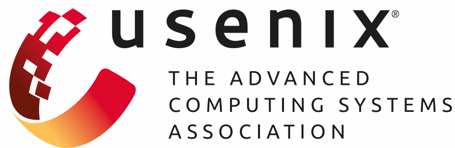

The Industrial Age of Hacking

Timothy Nosco, *United States Army;* Jared Ziegler, *National Security Agency;* Zechariah Clark and Davy Marrero, *United States Navy;* Todd Finkler, *United States Air Force;* Andrew Barbarello, *United States Navy;* W. Michael Petullo, *United States Army*

https://www.usenix.org/conference/usenixsecurity20/presentation/nosco

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.

The Industrial Age of Hacking

Tim Nosco
United States Army

Jared Ziegler
National Security Agency

Zechariah Clark
United States Navy

Davy Marrero
United States Navy

Todd Finkler
United States Air Force

Andrew Barbarello
United States Navy

W. Michael Petullo
United States Army

Abstract

There is a cognitive bias in the hacker community to select a piece of software and invest significant human resources into finding bugs in that software without any prior indication of success. We label this strategy depth-first search and propose an alternative: breadth-first search. In breadth-first search, humans perform minimal work to enable automated analysis on a range of targets before committing additional time and effort to research any particular one.

We present a repeatable human study that leverages teams of varying skill while using automation to the greatest extent possible. Our goal is a process that is effective at finding bugs; has a clear plan for the growth, coaching, and efficient use of team members; and supports measurable, incremental progress. We derive an assembly-line process that improves on what was once intricate, manual work. Our work provides evidence that the breadth-first approach increases the effectiveness of teams.

1 Introduction

Can we build a better vulnerability discovery process? Many researchers have proposed tools that aim to aid human work, including approaches that apply symbolic execution, fuzzing, taint tracing, and emulation to the problem of bug finding. These techniques automate bug finding in the sense that, with some up-front cost, they carry out a search over time of software states with little need for human intervention. The goal of each refinement or invention is to increase the effectiveness of tools when they are used on real software. Yet finding vulnerabilities at scale still appears out of reach, partly due to the human effort required to effectively setup automated tools.

Our work focuses on human processes that build on a foundation of automation. We choose to focus on autonomous technologies (as opposed to other vulnerability discovery techniques such as static analysis) because we view them as holding great promise for scalability. However, we by no means discourage the use of other techniques, either alone or in connection with autonomy.

We propose a minor change to Votipka's process [40] by creating a deliberate software selection step we call *targeting*. We encourage novice hackers to perform a breadth-first search of potential software targets to accomplish only the essential-but-preliminary tasks that allow automated analysis. We suggest bringing in more experienced hackers to perform a deeper but more costly analysis of select software only once novices have tried and failed with automation. Our approach focuses the most experienced practitioners on hard problems by delegating other work to hackers with less experience; they, in turn, generate work artifacts that are useful for informing more advanced analysis. Due to the volume of targets, all hackers have the opportunity to select software suitable for their skill level, and team members have a clear path for knowledge growth and coaching.

This paper describes our vulnerability-discovery process along with the repeatable experiment that we used to assess it. We found substantial evidence to claim a breadth-first search makes a superior targeting strategy in the presence of automation. We also measured significant improvement in the confidence of subjects who applied our process to a vulnerability-discovery campaign.

After surveying related work in §2, we introduce our process in §3. §3.1 describes a depth-first strategy, and §3.2 describes our breadth-first strategy. §4 lays out the design and execution of our experiment: the application of our process with two teams of hackers applying two strategies during two successive weeks. §5 describes our results, and §6 concludes.

2 Related work

Votipka, et al. studied the interplay between testers, who investigate software prior to release and hackers, who investigate software after release. They derived from their study a common vulnerability discovery process, which we build on here [40, §V].

Manès, et al. provide a survey of many of the techniques found in fuzzing tools [21]. For example, Mayhem [5] and Driller [36] address the path explosion problem in symbolic execution. Klees, et al. survey the fuzzing literature to comment on the required procedure for good scientific and evidence based research [20].

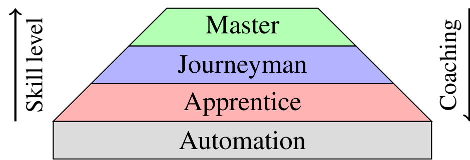

Figure 1: Practitioners, divided into apprentices, journeymen, and master hackers; each represents a higher level of skill and experience, and each mentors the level below

Avgerinos, et al. mention analysis at scale, specifically how scaling analysis to thousands of software artifacts makes any per-program manual labor impractical [1, §6.4]. Babic, et al. discuss a method to harness library code automatically and at great scale [2]. Sawilla and Ou proposed ASSETRANK, an algorithm that reveals the importance of vulnerabilities present in a system [31]. The strategy we propose builds on OSS-Fuzz's idea of passing indicators of vulnerability to human experts for remediation [32].

In this study, we extend Votipka's vulnerability discovery process, use modern tools referenced by Manès, accept some amount of manual labor to make finding bugs in real software artifacts tractable, and use statistical tests to extrapolate our observations to the hacker community.

3 Vulnerability discovery process

We aim to discover ways to increase the effectiveness of teams built on a foundation of automation (i.e., fuzzing and related technologies) whose goal is to find bugs in software. Most interesting to us are bugs exploitable in a way that circumvents a system's security. We consider both published and novel bugs, focusing on employed software where vulnerabilities—published (*n*-day) or not (0-day)—are the main concern. Here we describe our vulnerability discovery process, based on Votikpa's work. We also introduce distinct two strategies that our experiment compared.

Observations led us to divide bug finders into three categories: apprentices, journeymen, and masters, as depicted in Figure 1. Collectively, we refer to these three groups as hackers. Maximizing the productivity of each skill level while enabling a progression from apprentice to master over time was a key motivator to our process.

An apprentice hacker has a general computing background and a basic understanding of how to apply some number of automated software analysis tools. At the core of an apprentice's tool set are fuzzers. Apprentices have limited experience in modifying software, and they do not yet have a command of the internal workings of the various build systems used for software development.

A journeyman hacker adds the ability to manipulate a program to work with his tools. A journeyman can modify source code or use binary patching to deal with obstacles that thwart fuzzing, such as checksums, encryption, or non-deterministic functionality. A journeyman routinely modifies targets to expose their attack surfaces.

The highest skill level, master, adds the ability to manipulate or create tools in order to better investigate a target program. Many existing tools were written by masters in need of a specialized approach to a particular piece or class of software. We will use Alice as an apprentice, James as a journeyman, and Meghan as a master hacker.

Other actors include leaders, who make targeting decisions based on the work of hackers; analysts, who correlate technical work with other resources such as blogs and Common Vulnerabilities and Exposures (CVE); and system support personnel, who manage automation jobs and computing resources. Motivated by our observations of the skill levels that comprise vulnerability-discovery teams, we added a targeting step to Votipka's vulnerability discovery process [40], as shown in Figure 2.

Targeting Targeting selects software for investigation. The term *target* is common among bug finders because software targets are subject to an unusually careful inspection that resembles an attack [28]. The goal of the targeting phase is to divide a complex system or group of complex systems into targets that can be individually studied in later phases of the vulnerability-discovery process. Even monolithic software artifacts decompose into multiple targets: for example, a browser decomposes into media libraries, TLS and networking libraries, an HTML/CSS renderer, a JavaScript engine, and so on. Experience shows that many or most teams have multiple or many targets under consideration.

Only cursory information focused on how to perform this division should be collected during the targeting phase. Examples include the pervasiveness of existing security research focused on the target; the availability of target source code, bug trackers, and public developer forums; and the impact of finding a vulnerability in the target. The availability of the target itself; its dependencies (e.g., software, hardware, and supporting resources); and the tools necessary to interact with the target—both automatically and manually—are other considerations.

