Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It

Daniel Votipka
Kelsey R. Fulton
James Parker
Matthew Hou
Michelle L. Mazurek
Michael Hicks
University of Maryland
College Park, MD 20742, USA
dvotipka@cs.umd.edu
kfulton@cs.umd.edu
jprider@cs.umd.edu
mhou1@cs.umd.edu
mmazurek@cs.umd.edu
mwh@cs.umd.edu

Copyright is held by the author/owner. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee. Poster presented at the 15th Symposium on Usable Privacy and Security (SOUPS 2019).

Copyright held by the owner/author(s).
SOUPS, '19 Santa Clara, California USA
ACM x-xxxxx-xx-x/xx/xx.
http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

Abstract

Secure software development is a challenging task requiring programmers to consider many possible threats and mitigations. This paper investigates how and why programmers, despite having a baseline of security experience, make security-relevant errors. To do this, we conducted an in-depth analysis of 76 submissions to a secure programming contest designed to mimic real-world constraints: correctness, performance, and security. In addition to writing secure code, participants were asked to search for vulnerabilities in other teams' programs; in total, teams submitted 866 exploits against the submissions we considered. We found that simple mistakes were least common: only 26% of projects introduced such an error. Conversely, vulnerabilities arising from a misunderstanding of security concepts were significantly more common: 84% of projects introduced at least one such error.

Author Keywords

Secure Software Development; Vulnerability Discovery; Security Competitions

ACM Classification Keywords

H.5.m [Information interfaces and presentation (e.g., HCI)]: Miscellaneous; See [http://acm.org/about/class/1998/]: for full list of ACM classifiers. This section is required.

Research Questions:

RQ1: What *types* of vulnerabilities do developers introduce? Are they conceptual flaws in their understanding of security requirements or coding mistakes?

RQ2: How *severe* are the vulnerabilities? If exploited, what is the effect on the system?

RQ3: How *exploitable* are the vulnerabilities? What level of insight is required and how much work is necessary?

Introduction

Producing secure programs is a difficult task. NIST's National Initiative for Cybersecurity Education framework highlights the size of the challenge developers face, outlining 44 distinct areas of knowledge necessary for secure development [4]. Developers' limited time and attention — e.g., for security training, or for learning about and using new programming and testing tools — implies the importance of identifying the most critical and effective security interventions to prioritize.

In this paper, we make progress on this issue by investigating how and why programmers, despite a baseline of security experience, make security-relevant errors. To do this, we carried out a systematic, in-depth examination vulnerabilities present in 76 projects sampled from submissions to the *Build it, Break it, Fix it*[1] (BIBIFI) secure-coding competition series [5].

By studying code produced within security competitions, we can identify patterns in how different teams make security errors when approaching the same problem specification, enabling insights that are difficult to obtain when examining real-world programs with varying goals and requirements. At the same time, the contest format — in which teams had several weeks to build their project submissions, using any languages or tools they preferred — provides more ecological validity than lab studies in which developers complete small, highly specified programming tasks.

Our rigorous manual analysis of this dataset identified several interesting trends, with implications for improving secure-development training, security-relevant APIs [1–3], and tools for vulnerability discovery.

[1]https://builditbreakit.org

Simple mistakes, in which the developer attempts a valid security practice but makes a minor programming error, were least common: only 26% of projects introduced such an error. However, vulnerabilities arising from misunderstanding of security concepts were significantly more common: 84% of projects introduced at least one such error. Although these vulnerabilities were common, they proved difficult to exploit: only 76% were exploited by other teams (compared to 97% of simple mistakes), and our qualitative labeling identified 32% as difficult to exploit (compared to none of the simple mistakes).

Methods

We are interested in characterizing the vulnerabilities developers introduce when writing programs with security requirements. In particular, we pose several research questions as can be seen in the sidebar on page 2.

Answers to these questions can provide guidance about which interventions—tools, policy, and education—might be (most) effective, and how they should be prioritized. To obtain answers, we manually examined a sample of 76 BIBIFI projects (54% of all BIBIFI projects) and the 866 breaks submitted against them. The ultimate codebook we developed provides labels for vulnerabilities—their type, severity, and exploitability—and for features of the programs that contained them.

