SECURITY

Build It, Break It, Fix It Contests

Motivated Developers Still Make Security Mistakes

DANIEL VOTIPKA, KELSEY R. FULTON, JAMES PARKER, MATTHEW HOU, MICHELLE L. MAZUREK, AND MICHAEL HICKS

Daniel Votipka is a computer science PhD candidate at the University of Maryland. His research focuses on information security, with an emphasis on the human factors affecting security professionals. His most recent work focuses on understanding the processes and mental models of software vulnerability discovery to provide research-based improvements for education and automation to help develop and leverage human expertise. dvotipka@cs.umd.edu

Kelsey Fulton is a computer science PhD student at the University of Maryland. Her research explores the human factors of information security, with a focus on software developers and security professionals. Her most recent work centers on the barriers to adoption of secure programming languages in order to provide an empirical foundation for the future design of secure languages, APIs, and tools. kfulton@cs.umd.edu

James Parker is a Software Research Engineer at Galois. James earned his PhD in 2020 and was advised by Michael Hicks. His research spans verifying information flow control mechanisms, guaranteeing correctness of distributed systems, and studying secure development practices. jprider@cs.umd.edu

Secure software development is a challenging task requiring consideration of many possible threats and mitigations. We reviewed code submitted by 94 teams in a secure-programming contest designed to mimic real-world constraints—correctness, performance, and security. We found that the competitors, many of whom were experienced programmers and had just completed a 24-week cybersecurity course sequence with specific instruction on secure coding and cryptography, still introduced several vulnerabilities (182 across all teams), mostly due to misunderstandings of security concepts. We explain our methodology, discuss trends in the types of vulnerabilities introduced, and offer suggestions for avoiding the kinds of problems we encountered.

Developing secure software remains challenging, as evidenced by the numerous vulnerabilities still regularly discovered in production code [6]. There are many approaches that could be—and often have been—taken to improve this situation: building and deploying more automated tools for vulnerability discovery, expanding security education, or improving secure development processes.

But which of these interventions should we prioritize? While all are potentially helpful, we must carefully consider which provide the best return on investment, maximizing security while minimizing time, effort, and other resources, all of which are in short supply as developers are pressured to produce more new services and features.

A key part of this consideration is to understand the kinds and frequency of vulnerabilities that occur, and why developers introduce them, so that the root causes can be addressed. To this end, we performed a systematic, in-depth examination using best practices developed for qualitative assessments of vulnerabilities present in 94 project submissions by teams made up mostly of experienced programmers—many of whom had just completed a four-course program on secure development—to the Build It, Break It, Fix It (BIBIFI) secure-coding competition series [8, 10]. Our six-month examination considered each project's code and 866 total exploit submissions, corresponding to 182 unique security vulnerabilities associated with those projects.

Our findings suggest rethinking strategies to prevent and detect vulnerabilities, with more emphasis on conceptual difficulties rather than mistakes. This article provides an overview of our work. A more in-depth discussion of the methods followed, survey of related literature, and description of results can be found in our recent USENIX Security paper [10].

Build It, Break It, Fix It: A Happy Medium to Study

Our work to examine vulnerabilities introduced by software developers complements many prior efforts. Some researchers have performed large-scale analyses of open-source code and CVE reports, categorizing vulnerabilities found in production code [2, 3]; others have explored specific possible sources of error using controlled experiments with small, security-focused tasks [1, 7]. These field measures and lab studies represent two ends of a methodological spectrum. Field measures provide strong ecological validity, reflecting real-world

Matthew Hou is a first year computer science graduate student at the University of Maryland and is expecting to complete his master's degree next May. He recently graduated with honors from the University of Maryland with a BSc in computer science. His focus is on machine learning and artificial intelligence, leveraging cybersecurity principles. mhou1@cs.umd.edu

Michelle L. Mazurek is an Associate Professor in the Computer Science Department at the University of Maryland. Her research explores human aspects of information security and privacy, with a recent focus on improving security tools and processes for professionals, including software developers, network administrators, and reverse engineers. She also investigates how and why end users learn and adopt security and privacy behaviors, and she develops tools to increase transparency in online tracking and inferencing. mmazurek@cs.umd.edu

Michael Hicks is a Professor in the Computer Science Department at the University of Maryland. His research explores ways to make software more secure. He has a particular interest in securing low-level systems software, with a nearly 20-year stretch of work that started with the Cyclone safe C-like programming language (a significant influence on today's Rust programming language) and now involves contributions to Checked C, a safe-C extension based on clang/LLVM. He is also exploring synergies between cryptography and programming languages; techniques for better random (fuzz) testing and probabilistic reasoning; and high-assurance tools and languages for quantum computing. He blogs at https://www.pl-enthusiast.net/. mwh@cs.umd.edu

contexts, but provide no control over conditions like developer motivation and functionality being implemented that can affect results. In contrast, lab studies provide high levels of control but only limited ecological validity.

