Empirical Security & Privacy, for Humans

UPenn CIS 7000-010

9/23/2025

Measuring secure software development practices

Readings

Build It, Break It, Fix It: Contesting Secure Development

Andrew Ruef Michael Hicks James Parker
Dave Levin Michelle L. Mazurek Piotr Mardziel[†]

University of Maryland [†]Carnegie Mellon University

ABSTRACT

Typical security contests focus on breaking or mitigating the impact of buggy systems. We present the Build-it, Break-it, Fix-it (BIBIFI) contest, which aims to assess the ability to securely build software, not just break it. In BIBIFI, teams build specified software with the goal of maximizing correctness, performance, and security. The latter is tested when teams attempt to break other teams' submissions. Winners are chosen from among the best builders and the best breakers. BIBIFI was designed to be open-ended—teams can use any language, tool, process, etc. that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. During 2015, we ran three contests involving a total of 116 teams and two different programming problems. Quantitative analysis from these contests found that the most efficient build-it submissions used C/C++, but submissions coded in other statically-typed languages were less likely to have a security flaw; build-it teams with diverse programming-language knowledge also produced more secure code. Shorter programs correlated with better scores. Break-it teams that were also successful build-it teams were significantly better at finding security bugs.

1. INTRODUCTION

Cybersecurity contests [24, 25, 11, 27, 13] are popular proving grounds for cybersecurity talent. Existing contests

experts have long advocated that achieving security in a computer system requires treating security as a first-order design goal [32], and is not something that can be added after the fact. As such, we should not assume that good breakers will necessarily be good builders [23], nor that top coders necessarily produce secure systems.

This paper presents **Build-it, Break-it, Fix-it** (BIBIFI), a new security contest with a focus on *building secure systems*. A BIBIFI contest has three phases. The first phase, *Build-it*, asks small development teams to build software according to a provided specification that includes security goals. The software is scored for being correct, efficient, and feature-ful. The second phase, *Break-it*, asks teams to find defects in other teams' build-it submissions. Reported defects, proved via test cases vetted by an oracle implementation, benefit a break-it team's score and penalize the build-it team's score; more points are assigned to security-relevant problems. (A team's break-it and build-it scores are independent, with prizes for top scorers in each category.) The final phase, *Fix-it*, asks builders to fix bugs and thereby get points back if the process discovers that distinct break-it test cases identify the same defect.

BIBIFI's design aims to minimize the manual effort of running a contest, helping it scale. BIBIFI's structure and scoring system also aim to encourage meaningful outcomes, e.g., to ensure that the top-scoring build-it teams really produce secure and efficient software. Behaviors that would thwart such outcomes are discouraged. For example, break-it teams may submit a limited number of bug reports per build-it submission, and will lose points during fix-it for test cases that expose the same underlying defect or a defect also identified by other teams. As such, they are encouraged to

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, and Michael Hicks
University of Maryland
{dvotipka,kfulton,jprider1,mhou1,mmazurek,mwh}@cs.umd.edu

Abstract

Secure software development is a challenging task requiring consideration of many possible threats and mitigations. This paper investigates how and why programmers, despite a baseline of security experience, make security-relevant errors. To do this, we conducted an in-depth analysis of 94 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. Over an intensive six-month period, we used iterative open coding to manually, but systematically, characterize each submitted project and vulnerability (including vulnerabilities we identified ourselves). We labeled vulnerabilities by type, attacker control allowed, and ease of exploitation, and projects according to security implementation strategy. Several patterns emerged. For example, simple mistakes were least common: only 21% of projects introduced such an error. Conversely, vulnerabilities arising from a misunderstanding of security concepts were significantly more common, appearing in 78% of projects. Our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.

1 Introduction

Developing secure software is a challenging task, as evidenced by the fact that vulnerabilities are still discovered,

developers [16, 44, 77] is evidence of the intense pressure to produce new services and software quickly and efficiently. As such, we must be careful to choose interventions that work best in the limited time they are allotted. To do this, we must understand the general type, attacker control allowed, and ease of exploitation of different software vulnerabilities, and the reasons that developers make them. That way, we can examine how different approaches address the landscape of vulnerabilities.

This paper presents a systematic, in-depth examination (using best practices developed for qualitative assessments) of vulnerabilities present in software projects. In particular, we looked at 94 project submissions to the *Build it, Break it, Fix it* (BIBIFI) secure-coding competition series [66]. In each competition, participating teams (many of which were enrolled in a series of online security courses [34]) first developed programs for either a secure event-logging system, a secure communication system simulating an ATM and a bank, or a scriptable key-value store with role-based access control policies. Teams then attempted to exploit the project submissions of other teams. Scoring aimed to match real-world development constraints: teams were scored based on their project's performance, its feature set (above a minimum baseline), and its ultimate resilience to attack. Our six-month examination considered each project's code and 866 total exploit submissions, corresponding to 182 unique security vulnerabilities associated with those projects.

