It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots

Daniela Oliveira Marissa Rosenthal[†] Nicole Morin[†]
Kuo-Chuan Yeh[*] Justin Cappos[‡] Yanyan Zhuang[‡]
University of Florida Bowdoin College[†] Pennsylvania State University[*] NYU[‡]
daniela@ece.ufl.edu, mrosenth,nmorin@bowdoin.edu, yeh@cse.psu.edu, jcappos@nyu.edu, yyzh@cs.ubc.ca

ABSTRACT

Despite the security community's emphasis on the importance of building secure software, the number of new vulnerabilities found in our systems is increasing. In addition, vulnerabilities that have been studied for years are still commonly reported in vulnerability databases. This paper investigates a new hypothesis that software vulnerabilities are blind spots in developer's heuristic-based decision-making processes. Heuristics are simple computational models to solve problems without considering all the information available. They are an adaptive response to our short working memory because they require less cognitive effort. Our hypothesis is that as software vulnerabilities represent corner cases that exercise unusual information flows, they tend to be left out from the repertoire of heuristics used by developers during their programming tasks.

To validate this hypothesis we conducted a study with 47 developers using psychological manipulation. In this study each developer worked for approximately one hour on six vulnerable programming scenarios. The sessions progressed from providing no information about the possibility of vulnerabilities, to priming developers about unexpected results, and explicitly mentioning the existence of vulnerabilities in the code. The results show that (i) security is not a priority in software development environments, (ii) security is not part of developer's mindset while coding, (iii) developers assume common cases for their code, (iv) security thinking requires cognitive effort, (v) security education helps, but developers can have difficulties correlating a particular learned vulnerability or security information with their current working task, and (vi) priming or explicitly cueing about vulnerabilities *on-the-spot* is a powerful mechanism to make developers aware about potential vulnerabilities.

1. INTRODUCTION

Over the past decades, the security community has spent tremendous effort in emphasizing security awareness and building secure software. However, the number of new vulnerabilities keeps increasing in today's software systems. In 2013, the Symantec Internet Security report has announced that 5291 *new* vulnerabilities occurred in 2012, 302 more than in 2011 [1]. Despite the fact that vulnerabilities have been the focus of the security community for decades, frequently observed vulnerabilities such as buffer overflows and SQL injections are still repeatedly reported. With today's increasingly diverse software, and a society significantly dependent on networked computer systems, the inability to effectively handle software vulnerabilities will result in more serious security breaches in the future.

ACSAC'14 December 08 - 12 2014, New Orleans, LA, USA

http://dx.doi.org/10.1145/2664243.2664254.

Facing this huge number of security vulnerabilities, the research community has chosen to criticize the current security education and developers. As an example, SQL injection has caused issues that commonly lead to password database disclosures. When referring to the cause for SQL injection, Bruce K. Marshall stated that *"[T]he popularity of the language has led to the rapid deployment of PHP sites and PHP-based content management systems by people who lack an education in web application security. Even though the risk of SQL injection in PHP should be fairly well understood, some organizations still end up deploying code that doesn't implement proper security controls"* [2]. The frequent condemnation of security education and criticism on software developers, however, do not help to reason about the root causes of security vulnerabilities.

We argue that the nature of increasingly insecure software with well-studied vulnerabilities does not lie in the lack of security education from the developer's part. We seek to examine the hypothesis that software vulnerabilities are *blind spots* in developer's heuristic-based decision-making processes. Heuristics are computational models that do not use all information available to reach a particular decision or course of action. During their everyday programming tasks, developers use heuristics, either consciously or unconsciously, that do not include security or vulnerability information. As software vulnerabilities represent uncommon cases not completely understood by developers and exercise unusual information flows, they are usually left out from developer's heuristics. In spite of that, decision-making is adaptive. Therefore, another hypothesis we investigate in this paper is whether priming developers *on the spot* about the possibility of vulnerabilities will change their mindset towards security and make security-thinking part of their repertoire of heuristics.

Despite the abundant security tips for safe programming [3], developers may not be considering these security practices when writing code in their daily tasks. Psychological research documents that humans often engage in heuristic-based decision-making processes that are due to the limitations in a human's working memory capacity [4, 5]. Heuristics occur when a human is facing complex problems with a large amount of information, and thus she tends to make simplified, sub-optimal decisions regardless of the rich information available [6, 7]. Although heuristics is an adaptive tool, they can lead to software faults and deleterious consequences. Seeking security-related information while coding is usually not the case in software development. When programming, developers tend to focus on their immediate goals that usually involve functional and performance requirements. As a result, they do not expect the possibility of an adversary exploiting their code [8].

To validate this hypothesis, we conducted an IRB-approved study with 47 participants from a variety of backgrounds, including de-

velopers from the industry, CS undergraduate and graduate students, CS faculty, software developer managers, and software testers. The participants were asked to work on six programming scenarios with software vulnerabilities in a survey session. The session lasted approximately one hour. Our study leveraged psychological manipulation [9] where participants did not know in advance that the study was security-related, and were *progressively cued* about possible vulnerabilities. The goal was to validate the hypothesis that security information is usually not part of developers' heuristic thinking during their everyday programming; however, security information can become part of their repertoire of heuristics when developers are cued about the possibility of vulnerabilities. Analysis of participant's answers showed that stronger cues have stronger effect on developers.

In summary, this paper has the following contributions:

1. We present a hypothesis that views software vulnerabilities as blind spots in developers' heuristic-based decision-making processes, and propose a paradigm that advocates security information and education should reach developers when they need it, *on the spot*.
2. We conduct an IRB-approved study with 47 developers to validate our hypothesis and show that 60% of the developers thought that when primed about security they become aware of the security implications of their code, and 83% of the developers thought that explicitly mentioning the possibility of vulnerabilities on the spot changed their mindset towards security.
3. We present a thorough discussion of developers' interview answers that are analyzed with techniques widely used in the social, behavioral and economic sciences. The results show that (i) security is not a priority in software development environments, (ii) security is not part of a developer's mindset while coding, (iii) developers assume common cases for their code, (iv) security thinking requires cognitive effort, (v) security education helps, but developers can have difficulties correlating particular learned vulnerability or security information with their working task, and (vi) priming or explicitly cueing about vulnerabilities *on-the-spot* is a powerful mechanism to make developers aware of the potential vulnerabilities.

This paper is organized as follows. Section 2 describes humans' heuristic-based decision making processes and how they relate to software vulnerabilities. Section 3 presents the study method to validate our hypothesis. In Section 4 we detail the scenarios presented to the developers during our study, and the use of psychological manipulation. Section 5 presents the results obtained, and the Section 6 discusses these results and provides our recommendations for developers. Section 7 gives an overview of related work and Section 8 concludes.

