Vulnerabilities as Blind Spots in Developer's Heuristic-Based Decision-Making Processes

Justin Cappos
NYU
Dept. of Computer Science and Engineering
jcappos@nyu.edu

Yanyan Zhuang
NYU
Dept. of Computer Science and Engineering
yyzh@nyu.edu

Daniela Oliveira
University of Florida
Dept. of Electrical and Computer Engineering
daniela@ece.ufl.edu

Marissa Rosenthal
Bowdoin College
Dept. of Psychology and Dept. of Computer Science
mrosenth@bowdoin.edu

Kuo-Chuan Yeh
Pennsylvania State University
Dept. of Computer Science and Engineering
yeh@cse.psu.edu

ABSTRACT

The security community spares no effort in emphasizing security awareness and the importance of building secure software. However, the number of new vulnerabilities found in today's systems is still increasing. Furthermore, old and well-studied vulnerability types such as buffer overflows and SQL injections, are still repeatedly reported in vulnerability databases. Historically, the common response has been to blame the developers for their lack of security education. This paper discusses a new hypothesis to explain this problem by introducing a new security paradigm where software vulnerabilities are viewed as developers' *blind spots* in their decision making. We argue that such a flawed mental process is heuristic-based, where humans solve problems without considering all the information available, just like taking shortcuts. This paper's thesis is that security thinking tends to be left out by developers during their programming, as vulnerabilities usually exist in corner cases with unusual information flows. Leveraging this paradigm, this paper introduces a novel methodology for capturing and understanding security-related blind spots in Application Programming Interfaces (APIs). Finally, it discusses how this methodology can be applied to the design and implementation of the next generation of automated diagnosis tools.

1. INTRODUCTION

Despite the security community's emphasis on the importance of secure software, the number of new vulnerabilities found in today's systems is still increasing. According to the 2014 report by Symantec Internet Security [1], almost 7000 new vulnerabilities occurred in 2013. In addition, vulnerabilities that have been studied for years, such as buffer overflows and SQL injections, are still commonly reported. The inability to properly address security issues indicates that security problems will continue have impact for the foreseeable future.

Historically, a common response has been to blame the developers for their lack of security education. For example, in referring to the cause for the SQL injection issues that have lead to password database disclosures, one response was *"[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]. Decrying the lack of security education, and thus blaming the developer, is easy to do in the case of a vulnerability.

This paper proposes a new security paradigm where **software vulnerabilities are viewed as *blind spots* in developer's heuristic-based decision-making processes**. This paper's thesis is that the nature of insecure software and increasing number of well-studied vulnerabilities lies in the developers' mental processes. Instead of criticizing the lack of security education from the developer's part, we argue that the problem is that developers use heuristics during their everyday programming tasks. Heuristics are unconscious human actions that do not use all information available to reach a particular decision or course of action, just like taking shortcuts. When applied to everyday programming tasks, heuristics do not include security or vulnerability information, because software vulnerabilities represent uncommon cases not completely understood by developers and exercise unusual information flows. As a result, security thinking is usually left out from developer's heuristics.

It is known in psychology that humans have been hardwired during thousands years of evolution for employing shortcut, heuristic-based decision-making processes [3, 4]. When faced with many choices, possibilities, and information, humans become overwhelmed and begin to make sub-optimal decisions due to their short working memory [5, 6]. Heuristics are an adaptive human response in such an environment. However, they can lead to errors with deleterious consequences.

Based on this security paradigm, this paper introduces a new methodology to capture, understand, and address security-related blind spots in Application Programming Interfaces (APIs). APIs are a large avenue for blind spots with great impact in software security because they are often leveraged by developers from outside groups. A developer assumes a certain API behavior, which

NSPW'14 September 15 - 18 2014, Victoria, BC, Canada
Copyright 2014 ACM 978-1-4503-3062-6/14/09...$15.00
http://dx.doi.org/10.1145/2683467.2683472

does not hold in special cases or across different implementations of the API. In this paper such a misconception is called a *blind spot*, which often creates security vulnerabilities in software leveraging the API.

The main idea of the proposed methodology is to identify areas in APIs where the API behavior expectation for the API designer and the application developer who uses the API diverge. The behavior expectation of an API is a person's intuitive perception about the input to an API (causes), and the corresponding output (consequences). A blind spot is a divergence of the behavior expectation of the API designer and the application developer. It is similar to a blind spot in a vehicle, an area that cannot be directly observed by the driver and can potentially lead to hazards.

The methodology described in this paper specifically targets security-related blind spots in APIs. It focuses on APIs because they are particularly amenable to analysis: (i) they prescribe the methods by which a program interacts with external data, (ii) human-readable information such as documentation is frequently available about its behavior, (iii) there are common tools and techniques to log API calls and return values (*e.g.*, at the application and the OS level), and (iv) misunderstandings about API behavior are often the root cause of security vulnerabilities.