The predicted *P*rofit of a vulnerability-finding effort is proportional to the *L*ikelihood and *V*alue of success and inversely proportional to the projected *T*ime investment and required *S*kill level.

$$P = (L \times V) - (T \times S)$$

This model guides targeting and subsequent decisions about how to proceed while maximizing return on

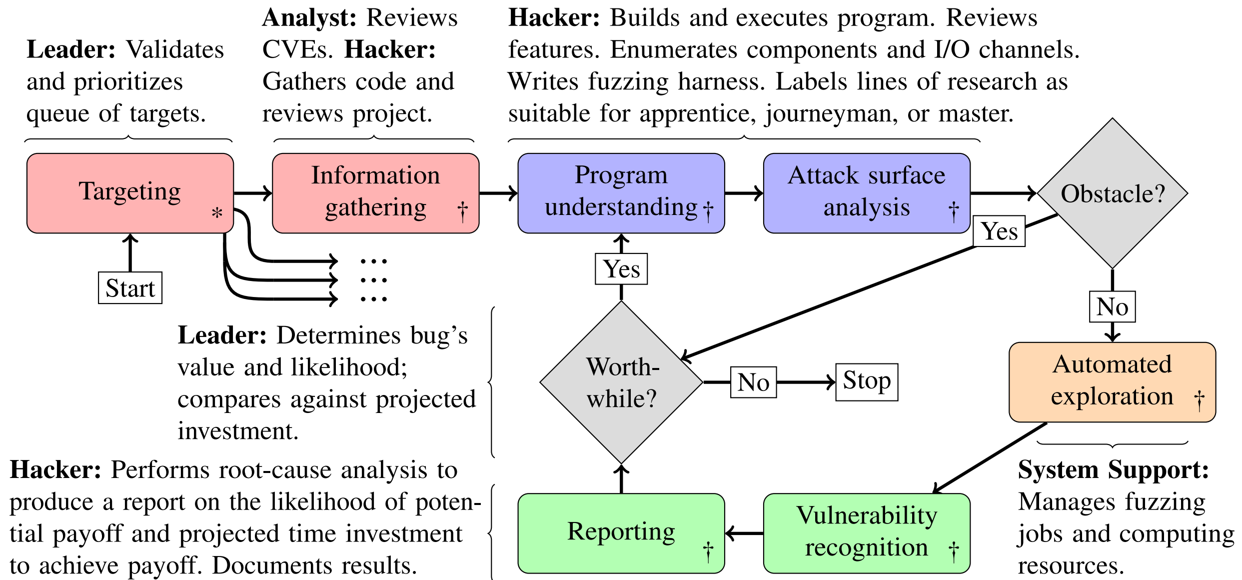

Figure 2: Our vulnerability-discovery process adds targeting (*) to the steps of Votipka, et al. (†) [40, §V].

investment.

Not all hackers are created equal, and building expertise in software security can take years of effort, experience, and coaching [28]. A targeting strategy ought to boost overall productivity across all skill levels. We wanted to derive a sufficiently large number of software targets to allow hackers of varying skill levels to select work that aligns with both their ability and interest.

Ultimately, we arrived at a strategy that coupled the freedom of target choice with a "fail fast" team culture and an incentive for producing rapid results. Thus our targeting phase allows teams to self-organize, and it enables a more effective use of journeyman- and master-level hackers' scarce time. We describe a depth-first strategy in §3.1 and our favored breadth-first strategy in §3.2.

Information gathering The first steps individual hackers and analysts take during the vulnerability-discovery process is to collect additional information about the target, this time with an eye toward decision making during later phases. Key among this information are general details about the target's development, prevalence, and known current or previous defects, along with any security research already complete [40].

Existing analysis can quickly advance the understanding of obstacles, along with the methods of overcoming them. For instance, work to fuzz the OpenSSH dæmon [26] describes eleven non-trivial techniques to harness targets for fuzzing. When considering a team of mixed proficiency, descriptive guides such as this allow a novice to begin work that would otherwise require a more experienced hacker.

> **Scenario A** Alice begins investigating a piece of software that provides an NTP service. She notes the version in common use, reviews the National Vulnerability Database for known vulnerabilities, and records the primary programming language used in the project.

Program understanding Hackers next focus on gaining knowledge of the target's operation and design. Of interest is how the target is used as it was intended, more advanced use cases and configuration options, and the general design of the target software. Information gathered during this phase can come from documentation, source code, online forums, users, developers [40], and other sources. Program understanding and the next phase, attack surface analysis, make up an iterative cycle within the vulnerability discovery process; Figure 2 illustrates this with the *Obstacle* decision point.

> **Scenario A (cont.)** Alice installs the NTP service by downloading its source code from an online repository and running `./configure; make`. She references the usage instructions to interact with the software.
> **Scenario B** Working on a separate project, James compiles a browser after reading preliminary notes by Alice. This takes some work as his Linux distribution did not provide a required library. He identifies the browser's JavaScript engine and HTML renderer, and he notes the libraries used to decode various media

formats. James also notes that the default build makes use of Address Space Layout Randomization (ASLR), non-executable stacks, and stack canaries.

Attack surface analysis Investigating a program's attack surface involves devising ways to provide input to portions of the target program. In many cases, this takes the form of a fuzzing harness, also known as a *driver application* [21], which directs the inputs a fuzzer generates to a portion of the program's attack surface. The practical execution of this phase diverges among hackers of varied skill.

Our process asks apprentices to apply known tools until an obstacle prevents them from further process. Their strategy is to give up quickly when progress stops, document their successful work, and move on to the next target.

Journeymen consume the documentation produced by the apprentices, allowing them to immediately apply higher-order analysis and continue the program understanding–attack surface analysis cycle.

Projects that reach the master level either are exceptionally important or have exceeded other hackers' ability to exploit despite clear indications of buggy behavior. A master should always enter the program understanding–attack surface analysis cycle with a plethora of documentation and other products generated by apprentices and journeymen. The master's time is thus spent doing tasks only a master could perform.

Some literature suggests that to even begin vulnerability discovery, a person must already have the skill we describe as a master's: "Although fuzzing tools are more common, people typically do not use off-the-shelf tools; they prefer making their own fuzzers ... [11]" We found counterexamples where apprentices and journeymen were able to progress through every phase of vulnerability discovery. In other cases, they provided clear value to later work by a master hacker. In either case, our process aims to maximize the contributions of less experienced hackers while making the employment of master hackers more efficient.

Scenario A (cont.) Alice learns the types of inputs her target accepts. These include input through network sockets as well as configuration files the server reads when started. The fuzzing tool she is familiar with doesn't support network fuzzing, so she makes a note for a future analyst to try network fuzzing. However, she knows how to start a fuzzing run based on file input.
Scenario B (cont.) James writes a fuzzing harness for the browser's more complicated media libraries, and he packages his work using a Dockerfile. Alice helps, as she had not yet learned how to use Docker.

Automated exploration Once a team learns how to manipulate the inputs of a program, it iteratively performs these manipulations to enumerate as much functionality of the program as possible. This maximizes the chance of finding a vulnerable condition. While "sometimes, a 'lucky' run-time failure leads to a vulnerability [11]," we focus most in this phase on testing the target program in a fuzzer using the harnesses produced by the previous phase. In order to make results repeatable, our team standardized the output of the attack surface phase to be a Dockerfile [3] that combined the target program and its fuzz harness.