The BIBIFI competition provides a unique and valuable vantage point for examining the vulnerability landscape, com-

Readings

Build It, Break It, Fix It: Contesting Secure Development

University of Maryland   Carnegie Mellon University

ABSTRACT

securely build software, not just break it. In BIBIFI, teams build specified software with the goal of maximizing correctness, performance, and security. The latter is tested when teams attempt to break other teams' submissions. Winners are chosen from among the best builders and the best breakers. BIBIFI was designed to be open-ended—teams can use any language, tool, process, etc. that they like. As such, contest outcomes shed light on factors that correlate with successfully building secure software and breaking insecure software. During 2015, we ran three contests involving a total of 116 teams and two different programming problems. Quantitative analysis from these contests found that the most efficient build-it submissions used C/C++, but submissions coded in other statically-typed languages were less likely to have a security flaw; build-it teams with diverse programming-language knowledge also produced more secure code. Shorter programs correlated with better scores. Break-it teams that were also successful build-it teams were significantly better at finding security bugs.

1. INTRODUCTION

Cybersecurity contests [24, 25, 11, 27, 13] are popular

computer system requires treating security as a first-order design goal [32], and is not something

a new security contest with a focus on *building secure systems*. A BIBIFI contest has three phases. The first phase, *Build-it*, asks small development teams to build software according to a provided specification that includes security goals. The software is scored for being correct, efficient, and feature-ful. The second phase, *Break-it*, asks teams to find defects in other teams' build-it submissions. Reported defects, proved via test cases vetted by an oracle implementation, benefit a break-it team's score and penalize the build-it team's score; more points are assigned to security-relevant problems. (A team's break-it and build-it scores are independent, with prizes for top scorers in each category.) The final phase, *Fix-it*, asks builders to fix bugs and thereby get points back if the process discovers that distinct break-it test cases identify the same defect.

BIBIFI's design aims to minimize the manual effort of running a contest, helping it scale. BIBIFI's structure and scoring system also aim to encourage meaningful outcomes, e.g., to ensure that the top-scoring build-it teams really produce secure and efficient software. Behaviors that would thwart such outcomes are discouraged. For example, break-it teams may submit a limited number of bug reports per build-it submission, and will lose points during fix-it for test cases that expose the same underlying defect or a defect also identified by other teams. As such, they are encouraged to

Overview

Round 1: **Build-it**
Teams build software to specification

2 weeks

Must satisfy basic **correctness** requirements; **optional features** and **good performance** for more points

Round 2: **Break-it**
Teams report bugs in submissions

2 weeks

Bug reports are (failing) executable **test cases**, including **exploits**

Round 3: **Fix-it**
Teams fix bugs found in their software

1 week

Doing so may **wipe out many bug reports in one go**: all count as the same bug

Last: Judges tally final results

Scoring System

- **Build-it Score**
  - **Gains** points for **good performance**
  - **Gains** points for implementing **optional features**
  - **Loses** points for *unique* **bugs found**
    - More points for (obviously) security-relevant bugs
- **Break-it Score**
  - **Gains** points for unique **bugs found**
    - Scaled by how many other teams found the same bug
- **Winners in both categories**

Build-it Round

- **Build software** according to the posted specification
  - Make it correct, feature-ful, efficient, and secure
    - The first three are assessed by (our) test cases (build-it round score)
    - The last is assessed by Break-it teams in round 2
- For many elements of the task, **teams may choose**
  - The software's internal **design** and **algorithms**
  - Which **optional features** to implement
  - What **programming language** to use
  - What development and testing **tools** to use
  - How to **divide tasks among team members**, etc.

Break-it Round

- Find **bugs and vulnerabilities** in submitted code
  - Provide an **exploit**, as defined by particular problem
- Teams will be given **access to the source code**
  - We provide scripts that teams can use to test projects against the standard tests, using a VM
- How teams go about this task is **up to them**, e.g.,
  - How to **divide up the task** among team members, and
  - whether (or how much) **testing** to use,
  - manual **code reviews**,
  - automated **dynamic/static analysis**, etc.

Fix-it Round

- Different teams may submit **different test cases** that identify the **same underlying bug**

```
void foo(char *str) {
  char buf[10];
  strcpy(buf,str);
}
```

- `str` = **"this is too long"**
- `str` = **"this is too long too"**
- `str` = **"and so is this string"**

- Build-it teams should only **lose points for each bug**, not for each test case that reveals it
- How to tell that test cases are "the same" ?