Codebook

To measure the types of vulnerabilities in each project, we characterized them across three variables as can be seen in the sidebar on page 4.

Results

Our manual analysis of 76 BIBIFI projects identified 172 unique vulnerabilities. We categorized each based on our

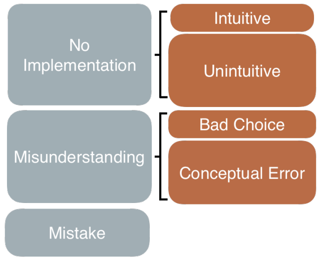
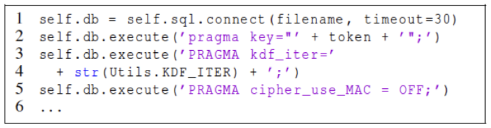


Figure 2: Classes of different issues



```
self.db = self.sql.connect(filename, timeout=30)
self.db.execute('pragma key="' + token + '";')
self.db.execute('PRAGMA kdf_iter='
  + str(Utils.KDF_ITER) + ';')
self.db.execute('PRAGMA cipher_use_MAC = OFF;')
...
```

Figure 1: Team that had a conceptual error by turning off integrity checks

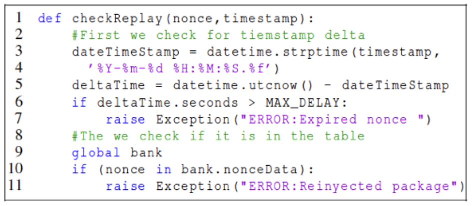
codebook into 23 different *Types*. We also saw some interesting relationships between vulnerability class and prevalence, severity, and likelihood to be exploited.

Classes

No Implementation A vulnerability type was classed as *No Implementation* when a team failed to even attempt to implement a necessary security mechanism, presumably because they did not realize it was needed. This class is further divided into the sub-classes *All Intuitive*, *Some Intuitive*, and *Unintuitive*. In the first two categories teams did not implement all or some, respectively, of the requirements that were either directly mentioned in the problem specification or were intuitive The *Unintuitive* category was used if the security requirement was not directly stated or was otherwise unintuitive.

Misunderstanding A vulnerability type was classed as *Misunderstanding* when a team attempted to implement a security mechanism, but failed due to a conceptual misunderstanding. We sub-classed these as either *Bad Choice* or *Conceptual Error*. An example of this can be seen in figure 1.

Mistake Finally, some teams attempted to implement the solution correctly, but made a mistake that led to a vulnerability. The mistake class is composed of five subclasses: insufficient error checking (P=12, V = 12), uncaught runtime



```
def checkReplay(nonce,timestamp):
    #First we check for tiemstamp delta
    dateTimeStamp = datetime.strptime(timestamp,
      '%Y-%m-%d %H:%M:%S.%f')
    deltaTime = datetime.utcnow() - dateTimeStamp
    if deltaTime.seconds > MAX_DELAY:
        raise Exception("ERROR:Expired nonce ")
    #The we check if it is in the table
    global bank
    if (nonce in bank.nonceData):
        raise Exception("ERROR:Reinyected package")
```

Figure 3: Team that made a made a mistake by forgetting to save the nonce

error (P=5, V=9), control flow mistake (P=5, V=10), skipped algorithmic step (P=5, V=7), and null write (P=1, V=1). An example of this can be seen in figure 3.

Prevalence

We observed a clear trend that teams often struggled to completely understand security concepts.

Teams often did not understand security concepts We found that both classes of vulnerabilities relating to a lack of security knowledge—*No Implementation* and *Misunderstanding* —were significantly more likely to be introduced than vulnerabilities caused by programming mistakes. These results indicate that efforts to address conceptual gaps should be prioritized. Focusing on these issues of understanding, we make the following observations.