2. HEURISTICS AND VULNERABILITIES

Psychology research has documented that during evolution, humans have become hardwired for shortcut and heuristic-based decision making processes [4, 5]. Heuristics are cognitive processes that humans use to make judgments, decisions and perform tasks [10]. They are simple computational models that allow one to quickly find feasible solutions and that do not necessarily use all information available. Heuristics rely on core human mental capacities, such as recognition, recall and imitation [11]. They represent an alternative to optimization models that use all information available and always compute the best solution.

As psychological processes, heuristics are very useful as they require less cognitive effort for a particular task. Humans have a short working memory, which makes cognitive processes difficult when too much information, possibilities, or choices are available [12]. In such cases, humans employ sub-optimal decision-making processes that can lead to mistakes [6, 7]. This argument is reinforced by Zipf's principle of least effort [13], which states that humans use as little effort as necessary to solve a problem. Heuristics are adaptive responses to human's short working memory. They have high predictive accuracy when information is scarce, but can lead to severe biases and errors in decision making or ensuring the correctness of tasks [10, 11].

Such heuristic-based decision-making processes also largely affect software security. According to Thorngate, humans tend to ignore information in heuristics because they do not notice certain issues of a particular problem, or there are small or infrequent decrements in reward that result from their ignorance or misuse of relevant information about the problem in hand [10]. In software development, this is reflected by the fact that functional and performance requirements usually have higher priority. Kieskamp and Hoffrage [14] also argue that under time pressure, a common situation in software development, humans are likely to adopt heuristics that are even simpler, and do not require much integration of information. In spite of that, a decision-maker is adaptive such that through proper feedback they can improve their repertoire of heuristics. However, there also exists a forgetting process where a particular piece of knowledge or strategy can be gradually wiped out from the decision-maker's repertoire of heuristics, if not properly reinforced.

Our hypothesis is that software vulnerabilities are blind spots (biases or errors) in developer's heuristic-based decision-making processes. Software vulnerabilities are introduced mostly because developers use heuristics to make decisions in their everyday tasks. When developers constantly make sub-optimal decisions, consciously or unconsciously, they are mostly concerned about finding *a* solution or an efficient solution to a particular problem. However, as software vulnerabilities often lie in the corner cases and unusual information flows, they tend to be left out from developers' heuristics. However, when properly primed about security, *on the spot* and with cues correlating with their current programming tasks, developers can properly develop a security mindset for the task at hand.

3. DEVELOPER'S STUDY METHOD

Forty-seven participants were tested in this study in exchange for a gift certificate. Participants were invited via direct e-mail sent to software development companies, universities and colleges in the United States and abroad.

3.1 Source Materials

Stimuli consisted of programming scenarios that included software vulnerabilities and that tested developer's understanding on the issue. The vulnerabilities in each programming scenario were: buffer overflow [15], cross-site scripting (XSS) [16], SQL injection [17], Python Secure Socket Layer (SSL) [18], time of check to time of use (TOCTTOU) [19], and brute force password exhaustion vulnerabilities [20]. Each scenario is an *exhibit question* in Qualitative Research [21], which sharpens the respondents' concentration by asking them to respond to a specific statement, story or artifact. A scenario is a short code snippet with comments that formulates the underlying vulnerability in an focused way, exercising it, adding all necessary context, and removing unnecessary noise. Noise was removed in a careful way, as vulnerabilities are often located in hidden cases and also in unusual information flows. Example programming scenarios will be given in Section 4.

3.1.1 Information Conditions

The scenarios are presented to developers with different information conditions: (i) with no information about security or vulnerabilities (controlled condition), (ii) with implicit information about possible unexpected results in code (priming condition), and (iii) with explicit information about the existence of vulnerabilities in code (explicit condition). Examples of these information conditions in our study are given in Sections **??** and **??**.

3.1.2 Pilot Study

Before inviting developers and distributing the survey, a pilot study was performed with undergraduate students from one of the author's institutions. In this study, students were asked to answer questions for each scenario and provide feedback in a live follow-up interview about their experience. Follow-up interviews were tape-recorded and included a thorough debrief on each vulnerability. Based on their feedback, the scenarios were adjusted to maximize their clarity. An online survey was then created that consisted of the six finalized scenarios, followed by open-ended questions that reflected the interview questions in the pretest. Stimulus presentation and the collection of responses were controlled using Qualtrics platform for online data collection [22].

3.2 Procedure of Study

Participants were instructed to answer questions in the survey as accurately and thoroughly as possible, without knowing that the study was security-related. To ensure that subjects were not suspicious of the aim of our study, the survey employed psychological manipulation techniques [9] that involved a cover story presented in a consent form signed by the participants. The cover story explained that this was a study of how developers think about programming in general, so that researchers of this study could build mental models about typical developers. They were informed that in the survey, there would be six programming scenarios and general questions about snippets of code, and they would be asked to perform small programming tasks to modify the code.

Subjects were not restricted to a particular time frame to complete the survey. However, they were informed that once they completed one scenario, they would not be able to return to previous pages. Following the last scenario, subjects responded to open-ended questions that addressed the manipulation of information, the subjects' personal experience with computer security, and their familiarity with each of the tested vulnerabilities. The examples of open-ended questions are given in Section **??**. To leverage the educational purposes of the study, subjects were debriefed on each security vulnerability.

3.3 Design

Psychological research experiments typically have both dependent and independent variables. In this study, the dependent variables are the participants' survey scores and their answers to the open-ended debrief questions. There are also two independent variables that are varied and manipulated by the experimenter in this study. The first one was the vulnerability scenario of a code snippet. The second is the level of information (condition) emphasized by each question and could be no information, implicit or explicit information. In a psychology experiment, researchers are interested in how the changes in an independent variable cause changes in the dependent variable [9]. Two one-way Analyses of Variance (ANOVAs) [23] across the three levels of information and six vulnerability scenario types were conducted on data accuracy.

Scenarios in the controlled and priming (implicit) conditions contained questions that addressed user input, code execution, and code modification. In the priming condition, subjects answered an additional question that asked whether unexpected results could arise from the presented snippet of code. This question aimed to prime participants to think about security-related vulnerabilities. In the explicit condition, subjects were cued to look for the vulnerability and were directly told that the code had a security flaw.

The order of the scenarios was randomized for each participant. However, because the manipulation intended for subjects to change their mindset, information conditions, or whether the scenario was in the controlled, implicit, or explicit condition, were presented to all participants in a specific order. The first two scenarios provided no information about the corresponding security flaws (controlled condition), the second two scenarios provided implicit information (priming) about the corresponding security flaws, and the last two scenarios explicitly addressed the fact that the snippets of code had security flaws (explicit).

```
1.  <? php
2.  session_start();
3.
4.  if(!isset($userinfo[$_POST['username']])) {
5.       // Invalid username
6.       echo "Authentication failed."
7.  }
8.  else {
9.
10.      if ($userinfo[$_POST['username']] ==
11.                  md5($_POST['password'])) {
12.          // Successful authentication
13.          echo "Welcome to Bank ABC."
14.      }
15.      else {
16.          // Invalid login
17.          echo "Login failed."
18.      }
19. }
20. // rest of the script
21. ?>
```

Figure 1: Brute force password exhaustion vulnerability scenario.