A hacker's proficiency, along with a consideration of the suitability of a given target determines the choice of a fuzzer. The effectiveness of a fuzzer includes the efficiency of harnessing the target and features (such as address sanitization, scalability, speed, and so on). Different fuzzers favor different types of targets. As an example, LibFuzzer aids in the work of writing a fuzz harness for a library, whereas American Fuzzy Lop (AFL) enables a hacker to begin fuzzing quickly given a binary target that reads its input from a file or the standard input stream.

Scenario A (cont.) Alice starts a fuzzing run on the unmodified NTP program with configuration files as the fuzzed input.
Scenario B (cont.) James deploys his browser media handling harnesses for fuzzing. They both work on other targets while the fuzzers run.

Vulnerability recognition Hackers who discover bugs while iterating through the process must confirm whether the bugs are vulnerabilities. A vulnerability exists when a bug is proven to be exploitable by an attacker [34]. This can be as simple as running the target program with the crashing input identified in the previous phase, or as complicated as setting up a complex system to observe the real-world effects of certain input. Automation in this phase might be necessary to balance the amount of human time that is required to review results, especially when a multitude of program crashes are discovered.

Scenario A (cont.) Alice begins another target.
Scenario B (cont.) Fuzzing discovers six inputs that cause the targeted browser to crash. James is not able to exploit these bugs, so Meghan takes on the task. James shifts his focus to fuzzing the browser's use of Transport Layer Security (TLS).

Reporting Finally, the hacker who finds a vulnerability prepares a report that allows developers to correct the bug. A clear description of the impact and prevalence of

the vulnerability allows software maintainers to prioritize their efforts. The report can take on different forms, but as *The CERT Guide to Coordinated Vulnerability Disclosure* states, the technical and practical details of the vulnerability and attack scenario should be well-documented [16]. To aid in the growth of other hackers, reports should be readily available and searchable.

> **Scenario B (cont.)** Meghan documents her findings, along with the findings of James. Meghan and James work together to package the exploit as a usable proof of concept. Later, the team discusses their results.

3.1 Depth-first strategy (S_D)

The most obvious targeting strategy resembles a depth-first search. First, hackers select a small set of targets based on some metric of operational impact. For each selected target, the team spends time auditing the software for bugs. This work flow is very natural: it focuses the team's effort on one software artifact at a time. Researchers select the target at the very beginning of their work and persistently look at that target for a notable period of time.

The depth-first work flow has found bugs in large software that requires a familiarization period [11]. For example, Google Project Zero researchers applied this strategy to find bugs in Apple's Safari browser. The researchers harnessed the underlying libraries used in Safari, and this required significant program understanding along with modifications to the build chain. They found 26 bugs over the course of one year using custom-built tools [13].

This strategy is straightforward from a management perspective. A team leader collects information from each hacker and distributes it to the teammates inspecting the same target. The leader divides work based on the approach of each team member. For example, one hacker might examine the unit tests distributed with the target software, modifying them to suit the team's aims; another could analyze the software with a popular static-analysis tool; and yet another could attempt to harness different parts of the target program to work with a fuzzer. The responsibility for scheduling the fuzzing jobs and subsequent review often falls on the author of a harness.

Hackers employing S_D record information collectively because it is immediately relevant to the other team members. To promote coaching, the team pairs novice hackers with experts hoping the novice will assimilate concepts and techniques from the expert.

The primary pitfall of S_D appears to be its inefficiency relative to the broad skill levels found on practical teams. With few software artifacts under scrutiny, the team will exhaust the easier tasks related to finding bugs. This leaves apprentices and possibly even journeymen less able to contribute. Simultaneously, masters might find themselves idle or performing tasks better suited for the other skill levels at the beginning of a project.

Another pitfall is the inefficient use of automation. After starting a fuzzing run, the team is left to continue working on the same target. They might build additional fuzzing harnesses or carry out in-depth manual analysis. Yet the automation might later uncover information that would have aided those processes, or it might even find the bugs they seek. Ploughing forward might waste human effort.

3.2 Breadth-first strategy (S_B)

We devised a new strategy that aims to address the pitfalls of S_D. Our goals were to scale the vulnerability-discovery process to support a growing team of hackers, reduce hacker fatigue, and increase the production of fuzz harnesses. To do this, our strategy relies on the idea of drastically increasing the pool of software targets. We encouraged hackers to produce the greatest number of fuzzing harnesses possible in each workday. We call this the breadth-first strategy (S_B).

S_B encourages apprentice-level hackers to give up when it becomes clear that harnessing a particular target would require a significant time investment. Rather than continue down a "rabbit hole," apprentices document any pertinent information about the target before moving it to a separate "journeyman" queue. This provides more experienced hackers material to review before applying their more experienced abilities.

We posit that the key to this strategy is to collect a large queue of targets and, for each target, have apprentices do the simplest possible thing and nothing more. Keeping apprentices out of rabbit holes allows more skilled hackers to more deeply investigate a target once it is accompanied by a report. In some cases, apprentices produce a working build or even a corpus of fuzzing outputs, but not if producing these artifacts exceeds their abilities. Ways to generate large pools of interesting targets include (1) dividing a device into its software components, (2) following a thorough analysis of the system-level attack surface, (3) enumerating library dependencies, and (4) investigating multiple bug-bounties. Having a large pool of targets allows apprentices to reject targets whose obstacles exceed their ability. Examples might include software with challenging run-time requirements, such as real-time operating systems running on niche hardware; programs that require dynamic network streams like FTP; programs requiring extensive system configurations; or programs that make use of a custom build process. With such a large

queue, prioritizing the targets so hackers spend more time on higher-value items becomes critical. For example, hackers on a penetration-testing team should prioritize a target that allows external network connections.

An important consideration in our study was figuring out how to train new members quickly, while at the same time allowing them to provide operational value to the team. A large queue of targets allows apprentices to select those compatible with the tools that they already know how to use. When they find that a target does not work with a tool they know, they can record what they learned and move it into a journeyman queue. Journeymen pick up targets that an apprentice had begun and push them into the exploration stage. The apprentice can, in turn, learn from that work. Each team member's work is thus frequently reviewed by more experienced people, and there is a clear path for someone to learn based on the experience of others. Similarly, master hackers record the problems that they overcome along with the types of solutions that they apply. These notes frequently help journeymen grow in knowledge too.

To make efficient use of automation, all work should stop on a particular target whenever a new automated job begins. Only once that job has completed (based on some predetermined measure of completeness) are the results reviewed, incorporated into the findings, and used to determine next steps. In this way, unnecessary human effort is minimized by relying on automation to the greatest extent possible.

4 Experiment

We designed a human study to investigate our two strategies: depth-first (S_D) and breadth-first (S_B). Our experiment took place over the course of ten days, as summarized in Figure 3. This counterbalanced design follows *The SAGE Encyclopedia of Communication Research Methods* [8] and includes between-subjects tests at the end of the first week and within-subjects tests at the end of the second week [6, 9, 27]. We ran our experiment on the business days from November 7 through November 22, 2019, taking the 8th and 11th off for Veteran's day. The detailed schedule of our experiment appears in Appendix B.

4.1 Subject selection

Our subjects drew from a pool of US Cyber Command personnel, each of whom had at least a basic understanding of the principles of system and software security. Our primary means of recruiting was a pamphlet posted throughout US Cyber Command work spaces, but we also invited promising candidates by email. We advertised our goal as identifying the best target-selection strategy for bug finding, and we indicated that selected subjects would spend two weeks working with expert hackers to analyze a range of real software. Finally, we noted that we would provide an AFL fuzzing tutorial for all participants. Our pamphlet asked for applicants who (1) had experience with Linux, (2) could work with open-source projects, (3) could conduct Internet-based target research, and (4) could read and modify C programs. 15 people indicated interest. Candidates signed a participation agreement and completed a self-assessment (Appendix A) used to assign teams.

