]
	M8. Incorrect assumption that developers should only perform functional testing and security testing is testing team's responsibility.	Assal et al. [17]

opment. We analyzed 104 vulnerabilities as reported by NVD[1]. We extracted information on coding defect, description, example code, and consequences of each vulnerability from NVD. We also gathered information from the literature survey and finally, built the taxonomy based on the information collected and the expertise of the authors. Section III-A describes the human errors classified under slips, followed by the lapses in Section III-B, and mistakes in Section III-C.

A. Slip Errors

Slips are caused by an action being carried out incorrectly while executing a planned sequence of events due to lack of attention or carelessness. Hence, a slip is an execution failure. The following two error classes were classified as slips:

1) Overlooking the design documentation: Developers often accidentally overlook or fail to refer to the design documentation that contain critical security measures such as complete mediation (every access to every object should be checked for authorization), and least common mechanism (minimize the amount of mechanism common to more than one user and depended on by all users). This oversight of software design documents can cause vulnerability injection in the final software product.

2) Overlooking API documentation before using the API: Developers usually do not like to read instruction manuals before getting started or they often simply overlook the API instructions before integrating the API code in their application. This can lead to vulnerability injection as the API manuals contain instructions about updating security settings.

B. Lapse Errors

Lapses happen when a goal is forgotten in the middle of a sequence of actions, or a step in the routine sequence is omitted. Like slips, lapses are also execution failures. Lapses are generally memory related failures (forgetting a step while executing a planned sequence of events). The following five errors committed by programmers were classified as lapses:

1) Forgetting to remove debug log files when software is transitioned from a debug state to production: In order to understand certain issues, during debugging, developers write sensitive information to log files (example of sensitive information includes user's name, credit card information, bank account information, etc.). Developers may sometimes forget to remove the code snippets that were specifically written for debugging purposes and transition the entire code to production. When this happens, sensitive information is continued to be written to the log files even after deployment. Attackers can exploit this to their gain.

2) Forgetting to remove the test code before deploying their applications: During the development process, developers sometimes need to insert test code in order to deal with situations which may hinder the development or unit testing process. Often, such test code remains in the final project code as the developers usually forget to remove it before deployment. This can lead to exploitable weaknesses in the final application.

3) Forgetting to fix those issues that are bookmarked in the earlier system versions (i.e., bookmarked to be fixed in later versions): During the development of early versions of the system, certain bookmarks are made by developers (or development teams) for system's further improvement. When these bookmarks are not realized (i.e., development teams forget to go back and fix the bookmarked issues), the bookmarked issues still remain in the final project code, thereby making the software vulnerable.

[1]https://nvd.nist.gov/vuln/categories

4) Forgetting to fix "self-injected" backdoors in the system: Sometimes developers specifically leave a "back door" for themselves. Often the reason for leaving these intentional backdoors is to ensure that they (i.e., the developers) are able to bypass normal security measures and get high-level user access (e.g., root access) on the software application that they are building. Developers create these backdoors mainly for troubleshooting purposes (i.e., resolving software issues). Developers can often forget to fix or plug the backdoors when delivering/deploying the software application. Attackers can find out these vulnerabilities and try to exploit them.

5) Forgetting to check every access to every object because security relevant code is distributed between functional code: Incomplete or incorrect access control validation is often caused due to the fact that security relevant code is usually distributed amid the functional code. This distributed implementation can lead to developers forgetting some access control checks (thus implementing the access control validation logic incorrectly).

C. Mistake Errors

Mistakes happen as a result of inadequate planning, which itself is a consequence of lack of knowledge or familiarity with the given problem-space. Mistakes are caused when the actions proceed as planned, but the plan itself is not adequate to achieve the intended outcome. The following eight programmer induced human errors were classified as mistakes:

1) Bounded Rationality when choosing libraries: When there are several libraries available that perform the same operation, developers usually tend to choose the library which they have used in the past or the library that is easiest to use. If the selected library happens to be insecure, it can make the application vulnerable at multiple places.

2) Incorrect assumptions or lack of knowledge about the type of environment in which his program would be running: If there is a lack of user requirements documentation, developers can make certain assumptions about the environment in which their program would be running. When an application is written for a fairly friendly environment (e.g., a mainframe or an intranet) but used in a hostile environment (like the Internet), major security issues can occur.