4. THE VULNERABILITY SCENARIOS

Each developer session exercised six vulnerabilities, where five are well-known and studied by the security community for years and one is relatively new, initially reported in vulnerability databases in 2013. In this section, we describe our IRB-approved study with these vulnerabilities, our psychological manipulation through different information conditions, and how the results are measured.

The five well-know vulnerabilities are buffer overflow [15], cross-site scripting (XSS) [16], SQL injection [17], time-to-check-to-time-to-use (TOCTTOU) [19], and finally an authentication vulnerability which can lead to brute-force password or dictionary attacks [20].

Figure 1 illustrates the brute force authentication vulnerability scenario. Developers were asked to consider a hypothetical banking application that requires users to authenticate themselves. A user is prompted to provide a user name and a password, which are verified before any access is granted. Developers were told that knowledge of PHP was not required, and the snippet of PHP code represented a simplified version of this authentication process. The scenario also mentioned that method `isset` determines whether a variable is set and is not NULL.

The code is vulnerable because different messages are given for valid and invalid user login, as shown in line 6, 13 and 17 in Figure 1. This helps an adversary to discover valid user names. In the **controlled condition**, developers were asked the following warm-up questions: *What is the user input to this program? What happens when this code executes?* These warm-up questions were present in all the scenarios in the controlled condition. Then the developer was asked to modify this code to allow a user to log in as *guest* if she enters a user name of `guest`. In this situation, the user will have restricted access to the system resources. The developer was told to consider places that might be improved while the developer was modifying the code. The goal was to give developers who were aware of any security concerns an opportunity to address them. This way, we also verify the developer's willingness or motivations to improve the code.

The last vulnerability is about the Python SSL API, which makes web applications vulnerable to man-in-the-middle attacks [18]. An example is given in Figure 2. In the figure, function `wrap_socket` is used by a client to open an SSL connection with a server. Normally when an application (SSL client) checks the server's SSL certificates, it ensures that there is a chain-of-trust from a root certificate (preloaded onto the system) to the provided certificate. While the end-to-end encryption of SSL provides integrity and confidentiality, certificate checking validates the authenticity of the server site. However, function `wrap_socket` in Python does not validate certificates received from the server. As a result of this vulner-

```
1.  import socket, ssl
2.
3.  sock = socket.socket()
4.  sock.connect((HOSTNAME, PORTNUMBER))
5.  # wrap socket to add SSL support
6.  ssl_sock = ssl.wrap_socket(sock,
7.   cert_reqs = ssl.CERT_REQUIRED,
8.   ca_certs = "/etc/ssl/certs/ca-certificates.crt")
9.
10. ...
11. print("Successful SSL connection")
12. # transfer data between client and server
13. ...
14. ssl_sock.close()
```

Figure 2: The Python SSL vulnerability scenario.

ability, any malicious party can provide a certificate claiming to be an official website, and the fake certificate will be trusted.

Figure 2 shows the Python SSL scenario in the survey. Developers first received basic information about how SSL works, and then were asked to consider an SSL client written in Python. In the **priming condition**, developers were asked the following questions: *Could a developer experience unexpected results when running such code? What could be examples of these unexpected results and where do they appear in the code?*

In the **explicit condition** for all scenarios, subjects were asked the following questions: *This code has a vulnerability (security flaw) that allows attackers to violate certain security policies of the [web application/program]. Can you pinpoint the problem? Please describe the vulnerability and where it occurs. Why do you think that developers have problems pinpointing this particular problem?*

Following the explicit conditions, subjects responded to **open-ended questions** that addressed the manipulation of information: (i) *Were all the programming scenarios and associated questions clear? If not, what confused you?* (ii) *When you were asked to modify the code, were you suspecting to find vulnerabilities? If not, why do you think you missed it?* (iii) *Did asking you about unexpected results cue you to think about potential vulnerabilities in the code?* (iv) *Did explicitly asking you to find the vulnerability force you to change your approach while examining the code and answering the subsequent questions? If so, explain.*

The participants were also asked about their personal experience with computer security (*have you taken computer security classes? If so did these classes help you pinpoint vulnerabilities in the snippets of code?*), their familiarity with each of the tested vulnerabilities (*Are you familiar with [the exemplified] vulnerabilities? If so please explain how your knowledge about these common vulnerabilities influenced or did not influence how you approached the scenarios?*), and whether they will be more aware of these vulnerabilities when developing new projects (*Now that you are aware of the different vulnerabilities, do you think you will think about vulnerabilities in future programming tasks?*). To leverage the educational purposes of the study, subjects were thoroughly debriefed on each of the security vulnerabilities and subsequently debriefed on the true purpose of the study.

Due to space limitations, only two scenarios are shown in Figure 1 and 2. The remaining four scenarios are available at <*removed for double-blind review*>.

In order to ensure a consistent measure for the accuracy of subjects' responses, each response was graded by at least two experimenters. Accuracy was measured for each response on a scale from zero (*i.e.*, completely incorrect) to two (*i.e.*, completely correct). A score of one signified an incomplete but correct response to the question, whereas a score of two indicated that the subject had correctly pinpointed the specific vulnerability. A separate score was included for each question signifying whether or not participants modified the code in a way that addressed the particular security vulnerability.

Occupation	Percentage
Developers from the industry	60%
Students (senior undergraduates and graduate students)	23%
CS Faculty	6%
Other occupations related to software development (*e.g.*, managers and testers)	11%

Table 1: Participant's occupations.

Educational background	Percentage
Doctoral degree	22%
Master's degree	34%
4-year college degree	38%
Not completed 4-year college degree	6%

Table 2: Participant's educational background.

5. RESULTS AND ANALYSIS

A total of 84 developers agreed to take the survey. From this set, 47 surveys were considered in this section as the respondents worked through all six scenarios and answered all the debrief open-ended questions. Table 1 shows the distribution of the participants according to their occupation, and Table 2 describes their educational backgrounds. Participants ranged from 20-52 years of age (M = 31) and approximately 81% of participants were male and 19% were female. The majority of the participants (86%) have a degree in computer science or related majors. All participants who reported not having a degree are senior undergraduate students working towards a degree. Approximately 66% of the participants have never taken a security class in college or any type of security training in the course of their careers.