4.2 Orientation

Twelve subjects were present on the first day of our experiment. We used the subjects' self-reported years of hacking experience to create groups. Then, we performed a representative random sample to assign the present subjects to two balanced teams of six. The distribution of the original fifteen applicants contained: eight subjects under one year of experience, two subjects between one and two years, two with four years, two with five years, and one subject who reported eight years experience. All applicants with over one year of experience claimed hacking was—at some point in time—part of their full-time job. The buckets are not uniform, but rather partition the reported skills in a way divisible into two teams. We assigned each team an investigator to serve as the leader, each with experience leading hacking teams.

We spent the first day providing introductions, presenting a class on the popular open-source fuzzing tool AFL [42], assessing the skills of our subjects, and describing our work flow and submission standards.

Period of instruction The class combined a lecture with exercises ranging from how to compile using `afl-gcc` to fuzzing `bzip2` using `afl-qemu`. We also provided a 30 minute lecture-only class on Docker [3].

Skill assessment Our self assessment was subjective, so we devised a more objective measurement of subject skill in the form of a series of technical skill assessment tests. We administered these tests three times: once immediately after the initial training course, once at the half-way mark (before the teams exchanged strategies), and once at the end of the experiment. One aim was to measure the amount of skill our subjects developed during the course of executing each strategy.

All three skill assessments followed the same form, consisting each time of a new set of five binaries taken

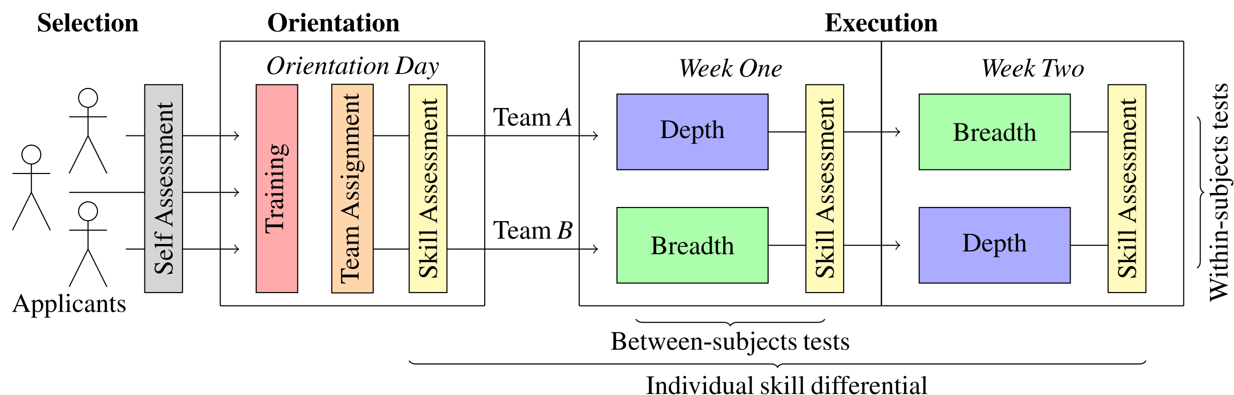

Figure 3: An overview of our experiment, divided into the phases of selection, orientation, and execution; we provide a detailed schedule in Appendix B

from a pool of fifteen. We took these binaries from three popular public corpora of fuzzing targets: the Trail of Bits adaptations of the Cyber Grand Challenge binaries [38], the MIT Lincoln Laboratory Rode0day bug-injection challenges [12], and Google's OSS-Fuzz project [32]. In some cases, we provided source code. By the end, each subject had investigated all fifteen binaries over the course of three skill assessments. We list the binaries in Appendix F. The binaries we selected represent a variety of practical challenges varying across a number of dimensions, including small versus large programs, pre-built versus complicated build systems, and artificial versus natural bugs.

Each of the targets employed in our skill assessments is freely available on the Internet. Also available on the Internet is an "answer key" for each target including, in some cases, a list of bugs and, in other cases, a pre-built fuzzing harness. Our intention was to emphasize that open source research is a key component of the vulnerability-discovery process and to acknowledge that known n-day vulnerabilities matter.

Subjects were given exactly one hour to make progress on these targets; clearly not enough time for a deep-dive into any of them. Their instructions emphasized two goals: (1) find bugs and (2) create fuzzing harnesses.

The motivation for finding bugs is self-evident, as it aligns with the goal of vulnerability research in general. The reason for the goal of creating fuzzing harnesses is to put subjects in the mindset of using automation as a primary strategy for achieving the first goal.

Target selection Selecting targets for this experiment was no easy task. Klees, et al. describe how selecting targets to evaluate a fuzzing tool is difficult [20, §8]. We encountered many of the same challenges when evaluating our hackers. After considering using the benchmarks in earlier work [12, 15, 20, 38], we decided on something else altogether. We chose to evaluate OpenWrt [10]. The packages available to OpenWrt are open source and serve diverse purposes. Each of our targets was real and thus representative of modern, complex, and deployed software.

Before the subjects began the vulnerability-discovery process, we ran a simple static analysis script that extracted some important information from every OpenWrt package. We collected each package's version, a listing of the files exported by the package, the results of running `file` [19, p. 46] on each item in the package, and the intersection of each ELF file's exported symbols with a set of frequently misused standard library functions such as `strcpy` and `gets`.

For S_D, we selected two targets: `dropbear` and `uhttpd`. Because these services are installed and listening on a network socket by default, they represent the most likely choices for a hacker performing S_D. For S_B targets, we allowed subjects to select any software the OpenWrt package manager provides, *except* for `dropbear` and `uhttpd`. We excluded those two during S_B so that both teams would start fresh on those targets during S_D. Two targets for S_D and a thousand for S_B does present an asymmetry; upon first inspection, this might appear unfair, as (1) the true number bugs in the underlying targets is biased and (2) the two S_D targets require more skill to analyze than the average of the S_B targets. Thus the reader might claim, "of course S_B can find more bugs, there are more bugs to find!" We agree. We argue this perceived unfairness is really intuition that S_B is more effective than S_D, because our selections represent real systems. Bugs exist, but over committing to a single target is not the easiest way to find them.

In order to aid the post-study analysis, we selected a four-year-old version of OpenWrt: 15.05.1. As others mention [20], there is no good substitute for real bugs found. Unique crashes as defined by program path or stack hash do not correlate to unique bugs. By choosing

an older version of OpenWrt, we hoped that subjects would find bugs that were patched by version 18.06.5, the modern release as of our experiment. This way, we could take crashes and categorize them more precisely. Because all targets are open source, we will use their issue trackers to report crashes still present in the modern version.

Work flow and tools Both strategies, S_D and S_B, require tools to manage the execution of the vulnerability-discovery process. We spent time during the orientation describing these tools and the manner of their use.

We relied on GitLab to manage our teams due its feature set and open-source availability. For each vulnerability-discovery campaign, we created a GitLab project, and for each proposed target we created a GitLab issue. We added the package information derived from our scripts to each issue's text.

We directed our subject teams to track their progress using a GitLab issue board, divided into lists related to each step in the vulnerability-discovery process. Each team's board contained one list (as defined in [14]) for each of *open*, *information gathering*, *program understanding*, *exploration*, and *journeyman*. We depict a snapshot of one such board in Figure 4. Many authors, including Newport [25], note the need for experts to be minimally interrupted, and this is why we did not include every step of our vulnerability-discovery process in our issue boards. Instead, we attempted to balance our subjects' need for concentration, our own need to track progress, and the teams' need to record important information. We felt a reasonable compromise would ask subjects to:

- drag a ticket from *open* to *information gathering* upon initiating work on a target;
- append to an issue relevant articles, blogs, source repositories, corpora, and other information uncovered during their search;
- move an issue from *information gathering* to *program understanding* once they create products worthy of committing to the target's GitLab repository;
- move an issue to the *exploration* list upon creating working fuzzing harness; and
- move an issue to the *journeymen* list if progress becomes too difficult. In this case, comments will explain the obstacles encountered.