Fix-it Round

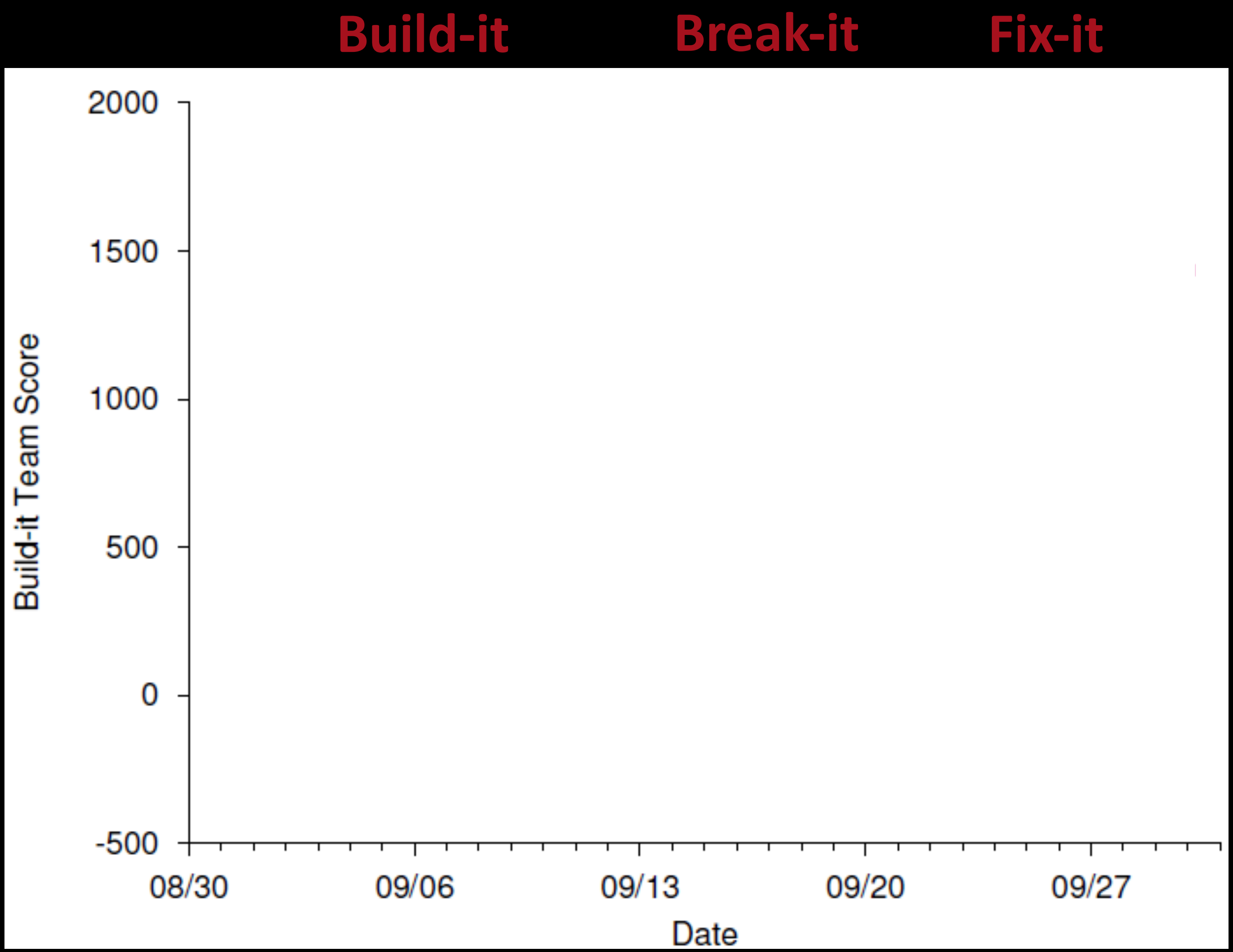
- Teams will receive the test cases during the fix-it round, and they can then **fix each bug identified**

```
void foo(char *str) {
  char buf[10];
  strlcpy(buf,str,10);
}
```

- `str` = “this is too long”
- `str` = “this is too long too”
- `str` = “and so is this string”

- **All test cases that pass** are unified to be the **same underlying bug**
  - Judges consider whether the fix is to a single bug
  - If not, the affected test cases will be scored individually