5.1 Statistical Results

This subsection presents the statistical results of our study. The results described in this section use analysis of variance (ANOVA) to analyze the differences between group means and their associated procedures (such as "variation" among and between groups) [23]. ANOVA is a generalization of t-tests to more than two groups. A two-sample t-test is a statistical test that examines whether two samples are different. It is commonly used when the variances of two normal distributions are unknown and when an experiment uses a small sample size. Both ANOVA and t-tests compute the ratio between the obtained difference between the means and the mean difference expected by chance. The goal is to determine whether the obtained difference between means are larger than expected by chance, which yields a larger F value.

In ANOVA, the denominator of the F-ratio is the error variance or mean squared error (MSe), and measures how much variance is expected if there are no systematic treatment effects and no individual differences contributing to the variability of scores. A value of $p < 0.05$ means that the variation seen due to the manipulation of the variables has a probability of at least 95% certainty. The Pearson correlation (r) measures the degree of linear relationship between two variables.

	N	Min	Max	Mean	Std. Dev.
Gender	47	1	2	1.19	0.398
Age	47	20	52	30.96	7.049
Occupation	47	1	4	1.68	1.002
Level of Education	47	1	8	4.68	1.200
Total Score	40	16	36	27.63	4.301
Explicit	44	1	2	1.11	0.321
Degree	44	1	2	1.14	0.347
Security Classes	44	1	2	1.66	0.479

Table 3: Descriptive statistics for the variables used in analysis.

		Gender	Age	Occupation	Level of Education	Total Score	Explicit	Degree	Security Classes
N	Valid	47	47	47	47	40	44	44	44
	Missing	0	0	0	0	7	3	3	3
Mean		1.19	30.96	1.68	4.68	27.63	1.11	1.14	1.66

Table 4: Frequencies for the variables used in the analysis.

		Occupation	Level of Education	Total Score	Degree	Security Classes
Occupation	Pearson Correlation	1	0.058	-0.262	0.320	0.058
	Sig. (2-tailed)		0.698	0.102	**0.034**	0.709
	N	47	47	40	44	44
Level of Education	Pearson Correlation	0.058	1	0.168	-0.168	-0.70
	Sig. (2-tailed)	0.698		0.301	0.277	0.653
	N	47	47	40	44	44
Total Score	Pearson Correlation	-0.262	0.168	1	-0.305	-0.313
	Sig. (2-tailed)	0.102	0.301		0.056	**0.050**
	N	40	40	40	40	40
Degree	Pearson Correlation	0.320	-0.168	-0.305	1	-0.133
	Sig. (2-tailed)	**0.034**	0.277	0.056		0.388
	N	44	44	40	44	44
Security Classes	Pearson Correlation	0.058	-0.070	-0.313	-1.133	1
	Sig. (2-tailed)	0.709	0.653	**0.050**	0.388	
	N	44	44	40	44	44

Table 5: Correlations among the variables used in the analysis. The highlighted correlations are considered significant.

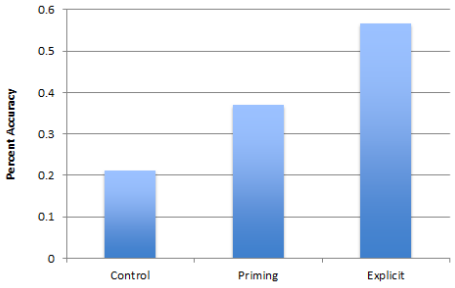

Figure 3: Mean percent accuracy for information conditions across all scenario types.

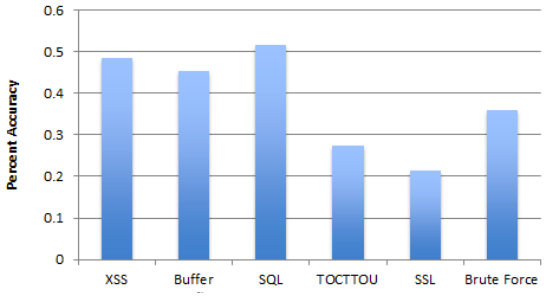

Figure 4: Mean percent accuracy for all scenario types across levels of information conditions.

	Control	Priming	Explicit	Total
Non-security related	6	6	0	12
Security related	2	4	4	10
Total	8	10	4	22

Table 6: Questionnaire breakdown into security and non-security questions.

5.1.1 Variables Used in Analysis

Table 4 summarizes the frequencies and mean for the variables used in this analysis. *Total score* is the maximum score that a participant could have obtained on a session. The *Missing* line represents particular questions within a scenario that were not answered by the participants. The *Explicit* variable indicates whether the respondent thought that explicitly asking about vulnerabilities while they worked on the scenarios changed their mindset towards security. The *Degree* variable indicates whether the respondent has a 4-year college degree in CS or related major. *Security Classes* indicates whether the respondent has ever taken security courses or training. Table 3 illustrates the descriptive statistics for these variables.

5.1.2 Result Interpretation

Because the accuracy for each response was graded by two separate experimenters, accuracy result was averaged across the two scores. For each subject, mean accuracy data was obtained for each of the six scenarios and subsequently averaged across all subjects. Two one-way ANOVAs revealed significant effects across the three levels of information ($F(2, 36) = 11.84$, $MSe = 0.30$, $p < 0.05$) and the six scenario types ($F(5, 80) = 7.80$, $MSe = 0.23$, $p < 0.05$).

With regard to the levels of information: Follow-up t-tests, which compares the means between the controlled, implicit, and explicit information conditions, revealed that subjects were significantly less accurate in controlled condition ($M = 0.31$) than in implicit ($M = 0.90$) and explicit ($M = 1.10$) conditions ($t(18) = 3.43$, $SE = 0.17$, $p < 0.05$; $t(19) = 5.86$, $SE = 0.14$, $p < 0.05$). As shown in Figure 3, these findings suggest that prompting developers to think about how other developers could encounter potential vulnerabilities with a particular snippet of code increases the likelihood that they will recognize the security flaw.

With regard to the types of vulnerability scenarios: Follow-up t-tests revealed that subjects were most accurate in identifying the SQL injection vulnerability ($M = 1.27$) and least accurate in identifying the SSL Python vulnerability ($M = 0.37$) ($t(17) = 4.54$, $SE = 0.19$, $p < 0.05$). As shown in Figure 4, these results suggest that the nature of the scenarios themselves may have a correlation with cognitive blind-spots. To further investigate how previous knowledge and experience may influence

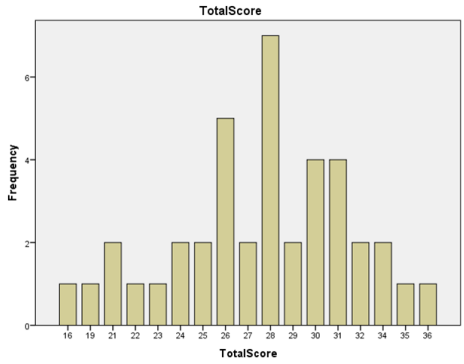

Figure 5: Participant's score distribution.