We attempt to balance ecological validity and experimental control by studying vulnerabilities in the context of BIBIFI competition projects. A BIBIFI competition has three phases. In the *build it* phase, teams are given just under two weeks to build a project that (securely) meets a given specification. Team scores depend on the project's correctness and efficiency, based on provided test cases. Submitted projects may be written in any programming language and can use any open-source libraries, as long as they can be built on a standard Ubuntu Linux VM. In the *break it* phase, teams receive access to their competitors' source code in order to search for vulnerabilities. Teams can submit test cases, known as *breaks*, to demonstrate exploitation. Successful breaks add to the exploiting team's break-it score, while reducing the victim's build-it score. The final *fix-it* phase allows teams to fix identified vulnerabilities in order to gain back a portion of their lost build-it points.

BIBIFI data therefore strikes a unique balance between ecological validity and control. Many implementations of the same functionality, created under similar circumstances, provide more confidence than field data does to help us understand what happened and why. On the other hand, teams had weeks (rather than hours) to develop their projects, could use their choice of languages and libraries, and were incentivized to consider constraints like performance and functionality as well as security, creating more ecological validity than many lab studies. While we know BIBIFI does not provide a perfect view into the development process (see our original paper [10] for a detailed discussion of limitations), it provides a new and valuable vantage point for examining the vulnerability landscape and informing future work.

The Competition's Projects

We analyzed projects from four BIBIFI competitions, covering three different programming problems: *secure log*, *secure communication*, and *multiuser database*. Each problem specification required the teams to consider different security challenges and attacker models.

Secure log (SL). This problem requires teams to implement two programs: one to securely append records to a log, and one to query the log's contents. Teams must protect against a malicious adversary with access to the log and the ability to modify it. The adversary does not have access to the keys used to create the log. Teams are expected (but not told explicitly) to utilize cryptographic functions to encrypt the log and protect its integrity.

Secure communication (SC). This problem requires teams to create client/server programs representing a bank and an ATM. The ATM initiates transactions, including account creation, deposits, and withdrawals.

Teams must protect bank data integrity and confidentiality against an adversary acting as a man-in-the-middle (MITM), with the ability to read and manipulate communications between the client and server. Once again, build teams were expected to use cryptographic functions and to consider challenges such as replay attacks and side-channels.

Multiuser database (MD). This problem requires teams to create a server that maintains a secure key-value store. Clients submit scripts written in a domain-specific language. A script authenticates with the server and then submits a series of commands to read and write data stored there. Data is protected by role-based access control policies customizable by the data owner, who may (transitively) delegate access control decisions to other principals.

The problem assumes that an attacker can submit commands to the server but not snoop on communications.

Vulnerabilities: Type and Prevalence

We manually analyzed 94 (out of 142) BIBIFI projects and 866 exploit submissions against them, ultimately identifying 182 unique vulnerabilities (some of which had not been identified during the contests). We grouped these vulnerabilities according to three main types: *no implementation*, *misunderstanding*, and *mistake*. Table 1 shows how many vulnerabilities, from how many projects, we identified for each type. This section describes each type, with examples.

No Implementation

The first step in building a secure system is to *attempt* to implement necessary security mechanisms. Unfortunately, half of all teams introduced a *no implementation* vulnerability, failing in this first step for at least one required security mechanism. This is presumably because they did not realize the security mechanism was needed. We further divided *no implementation* vulnerabilities based on how *obvious* the need was, depending on whether it was directly mentioned in the problem specification or just implied. For example, in the secure log problem, where teams were asked to ensure an attacker with read/write file access could not read or make changes to a confidential log, we considered it obvious that encryption was needed to provide confidentiality, but unintuitive that a Message Authentication Code (MAC) should be used as an integrity check.

Unintuitive security requirements are commonly skipped. Of the *no implementation* vulnerabilities, we found that teams were much more likely to skip *unintuitive* security requirements (45% of projects) than their intuitive counterparts (16% of projects). 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. Therefore, they regularly leave out some necessary controls.