This methodology leverages cognitive puzzles to capture, understand, and address blind spots in APIs. To get meaningful blind spot data for analysis, a large set of *cognitive puzzles* is generated from reports found in vulnerability databases and bug trackers. A cognitive puzzle is an exhibit question, which sharpens a respondent's concentration by asking them to respond to a specific artifact. In this methodology, a puzzle is a question about a snippet of code capturing the vulnerability with extraneous information removed and necessary context information added. Sets of puzzles representing real vulnerabilities are distributed to developers through automated online surveys. After developers' answers to the puzzles are collected, they are statistically analyzed to find the actual blind spots, their features and correlations among each other. If open-ended questions are also employed in the puzzle session, they can be analyzed with standard qualitative research triangulation methods [7] to increase the confidence of results interpretation. The features collected about blind spots can be used to build a supervised learning model [8] to find other blind spots in the same API. Figure 1 shows a high level view of the methodology.

This methodology can be leveraged to build automated tools to diagnose blind spots in APIs and perform other types of analyses, such as blind spot classification and estimation of the likelihood that an API contains a certain set of blind spots. Blind spot knowledge can also be employed in automatic tools that cue developers *on-the-spot* about blind spot functions in an API during coding. Using a better understanding of blind spots, these tools can analyze program behavior with the blind spots that developers create mentally and use practically. This knowledge can also lead to a better understanding of how to design and construct APIs that have as few blind spots as possible, which will reduce the occurrence of vulnerabilities.

In summary, this paper has the following contributions:

1. A new security paradigm that views software vulnerabilities as blind spots (biases) in the heuristic-based decision making processes used by developers.
2. A methodology leveraging this paradigm for capturing, understanding and addressing blind spots in APIs.
3. A discussion of how such methodology can be applied in the design of the next generation of automated tools for vulnerability prevention, detection and analysis in code leveraging APIs.

This paper is organized as follows. Section 2 discusses the heuristic-based decision making processes used by humans and how it relates to software vulnerabilities. Section 3 describes in details the proposed methodology for capturing, understanding and addressing blind spots in APIs. Section 4 discusses how this methodology can be leveraged to build automated diagnosing tools for APIs. Section 5 discusses related work. Section 6 summarizes the paper's lively discussion among the NSPW attendees. Finally, and section 7 concludes the paper.

2. SECURITY PARADIGM: HEURISTICS AND VULNERABILITIES

Psychology research has shown that during evolution, humans have become hardwired for shortcut and heuristic-based decision-making processes [3]. Heuristics are cognitive processes that people use to make judgments, decisions and perform tasks [9]. They are simple computational models that allow one to quickly find feasible solutions and that do not necessarily use all information available. Heuristics 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 to perform a particular task. Humans have a short working memory, which makes some cognitive processes difficult [10] when they are confronted with too much information, possibilities, and choices. In such cases, humans employ sub-optimal decision-making processes and tend to make mistakes [5]. This argument is reinforced by Zipf's principle of least effort [11], which states that people use as little effort as necessary to solve a problem. Heuristics are an adaptive response to human's limited working memory. They tend to have high predictive accuracy, especially when information is scarce, but they can lead to severe biases and errors [9] in decision making and ensuring correctness of tasks.

Kieskamp and Hoffrage [12] also argue that under time pressure, a common situation in software development environments, people tend to adopt heuristics that are even simpler, and that do not require much integration of information. According to Thorngate [9], people tend to ignore information or leverage a small amount of information in their heuristics because (i) they do not notice certain issues of a particular problem, (ii) they do not care about the problem being solved that much, (iii) there are small or infrequent decrements in reward that result from their ignorance or misuse of relevant information about the problem in hand, and (iv) the time and effort to use it properly may be more costly than any decrease in payoffs associated with their occasional sub-optimal choices.

The security paradigm introduced in this paper views software vulnerabilities as blind spots (biases or errors) in developer's heuristic-based decision-making processes. In other words, developers introduce vulnerabilities in software because they use heuristics to make decisions and perform their tasks. When program developers are constantly making heuristic-based decisions, consciously or unconsciously, these heuristics are mostly about finding *a* solution or an efficient solution to a particular problem. As software vulnerabilities lie in corner cases and unusual information flows, security thinking tends to be left out of the heuristics adopted by developers.

3. UNDERSTANDING SECURITY BLIND SPOTS WITH COGNITIVE PUZZLES

This section describes a methodology for capturing and understanding blind spots in APIs. This methodology is based on the proposed paradigm that software vulnerabilities are blind spots in developers' heuristic-based decision making processes.

Knowledge of blind spots improves our understanding of the information developers include and leave behind while making programming decisions. This understanding can be leveraged to improve software security by building automated diagnosis tools that focuses on these blind spots (Section 4) and designing APIs by taking into account these common developer's biases. The proposed methodology focuses on API blind spots because APIs are particularly amenable to such type of analysis. Due to ambiguity in portions of API documentations and hidden assumptions, developers often have misconceptions about API behavior, which leads to vulnerabilities in software leveraging the API [13].