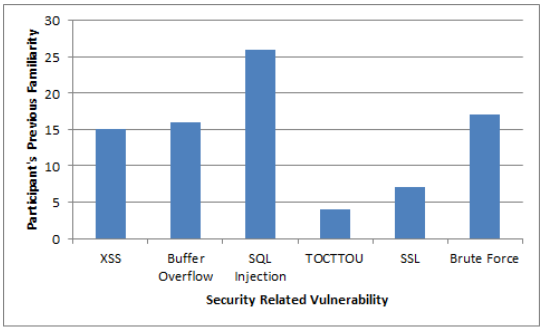

Figure 6: Participant's familiarity with vulnerabilities exercised in the questionnaire.

subjects' response accuracy, a correlation analysis revealed that the number of vulnerabilities reported as familiar to subjects was positively correlated with higher accuracy ($r(47) = 0.4, p < 0.05$).

Because familiarity is linked to experience, we conducted a correlation analysis on the dimensions of subjects' experience, including their occupation, whether or not they had a degree in computer science, and whether or not they had taken computer security courses and the number of known vulnerabilities. Results revealed that the number of known vulnerabilities was moderately correlated with having taken computer security classes ($r(47) = 0.27$, $p < 0.05$). These results indicate that developers are more likely to find security-related blind-spots if they have been formally educated on security related issues. Table 5 presents the correlations among the variables and shows how the score obtained by a respondent is highly correlated with whether they took some security training or courses.

5.2 Results of Experiment Manipulation

The debrief open-ended questions asked participants how the manipulation used in the experiment worked and its effects on how well the participants answered the security-related questions. Table 6 details how each questionnaire was graded. Each questionnaire contained 22 questions where 12 (54%) were not security related and 10 (45%) were security related. The non-security questions were warm-up questions or questions used to aid in the psychological manipulation. Two questions in the explicit condition were open-ended and were not graded (*Why do you think developers have problems pinpointing this particular problem?*). Thus, the maximum score a respondent could obtain was 40. Figure 5 shows the distribution of participants scores.

5.2.1 *Familiarity with Vulnerabilities*

The debrief questions also asked the respondents whether they were familiar with the vulnerabilities exercised in the study and Figure 6 summarizes their answers. The goal of these questions was to manually correlate a respondent score on a scenario with the familiarity they reported for the exercised vulnerability. For example, how the participants that declared knowledge of the SQL injection vulnerability scored on the scenario that exercised this vulnerability? Were they able to correlate the scenario with the vulnerability? In other words, were they able to apply their previous knowledge about the vulnerability *at the time* they needed? When given the chance to improve the code containing the corresponding vulnerability did they remove the vulnerability? Table 7 summarize this analysis.

The results in Table 7 show that many developers who were familiar with the exercised vulnerabilities had difficulties correlating the information about them to the current programming scenario or failed to fix vulnerable code when given an opportunity. For example, 53% of the participants knew a particular vulnerability but did not correlate it to working scenario. This is because the needed security information or the security thinking was not included in their heuristics at the moment.

5.2.2 *Effectiveness of Psychological Manipulation*

Despite the familiarity with vulnerabilities is weakly correlated to secure programming, the results in Table 8 show that when primed

	Frequency	**Overlooked Vulnerabilities**
Knew vulnerability but did not correlate it to working scenario	25 (53%)	Brute force (17 instances - 68%) SQL injection (6 instances - 24%) Buffer overflow (5 instances - 20%) XSS (4 instances - 20%) TOCTTOU (3 instances - 12%) Python SSL (2 instances - 8%)
Knew vulnerability but did not remove it from code when given a chance	17 (36%)	Buffer overflow (7 instances - 41%) SQL injection (7 instances - 41%) Brute force, TOCTTOU and Python SSL (1 instance each - 0.05%)

Table 7: Vulnerability knowledge and the application of this knowledge when needed it.

	Yes	**No**	**Maybe/Unsure**
Suspecting of finding vulnerabilities	15 (32%)	30 (64%)	2 (0.04%)
Asking about unexpected results cued to think about vulnerabilities	28 (60%)	19 (40%)	0 (0%)
Explicitly mentioning vulnerabilities changed your approach to a security-mindset	39 (83%)	5 (10.6%)	3 (6.4%)

Table 8: Effectiveness of the psychological manipulation and priming to change developer's approach to security.

Theme	Mentioned by developers	Representative Quotes
1) Specifically mention vulnerabilities changed developer approach	39 (83%)	*"it it put me in the mindset that this code would be very easy to infiltrate"* *"Until the words vulnerability and/or security were used, I had not thought of security risks yet"*
2) Priming about security changes developer's mindset	28 (60%)	*"Yes, I expected to due to the power of suggestion"* *"Once we were looking for unexpected results, I immediately started thinking of what a dumb and/or malicious user would attempt to do with the program"*
3) Security is not a priority / Developer's mindset does not include security	23 (48%)	*"Developers usually focus in delivering functional requirements"* *"I generally don't look for vulnerabilities in code"* *"developers look for the most immediate solution to the current problem they are facing, such as copying one buffer to a second, and run with whatever solution Google brings them first"*
4) Developers assume common cases	16 (34%)	*"We usually try to solve the problem for a set of inputs, not for all possible inputs."* *"Developers seem to be constantly reminded of the fact that users are "dummies" and any mistake they can make, they will make, etc. However, we do not tend to think of the user as evil"*
5) Security thinking requires cognitive effort	14 (30%)	*"It's hard to understand the fact that the user input can directly affect the execution of the code, changing what it was supposed to do to something else."* *"Remembering to sanitize input is tedious"* *"In general, security and vulnerability problems are hard to find"*
6) Developers trust APIs	14 (30%)	*"It's not straightforward that misusing strcpy can lead to very serious problems. Since it's part of the standard library, developers will assume it's ok to use. It's not called unsafe_strcpy or anything, so it's not immediately clear that that problem is there"* *"normally people expects that an official library does not have this kind of vulnerability as default.* *"The security check made prior to accessing the file gives a false impression that the any non-permitted users will be denied access upfront. [For TOCTTOU scenario]"*
7) For certain tasks or CS fields, security is not an issue	6 (13%)	*"In terms of academic AI code, security is really not an issue"* *"But I don't do a lot of network programming so I feel like it matters a bit less [likely to think about security in the future]"* *"yes [i would think about security in the future], if the app is going to be exposed to general public. no , for internal apps, running inside a firewall."*
8) Economic incentives	6 (13%)	*"The reward for making a code safer is not easily seen by others and may come only on the long run, when the code needs less maintenance."* *"there's the economic side. Developers are measured and paid for the features delivered without functionality bugs"*
9) Security education	6 (13%)	*"When we are aware of a vulnerability it starts to be a part of our checklist"*

Table 9: Open-ended answers analysis - Coding.

about the possibility of finding vulnerabilities developers changed their mindset towards security. In particular, 83% of the participants stated that explicitly mentioning vulnerabilities changed their programming approach and security-mindset. Table 8 shows the effectiveness of the manipulation and the use of priming can change a developer's approach to coding. This is an encouraging result.