Builder score, as the contest progresses



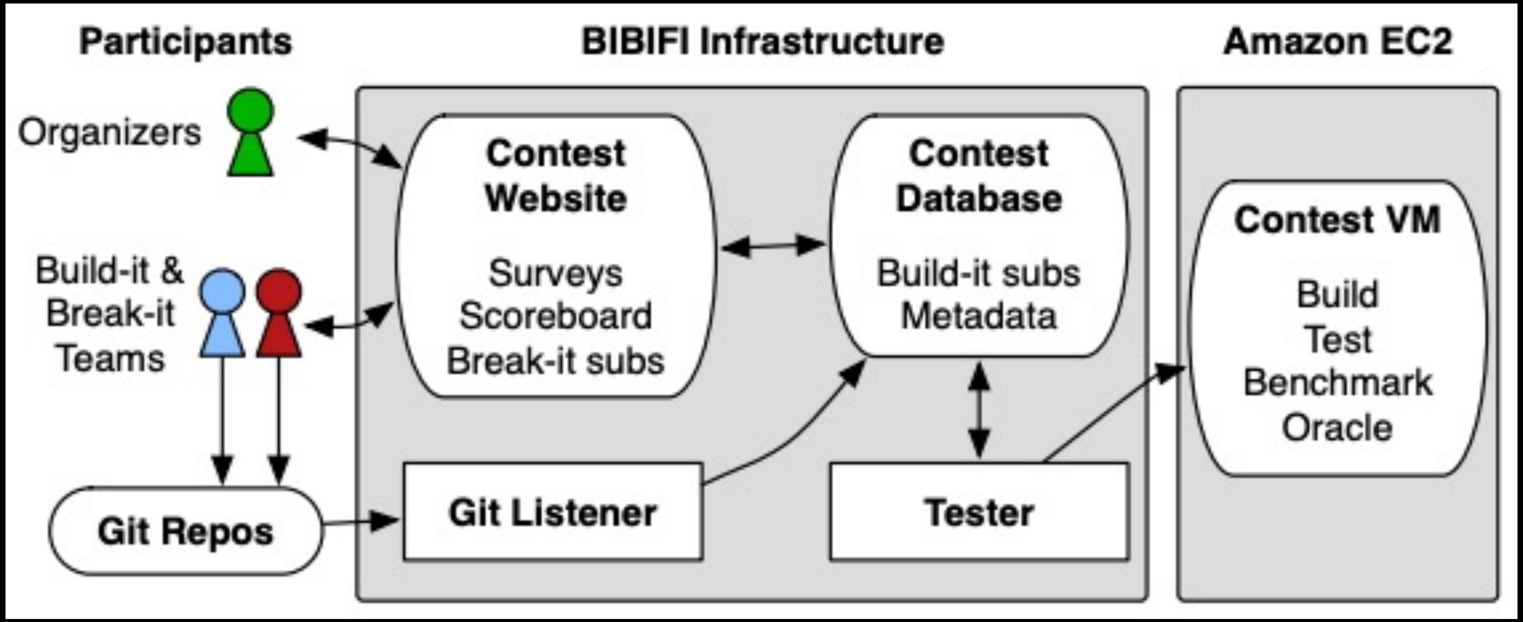
BIBIFI incentives approximate the real world

- Build-it teams are incentivized to
  - **meet near-term requirements** (**correctness** and **performance**)
    - Under time pressure, and ”no points for no submission” pressure
  - but while **not neglecting security**
    - Which will soon matter
- Break-it teams are incentivized to
  - find **unique security bugs** (more points than correctness)
    - Duplicate submissions consume exploit budget
  - that are **hard to find** or in **neglected submissions**
    - no point sharing if others miss
  - Upshot: Aim to **cover all submissions, in depth**
    - But only bugs that are exploitable via contest infrastructure

Contest problems

- Secure log of events at an art gallery
  - Commands to append records and query the log contents
  - Threat model:
    - Attacker has access to the log
    - Should be tamperproof and protect confidentiality
- Secure ATM
  - ATM communicates with Bank server to carry out transactions
  - Threat model: MITM can observe, send, drop messages, and simulate the ATM
- Multi-user DB [in extended version of paper]
  - Scriptable key-value store with RBAC policies with delegation
  - Threat model: Attacker as client; writes scripts to try to break RBAC implementation