Specifically to APIs, the hypothesis addressed by this methodology is that blind spots are programmer's misconceptions about certain portions of an API and are caused by the developer's heuristic-based decision-making processes. The goal of the methodology is to locate security-related blind spots and understand their causes, where they are commonly located, and how to detect and address them. This methodology, detailed in the next subsections, is illustrated in Figure 1.

Please notice that the focus of this methodology is on API-related blind spots, which often cause vulnerabilities in software leveraging the API. However, there are software vulnerabilities which are not API-related and are not considered in the proposed methodology. For example, consider a web application that prompts a login authentication page. Now assume that the authentication module provides different messages for the case of an inexistent user name (*"Invalid user name!"*) and valid user name, but wrong password (*"Authentication failed!"*). This program contains a security vulnerability because it allows an adversary to discover valid user names, which facilitates later brute-force or dictionary attacks [14]. However, this type of vulnerability is out of the scope of the proposed methodology. We only focus on API-related vulnerabilities, where the vulnerability lies on a blind spot related to the behavior of an API function. Classic examples of API-related vulnerabilities are buffer overflows [15] (`strcpy() function`) and Time-To-Check-To-Time-To-Use (TOCTTOU) [16] (`access() and open() functions`) vulnerabilities.

This work considers software vulnerabilities as a subset of software bugs [17]. A software bug is an error or a fault in the program's design or implementation that causes the program to act in an unpredictable or erroneous way. A general bug is usually a failure from the part of the developer to implement certain functionality according to the software specification of requirements. A software vulnerability is a weakness in the design or implementation of a piece of software that allows it to be exploitable by an adversary in a way that compromises security: the program integrity, the confidentiality of sensitive data handled by the program, or the availability of functionality to the end users. Vulnerabilities are also a defect or bug because they cause the software to behave in ways its developers did not intend it to behave. As security is usually not treated as a first class citizen in most software development projects, security requirements are seldom specified. As argued by this paradigm, due to the heuristic-based decision-making processes adopted by people, non-specified and non-priority aspects of software development will likely to be left out of developers working memory. On the other hand, if a security requirement is specified but not met, it is considered a software defect that is not a vulnerability. Figure 2 illustrates these relationships.

3.1 Extracting Potential Blind Spots from Common APIs

Potential blind spots can be extracted in one of two ways. The first possibility is to manually collect a great number of potential blind spots in the form of vulnerabilities and bug reports from common APIs (JavaScript, POSIX, C library, *etc.*) as the primary data to be analyzed. These vulnerabilities can be collected from vulnerability databases like SecurityFocus [18], NVD [19] and OSVDB [20], and bug trackers of popular software projects, such as Firefox, Apache, Thunderbird, SeaMonkey, *etc.* The main idea in this approach is to discover whether a particular blind spot is common. In essence, this is checking to see if known vulnerabilities are likely to be repeated by multiple developers. In this case, all vulnerabilities and bugs collected are very likely to be blind spots.

The second technique is to look at the possible calls to an API and use this to find potential blind spots. One way of performing this is to use fuzzing [21] or symbolic execution [22] to find out which variations of parameters impact the output in non-obvious ways. One could then derive potential blind spots for these cases and use them to understand blind spots in general. This essentially sweeps the space of possible blind spots and tests programmers on their understanding. While the search space may be impractically large for complex APIs with a great number of functions, this will have more complete coverage than the prior technique.

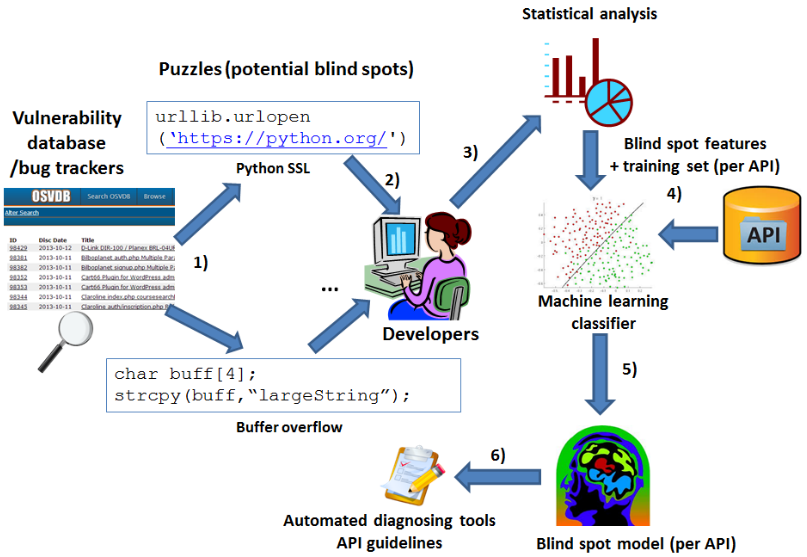
3.2 Converting Vulnerabilities into Cognitive Puzzles

After the collection phase, these potential blind spots are manually transformed into cognitive puzzles that can capture the developer's understanding (or misunderstanding) about them.