5.3 Coding

Coding is a technique used in qualitative analysis studies in the social, behavioral and economic sciences for analysis of data. The main idea of coding is to associate what the respondent said in an interview or survey with a set of themes, concepts or categories [24]. Coding is done while reading the transcripts or an interview or the contents of an online survey. A code is a word or a short phrase that captures a datum's primary content or essence [25].

In this study the respondents' comments and answers to open-ended questions were coded according to seven themes. Table 9 shows the themes with the percentage of developers that mentioned the theme and some representative quotes from different developers.

6. DISCUSSION AND RECOMMENDATIONS

In this section, we discuss the results obtained in this work, and provide our recommendations for developers based on these results.

6.1 Discussion of Study Results

The results of this study corroborated our hypothesis that security is not part of the heuristics used by developers in their daily programming tasks. Humans have a short working memory, and can only keep a limited number of mental elements readily available at any time [26]. For a variety of reasons, security information is generally not included among these elements. Developers mostly focus on functionality requirements and performance. Developers are trained, evaluated and paid for delivering feature-rich software with good performance levels. They do not see an economic incentive for *squeezing* security thinking into their working memories and producing safe code. Developers and managers see this issue as a zero-sum game, and time spent on *"quality"* will adversely affect function points[1], deadlines, timetables and budgets.

Also, developers usually assume common cases for the inputs a piece of code will receive and the possible states the program can reach. Vulnerabilities lie in uncommon cases overlooked by the developers and exploited by a clever adversary. Attackers make uncommon code paths happen, whereas system designers focus on the common code paths that they know about and are often not aware of the attack code path until the carefully crafted input is presented to them. Static analysis methods, such as Denning's lattice model [27], overcome this limitation by analyzing all possible code paths. This can be effective (despite being formally undecidable in the general case), but have limitations on the type of programs they can be applied and the type of vulnerabilities they can pinpoint.

Developers' failure to address uncommon information flows is also caused by the complexity of fault analysis. Therefore, security thinking requires significant cognitive effort, while people use as little effort as necessary to solve a problem [13].

This study also shows that developers tend to blindly trust code

[1]A unit of measurement of the amount of functionality an information system (as a product) provides to a user.

from a reputable source, *e.g.*, API code. Given the way humans think and use shortcuts, simply assuming the correctness of third party code from a reputable source simplifies developers heuristics and their cognitive efforts to do their work. This also has to do with attribution, as if anything goes wrong, developers are not to be blamed as they were just using a well-known API or component.

To make matters worse, our society provides perverse economic incentives for a market of insecure software. As discussed by Ross Anderson in his classic ACSAC 2001 paper [28], the party who is in a position to protect a system is not the party that suffers the results of security failure. The computer software and systems market is not regulated: they select software and systems that reach the user as quickly and as feature-rich as possible. Moreover, information warfare also plays a role because the same software system that can leave millions of people vulnerable to attacks can be leveraged by national states to conduct cyber warfare and espionage. Finally, Anderson argues that it is much easier to attack than to defend [28]. This shows that there is not a "silver bullet" to solve the technical, economical, legal problems, and psychological challenges of vulnerable software as it involves.

6.2 Recommendations for Developers

In spite of the above, from a Psychology viewpoint, this paper advocates that **security information should reach users when they need it, on the spot, and not the other way around**. It is commonly assumed that developers should educate themselves about security and then apply the acquired knowledge when needed. However, this assumption goes against how the human brain naturally behaves. Our security solutions would be most effective if they leveraged how humans think. This study showed how priming security information when developers need it, *on the spot*, changed their approach towards security and adapted them to include security thinking in their repertoire of heuristics.

This is not an argument against previous and general security education. On the contrary, security education is essential and the results of this study showed that participants who had previous security training performed better in the security-related questions than participants that have never looked for security information. Bringing security information to developers and not expecting them to look for it, as advocated here, will streamline the process of retrieving previously acquired security information. The results of this study also showed that many developers who were familiar with the exercised vulnerabilities had difficulties correlating the information about them to the current programming scenario or failed to fix vulnerable code when given an opportunity. This is because the needed security information or the security thinking was not included in their heuristics at the moment. However, when primed about the possibility of finding vulnerabilities developers changed their mindset towards security.

Given the importance of security in matters, such as cyber crime, cyber warfare and privacy, we recommend that systems and software that interface with the developer (IDEs, text editors, browsers, compilers, *etc.*) bring security information or prime developers *on the spot* when they need it: *while coding*. This priming information should be closely related to their current working scenario to increase the chances that this security cue will be included in the developer's heuristics. Our insight will influence the next generation of tools and applications for developers so that more secure software reach the market. While such paradigm does not solve the multifaceted challenges of cyber security, it can illuminate developers' mental blind spots in vulnerabilities and secure programming.

7. RELATED WORK

The work presented in this paper intersects with the areas of vulnerability analysis, information security perception, and cognitive and human factors. This section discusses the related work in these areas.

7.1 Vulnerability Studies

The first effort towards understanding software vulnerabilities appeared in the 1970's through the RISOS Project that investigates security flaws in operating systems [29]. Around the same time, Protection Analysis study [30] focused on developing vulnerability detection tools to assist developers. Other vulnerability studies followed, such as the taxonomies by Landwehr *et al.* [31] and Aslam [32]. In the 1990's, Bishop and Bailey [33] analyzed current vulnerability taxonomies and concluded that they are imperfect: depending on the layer of abstraction that a vulnerability was considered, it could be classified in multiple ways. More recently Crandall and Oliveira [34] proposed a view of software vulnerabilities as fractures in the interpretation of information as it flows across boundaries of abstraction.

There are also discussions about the theoretical and computational science of exploit techniques and proposals for explicit parsing and normalization of inputs. Bratus *et al.* [35] discussed a view that the theoretical language aspects of computer science lie at the heart of practical computer security problems, especially exploitable vulnerabilities. Samuel and Erlingsson [36] proposed input normalization via parsing as an effective way to prevent vulnerabilities that allow attackers to break out of data contexts. Garg and Camp [37] identified systematic errors by decision-makers leveraging heuristics as a way to improve security designs for risk averse people.

Researchers have also studied vulnerability trends. Browne *et al.* [38] determined that the rates at which incidents were reported to CERT could be mathematically modeled. Gopalakrishna and Spafford [39] analyzed software vulnerabilities in five critical software artifacts using information from public vulnerability databases to predict trends. Alhazmi *et al.* [40] presented a vulnerability discovery model to predict long and short term vulnerabilities for several major operating systems. Anbalagan and Vouk [41] analyzed and classified thousands of vulnerabilities from OSVDB [42] and discovered a relationship between vulnerabilities and exploits. Wu *et al.* [43] performed an ontology-guided analysis of vulnerabilities and studied how semantic templates can be leveraged to identify further information and trends. Zhang *et al.* [44] analyzed vulnerabilities from the NVD database using machine learning to unsuccessfully discover the time to the next vulnerability for a given software application.