Unintuitive security requirements are commonly skipped Of the *No Implementation* vulnerabilities, we found that the *Unintuitive* subclass was much more common than its *All Intuitive* or *Some Intuitive* counterparts. This indicates that developers do attempt to provide security — at least when incentivized to do so — but struggle to consider all the unintuitive ways an adversary could attack a system.

Codebook:

Type: Characterizes the underlying source of a vulnerability (RQ1)

Severity: Characterizes the impact of a vulnerability's exploitation (RQ2) as either a full compromise or a partial one

Likelihood of exploitability (RQ3): Characterized using two variables, discovery difficulty and exploit difficulty

- **Discovery difficulty**- characterizes the amount of knowledge the attacker must have to find the vulnerability
- **Exploit difficulty**- describes the amount of work needed to exploit the vulnerability once discovered

Teams often used the right security primitives, but did not know how to use them correctly Among the *Misunderstanding* vulnerabilities, we found that the *Conceptual Error* sub-class was significantly more likely to occur than *Bad Choice* This indicates that despite knowing the the correct primitives to use, developers do not always conform to the assumptions of "normal use" made by the library developers.

Complexity breeds *Mistakes* We found that complexity within both the problem itself and also the approach taken by the team has a significant effect on the number of *Mistakes* introduced. *Mistakes* were most common in the multiuser database problem and least common in the secure log problem. Teams were $5.79\times$ more likely to introduce *Mistakes* in multiuser database than in the baseline secure communication case. This effect likely reflects the fact that the multiuser database problem was the most complex, requiring teams to write a command parser, handle network communication, and implement nine different access control checks.

Effect of Exploitation
To answer RQ2 and RQ3, we compare the severity and likelihood of exploitation of the identified vulnerabilities.

***Misunderstandings* are hard to find** Identifying *Misunderstanding* vulnerabilities often required the attacker to determine the developer's exact approach and have a good understanding of the algorithms, data structures, or libraries they used. As such, we coded *Misunderstanding* vulnerabilities as hard to find significantly more often than both *No Implementation* and *Mistake* vulnerabilities.

***No Implementations* are easy to find** Unsurprisingly, a majority of *No Implementation* vulnerabilities were coded as easy to find (V=41, 60% of *No Implementations*). None of the *All Intuitive* or *Some Intuitive* vulnerabilities were coded as difficult to exploit; however, 42% of *Unintuitive* vulnerabilities were (V=19).

***Mistakes* are easy to find and exploit** We coded *Mistakes* as easy to exploit significantly more often than either *No Implementation* or *Misunderstanding* vulnerabilities. Further, all *Mistakes* were coded as easy to exploit. Fortunately, code review may be sufficient to find many of these vulnerabilities: *Mistakes* were also found and exploited during the Break It phase significantly more often than either *Misunderstanding* or *No Implementation* with only one *Mistake* (0.03%) not found by any Break It team.

Conclusion
With the goal of understanding which security errors programmers tend to make and why, this paper has presented a systematic, qualitative study of 76 program submissions to the build it, break it, fix it secure programming contest. Considering the 866 exploits against these submitted programs, we labeled 172 unique security vulnerabilities, according to type, severity, and ease of exploitability as well assecurity aligned project features. We found implementation mistakes were comparatively less common than failures in security understanding.Our results have implications for improving APIs and documentation, vulnerability-finding tools, and security education.

Acknowledgements
This material is based upon work supported by the National Science Foundation under Grant No. CNS-1801545.

REFERENCES
1. Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and

Christian Stransky. 2017. Comparing the usability of cryptographic apis. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 154–171.

2. Matthew Green and Matthew Smith. 2016. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy* 14, 5 (2016), 40–46.
3. Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 311–328.
4. William Newhouse, Stephanie Keith, Benjamin Scribner, and Greg Witte. 2017. *NIST Special Publication 800-181, The NICE Cybersecurity Workforce Framework*. Technical Report. National Institute of Standards and Technology.
5. Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. 2016. Build It, Break It, Fix It: Contesting Secure Development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 690–703. DOI: **http://dx.doi.org/10.1145/2976749.2978382**