A puzzle is an exhibit question, which sharpens the respondents concentration by asking them to respond to a specific statement, story, or artifact. The goal is to give the respondent a hint to examine and draw out a recollection, interpretation or judgement [7]. The puzzle is not a mere reproduction of a vulnerability (blind spot). It is a short code snippet that formulates the underlying problem in an intelligent way, exercising it, adding all necessary context, and removing unnecessary noise. Noise is removed in a careful way, as vulnerabilities are often located in hidden cases and also in unusual information flows.

3.2.1 A Blind Spot and Puzzle

As an example of what a blind spot is and how its corresponding puzzle can be generated, consider a recently reported vulnerability in the Python `urllib` library that makes web applications vulnerable to man-in-the-middle attacks [23]. The `urllib` library provides a SSL protocol API to Python applications. However, the function `urlopen` (used by a client to open a SSL connection with a server) does not validate certificates received from the server. Normally when an application (SSL client) or API checks SSL certifi-


Statistical analysis
Puzzles (potential blind spots)
Vulnerability database /bug trackers
OSVDB
Python SSL
1)
2)
3)
...
Developers
Blind spot features + training set (per API)
4)
API
Machine learning classifier
5)
Buffer overflow
6)
Automated diagnosing tools API guidelines
Blind spot model (per API)

```
urllib.urlopen
('https://python.org/')
```

```
char buff[4];
strcpy(buff,"largeString");
```

Figure 1: Capturing and understanding blind spots - Overview.

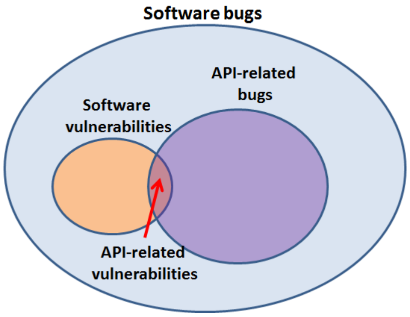


Figure 2: Software bugs and vulnerabilities.

cates, 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 site. As a result of this vulnerability, any malicious party can provide a certificate claiming to be an official website, and the fake certificate will be trusted.

Consider the snippet of code below from a hypothetical Python SSL client, where the client needs to connect to a server via SSL, fetch the certificate, check whether it is valid (signed properly) and belongs to the server the client is attempting to connect. Also, suppose a web server exists with a valid certificate issued by a root CA for *puzzles.poly.edu*.

```
responsefileobj = urllib.urlopen('https://puzzles.poly.edu')
```

This code snippet can be presented to a developer as a puzzle, where the assumptions mentioned above are provided and the following questions (open-ended and/or multiple choice) are asked.

1. Will the `urlopen` call succeed or fail? Why?
2. If the client uses the site's IP address (1.2.3.4) instead of the server name, will the `urlopen` call succeed or fail? Why?
3. What do you expect to happen if your web browser attempts to establish an HTTPS connection with the host *puzzles.poly.edu* using the server name?
4. What do you expect to happen if your web browser attempts to establish a connection over HTTPS using the server's IP address?
5. What happens if a web server with a self-signed certificate for *puzzles.poly.edu* intercepts and services the request of the Python SSL client above instead?

For question 1, the connection succeeds because `urlopen` correctly establishes an SSL connection for the given page. This is a basic question that we expect any programmer to answer correctly if they understand the basics of the API and SSL. For question 2, one might expect the connection to fail because the server will not have a certificate for its IP address. However, the connection will succeed because `urlopen` does not validate certificates. This questions checks whether the developer understands how SSL works in general: the certification validation process involves checking whether the name on the certificate (`puzzles.poly.edu`) is the same as the host that the client is attempting to connect via SSL (1.2.3.4). For questions 3, the SSL client is a web browser that validates certificates. As `puzzles.poly.edu` will send a valid certificate during the SSL handshake process, the connection will succeed. For question 4, the connection will fail because the browser cannot match the host being connected (1.2.3.4) with the server name on the certificate. The situation described in question 5 is a classic man-in-the-middle attack. The connection with the bogus web server will succeed because `urlopen` does not validate certificates. If programmers have a blind spot about the fact that `urlopen` fails to validates SSL certificates, we would expect them to answer questions 1, 3, and 4 correctly, but give incorrect answers for 2 and 5.

On applying such methodology, puzzles can be given to developers in three formats: multiple choice, open-ended questions, or a combination of both. The use of only multiple choice questions streamlines the analysis of the results. On the other hand open-ended questions can provide richer insights on the nature of blind spots.