3) Lack of knowledge about handling exceptional conditions: Exceptional conditions are typically handled outside the normal control flow of programs. Developers need to follow additional strategies to handle the security-related consequences of those non-default control flows. Insecure handling of exceptional conditions is often caused due to either: (1) developer's lack of knowledge about how to handle such conditions, or (2) lack of awareness about all the exceptional conditions that might occur during program's runtime.

4) Wrong assumptions about the potential program inputs: Developers regularly make assumptions about the inputs to their programs. Attackers can invalidate these assumptions to their gain. Developers often do not consider all potential inputs to their programs and hence do not implement adequate input validations for all possible input types. The developers implement validations for commonly known inputs such as those given to a program through files, network connections, interaction with a user, and environment variables. Due to incorrect assumptions or lack of awareness, other input types are not overlooked. For example, method calls in a program can cross protection domains (as in Java). Hence, even method parameters should be considered by the developers as non-trustworthy input that needs careful validation. Some potential areas where non-trustworthy inputs can enter include: cookies, parameters or arguments, anything being read from the network, reverse DNS lookups, request headers, query results, URL components, filenames, e-mail, databases, and any external software systems that provide data to the application.

5) Wrong assumption about user authorization: Developers sometimes incorrectly assume that a web application can only be reached through a given navigation path. Due to this assumption, the developers might only implement authorization at certain points in the path. This can make the web applications prone to direct request attacks.

6) Blind trust on code from reputable sources (e.g. API code): While completing their assigned development tasks, developers tend to use shortcuts (e.g., integrating third party code into their programs). Assuming the correctness and reliability of code gleaned from a reputable third-party source simplifies developers' heuristics and reduces their cognitive efforts to do their work. This is also related to attribution. That is, if anything goes wrong, developers are not blamed as they were simply using a well-known API or code-repository.

7) Incorrect assumption that blindly following the specifications generated during the design stage of SDLC guarantees security: Developers' mental model of security is created mainly based on the security protocols considered during the Architecture and Design stage (e.g., design documents specify the adequate client-server communication protocol). However, design specifications may not always include software security. Therefore, incorrect assumption and blind faith of the developers on the design specifications may lead to vulnerability injection in the code. By simply assuming the design specifications as a security guarantee, the developer may be referring to security functions (e.g., using passwords for authentication) that they would implement as identified by the design specification. However, the developer also needs to focus on those vulnerabilities that are injected during implementation and are not necessarily mentioned or noted in the design specifications. As an example, a common vulnerability introduced during implementation is revealing error handling information (such as stack traces) to the attackers.

8) Incorrect assumption that developers should only perform functional testing and security testing is testing team's responsibility: This is caused due to misplaced priorities on the part of the developers. The Developers incorrectly assume that their only objective is to ensure the complete functioning of the code by fulfilling all functional requirements in the implementation stage. Even during the unit testing, they only attempt to ensure that their new code satisfies FRs and does not break any existing code. Furthermore, even the tests run by the

developers may vary in quality. Some developers perform ad hoc testing or run very simple sanity-checks wherein they only verify positive test cases using valid inputs. In some cases, developers may deliberately test only ideal case scenarios and fail to identify and test worst case scenarios. Many developers may not even view security as their responsibility in the implementation stage; instead they rely on the later SDLC stages. That is, security testing is considered to be the responsibility of the testing team. The assumption is that the testing team will have more knowledge in this area.

IV. Mapping Security Bugs to Human Errors

This section provides three examples of security bugs that can be traced back to a human error class from Dev-HET. We have organized the examples based on the three Error Categories (slip-lapse-mistake) of Dev-HET. Due to space constraints, we restrict our discussion to just three examples. The examples discussed in this section have been collected from Common Weaknesses Enumeration (CWE) [14] database.

A. Overlooking the design documentation – Slip Error

This error occurs when developers overlook or disregard the design documentation while coding. This is an important step in the implementation stage and programmers often fail to carefully follow all design instructions/specifications. This oversight can lead to vulnerability injection in the code as the design documentation usually contains critical security measures such as complete mediation. By not referring to the design documents, the developer will fail to implement such security measures in their code.

Example Fault: This fault is related to the following code excerpt [19]:

```
cookie.response.addCookie( new
Cookie("userAccountID", acctID);
```

Even though the design documentation clearly specified the data encryption tactic that Account ID should not be in plaintext and needs to be encrypted, the developer failed to implement the encryption measure by being careless and ignoring to read the design documentation thoroughly. This is a frequently committed error as developers routinely write this kind of code and often cognitive controls (i.e., concentration and focus) are by default not enforced while performing routine tasks. In the above-mentioned code-snippet, as the account ID is in plaintext, the account information of the user will be exposed if the attacker can compromise the weakness.