We gave each subject an Internet-connected workstation co-located with their team members. The workstations contained tools for our subjects, including: Ghidra [18], AFL [42], Honggfuzz [37], Docker [3], and Mayhem [5]. Each workstation also contained monitoring software and was thus tied to our data collection. We further allowed the subjects to use any bug-finding tool they desired, but we encouraged them to use dynamic-analysis tools. We also provided subjects a Docker container that emulates the OpenWrt 15.05.1 filesystem and services (adapted from other work [35]).

4.3 Execution

Our experiment involved two iterations of our vulnerability-discovery process. During the first iteration, Team *A* applied S_D, and Team *B* applied S_B. Roughly each hour, we stopped work and asked the subjects to complete a survey (Appendix C). The teams traded their strategies for the second iteration, and we repeated the skill assessment after each iteration. Each day ended with an end-of-day survey (Appendix D), and the final day included an end-of-experiment survey (Appendix E).

For the next four business days, subjects on each team—lead by an investigator—worked in their assigned strategy. We enforced that each group use their assigned strategy by selecting only two targets for S_D and approximately 1,000 targets for S_B. The team lead encouraged S_B subjects to give up quickly and select targets that they could reasonably accomplish in two hours of work. We gave subjects the intermediate skill assessment before they traded strategies for the final four business days. On the final day, subjects took the final skill assessment.

Limitations Our sampled population consisted solely of US Cyber Command personnel, but we posit our results are applicable to other organizations. Both teams knew on day one the software they would target for both weeks using our two strategies; this could have resulted in looking ahead at a future target, but team leads mitigated this by focusing work. Our two team leaders did double as investigators, but they tried to mitigate any bias towards S_B as they guided their teams.

Other aspects of our study were difficult if not impossible to control. Some subjects missed work due to unforeseen emergencies, although the collective time for both teams appeared to be about equal. At times, our Internet connection became prohibitively slow. This affected both teams and seemed to persist during both weeks of the study. Sometimes subjects would restart their workstation or it would crash from an unwieldy fuzz job. This affected our ability to collect and log data about the participant's actions. We also discovered during the experiment that our X11 monitoring tool did not capture

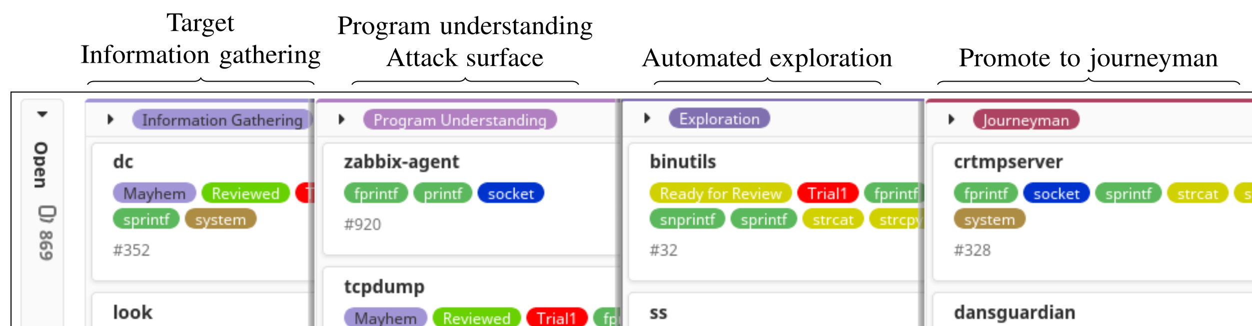

Figure 4: The use of Gitlab to track the progress of a vulnerability-discovery campaign; we used a variant of Kanban with bins that corresponded to groups of steps in our vulnerability-discovery process; each issue corresponds to a target

time spent in the X11 lock screen.

Human research standards and data collection We obtained a DoD Human Research Protection Program (HRPP) determination before executing the research described by this paper. This included an examination by our Institutional Review Board (IRB). All recruitment was voluntary and minimized undue influence. We assigned each subject a two-word pseudonym that was also their machine's host name, their Rocket.Chat user name, their survey response name, and their GitLab user name. Recorded data bore this pseudonym, and it was in no way linked to the subject's real name. We collected skill assessments, surveys, GitLab commits, comments, and work products. We also collected data using execsnoop, which logged programs started by the subjects, and x11monitor, which monitored the subjects' X11 cursor focus.

5 Results

Our analysis of the experiment's results involves four categories: survey questions, determining the number of bugs found, measuring the subjects' hacking skill, and ancillary data. We present this analysis here before commenting on our two strategies.

5.1 Surveys

We use Mann-Whitney u-test p-value (MW). That is, the probability that the statement listed is *not* true given our observation. We use this test to compare the means of survey responses and conform to the necessary assumptions [24, §1.2] except that each entry is an independent trial. This is violated because we sample each subject multiple times over the course of each method. We expect there is variation within a single subject's responses and thus we conducted multiple samplings. Potentially, some other tests such as repeated measures ANOVA [17] or Wilcoxon signed-rank test [41] are more fitting, but not quite right and not the focus of this paper. We choose Mann-Whitney mainly because it is a non-parametric test with minimal assumptions about the data's distribution and allows us to test the signed difference of means between two groups: S_D and S_B. B is the Bernoulli Trial as described by Papoulis et al. [30]. We must assume our sample of 12 is "large enough". To balance the number of tests with our small sample, we use an acceptance criteria of 0.020.

Hourly survey outcomes When comparing between subjects from both teams during the first week, subjects performing S_B felt less surprised (MW=0.003), less frustrated (MW=3×10^{-4}), and less doubtful (MW=0.004) than those performing S_D. They also spent more time interacting with tools (MW=5×10^{-7}) and more time harnessing (MW=0.002).

After the second week, we compared within-subjects on the team that transitioned from S_D to S_B. These subjects reported that S_B left them spending less time on research (MW=1×10^{-4}) and feeling less frustrated (MW=0.007), doubtful (MW=0.001), and confused (MW=0.009). S_B found them interacting with tools (MW=0.008) and harnessing (MW=0.009) more.

End-of-experiment outcomes Subjects felt S_D was less effective than S_B overall (B=0.019) and was a less effective use of their team's skills (B=0.003). When asked which method they would prefer to lead, subjects were less likely to choose S_D (B=0.003). Subjects felt breadth-first work was more independent but left them feeling less a part of a team (B=0.003). The subjects claimed S_B was less frustrating (B=0.003), and they unanimously said it was easier to get started with (B=2.400×10^{-4}) and easier for a novice to contribute to (B=2.400×10^{-4}). Subjects also unanimously claimed they learned something during the experiment (B=2.400×10^{-4}). Subjects felt more prepared (MW=0.010) and more interested (MW=0.015)

in hacking after the experiment than before. Every participant reported finding at least one bug (B=2.400×10^{-4}).

5.2 Determining number of bugs

As Klees et al. discuss in depth, many papers fail to provide control for randomness in fuzzing results [20]. Our approach was to collect subject harnesses and run each in three independent trials for 24 hours using the corpora and fuzzer selected by the harness creator. While Klees et al. also discuss finding "real bugs," the process of iteratively patching is extensive and time consuming. As a compromise, we settled on an approximation. In lieu of "real bugs," we decided to use the bug de-duplication mechanism in Mayhem [1, 5].

Statistical tests We use MW to test the significance of mean difference in coverage and bug metrics and conform to all required assumptions [24, §1.2]. We chose this test to measure the difference in bugs found by S_D and S_B primarily for the reasons suggested by Klees [20, §4].