3.3 Understanding Blind Spots

Each potential blind spot collected and transformed into a puzzle will be related to an API function, which will have a number of attributes, such as number, types, and order of parameters, category (memory, I/O, string, network, *etc.*), frequency of changes, and popularity. The answers to the puzzles can provide a deeper understanding about security-related blind spots, which include:

1. Which potential blind spots are real blind spots? In other words, which API functions have behavior not well-understood by the majority of developers?
2. How do blind spots correlate with one another? For example, do developers that have blind spots for memory functions with a certain number and types of parameters, also have blind spots for the same type of I/O functions from the same API?
3. What features all blind spots have in common?
4. Which API areas, for instance, memory, I/O, string, network *etc*, are more likely to cause blind spot in developers?
5. Do developers tend to make the same type of assumptions in functions they have blind spots?

In statistics, one way to abstract a concrete problem is to extract some features to create some abstract dimensions. In the case of potential blind spots, examples of features are the attributes of the API functions and developer's score on the puzzles. Each feature is a dimension in an abstract high-dimensional space. Statistical methods such as Analysis of Variance (ANOVA) [24] and Principal Component Analysis (PCA) [25] can be used to analyze the data in this space and measure correlation among data samples. In particular, PCA can find a *principal component* (a vector in the d-dimensional space) that along this vector, data points are highly correlated. It can be used to find what developers think in common, and how their thoughts deviate in other dimensions.

Features extracted from actual blind spots and the set of actual blind spots can be used to train a machine learning classifier that can find other blind spots in the studied API. This blind spot knowledge can also be used to estimate the likelihood that an unseen API contains a certain set of blind spots.

4. APPLICATION TO DIAGNOSIS TOOLS

This section discusses applications of the proposed methodology to capture and understand blind spots in APIs: making use of blind spots's knowledge from developers to prevent, detect and analyze vulnerabilities in software using the API. This knowledge also allows for the inference of unknown blind spots in other APIs.

4.1 Blind Spot Checking

The collection of blind spot functions, their expected values according to developers and ground-truth values specified by an API expert can serve as input to dynamic diagnosis tools. An intuitive direction for an automated tool for blind spot detection is to construct a model capturing the characteristics of blind spots, and compare this model with developer's expectations of behaviors for the same API. This approach is advantageous compared to traditional ones. By leveraging the knowledge about blind spots, such tool can target more general APIs and can be used to check vulnerabilities in a wide variety of applications. Rather than finding the *generic* root cause of software vulnerabilities, the majority of today's diagnosis tools target at domain-*specific* problems by applying application-specific techniques, such as log analysis [26], program system call graphs [27], reverse engineering [28], and model checking [29].

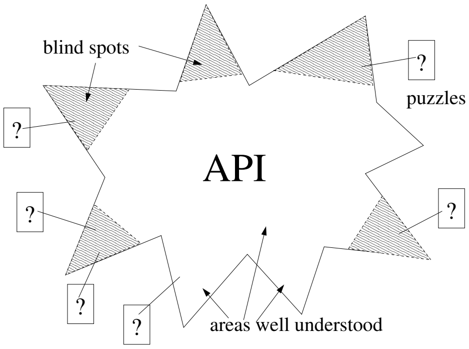


Figure 3: Puzzles and blind spots: APIs have blind spots, and the puzzles exercise portions of the API that are not well-understood. These API portions may or may not have blind spots. The proposed methodology defines as a blind spot an area of an API not well-understood by the majority of the developers.

One possibility is to compare program execution traces with blind spots (found through the methodology described in Section 3) during application execution. For example, with the information collected about blind spots a model of blind spot functions in the API can be built. As part of this model, the tool will have access to which functions are actual blind spots and what values and types of parameters indicate a misunderstanding of the API blind spot function from the developer's part as shown at runtime. For functions identified as blind spots, the majority (but not all) of the developers will use the function in a way that will cause unintended consequences, which can lead to vulnerabilities. These cases should be detected and flagged by the tool at runtime.

Therefore, the blind spot model generated for APIs will cover the portions of APIs that are not well-understood by the majority of developers. As shown in Figure 3, areas well understood are the facts about an API that are widely recognized and accepted, and thus encoded in the developer's heuristics. In the methodology proposed here, the puzzles exercise portions of the API that may or may not be blind spots, with the goal to find actual blind spots, *i.e.*, portions of an API not well-understood by the majority of the developers.

The knowledge about a particular API's blind spots can also be used in tools that automatically cue developers *on-the-spot* about the usage of blind spot functions with examples of poorly-understood usage. This type of tool can be integrated with IDEs and popular coding text editors, such as VI or emacs.

4.2 Inferring Unknown Blind Spots

A promising approach to analyze and infer unknown blind spots is to leverage Fuzzy Clustering [30] theory to estimate the *likelihood* that an API contains a certain set of blind spots. The blind spots can be discovered through the proposed methodology in Section 3. The input is a particular API, which can be classified into having more than one blind spot, with a goodness score indicating the strength of the association between the input API and a particular blind spot. Fuzzy Set theory [31] is particularly relevant for this problem because it allows one to define the likelihood of an element belonging to a set, by the degree of membership.