Misunderstandings

After realizing a security mechanism should be implemented, teams then needed to make sure they implemented it correctly. We found that most teams failed at this point in the secure development process, most commonly due to a conceptual misunderstanding (56% of projects). We sub-typed these as either *bad choice* or *conceptual error*.

A *bad choice* occurs when a team decides to use a known-insecure algorithm or library—likely because they did not realize its inherent flaw (12% of vulnerabilities). In another secure log problem example, one team realized they needed to encrypt their log, but chose to simply XOR key-length chunks of the log with the user-provided key to generate the final encrypted version of the log. This method of encryption is inherently insecure, as the attacker can simply extract two key-length chunks of the ciphertext, XOR them together, and produce the key, allowing them to decrypt the entire log easily.

Assuming a team did choose a secure algorithm or library, next they had to know how to use it properly. We observed several cases where teams introduced vulnerabilities by not using the algorithm or library as intended, owing to a conceptual misunderstanding (27% of vulnerabilities). We classified these as *conceptual error* vulnerabilities. For example, one team made the reasonable choice to use AES encryption but used a fixed value for its initialization vector (IV); see code in Listing 1. A fixed IV, rather than a random one, allows an attacker to break the encryption and read the secret log.

```
1  def fillercrypter (sharedkey, text):
2      ...
3      encryption_suite = AES.new (sharedkey,
4       AES.MODE_CBC, 'This is an IV456')
5.     ...
```

Listing 1: One team generated a *conceptual error* vulnerability by using a hardcoded IV.

Type	Sub-Type	Projects (94)	Vulnerabilities (182)
No implementation	Intuitive	15 (16%)	23 (13%)
	Unintuitive	42 (45%)	49 (27%)
	Total	47 (50%)	72 (40%)
Misunderstanding	Bad choice	20 (21%)	22 (12%)
	Conceptual error	41 (44%)	49 (27%)
	Total	53 (56%)	71 (39%)
Mistake	—	20 (21%)	39 (21%)

Table 1: Number of vulnerabilities for each type and the number of projects each vulnerability was introduced in. Note, because projects can have multiple vulnerabilities, the total number of projects introducing a vulnerability for each type may not be the sum of sub-type project counts.

```
1  self.db = self.sql.connect(filename, timeout=30)
2  self.db.execute('pragma key="' + token + '";')
3  self.db.execute('PRAGMA kdf_iter='
4    + str(Utils.KDF_ITER) + ';')
5  self.db.execute('PRAGMA cipher_use_MAC=OFF;')
6  ...
```

Listing 2: Another team disabled the automatic MAC in SQLCipher library.

In another interesting example, one team simply disabled protections provided transparently by their chosen library. They initially made a secure choice by using the SQLCipher library, which provides encryption and integrity checks in the background without developer effort, but then explicitly disabled the library's MAC protection; see line 5 in Listing 2.

Teams often used the right security primitives but did not know how to use them correctly. Among the *misunderstanding* vulnerabilities, we found that *conceptual error* vulnerabilities (44% of projects) were significantly more likely to occur than *bad choice* vulnerabilities (21% of projects). This indicates that if developers know what security controls to implement, they are often able to identify (or are guided to) the correct primitives to use. However, they do not always conform to the assumptions of "normal use" made by library developers.

Mistakes

Finally, some teams chose the correct algorithm or library, and appeared to understand how to correctly use it, but made a simple mistake that led to a vulnerability (21% of vulnerabilities). For example, some teams did not properly handle errors, leaving the program in an observably bad state. Other mistakes led to logically incorrect execution behaviors. Such mistakes were often related to control flow logic or missed steps in an algorithm. For example, if a team correctly encrypted their log, but accidentally wrote the plaintext log to file instead of the ciphertext, this would be a *mistake*.

Complexity breeds mistakes. We found that the frequency of *mistakes* was affected by complexity, within both the problem itself and also the approach taken by the team. First, we found that teams were 6.68× more likely to introduce *mistakes* in the *multiuser database* than in the *secure communication* problem. This 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. Similarly, teams were only 0.06× as likely to make a mistake in the comparatively simple *secure log* problem compared to the *secure communication* problem.

Additionally, choosing not to reimplement security-relevant code multiple times was associated with only 0.36× as many *mistakes*, suggesting that violating the "Economy of Mechanism" principle [9] by adding unnecessary complexity leads to *mistakes*.

As an example of this effect, one team implemented their access control checks four times throughout the project. Unfortunately, when they realized the implementation was incorrect, they only updated it in one place.