Bug outcomes After using a total 18,432 compute-hours to test each harness three independent times for 24 hours and two cores each, we collected the results. The following table shows the cumulative number of unique bugs found in each independent fuzzing trial T_x.

Team	Method	Harnesses	T_0	T_1	T_2
A	S_D	8	3	2	3
A	S_B	42	31	23	40
B	S_B	61	42	49	40
B	S_D	12	4	4	4

Testing $f(S_D) < f(S_B)$ reveals some potentially coincidental results. Team A within-subjects, found a p-value of $(0.038 > 0.020)$; Within-subjects for team B, $(0.032 > 0.020)$. For the between-subject test of week one, $(0.032 > 0.020)$. However, combining both team's findings, we find significant evidence to claim $f(S_D) < f(S_B)$ with a p-value of $(0.002 < 0.020)$.

In addition to finding more bugs, the categories of bugs found by S_B are significantly more diverse and security-related than the bugs found in S_D. Both S_B sessions found multiple out-of-bounds write primitives as described in the Common Weakness Enumeration (CWE) database [23], while none were found by S_D. Both strategies found out-of-bounds reads [22], but S_B found significantly more and some that could lead to information disclosure. For bug-bounty hunters, this is important because bug criticality determines compensation [28].

5.3 Skill assessment

After each assessment, we collected the subjects' work products and notes and graded them with the goal of determining three objective measures: (1) number of working harnesses, (2) number of bugs found, and (3) number of bugs reproduced. We defined a fuzzing harness as *working* if, after a short while, it discovers new paths through the target program. We defined a bug as any program terminated by a signal that might result in a core dump. Some commonly-encountered examples include: `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGSEGV`, and `SIGTRAP`. Finally, it is possible for a subject to find a bug—either through static analysis or information gathering—but not reproduce it. Reproducing a bug requires the subject to successfully run the program with the crashing input.

After collecting each objective measure, we combined them into a single score for each participant for the purpose of analysis. While one could imagine assigning differing weights to each category, those weights would likely be chosen based on model fitting from training data. Perhaps a future researcher might use data from sources like HackerRank [39]. Given the large scope of this study, we chose to weight each category equally. A participant's score, then, is the sum of all measures: $h+b+r$.

Statistical tests and outcomes Our assessment of subjects before the study and after each strategy makes for a good candidate for the Friedman signed-rank test [33]. We chose this test over others such as repeated-measures ANOVA [17] because this test does not require an assumption about the underlying distribution of our data. In our case, this is important because we neither know the distribution of test scores nor think it reasonable to assume the distribution is normal. We again use an acceptance criteria of 0.020.

The Friedman test unfortunately revealed no statistically significant mean difference between the three assessments. When testing all twelve participants, we receive a p-value of 0.02024; for group one, 0.10782; and for group two, 0.12802. A larger sample of subjects might reveal more significant results.

5.4 Ancillary data

Browsing the web vs. strategy Dividing the work time into hour-long windows to bin time spent with the X11 focus on Firefox (the pre-installed web browser) and grouping the values by strategy S_D or S_B was not significant according to Wilcoxon signed-rank test [41]. The number of entries in Firefox's history and the team's strategy were also not significantly related.

Materials produced Figure 5 describes the number of materials produced by both teams under both strategies. Both teams produced more materials under S_B than S_D: Team *A* produced 151 and 588 products under S_D and S_B, respectively; and Team *B* produced 177 and 387 products under S_D and S_B, respectively.

5.5 Depth-first strategy discussion

This section, along with §5.6, records observations made during the daily team discussions with subjects. A number of factors challenge S_D in a semi-autonomous, team-based analysis environment. The process of investing significant resources into a single target can reveal novel flaws or no flaws at all; a hacker will not know which without first consuming considerable time and effort.

Minimum skill threshold Apprentice hackers are prone to falling in rabbit holes. Votipka described this thusly: "Without prior experience guiding triage, our practitioners relied on stumbling across vulnerabilities incidentally; or on their curiosity, personal creativity, and persistence with ample time to dig through the complexity of a program. Such incidental discovery is time consuming and haphazard, with little consistency in results [40, §VI.A.1]."

S_D made recruiting more difficult because of the extensive list of prerequisite knowledge required to get started with some of our targets. Considering the two depth-first projects mentioned in this paper, we sought experience in: (1) software reverse engineering and assembly architectures (2) C software development (3) understanding and modifying software build tool chains (4) binary patching (5) source auditing (6) bug finding (7) the use of static analysis tools (8) fuzzing We also aimed to find self-motivated problem solvers.

Unsurprisingly, S_D overwhelmed the less-skilled subjects. Subjects performing S_D felt more surprised, more frustrated, and more doubtful than during S_B. Subjects also claimed S_D was a less effective use of their team's skills than S_B. We posit that these sentiments resulted from the quick exhaustion of novice work at the beginning of a bug-finding session, leaving tasks requiring a more skilled practitioner. Very early on, when looking at `uhttpd` and `dropbear`, novice subjects found valuable information from Internet research, but for the remainder of the week, they contributed significantly less to team progress.

Feedback Loop When our teams were assigned a single target, they continued working on that problem even when automation might be on the path to a solution. At some point, the human will be doing work eventually rendered unnecessary due to that automation. This is inefficient because, in general, human time is expensive while computer time is inexpensive.

S_D left subjects less time to interact with tools and less time harnessing than S_B. This means hackers are not able to maximize the time spent producing new harnesses to test new code. There is a natural break where—once a harness is complete—it is inefficient for the hacker to continue work until they know what automation will discover.

Knowledge sharing and tasking A team of humans simultaneously investigating the same target incurs a high synchronization overhead. Some findings are of general interest and should be shared as soon as possible, but other information might not be of broad interest. Communicating incurs overhead, but under-communicating leads to duplicate work. How to balance this is not always immediately clear. Feedback from subjects indicated that S_B left them feeling less a part of a team than S_D. We believe this stems from the fact that S_B naturally leads to more independent work and a reduction in real-time communications in favor of asynchronous communication, such as notes and code submissions. This position is bolstered by teaming research in a related discipline that found the most productive teams in cyber defense exercises have the *fewest* direct human interactions [4].

The discrete tasks in the fuzzing process seem conducive to parallelization. In practice, these tasks turn out to be a pipeline, with progress on one task being necessary in order to advance to the next. With some targets, such as ubus [29], emulating the target is a nontrivial prerequisite to fuzzing. The narrow target selection of S_D does little to help with parallelizing the fuzzing pipeline.

Output Ultimately, Team *A* found zero bugs in `uhttpd` and three bugs in `dropbear`; Team *B*, zero and four. With S_D, hackers tended to go down "rabbit holes," investing significant time and effort into analyzing complex components of a target. The more time spent delving into a particular component, the more a sort of tunnel vision would develop. This left other components of the target ignored. Ultimately, deadlines led to overlooked bugs that might have been easy to find using automation techniques and minimal human effort.