There are also studies on developer's practices. Meneely and Williams [45] studied developers collaboration and unfocused contributions into developer activity metrics and statistically correlated them. Schryen [46] analyzed the patching behavior of software vendors of open-source and closed-source software, and found that the policy of a particular software vendor is the most influential factor on patching behavior.

However, none of the above research efforts directly leveraged the human factor to understand software vulnerabilities as proposed in this paper.

7.2 Information Security Perception

Asghapours *et al.* [47] advocate the use of mental models of computer security risks for improvement of risk communication to naive end users. Risk communication consists of security experts messages to non-experts and a mental model in a simplified internal concept of how something works in reality. In their study they leveraged five conceptual models from the literature: Physical Safety, Medical Infections, Criminal Behavior, Warfare, and Economic Failures.

Huang *et al.*[48] studied ways to adjust people's perception of information security to increase their intention to adopt IT appliances and compliance with security practices. Their user study involving e-banking and passwords showed that knowledge is a key factor influencing the gap between people's perceived security and a system real security.

Garg and Camp [49] leverage the classic Fischhoff's canonical nine dimensional model of offline risk perception [50] to better understand online risk perceptions. They found that the results obtained for online risks differed from the ones obtained for offline

risks and that the severity of a risk was the biggest factor in shaping risk perception.

Research has also been done in the area of computer warnings for end users [51, 52]. In computer security, warnings are designed to protect people from becoming victims of attacks, such as phishing, malware installation, e-mail spam, *etc.* Researchers have found that people tend to not pay attention to messages that do not map well onto a clear course of action [51]. This corroborates our hypothesis that unless the cue is related to the working scenario at hand, it will likely to be left out from the decision-maker's repertoire of heuristics.

All these studies consider information security perception from the non-expert end user viewpoint and not developers.

7.3 Human Factors in Software Development

Using human factors in technology research is not a new concept. Curtis, Krasner, and Iscoe [53] studied the software development processes by interviewing programmers from 17 large software development projects. They tried to understand the effect of behavioral and cognitive processes in software productivity. They believed software quality in general could be improved by attacking the problems they discovered in this exploratory research. They summarized the study by describing the implication of their interviews and observations on different aspects of the software development process, including team building, software tools and development environment and model.

Others also recognized the role of cognition in program representation and comprehension [54, 55], design strategies and patterns [56, 57], and software design [58, 59]. These studies show the evolution of design paradigm and development tools from task-centered to human-centered. Current software development tools are very good at pinpointing errors and making sensible suggestions to avoid problems later. New derivatives are created to assist programmers. They have helped the software development process to be less error-prone in general. These studies paved the way for secure software development from the human aspect.

We believe this paper leverages those studies as stepping stones and investigate deeper in the human factor issues, in particular, to understand the impact of cognitive processes on software vulnerabilities. Like previous studies that lead to better software development tool, faster turnout, and more robust software integration, the insights obtained from this study can help the software security community gain insights, improve software security, better the design of guidelines, and build more effective vulnerability blind spot tools.

8. CONCLUSIONS

This paper investigated a hypothesis that software vulnerabilities are blind spots in developers' heuristics in their daily coding activities. Humans have been hardwired through evolution for adopting shortcuts and heuristics in decision making due to the limitations in their working memory. As vulnerabilities lie in uncommon code paths and we have a market that generates perverse incentives for insecure software, security information is often left behind from developer's repertoire of coding strategies.

A study with 47 developers using psychological manipulation was conducted to validate this hypothesis. In this study each developer worked for approximately one hour on six programming scenarios that contained vulnerabilities. The developers were told that the study's goal was to understand developer's mental models while coding. The sessions progressed from providing no information at all about possible vulnerabilities, to priming developers about unexpected results, and explicitly mentioning the existence of vulnerabilities in the code. The results show that developers in general changed their mindset towards security when primed about vulnerabilities on the spot. When not primed, even developers familiar with certain vulnerabilities failed to correlate them with their working scenario and fix them in the code when given a chance. Therefore, the assumption that developers should be educated about security and then apply what they learned while coding goes against the way the human brain behaves. This paper advocates that this assumption be reversed and that security information should reach developers when they need it, *on the spot* and correlated to their tasks at hand. The authors hope that these insights can influence the next generation of tools interfacing developers, such as IDE's, text editors, browsers and compilers so that more secure software reach the market.

Plans for future work include an investigation of the best methodologies for cueing developers and an analysis of the correlation of cueing effectiveness and previous security education.

Acknowledgments

We would like to thank the developers that participated in this study and the anonymous reviewers for valuable feedback. This research is funded by the National Science Foundation under grants CNS-1149730, CNS-1223588, CNS-1205415.

9. REFERENCES

[1] "Symantec Internet Security Threat Report 2013." `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf`.

[2] B. K. Marshall, "PasswordResearch.com Authentication News: Passwords Found in the Wild for January 2013." `http://blog.passwordresearch.com/2013/02/passwords-found-in-wild-for-january-2013.html`.

[3] "Seventeen steps to safer C code." `http://www.embedded.com/design/programming-languages-and-tools/4215552/Seventeen-steps-to-safer-C-code`.

[4] D. Kahneman and A. Tversky, "On the reality of cognitive illusions," *Psychological Review*, pp. 582–591, 1996.

[5] G. Gigerenzer, R. Hertwig, and T. Pachir, *Heuristics: The Foundations of Adaptive Behavior*. Oxford University Press, 2011.

[6] B. Schwartz, "The tyranny of choice," *Scientific American*, pp. 71–75, 2004.

[7] S. Botti and S. S. Iyengar, "The dark side of choice: When choice impairs social welfare," *American Marketing Association*, pp. 24–38, 2006.

[8] C. Kern, A. Kesavan, and N. Daswani, *Foundations of security: what every programmer needs to know*. Apress, 2007.

[9] E. Harmon-Jones, D. M. Amodio, and L. R. Zinner, *Social psychological methods of emotion elicitation (Handbook of Emotion Elicitation and Assessment)*. Oxford University Press, 2007.

[10] W. Thorngate, "Efficient decision heuristics," *Behavioral Science*, vol. 25, no. 3, pp. 219–225, 1980.

[11] K. V. Katsikopoulos, "Efficient decision heuristics," *Decision Analysis*, vol. 8, no. 1, pp. 10–29, 2011.

[12] J. W. Payne, J. R. Bettman, and E. J. Johnson, *The Adaptive Decision Maker*. Cambridge University Press, 1993.