Contest implementation



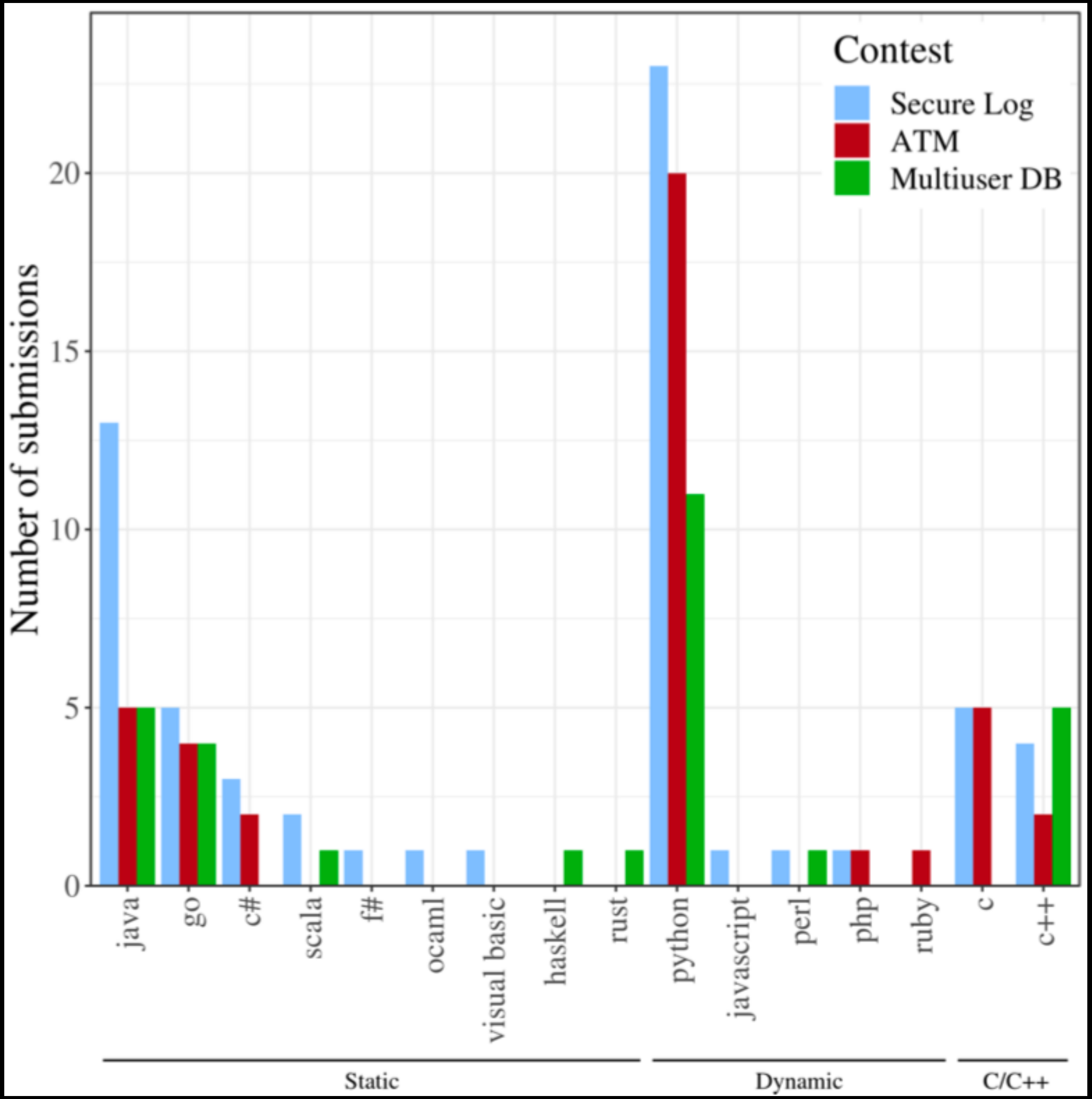
- Website written in **Yesod, Haskell**-based web framework
  - www.yesodweb.com
- Extended with a custom version of **LIO, for security enforcement**

Participant demographics

Contest	USA	India	Russia	Brazil	Other
Spring 2015	30	7	12	12	120
Fall 2015	64	14	12	20	110
Fall 2016	44	13	4	12	103

- Worldwide participant pool (mostly non-US)
  - 156 teams, 406 people
- Average 9 years programming experience
  - 1/3 have CS degree
- Most participants part of MOOC
  - Four courses of security training
- Average team size: ~2

Contest	Spring 15	Fall 15	Fall 16
Problem	Secure Log	ATM	Multiuser DB
# Contestants	156	145	105
% Male	91 %	91 %	84 %
% Female	5 %	8 %	4 %
Age (mean/min/max)	34.8/20/61	32.2/17/69	29.9/18/55
% with CS degrees	35 %	35 %	39 %
Years programming	9.6/0/30	9.4/0/37	9.0/0/36
# Build-it teams	61	40	29
Build-it team size	2.2/1/5	3.1/1/6	2.5/1/8
# Break-it teams (that also built)	65 (58)	43 (35)	33 (22)
Break-it team size	2.4/1/5	3.1/1/6	2.6/1/8
# PLs known per team	6.8/1/22	9.1/1/20	7.8/1/17
% MOOC	100 %	84 %	65 %



Submission features

- **Count** by language
- Grouped by **category**
  - **Static**ally type-safe (49)
  - **Dynamic**ally typed (60)
    - 54 are Python!
  - **C/C++** (21)