The main idea is to categorize the features of blind spots to form a feature space, from which feature vectors for each API and blind spots can be created. For example, the number, type and order of parameters, returned values and type of an API indicate the input and output. Allocated memory, network and I/O operations are the behavioral features of an API. These discrete features can be projected into a high-dimensional numeric space where classification algorithms can be applied. Based on the feature vectors, the distance between the API and blind spots can be calculated according to a membership function. The classification process takes the distance into account to determine how likely an API has a blind spot. The assigned membership level indicates the strength of the association between an unseen API and the severity of a particular blind spot.

By identifying potential blind spots, this information can be used to further examine the API using puzzles (Section 3), thus increasing the accuracy of diagnosing tools for that particular API.

5. RELATED WORK

The security paradigm and methodology proposed leverages knowledge from the areas of vulnerability analysis, human factors in software development, information security perception and economics, and automated diagnosis tools. This section discusses relevant related work in these areas.

5.1 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 [32]. Around the same time, the Protection Analysis study [33] focused on developing vulnerability detection tools to assist developers. Other vulnerability studies followed, such as the taxonomies by Landwehr *et al* [34] and Aslam [35]. In the 1990's, Bishop and Bailey [36] 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. Crandall and Oliveira proposed a view of vulnerabilities as fractures in the interpretation of information as it flows across boundaries of abstraction [37].

There are also discussions about the theoretical and computational science of exploit techniques and proposals for explicit parsing and normalization of inputs [38–40]. Bratus *et al.* [38] discussed the view that the theoretical language aspects of computer science lie at the heart of practical computer security problems, especially exploitable vulnerabilities. Samuel and Erlingsson [40] proposed input normalization via parsing as an effective way to prevent vulnerabilities that allow attackers to break out of data contexts.

Researchers have also studied vulnerability trends. Browne *et al* [41] determined that the rates at which incidents were reported to CERT could be mathematically modeled. Gopalakrishna and Spafford [42] analyzed software vulnerabilities in five critical software artifacts using information from public vulnerability databases to predict trends. Alhazmi *et al* [43] presented a vulnerability discovery model to predict long and short term vulnerabilities for several major operating systems. Anbalagan and Vouk [44] analyzed and classified thousands of vulnerabilities from OSVDB [20] and discovered a relationship between vulnerabilities and exploits. Wu *et al* [45] performed an ontology-guided analysis of vulnerabilities and studied how semantic templates can be leveraged to identify further information and trends. Zhang *et al* [46] analyzed vulnerabilities from the NVD database using machine learning to discover the time to the next vulnerability for a given software application.

There are also studies on developer's practices. Meneely and Williams [47] studied developers collaboration and unfocused contributions into developer activity metrics and statistically correlated them. Schryen [48] 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.

The difference between these works and the proposed paradigm is that it leverages human psychology to understand the nature of software vulnerabilities.

5.2 Human Factors in Software Development

Using human factors in technology research is not a new concept. Curtis, Krasner, and Iscoe [49] 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 [50], design strategies and patterns [51], and software design [52]. 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 understanding secure software development from the human viewpoint, as being proposed in this paper.

Wurster and van Oorschot [53] argue that the security community should conservatively view developers as the enemy and take the security of systems out of their hands. This is achieved by developing security technical solutions instead of assuming developers will be educated to do the right thing. The paradigm proposed here is complementary to the author's claim by proposing to increase the probability of developers thinking about security issues through cueing *on-the-spot*.

5.3 Information Security Perception

The concept of blind spots is intimately related to the way developers perceive security and the risk of software vulnerabilities while programming. Risk is the probability that an undesirable event will occur and it has been shown to have the potential to influence people's attitudes and attention [54]. There is broad research addressing risk and information perception of non-expert Internet users.

Asghapours *et al* [55] advocate the use of mental models of computer security risks for improvement of risk communication to non-expert end users. Based on the literature, the authors leveraged five mental models, as simplified internal concepts of how something works in reality: Physical Safety, Medical Infections, Criminal Behavior, Warfare, and Economic Failures. Their user study showed that people's mental models of security risks correlated with their level of expertise.

Garg and Camp [56] adopted the classic Fischhoff's canonical nine dimensional model of offline risk perception [57] to better understand online risk perceptions. A user study was performed to identify the dimensions of online risk perceptions of end users. Results obtained for online risks differed from the ones obtained for offline risks. In addition, the severity of a risk was the biggest factor in shaping risk perception.

In the area of decision-making Garg and Camp [58] identified systematic errors by decision-makers leveraging heuristics as a way to improve security designs for risk averse people.

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

5.4 Economics of Computer Security

One of the arguments of this paper is that due to people's working memory limitations, basic functionality and performance issues leave very little room for security thinking while developing software. Psychological research [12] also documents that under time pressure, people tend to use even simpler heuristics requiring less integration of information. This hypothesis corroborates many studies in the area of economics of computer security that argue that developers have "perverse" incentives [59] to develop insecure software, such as time-to-market pressures and lack of accountability.