[13] G. K. Zipf, *Human Behavior and The Principle of Least Effort*. Addison-Wesley, 1949.

[14] J. Rieskamp and U. Hoffrage, *Simple Heuristics that Make Us Smart*. Oxford University Press, 1999.

[15] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security*, pp. 63–78, Jan 1998.

[16] G. Wassermann and Z. Su, "Static Detection of Cross-site Scripting Vulnerabilities," in *30th International conference*

on Software engineering, ICSE '08, (New York, NY, USA), ACM, 2008.

[17] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, (New York, NY, USA), pp. 372–382, ACM, 2006.

[18] "Urllib and validation of server certificate." `http://stackoverflow.com/questions/6648952/urllib-and-validation-of-server-certificate`.

[19] W. S. McPhee, "Operating System Integrity in OS/VS2," *IBM Systems Journal*, vol. 13, no. 3, pp. 230–252, 1974.

[20] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," *ACM CCS*, 2005.

[21] R. E. Stake, *Qualitative Research: Studying How Things Work*. The Guilford Press, 2010.

[22] "Qualtrics (http://www.qualtrics.com/)."

[23] F. Gravetter and L. Wallnau, *Statistics for the Behavioral Sciences*. Wadsworth/Thomson Learning, 8th ed., 2009.

[24] R. S. Weiss, *Learning from Strangers - The Art and Method of Qualitative Interview Studies*. The Free Press, 1994.

[25] J. Saldana, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2012.

[26] A. Newell and H. Simon, *Human Problem Solving*. Prentice Hall, 1972.

[27] D. Denning, "A lattice model of secure information flow," *Communications of ACM*, 1976.

[28] R. Anderson, "Why information security is hard - an economic perspective," *ACSAC*, 2001.

[29] R. P. Abbot, J. S. Chin, J. E. Donnelley, W. L. Konigsford, and D. A. Webb, "Security Analysis and Enhancements of Computer Operating Systems," *NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards*, 1976.

[30] R. B. II and D. Hollingsworth, "Protection Analysis Project Final Report," *ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute*, 1978.

[31] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys*, vol. 26, no. 3, 1994.

[32] T. Aslam, "A Taxonomy of Security Faults in the UNIX Operating System," 1995.

[33] M. Bishop and D. Bailey, "A Critical Analysis of Vulnerability Taxonomies," *Technical Report CSE-96-11, University of California at Davis*, 1996.

[34] J. Crandall and D. Oliveira, "Holographic Vulnerability Studies: Vulnerabilities as Fractures in Interpretation as Information Flows Across Abstraction Boundaries," *New Security Paradigms Workshop (NSPW)*, 2012.

[35] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation." USENIX *;login*, December 2011.

[36] M. Samuel and U. Erlingsson, "Let's Parse to Prevent pwnage (invited position paper)," in *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*, LEET'12, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2012.

[37] V. Garg and L. J. Camp, "Heuristics and biases: Implications for security," *IEEE Technology & Society*, March 2013.

[38] H. K. Browne, W. A. Arbaugh, J. McHugh, and W. L. Fithen, "A trend analysis of exploitations," *IEEE Symposium on Security and Privacy*, 2001.

[39] R. Gopalakrishna and E. H. Spafford, "A Trend Analysis of Vulnerabilities," *CERIAS Tech Report 2005-05*, 2005.

[40] O. H. Alhazmi and Y. K. Malaiya, "Prediction capabilities of vulnerability discovery models," *IEEE Reliability and Maintainability Symposium (RAMS)*, pp. 86–91, 2006.

[41] O. H. Alhazmi and Y. K. Malaiya, "Towards a unifying approach in understanding security problems," *IEEE International Conference on Software Reliability Engineering (ISSRE)*, pp. 136–145, 2009.

[42] "Open Source Vulnerability Database (http://www.osvdb.org/)."

[43] Y. Wu, R. A. Gandhi, and H. Siy, "Using Semantic Templates to Study Vulnerabilities Recorded in Large Software Repositories," *ICSE Workshop on Software Engineering for Secure Systems*, 2010.

[44] S. Zhang, D. Caragea, and X. Ou, "An Empirical Study on using the National Vulnerability Database to Predict Software Vulnerabilities," *International Conference on Database and Expert Systems Applications (DEXA)*, 2011.

[45] A. Meneely and L. Williams, "Secure Open Source Collaboration: An Empirical Study of Linus' Law," *ACM CCS*, pp. 453–462, 2009.

[46] G. Schryen, "A comprehensive and comparative analysis of the patching behavior of open source and closed source software vendors," *IMF*, 2009.

[47] F. Asgapour, D. Liu, and L. J. Camp, "Mental models of computer security risks," *Financial Cryptography and Data Security Lecture Notes in Computer Science*, vol. 4886, pp. 367–377, 2007.

[48] D.-L. Huang, Pei-Luen, P. R. abd Gavriel Salvendya, F. Gaoa, and J. Zhoua, "Factors affecting perception of information security and their impacts on it adoption and security practices," *International Journal of Human-Computer Studies*, vol. 69, no. 12, 2011.

[49] V. Garg and L. J. Camp, "End user perception of online risk under uncertainty," *Hawaii International Conference On System Sciences*, vol. 4886, 2012.

[50] B. Fischhoff, P. Slovic, S. Lichtenstein, and B. C. Stephen Read, "How safe is safe enough? a osychometric study of attitudes towards technological risks and benefits," *Policy Sciences*, vol. 9, no. 2, 1978.

[51] C. Bravo-Lillo, L. Cranor, J. Downs, and S. Komanduri, "Bridging the gap in computer security warnings: A mental model approach," *IEEE Security and Privacy*, vol. 9, no. 2, 2011.

[52] K. Witte, "Putting the fear back into fear appeals: The extended parallel process model," *Communication Monographs*, vol. 59, no. 4, pp. 329–349, 1992.

[53] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.

[54] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.

[55] H. C. Purchase, L. Colpoys, M. McGill, D. Carrington, and C. Britton, "Uml class diagram syntax: an empirical study of comprehension," in *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation-Volume 9*, pp. 113–120, Australian Computer Society, Inc., 2001.

[56] A. Chatzigeorgiou, N. Tsantalis, and I. Deligiannis, "An empirical study on students ability to comprehend design patterns," *Computers & Education*, vol. 51, no. 3, pp. 1007–1016, 2008.

[57] W. Visserl, J.-M. Hocz, and F. Chesnay, "Expert software design strategies," 1990.

[58] R. Jeffries, A. A. Turner, P. G. Polson, and M. E. Atwood, "The processes involved in designing software," *Cognitive skills and their acquisition*, pp. 255–283, 1981.

[59] B. Adelson and E. Soloway, "A model of software design," *International Journal of Intelligent Systems*, vol. 1, no. 3, pp. 195–213, 1986.