Summary of data analysis

- Build-it
  - **Best performance**: coded in C/C++
  - **Lower chance of a security flaw** (11x): coded in statically type-safe language
  - Diverse programming background, shorter code, team size, knowledge of C factor in less significantly

- Break-it
  - **Increased chance of finding security bug**: Larger team
  - **Higher overall bug count**: Larger team, took part in build-it
  - Advanced techniques made no measurable difference

Discussion

- How does this format of study compare to alternatives?
  - What variations on the contest design that might improve it?
- Why might we trust, or not trust, these results?
  - E.g., non-fixed breaks could overpenalize teams in a certain category
  - Could not assess availability problems (hangs)
  - Lack of data does not imply non-effect
- Could this contest setup be applied to other problems?

Readings

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, and Michael Hicks
University of Maryland
{dvotipka,kfulton,jprider1,mhou1,mmazurek,mwh}@cs.umd.edu

Abstract

Secure software development is a challenging task requiring consideration of many possible threats and mitigations. This paper investigates how and why programmers, despite a baseline of security experience, make security-relevant errors. To do this, we conducted an in-depth analysis of 94 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. Over an intensive six-month period, we used iterative open coding to manually, but systematically, characterize each submitted project and vulnerability (including vulnerabilities we identified ourselves). We labeled vulnerabilities by type, attacker control allowed, and ease of exploitation, and projects according to security implementation strategy. Several patterns emerged. For example, simple mistakes were least common: only 21% of projects introduced such an error. Conversely, vulnerabilities arising from a misunderstanding of security concepts were significantly more common, appearing in 78% of projects. Our results have implications for improving secure-programming APIs, API documentation, vulnerability-finding tools, and security education.

1 Introduction

Developing secure software is a challenging task, as evidenced by the fact that vulnerabilities are still discovered, developers [16, 44, 77] is evidence of the intense pressure to produce new services and software quickly and efficiently. As such, we must be careful to choose interventions that work best in the limited time they are allotted. To do this, we must understand the general type, attacker control allowed, and ease of exploitation of different software vulnerabilities, and the reasons that developers make them. That way, we can examine how different approaches address the landscape of vulnerabilities.

This paper presents a systematic, in-depth examination (using best practices developed for qualitative assessments) of vulnerabilities present in software projects. In particular, we looked at 94 project submissions to the *Build it, Break it, Fix it* (BIBIFI) secure-coding competition series [66]. In each competition, participating teams (many of which were enrolled in a series of online security courses [34]) first developed programs for either a secure event-logging system, a secure communication system simulating an ATM and a bank, or a scriptable key-value store with role-based access control policies. Teams then attempted to exploit the project submissions of other teams. Scoring aimed to match real-world development constraints: teams were scored based on their project's performance, its feature set (above a minimum baseline), and its ultimate resilience to attack. Our six-month examination considered each project's code and 866 total exploit submissions, corresponding to 182 unique security vulnerabilities associated with those projects.

The BIBIFI competition provides a unique and valuable vantage point for examining the vulnerability landscape, com-

Approach

- Examined each project and vulnerability in detail
  - 94 projects
  - Breaker-identified (866 submitted exploits) and researcher-identified (manual analysis)
  - In total, 182 distinct vulnerabilities
- Iterative open and axial coding
  - Two+ independent coders
  - High agreement: Krippendorff's $\alpha > 0.8$
- Qual and quant analysis on resulting categories