Exploit Difficulty

In addition to examining vulnerability types and their frequency, we also assessed how *difficult* it would be for an attacker to find and exploit the vulnerability. Even if a vulnerability was quite common, if it was very difficult to identify, requiring esoteric knowledge or practically impossible to exploit, its resolution might be lower priority than a less common but more exploitable vulnerability.

We considered three metrics of difficulty: our qualitative assessment of the difficulty of finding the vulnerability (*discovery difficulty*); our qualitative assessment of the difficulty of exploiting the vulnerability (*exploit difficulty*); and whether a competitor team actually found and exploited the vulnerability (*actual exploitation*). For convenience of analysis, we binned *discovery difficulty* into *easy* (execution) and *hard* (source, deep insight). We similarly binned *exploit difficulty* into *easy* (single-step, few steps) and *hard* (many steps, deterministic or probabilistic). Figure 1 shows the number of vulnerabilities for each type with each bar divided by *exploit difficulty* and bars grouped by *discovery difficulty*.

***Misunderstandings* are rated as hard to find, while *no implementations* are rated as easy 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 rated *misunderstanding* vulnerabilities as hard to find significantly more often than other vulnerability types.

Unsurprisingly, a majority of *no implementation* vulnerabilities were considered easy to find. For example, in the secure log problem, an auditor could simply check whether encryption and an integrity check were used. If not, then the project can be exploited.

Easy to find doesn't mean easy to exploit. Interestingly, we did not observe a significant difference in actual exploitation between *misunderstandings* and *no implementations*. Some *misunderstandings* were rated as difficult to find, while others were rated as difficult to exploit. In one team's use of homemade encryption, the vulnerability took some time to find, because the implementation code was difficult to read. However, once an attacker realized the team had essentially reimplemented the Wired Equivalent Protocol (WEP), a simple check of Wikipedia revealed the exploit. Conversely, seeing that a non-random IV was used for encryption is easy, but successful exploitation of this flaw can require significant time and effort.

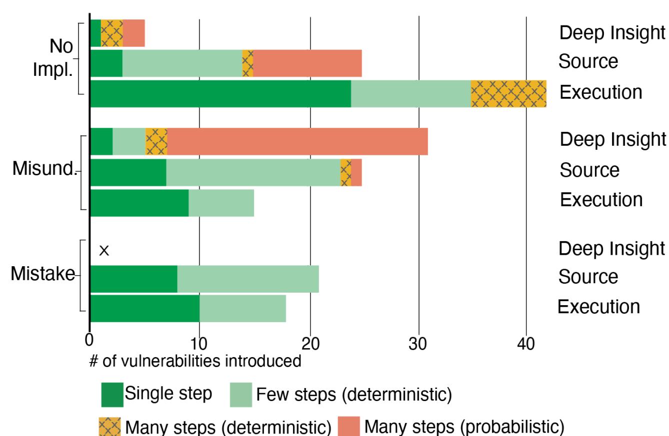

Figure 1: Number of vulnerabilities introduced for each type divided by *discovery difficulty* and *exploit difficulty*

As a *no implementation* example, one secure log team did not use a MAC to detect modifications to their encrypted files. This mistake is very simple to identify, but it was not exploited by any of the BIBIFI teams. This is likely because the team stored log data in a JSON blob before encrypting, meaning that any modifications to the encrypted text must maintain the JSON structure after decryption to succeed. This attack could require a large number of tests to find a suitable modification.

***Mistakes* are rated as easy to find and exploit**. We rated all *mistakes* as easy to exploit. This is significantly different from both *no implementation* and *misunderstanding* vulnerabilities, which were rated as easy to exploit less frequently. Similarly, *mistakes* were actually exploited during the Break It phase significantly more often than other vulnerability types. In fact, only one *mistake* was not actually exploited by any team. These results suggest that although *mistakes* were least common, any that do find their way into production code are likely to be found and exploited. Fortunately, our results also suggest that code review may be sufficient to find many of these vulnerabilities. We note that this assumes that the source is available, which may not be the case when a developer relies on third-party software.

Discussion and Recommendations

So what do these results mean for improving secure development? We believe they add weight to existing recommendations and suggest prioritizations of possible solutions.

Get the help of a security expert. In some large organizations, developers working with cryptography and other security-specific features might be required to use security-expert-determined tools and patterns or have a security expert perform a review. Our results reaffirm this practice, when possible, as participants were most likely to struggle with security concepts avoidable through expert review.

Security education. Better education should help developers better help themselves. However, across all vulnerability types, we observed no difference in vulnerabilities introduced related to prior security training or years of prior development experience. It therefore seems that increased development experience and (traditional) security training have, at most, a small impact.