There are several studies that have investigated issues related to the economics of computer security. In a classic paper, Anderson [59] discusses how society provides very high economic incentives for a market of insecure software. The party who is in a position to protect a system is not the party that suffers the results of a security failure. The computer software and systems market selects software and systems that reach the user quickly and as feature-rich as possible. In a similar fashion, Herley [60] argues that the user's rejection of security advice is rational economically. On one one hand, the received advice offers users protection from the direct costs of attacks, such as identity theft. On the other hand, it burdens the users with indirect costs in the form of cognitive effort, such as time invested trying to understand complex security tips. Similarly, Herley argues [61] that "more is not the answer", but that security advice should be effort neutral.

5.5 Diagnosis Tools

Rather than finding the generic root cause of software vulnerabilities, the majority of today's diagnosis tools target at domain-specific problems by applying application-specific techniques. There are several categories of these tools.

The work in validation of API behaviors for vulnerability detection include runtime verification [62], logic formalisms [63] and model-based testing [64]. These techniques are effective at finding bugs that persist in an API, but many of them are difficult or impractical to use.

Subtle and complicated vulnerabilities in software systems depend on specific sequences of APIs execution. Some work has focused on centralizing distributed processes in order to avoid the asynchronous communication and to properly apply application-specific model checking [65].

Several tools have been developed for fault or vulnerability detection that use machine learning algorithms [66]. Supervised machine learning (ML) derives a signature from application traces or network packet traces [67].

6. WORKSHOP DISCUSSION

In the lively discussion of this paper during the workshop, Bob Blakley noted that the psychological difficulty of choice is what Alvin Toffler called "future shock". Abe Singer suggested that Google could try giving developers the right answers to secure code, as developers heavily rely on Google results while coding. He also noted that API names and documentation are actively misleading. Ben Edwards argued that developers are responsible for not checking security issues, but there is also a testing failure involved.

Nathaniel Husted thinks that there is a chain of custody issue that holds API developers responsible for security issues involving the API. Jeremy Epstein argued that the threat environment might change between time of API design and use. Sean Peisert wonders whether developers might be "lazy" or afraid of what they might find in code if they examine it further. Michael Locasto pointed out that the goal of using an API is to reduce developers' workload via abstraction. Thus, developers shouldn't have to break out of the idea of encapsulation and examine API issues further.

Konstantin Beznosov suggested looking at work on use of checklists in healthcare as a source of techniques for helping developers making good decisions. Tom Longstaff noted that the decision-making literature assumes developers know where the blind spots are. Richard Ford stressed the importance of cueing reinforcement and Anil Somayaji suggested employing cues like a game process to increase cueing effectiveness.

Jeremy Epstein also mentioned that people used to write code which was designed to suppress warnings without fixing the associated problems. He gave the example of a Debian developer who created a vulnerability in OpenSSL by making a modification to suppress a warning.

Mary Ellen Zurko wonders which one is harder: finding blind spots or cueing developers. Paul van Oorschot thinks that there is also an incentive problem, as developers are rewarded based on productivity.

Wolter Pieters thinks that the authors are actually proposing to change the content of developer's heuristics to include security information, which the authors agree. Andreas Poller questioned whether the blind spots are simply lack of knowledge, or developers know that they are in a context where the security knowledge applies, but they are not incentivized to use this knowledge. Tom Longstaff thinks that the authors need to consider the demographics and education level of developers taking puzzles, as it will change the set of blind spots discovered. Ben Edwards thinks that the blind spots are not programming language agnostic. Finally, Steve Myers suggests a clear separation of blind spot discovery (crowdsourcing) and cueing developers.

7. CONCLUSIONS

This paper re-evaluated the root cause of software vulnerabilities as blind spots in the heuristics developers employ during their everyday decision-making processes. Evolution has hardwired humans for shortcut heuristic-based decision-making processes as an adaptive defense against their short working memories. Heuristics require less cognitive effort to solve problems because they do not use all information available, but they can lead to serious errors in judgment.

This paper's thesis is that as vulnerabilities lie in hidden cases and uncommon information flows, security thinking is not included in developers' heuristics. Then, programming paths with hidden assumptions become blind spots. The paper also proposed a methodology for capturing and understanding security-related blind spots in APIs that leverages this paradigm. This methodology involves the creation of cognitive puzzles exercising potential blind spots, which are presented to a large number of developers.

Considering vulnerabilities from a Psychology viewpoint is not well explored and the authors believe that this hypothesis helps explain why the number of vulnerabilities keeps increasing, and why well-understood vulnerabilities are still commonly reported.

Acknowledgments

We would like to thank the NSPW anonymous reviewers, our shepherd, Julie Ard, and all the workshop attendees for valuable feedback. Justin Cappos is supported under grant CNS-1223588. Daniela Oliveira and Marissa Rosenthal are supported under grant CNS-1149730.