Vulnerability classes

No Implementation

Misunderstanding

Mistake

Intuitive

Unintuitive

Bad Choice

Conceptual Error

Vulnerability classes

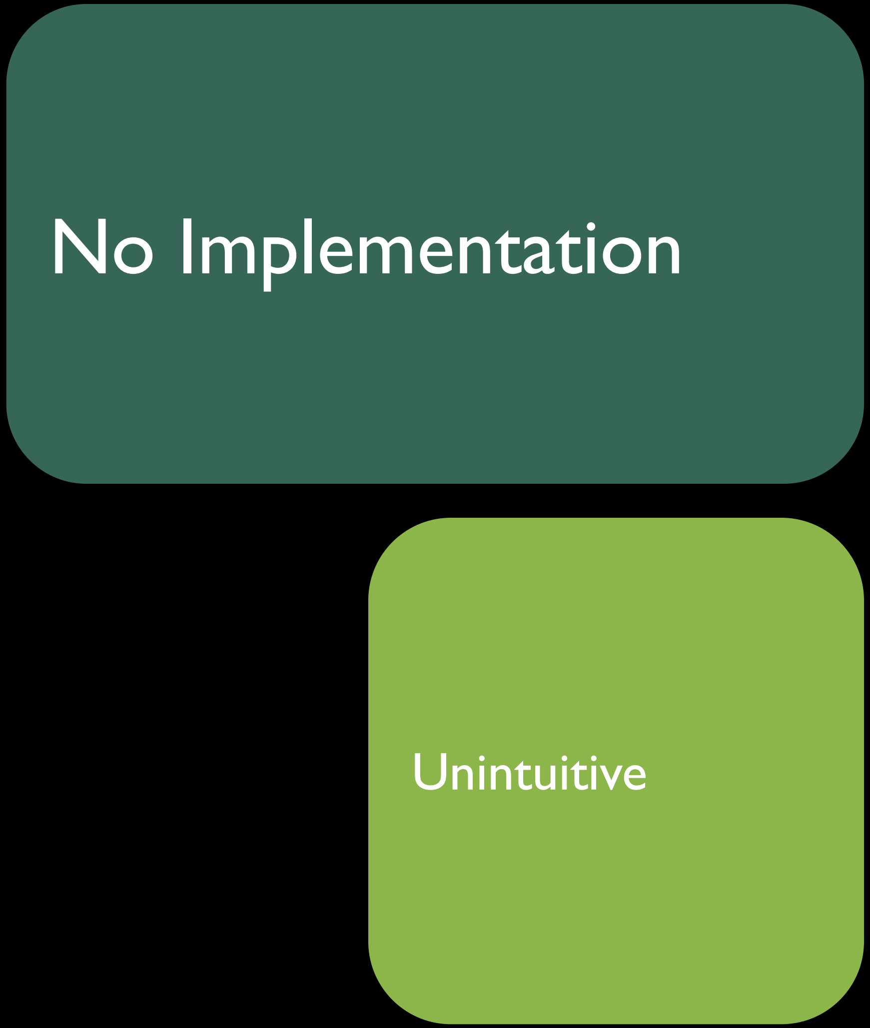
No Implementation

Intuitive

Missed something "Intuitive"

- No encryption
- No access control

Vulnerability classes



Missed something "Unintuitive"

- No MAC
- Side channel leakage
- No replay prevention

45% of projects

Vulnerability classes

Made a "bad choice"

- Weak algorithms
- Homemade encryption
- strcpy()

Vulnerability classes

Misunderstanding

Conceptual Error

Made a “conceptual error”

- Insufficient randomness
- Disabling default protections

44% of projects

Vulnerability classes

Made a programming “mistake”

- Control flow error
- Skipped algorithmic step

21% of projects

Mistake

Summary of data analysis

- No implementation & misunderstanding more common (78%) than mistake (21%)
  - Mistake: control error, skipped step
- Unintuitive requirements missed or implemented incorrectly much more often (45%) than intuitive ones
  - Unintuitive: MAC; avoiding side channels and/or replays
  - Intuitive: Encryption for privacy; access control
- Implementation complexity breeds mistakes
  - Failure to localize functionality, minimize TCB, completely mediate
- Mistakes readily exploited
  - Almost always result in contestant attacks