5.6 Breadth-first strategy discussion

Minimum skill threshold and feedback Our apprentice hackers were both more prolific and more effective while employing S_B. S_B allows the human to completely

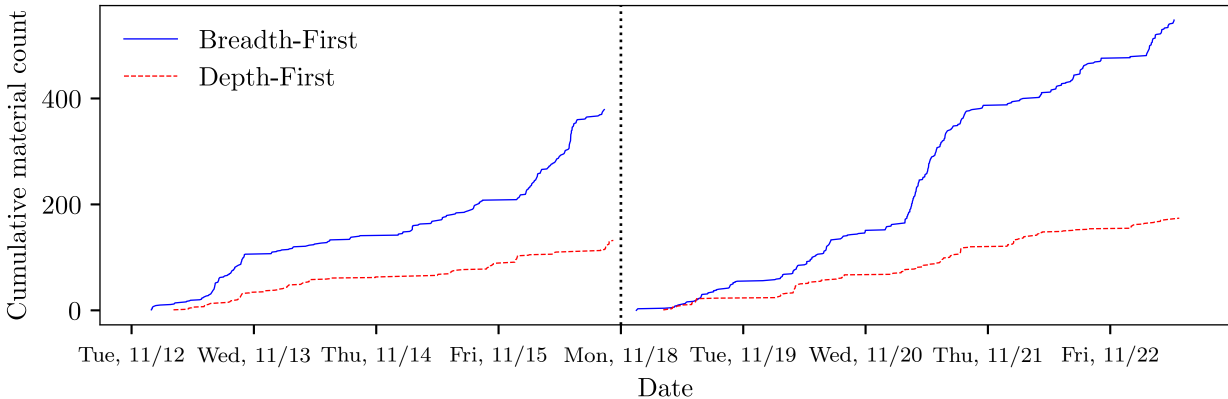

Figure 5: Total number of materials (Git commits, GitLab comments, GitLab projects, issues, issue tags, and Rocket.Chat messages) produced per team over time; the vertical dotted line represents the transition between strategies

hand off work to the machine and only continue work on that target once the machine had a chance to discover a solution. Such a model allows for a feedback loop from the human to the machine and back, minimizing human time spent, and iterating until reaching a desired outcome.

Knowledge sharing and tasking S_B allows team members to work with confidence on independent tasks, make progress until they understand the key pieces of information, and then communicate those pieces of information in an asynchronous way. This reduces overhead and redundancy while resulting in a continually growing record of findings, each feeding into the next. With respect to coaching, pairing a novice with an expert frequently resulted in the expert spending more time teaching then hacking. In a model where team members can record and convey their problem solving, more expert people can review those problems and suggest paths forward based on their experience. S_B's large set of targets means that hackers can create a collection of fuzzing pipelines as part of a parallel strategy.

5.7 Subsequent and future work

We applied our breadth-first strategy to other large-scale projects after our experiment, and we record here some additional lessons. We also suggest areas of future work.

Targeting We have further automated our targeting stage to make leaders more efficient. In one project, a team was asked to analyze four interesting devices. We wanted to apply S_B, so we wrote a script to enumerate the binaries on each device and establish issues on GitLab for each. This eased deciding what to work on, and it simplified the tracking of progress.

Future experiments might benefit from prioritizing targets. The targets in our experiment's queue were unsorted. Thus analysts tended to work through the *Open* column in GitLab from top to bottom, suggesting that sorting the queue would result in more time spent analyzing the highest-priority software.

Information gathering Future work could investigate using web scrapers to perform common research tasks. For example, if the target was `objdump`, a script could collect the results of searching for "objdump CVE", or "fuzzing objdump." These tools could easily append this information to each target's GitLab issue.

Program understanding There is a great deal of further research to be done in the area of program understanding and its impact on decision making. Automated tools should identify indicators of potential bugs. These indicators would justify additional time spent improving harnesses and diving deeper into understanding a target program. Without them, scaling becomes difficult if not impossible, as analysts tend to spend too much time focusing on challenging targets, possibly overlooking easier-to-find bugs in other targets. This is not to say that challenging targets should be ignored, but that team leaders should make an evidence-based determination of how much time to dedicate to a challenging target before the manpower cost outweighs the benefit of finding a bug.

Obvious examples of other information that tools could add to targets' GitLab issues include: the lack of basic run-time protection mechanisms like stack canaries, PIE, RELRO, and non-executable stacks; the presence of the SUID bit; `--help` and `--version` outputs; and whether the program listens on a port (i.e., `netstat` output) or runs automatically (i.e., `ps` output). This information

would help leaders prioritize targets or hackers select them, and there is clearly room for more ideas.

Attack surface analysis During a pilot study that preceded our experiment, running fuzz harnesses on our dedicated cluster required transferring the harnesses to a separate network where a team member managed fuzzing jobs. This proved to be a significant undertaking. The quality of the documentation provided by our hackers varied, and thus reproducing the harness occasionally failed. Failures led to several hours of rework. As a remedy, we adopted the use of Docker [3]. A Dockerfile able to build the targeted program on a base Ubuntu image with AFL installed has since then accompanied each new harness. We made the hackers responsible for performing test builds of their Dockerfile. The switch to Dockerfiles as a deliverable drastically reduced the overhead incurred when transferring the harnesses to a different network for fuzzing. We later expanded this architecture so that hackers could produce docker images that used any arbitrary fuzzer.

Automated exploration Automation in this stage consists of taking the completed harnesses and running them on computing resources. An architecture such as Clusterfuzz [32] matches our intent. During the depth-first strategy, we attempted to use our computing resources by having a single fuzz job run on many nodes of our cluster. When we transitioned into having many targets, we needed a simpler structure that would allow us to quickly run many jobs. We decided after our pilot study that a job running on a single node and employing all cores on the node would fit our needs. Not only is this an easier architecture to implement and maintain, but Cioce et al. show the diminishing returns of additional fuzz-cores make this a more efficient use of our computing resources [7].

Vulnerability recognition While our experiment was focused on building teams around the process of harnessing target applications, we realize that more work needs to be done to establish processes for managing the results of the fuzzing campaign—vulnerability recognition at scale. Some applications produced numerous crashes, with one application producing thousands of crashes. Techniques for dedicating sufficient time to crash triage while also continuing to harness new targets must be developed. With limited manpower, this is a challenging problem for which we are still working on a solution.

Other researchers who choose to extend our work should attempt to assign criticality scores to the bugs found. They might also wish to determine—before their experiment—the number of known bugs in the targets used.

Other We found overheads in S_B that were much less impactful to S_D. Small things such as enforcing GitLab policies or shepherding targets on and off of our computing resources became time-consuming with many projects happening simultaneously. Fuzzing, archiving and reviewing results was difficult to balance with other targets in the queue. Also, in our actual operational environment, higher leadership would add to our target queue, leaving us to figure out how assign priorities while balancing ongoing work. As with many endeavors, these practical matters are a ripe area for future work.

6 Conclusion

Frustrated with the pitfalls of S_D, we sought a better approach, and we found one. Evidence indicates S_B is more effective at finding bugs, and we found some positive side effects as well. S_B more efficiently employs hackers of varying skill levels. It also boosts the amount of documentation and learning resources available to hackers and leaders, cultivating technical growth. S_B better applies automated bug-finding tools, and it more clearly defines work roles and unit tasks. Our experiment to test S_D and S_B is repeatable and thus allows researchers to test other hypotheses related to the hacking process in a similar environment. Finally, we learned, coached, and hacked for fun and profit.

Acknowledgments

We are grateful for the aid Leslie Bell provided while we sought approval of our experimental approach using human subjects. Temmie Shade helped review our survey questions, and James Tittle coached us on the counter-balanced design of our experiment. Andrew Ruef also gave his time to discuss many of our early ideas. Richard Bae and ForAllSecure provided us with Mayhem installation, support, and notable computing resources. The staff at Dreamport (`https://dreamport.tech`) hosted our pilot and experiment, providing space, computing resources, and support. We thank our participants in both the actual study and the pilot. This paper would not have been possible without their help.

Our work was performed in part during a segment of the NSA's Computer Network Operator Development Program, and both the investigators and many of our subjects came from three military services: the Army, Navy, and Air Force. We are grateful for our services' support towards advancing vulnerability discovery.

A Self assessment

On a scale of 0–5, how comfortable do you feel . . .