8. REFERENCES

[1] "Symantec - Internet Security Threat Report (http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_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] D. Kahneman and A. Tversky, "On the Reality of Cognitive Illusions," *Psychological Review*, pp. 582–591, 1996.

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

[5] B. Schwartz, "The Tyranny of Choice," *Scientific American*, pp. 71–75, 2004.

[6] S. Botti and S. S. Iyengar, "The Dark Side of Choice: When Choice Impairs Social Welfare," *American Marketing Association*, pp. 24–38, 2006.

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

[8] I. W. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2005.

[9] W. Thorngate, "Efficient Decision Heuristics," *Behavioral Science*, vol. 25, no. 3, pp. 219–225, 1980.

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

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

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

[13] "Openssh 5.1." Accessed May 2nd, 2012 http://www.openssh.com/txt/release-5.1.

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

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

[17] M. Pinzger, N. Nagappan, and B. Murphy, "Can Developer-Module Networks Predict Failures?," *ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 2–12, 2008.

[18] "SecurityFocus (http://www.securityfocus.com/)."

[19] "National Vulnerability Database (http://nvd.nist.gov/home.cfm)."

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

[21] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services," tech. rep., 1995.

[22] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

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

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

[25] I. T. Jollife, *Principal Component Analysis*. Springer, 2002.

[26] I. Beschastnikh, "Inferring networked system models from behavior traces," in *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pp. 13–14, ACM, 2012.

[27] E. Eskin, W. Lee, and S. J. Stolfo, "Modeling system calls for intrusion detection with dynamic window sizes," in *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, vol. 1, pp. 165–175, IEEE, 2001.

[28] A. Ulrich and A. Petrenko, "Reverse engineering models from traces to validate distributed systems–an industrial case study," in *Model Driven Architecture-Foundations and Applications*, pp. 184–193, Springer, 2007.

[29] J. Yang, C. Sar, and D. Engler, "Explode: a lightweight, general system for finding serious storage system errors," in *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, (Berkeley, CA, USA), pp. 131–146, USENIX Association, 2006.

[30] E. H. Ruspini, "A new approach to clustering," *Information and control*, vol. 15, no. 1, pp. 22–32, 1969.

[31] "Fuzzy set." http://en.wikipedia.org/wiki/Fuzzy_set.

[32] 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.

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

[34] 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.

[35] T. Aslam, “A Taxonomy of Security Faults in the UNIX Operating System,” 1995.

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

[37] 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.

[38] 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.

[39] W. Pieters and L. Consoli, “Vulnerabilities As Monsters: The Cultural Foundations of Computer Security,” *Proceedings of the European Computing and Philosophy Conference (E-CAP)*, 2006.

[40] 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.

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

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

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

[44] 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.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] J.-M. Hoc, “Role of mental representation in learning a programming language,” *International Journal of Man-Machine Studies*, vol. 9, no. 1, pp. 87–105, 1977.

[51] 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.

[52] 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.

[53] G. Wurster and P. C. van Oorschot, “The Developer is the Enemy,” *New Security Paradigms Workshop (NSPW)*, pp. 89–97, 2008.

[54] M. S. Wogalter, D. DeJoy, and K. R. Laughery, *Warnings and Risk Communication* . CRC Press, 1999.

[55] 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.

[56] V. Garg and L. J. Camp, “End User Perception of Online Risk Under Uncertainty,” *Hawaii International Conference On System Sciences*, vol. 4886, 2012.

[57] 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.

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

[59] R. Anderson, “Why Information Security is Hard - An Economic Perspective,” *ACSAC*, 2001.

[60] C. Herley, “So Long, and No Thanks for the Externalities: the Rational Rejection of Security Advice by Users,” *New Security Paradigms Workshop (NSPW)*, 2009.

[61] C. Herley, “More is Not the Answer,” *IEEE Security & Privacy magazine*, 2014.

[62] S. Park, S. Lu, and Y. Zhou, “Ctrigger: exposing atomicity violation bugs from their hiding places,” in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS ’09, (New York, NY, USA), pp. 25–36, ACM, 2009.

[63] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, “Rule-based runtime verification,” in *VMCAI*, pp. 44–57, 2004.

[64] F. Dadeau, A. Kermadec, and R. Tissot, “Combining scenario- and model-based testing to ensure posix compliance,” in *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, ABZ ’08, (Berlin, Heidelberg), pp. 153–166, Springer-Verlag, 2008.

[65] C. Artho and P.-L. Garoche, “Accurate centralization for applying model checking on networked applications,” in *Proceedings of the 21st IEEE/ACM International conference on Automated Software Engineering (ASE)*, 2006.

[66] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker, “Netprints: diagnosing home network misconfigurations using shared knowledge,” in *NSDI*, 2009.

[67] M. Attariyan and J. Flinn, “Using causality to diagnose configuration bugs,” in *USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC)*, 2008.