Further, many of the BIBIFI teams had previously completed a four-course cybersecurity training during which all needed security controls were discussed, but a majority of these teams nevertheless botched *unintuitive* requirements. Were the topics not driven home sufficiently? An environment like BIBIFI, where developers practice implementing security concepts and receive feedback regarding mistakes, could help. Future work should consider how well competitors from one contest do in follow-on contests.

API design. Our results support the basic idea that security controls are best applied transparently, e.g., using simple APIs [4]. However, while many teams used APIs that provide security (e.g., encryption) transparently, they were still frequently misused (e.g., failing to initialize using a unique IV or failing to employ stream-based operation to avoid replay attacks). It may be beneficial to organize solutions around general use cases, so that developers only need to know the use case and not the security requirements.

API documentation. API usage problems could be a matter of documentation, as suggested by prior work [1, 7]. For example, two teams used TLS socket libraries but did not enable client-side authentication, necessary for the problem. This failure appears to have occurred because client-side authentication is disabled by default, but this fact is not mentioned in the documentation [11, 12]. Defaults within an API should be safe and without ambiguity [4]. Returning to the example from Listing 2, the team disabled the automatic integrity checks of the SQLCipher library. Their commit message stated, "Improve performance by disabling per-page MAC protection." We know this change was made to improve performance, but it is possible they assumed they were only disabling the "per-page" integrity check while a full database check remained. The documentation is unclear about this (https://www.zetetic.net/sqlcipher/sqlcipher-api/#cipher_use_MAC).

Vulnerability analysis tools. There is significant interest in automating security vulnerability discovery (or preventing vulnerability introduction) through the use of code analysis tools. Such tools may have found some of the vulnerabilities we examined in our study. For example, static analyses, symbolic executors, fuzzers, and dynamic analyses could have uncovered vulnerabilities relating to memory corruption, improper parameter use (like a fixed IV), and missing error checks. However,

they would not have applied to the majority of vulnerabilities we saw, which were often design-level, conceptual issues.

How could automation be used to address security requirements at design time? More research is needed, but one possible direction forward is to consider analysis development in tandem with improvements to API design. One example is Google's efforts to restrict the ways developers can potentially introduce certain vulnerabilities (e.g., XSS, SQL-injection) through API design, limiting the required complexity of vulnerability discovery analysis [5].

Conclusion

Secure software development is challenging, with many proposed remediations and improvements. To know which interventions are likely to have the most impact requires understanding which security errors programmers tend to make and why. In our review of 94 submissions to a secure-programming contest, each implementing one of three non-trivial, security-relevant programming problems, we found implementation mistakes were comparatively less common than failures in security understanding. Our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.

References

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the Usability of Cryptographic APIs," in *Proceedings of the IEEE Symposium on Security and Privacy* (2017), pp. 154–171.

[2] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications," in *Proceedings of the ACM Conference on Computer and Communications Security* (2013), pp. 73–84.

[3] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software," in *Proceedings of the ACM Conference on Computer and Communications Security* (2012), pp. 38–49.

[4] M. Green and M. Smith, "Developers Are Not the Enemy!: The Need for Usable Security APIs," *IEEE Security & Privacy*, vol. 14, no. 5 (Sept.–Oct. 2016), pp. 40–46.

[5] C. Kern, "Preventing Security Bugs through Software Design," 24th USENIX Security Symposium: https://www.usenix.org/conference/usenixsecurity15/symposium-program/presentation/kern.

[6] D. R. Kuhn, M. S. Raunak, and R. Kacker, "An Analysis of Vulnerability Trends, 2008–2016," in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security Companion*, pp. 587–588.

[7] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith, "Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study," in *Proceedings of the ACM Conference on Computer and Communications Security* (2017), pp. 311–328.

[8] A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build It, Break It, Fix It: Contesting Secure Development," in *Proceedings of the ACM Conference on Computer and Communications Security* (2016), pp. 690–703.

[9] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," in *Proceedings of the Symposium on Operating System Principles* (ACM, 1975), pp. 1278–1308..

[10] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks, "Understanding Security Mistakes Developers Make: Qualitative Analysis from Build It, Break It, Fix It," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, pp. 109–126.

[11] TLS socket documentation: https://golang.org/pkg/crypto/tls/#Listen and https://www.openssl.org/docs/manmaster/man3/SSL_new.html.

[12] SQLCipher documentation: https://www.zetetic.net/sqlcipher/sqlcipher-api/#cipher_use_MAC.