B. Forgetting to remove debug log files when software is transitioned from a debug state to production – Lapse Error

Developers often write code snippets for debugging purposes. These code snippets help developers understand issues in their source code by letting them step through their code and view outputs at certain critical points. Developers often create the debug code in such a way that the output is written to the log files and these interim log files may contain sensitive information. They remove these debugging code-snippets once they identify the issue in the code. When a developer forgets to remove the debug code before deployment, the sensitive information will continue to be written in the log files, thus posing a major security risk for the user of the application.

Example Fault: In the following code snippet [20], a user's full name and credit card number are written to a log file:

```
logger.info("Username: " + usernme + ",
CCN: " + ccn);
```

The developer intended to delete this logging event before transitioning the source code to production state, but forgot to do so. While logging this information may have been helpful during the development stage, it is important that logging levels are set appropriately before a software product ships so that sensitive system information and user data are not accidentally exposed to potential attackers.

The example shown above highlights the fact that software development organizations can use the Dev-HET (shown in Table I) as a checklist for the programmers. This checklist, among other things, will ensure that programmers do not forget to remove the debug code before deployment.

C. Bounded Rationality when choosing libraries – Mistake Error

Many programming platforms offer preexisting libraries that can reduce the programmer's coding effort. Sometimes, for performing the same operation, several libraries might exist and the programmer has to make an informed decision about which library to use. Instead of considering the level of security offered by a library, programmers tend to use those libraries that they have either used in the past or those that are easiest to use. This can lead to insecure libraries being used in the source code and thus rendering the software application weak/vulnerable at multiple places.

Example Fault: Consider the following two code-snippets (obtained from [21]):

```
Random random = new
Random(System.currentTimeis());
int accountID = random.nextInt();
```

The random number functions used in the above two examples, Random() and random.nextInt(), are not considered cryptographically strong. However, the programmer still used these functions because they may have used them before or somehow the functions were familiar to them. A potential attacker may be able to predict the random numbers that are generated by these functions. Instead of these functions, programmers should have used the more secure hardware-based random number generation functions like CryptGenRandom on Windows, or hw_rand() on Linux, or SecureRandom() from java.security library.

V. A Comparison of Current Research with Existing Human Factors Based Security Research

In this section, we provide a brief discussion on existing research efforts that have tried to integrate human factors and human errors to improve software security.

Multiple research studies have focused on the human and organizational factors that are related to cybersecurity breaches [22]–[25]. These factors are mostly related to the potential risks that are caused by the organization's employees and end users. In [23], the authors identified that lack of training or inadequate training of the insiders increases the probability of cyber attacks. Boyce et al [22] found additional human factors including lack of motivation of the employees, disconnection between developers and user needs, lack of security compliance as being related to the cybersecurity breaches.

It should be noted that the human factors in the above-mentioned studies are not directly associated with the software development process (more specifically, the implementation phase). However, in our study, we focused on the human errors that are related to the implementation phase so that developers can write secure code by being more conscious about software security during the software development process itself. The Dev-HET will help organizations in making their programmers aware of human errors that are commonly committed during implementation. We anticipate that an awareness of common human errors will help programmers avoid the errors, thereby reducing the chances of possible security bug injection.

VI. Conclusion and Future Work

In this study, we identified human errors that are frequently committed by programmers during the implementation phase of software development lifecycle (SDLC). These human errors occur during implementation phase and lead to injection of security bugs in the source code of the final software product. Next, we classified the identified errors under three human error categories (Slip, Lapse, and Mistake) as proposed in James Reason's human error model [8]. We believe that the resulting taxonomy of programmer-induced human errors (i.e., the Dev-HET) provides a novel perspective on security vulnerabilities and their root causes.

The Dev-HET can help explain how several different flaws or faulty coding practices may stem from the same erroneous behavior patterns (slip/lapse/mistake) exhibited by programmers. We anticipate that software organizations will be able to use the Dev-HET to create intervention tools (such as checklists) to help programmers avoid the common coding pitfalls that inject security bugs in the code. Another way that we envision that organizations will use Dev-HET is to perform a trace-back from vulnerabilities in their software products to the human errors in the Dev-HET. This trace-back can help organizations understand which type of human errors are being committed more frequently by their programmers. Next, organizations can create focused measures to prevent those human errors that are specific to their environment.