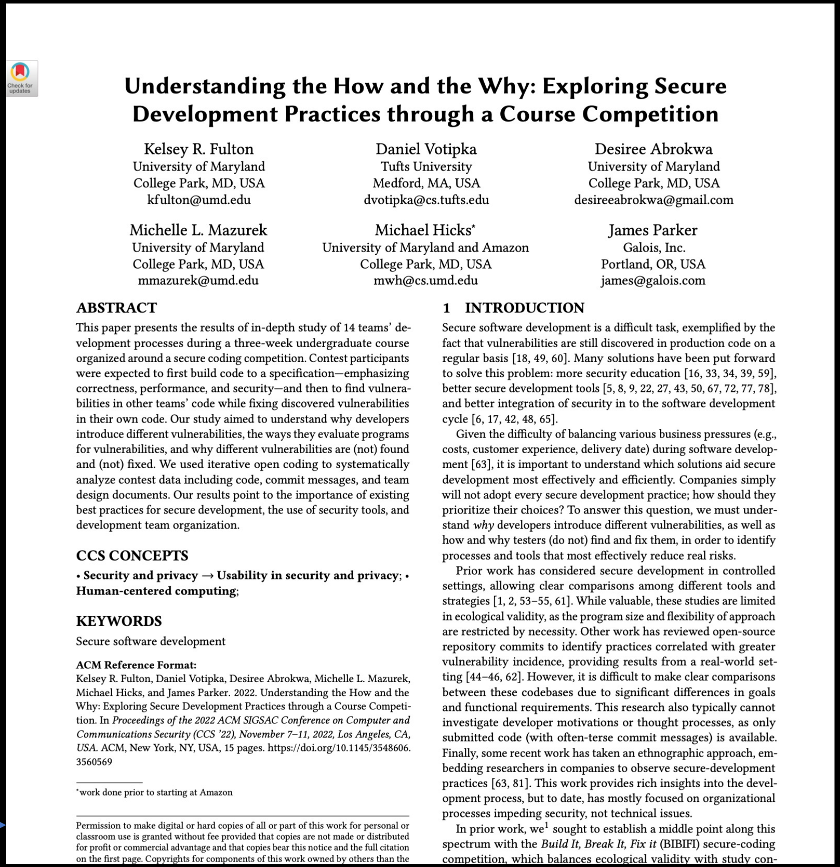
Recommendations

- Simplify API design
  - Build in security primitives and focus on common use-cases
- Indicate security impact of non-default use in API documentation
  - Explain the negative effects of turning off certain things
- Expand capabilities of vulnerability analysis tools
  - More emphasis on design-level conceptual issues

Discussion

- What is your take on what the data seems to be saying?
- What sort of follow-on studies could we do to support or refute the stated conclusions?
- How does this study's results speak to the big picture of securing software?
- What technical next steps do you think might be worth pursuing, to improve the state of affairs?

One more paper in this series!



Understanding the How and the Why: Exploring Secure Development Practices through a Course Competition

Kelsey R. Fulton
University of Maryland
College Park, MD, USA
kfulton@umd.edu

Daniel Votipka
Tufts University
Medford, MA, USA
dvotipka@cs.tufts.edu

Desiree Abrokwa
University of Maryland
College Park, MD, USA
desireeabrokwa@gmail.com

Michelle L. Mazurek
University of Maryland
College Park, MD, USA
mmazurek@umd.edu

Michael Hicks*
University of Maryland and Amazon
College Park, MD, USA
mwh@cs.umd.edu

James Parker
Galois, Inc.
Portland, OR, USA
james@galois.com

ABSTRACT

This paper presents the results of in-depth study of 14 teams' development processes during a three-week undergraduate course organized around a secure coding competition. Contest participants were expected to first build code to a specification—emphasizing correctness, performance, and security—and then to find vulnerabilities in other teams' code while fixing discovered vulnerabilities in their own code. Our study aimed to understand why developers introduce different vulnerabilities, the ways they evaluate programs for vulnerabilities, and why different vulnerabilities are (not) found and (not) fixed. We used iterative open coding to systematically analyze contest data including code, commit messages, and team design documents. Our results point to the importance of existing best practices for secure development, the use of security tools, and development team organization.

CCS CONCEPTS

• **Security and privacy → Usability in security and privacy**; • **Human-centered computing**;

KEYWORDS

Secure software development

ACM Reference Format:
Kelsey R. Fulton, Daniel Votipka, Desiree Abrokwa, Michelle L. Mazurek, Michael Hicks, and James Parker. 2022. Understanding the How and the Why: Exploring Secure Development Practices through a Course Competition. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560569

*work done prior to starting at Amazon

1 INTRODUCTION

Secure software development is a difficult task, exemplified by the fact that vulnerabilities are still discovered in production code on a regular basis [18, 49, 60]. Many solutions have been put forward to solve this problem: more security education [16, 33, 34, 39, 59], better secure development tools [5, 8, 9, 22, 27, 43, 50, 67, 72, 77, 78], and better integration of security in to the software development cycle [6, 17, 42, 48, 65].

Given the difficulty of balancing various business pressures (e.g., costs, customer experience, delivery date) during software development [63], it is important to understand which solutions aid secure development most effectively and efficiently. Companies simply will not adopt every secure development practice; how should they prioritize their choices? To answer this question, we must understand *why* developers introduce different vulnerabilities, as well as how and why testers (do not) find and fix them, in order to identify processes and tools that most effectively reduce real risks.

Prior work has considered secure development in controlled settings, allowing clear comparisons among different tools and strategies [1, 2, 53–55, 61]. While valuable, these studies are limited in ecological validity, as the program size and flexibility of approach are restricted by necessity. Other work has reviewed open-source repository commits to identify practices correlated with greater vulnerability incidence, providing results from a real-world setting [44–46, 62]. However, it is difficult to make clear comparisons between these codebases due to significant differences in goals and functional requirements. This research also typically cannot investigate developer motivations or thought processes, as only submitted code (with often-terse commit messages) is available. Finally, some recent work has taken an ethnographic approach, embedding researchers in companies to observe secure-development practices [63, 81]. This work provides rich insights into the development process, but to date, has mostly focused on organizational processes impeding security, not technical issues.

In prior work, we[1] sought to establish a middle point along this spectrum with the *Build It, Break It, Fix it* (BIBIFI) secure-coding competition, which balances ecological validity with study con-