We collected Dev-HET's human errors from three sources: (1) existing literature, (2) Common Weaknesses Enumeration database, and (3) authors' experience in cognitive and software security research. Authors note that there is a validity threat to this study in that we may be missing certain errors that were not reported in the existing literature. In the future, we intend to conduct ethnographic studies involving professional software developers with the goal of collecting more human errors. This will help in growing our corpus of developer human errors, thereby improving the Dev-HET.

References

[1] A. J. Ko and B. A. Myers, "A framework and methodology for studying the causes of software errors in programming systems," *Journal of Visual Languages & Computing*, vol. 16, no. 1-2, pp. 41–84, 2005.

[2] I. S. Association *et al.*, "Systems and software engineering—vocabulary iso/iec/ieee 24765: 2010," *Iso/Iec/Ieee*, vol. 24765, pp. 1–418, 2010.

[3] D. Wiegmann, T. Faaborg, A. Boquet, C. Detwiler, K. Holcomb, and S. Shappell, "Human error and general aviation accidents: A comprehensive, fine-grained analysis using hfacs," 2005.

[4] S. E. McDowell, H. S. Ferner, and R. E. Ferner, "The pathophysiology of medication errors: how and where they arise," *British journal of clinical pharmacology*, vol. 67, no. 6, pp. 605–613, 2009.

[5] W. Hu, J. C. Carver, V. Anu, G. S. Walia, and G. L. Bradshaw, "Using human error information for error prevention," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3768–3800, 2018.

[6] W. R. Daugherty, "Human error is to blame for most breaches." [Online]. Available: http://www.cybersecuritytrend.com/topics/cyber-security/articles/421821-human-error-to-blame-most-breaches.htm

[7] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang, "It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14, 2014, p. 296–305.

[8] J. Reason, *Human error*. Cambridge University Press, 1990.

[9] V. Anu, W. Hu, J. C. Carver, G. S. Walia, and G. Bradshaw, "Development of a human error taxonomy for software requirements: a systematic literature review," *Information and Software Technology*, vol. 103, pp. 112–124, 2018.

[10] V. Anu, G. Walia, and G. Bradshaw, "Incorporating human error education into software engineering courses via error-based inspections," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 2017, pp. 39–44.

[11] K. Manjunath, V. Anu, G. Walia, and G. Bradshaw, "Training industry practitioners to investigate the human error causes of requirements faults," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 53–58.

[12] A. Revnivykh and A. Fedotov, "Main reasons of information systems vulnerability," vol. 12, pp. 2133–2142, 01 2016.

[13] M. Green and M. Smith, "Developers are not the enemy!: The need for usable security apis," *IEEE Security Privacy*, vol. 14, pp. 40–46, 2016.

[14] MITRE, "Common weakness enumeration," *URL: http://cwe. mitre. org*.

[15] F. Piessens, "A taxonomy of causes of software vulnerabilities in internet software," 2002.

[16] G. Wurster and P. C. van Oorschot, "The developer is the enemy," in *NSPW '08*, 2008.

[17] H. Assal and S. Chiasson, "Security in the software development lifecycle," in *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, 2018, pp. 281–296.

[18] NIST, "National vulnerability database," *URL: https://nvd.nist.gov/*.

[19] CWE, "Common weakness enumeration," *URL: http://cwe.mitre.org/data/definitions/312.html*.

[20] MITRE, "Common weakness enumeration," *URL: http://cwe.mitre.org/data/definitions/532.html*.

[21] CWE, "Common weakness enumeration," *URL: https://cwe.mitre.org/data/definitions/338.html*.

[22] M. W. Boyce, K. M. Duma, L. J. Hettinger, T. B. Malone, D. P. Wilson, and J. Lockett-Reynolds, "Human performance in cybersecurity: A research agenda," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 55, no. 1, pp. 1115–1119, 2011.

[23] J. W. Coffey, "Ameliorating sources of human error in cybersecurity: technological and human-centered approaches." in *The 8th International MultiConference on Complexity, Informatics, and Cybernetics*, Pensacola, Florida, 2017, pp. 85–88.

[24] S. Kraemer, P. Carayon, and J. Clem, "Human and organizational factors in computer and information security: Pathways to vulnerabilities," *Comput. Secur.*, vol. 28, no. 7, p. 509–520, Oct. 2009.

[25] A. J. Widdowson and P. B. Goodliff, "Cheat, an approach to incorporating human factors in cyber security assessments," in *10th IET System Safety and Cyber-Security Conference 2015*, Oct 2015.