- Programming?
  - With the C programming language?
    - Writing a program from start to finish?
    - Reading and understanding a large program?
    - Modifying a large program?
  - With the Python programming language?
    - Writing a program from start to finish?
    - Automating data processing tasks?
    - Implementing algorithms and data structures?
  - Collaborating with a software development team?
- Using open-source software?
  - Compiling large software packages written in C?
  - Using `make`? `cmake`? GNU auto-tools? `git`?
  - Making small modifications to software?
  - Making large modifications to software?
- Using Linux?
  - Using `bash`?
  - Using Debian-based Linux?
    - Configuring a Debian-based Linux system?
    - Using APT?
  - Understanding system calls?
  - Understanding exit signals such as `SIGSEGV`?
- Using Docker?
- Using dynamic analysis tools such as fuzzers?
  - Using QUEMU?
  - Using a AFL?
    - Modifying a binary-only target to work with AFL?
    - Modifying an open-source target to work?
  - Using Libfuzzer? Honggfuzz?
  - Using CISCO-TALOS/Mutiny?
  - Using an unlisted dynamic-analysis tool?
  - Understanding run-time instrumentation?
  - Understanding compile-time instrumentation?
  - Writing your own custom purpose fuzzer?
  - Understanding differed forking?
  - Understanding persistent fuzzing?
  - Enumerating all possible program input methods?
- Recognizing a software security flaw?
  - Reading articles on new software vulnerabilities?
  - Reproducing research on software vulnerabilities?
  - Understanding DEP, ASLR, and stack canaries?
  - Overcoming these protections?
  - Exploiting control over the instruction pointer?
  - Exploiting control over `printf` arguments?
  - Exploiting a program that misuses `strcpy`, `memcpy`, or `sprintf` with a stack destination?
  - Attacking programs that misuse `system`?
  - Understanding the implications of a SUID program?
  - Exploiting with heap-metadata overwrites?
  - Finding information on the Internet?
- Using the scientific method?
- With Assembly Languages?
  - Reading Intel x86? Writing Intel x86?
  - Using different calling convention such as stdcall, fastcall, and cdecl?
- Reverse engineering?
  - Using debuggers? Using disassemblers?
  - Collaborating with a team, reversing a large target binary?

B Schedule

November 7, 2019

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00 am				Introductions	
9:00 am				AFL class	
10:00 am					
11:00 am					
12:00 noon					
1:00 pm				Skill assessment	
2:00 pm					
2:30 pm				Docker and submission standards	
3:00 pm					
4:00 pm					

November 12–15, 2019

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00 am		Introductions	Introductions	Introductions	Introductions
8:30 am					
9:00 am		Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch
9:30 am					
10:00 am					
10:30 am					
11:00 am					
11:30 am					
12:00 noon					
12:30 pm					
1:00 pm					
1:30 pm					
2:00 pm					
2:30 pm					
3:00 pm					
3:30 pm		Team synchronization	Team synchronization	Team synchronization	Team synchronization
4:00 pm					

November 18–22, 2019

	Monday	Tuesday	Wednesday	Thursday	Friday
8:00 am	Discussion • Team interview – Utility – Interaction – Success – Failure	Introductions	Introductions	Introductions	Introductions
8:30 am					
9:00 am		Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch	Sprint hours • Apply targeting strategy • Hourly survey • Lunch
9:30 am					
10:00 am	Skill Assessment • Targets prepared • Observers ready • 15-minute overview • 60-minute test				
10:30 am					
11:00 am					
11:30 am					
12:00 noon					
12:30 pm	Introductions				Discussion • Team interview – Utility – Interaction – Success – Failure
1:00 pm					
1:30 pm	Sprint hours • Apply targeting strategy • Hourly survey • Lunch				
2:00 pm					
2:30 pm					Skill assessment
3:00 pm					
3:30 pm		Team synchronization	Team synchronization	Team synchronization	
4:00 pm					

C Hourly questions

- What is your pseudonym?
- How many minutes were spent interacting with tools?
- How many minutes were spent harnessing?
- How much time was spent on research?
- Are you feeling productive?
- Are you feeling surprised?
- Are you feeling frustrated?
- Are you feeling doubtful?
- Are you feeling confused?

D End-of-day questions

- What is your pseudonym?
- I learned something today.
- I felt frustrated today.
- I worked with another team member today (team lead excluded).
- I accomplished something today.
- I feel exhausted today.
- I enjoyed my work today.
- I learned a new skill today.
- I was bored today.

E End-of-experiment questions

(1) Which vulnerability-discovery method do you feel was more effective?
(2) Which vulnerability-discovery method made you feel like you were part of a team?
(3) Which vulnerability-discovery method made the best use of your personal skill?
(4) Which vulnerability-discovery method do you think made the best use of your team's skill?
(5) Which vulnerability-discovery method did you think was easier to get started with?
(6) Which vulnerability-discovery method do you think is easier for a novice to contribute to?
(7) Did you learn any valuable skills during the experiment?
(8) Which vulnerability-discovery method did you learn more during?
(9) Which vulnerability-discovery method did you enjoy more?
(10) Which vulnerability-discovery method frustrated you the most?
(11) If you were asked to lead a vulnerability-discovery project, which method would you choose?
(12) How prepared do you think you were for the vulnerability-discovery work you were asked to do during the experiment, before initial training?
(13) How prepared do you think you were for the vulnerability-discovery work you were asked to do during the experiment, after initial training?
(14) How prepared do you think you were for the vulnerability-discovery work you were asked to do during the experiment, after the experiment?
(15) What was your interest in doing vulnerability-discovery work, before the experiment?
(16) What was your interest in doing vulnerability-discovery work, after the experiment?
(17) How many unique bugs did you find during the experiment?
    - 0
    - 1–5
    - 5–10
    - 10–20
    - 20+
(18) Which method of learning was best for you during the experiment?
    - Instructor-led training
    - Hands-on experience
    - Other
(19) Were there any external factors that affected your or your team's performance during the experiment? (For example, network outages, room temperature, experiment hours, and so on.)
(20) Do you have any thoughts or comments you would like us to consider?

F Skill assessment binaries

Collection	Binary
Cyber Grand Challenge	`Childs_Game`
	`Game_Night`
	`Casino_Games`
Rode0day (binary only)	`tcpdumpB`
	`fileB`
	`audiofileB`
Rode0day (with source)	`bzipS`
	`jqS`
	`jpegS`
OSS-Fuzz (with source)	`vorbis`
	`libarchive`
	`libxml2`
	`c-ares`
	`freetype2`
	`openssl`

References

[1] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. *Communications of the ACM*, 59(6):93–100, May 2016.

[2] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: Fuzz driver generation at scale. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 975–985, New York, New York, USA, 2019. ACM.

[3] Carl Boettiger. An introduction to Docker for reproducible research. *Operating Systems Review*, 49(1):71–79, January 2015.

[4] Norbou Buchler, Prashanth Rajivan, Laura R Marusich, Lewis Lightner, and Cleotilde Gonzalez. Sociometrics and observational assessment of teaming and leadership in a cyber security defense competition. *Computers & Security*, 73:114–136, 2018.

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[6] Gary Charness, Uri Gneezy, and Michael A Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012.

[7] Christian Cioce, Daniel Loffredo, and Nasser Salim. Program fuzzing on high performance computing resources. Technical Report SAND2019-0674, Sandia National Laboratories, Albuquerque, New Mexico, USA, January 2019. `https://www.osti.gov/servlets/purl/1492735` [Accessed January 23, 2020].

[8] Elena F. Corriero. Counterbalancing. In Mike Allen, editor, *The SAGE Encyclopedia of Communication Research Methods*, volume 1. SAGE Publications, Thousand Oaks, California, USA, 2017.

[9] Richard Draeger. Within-subjects design. In Mike Allen, editor, *The SAGE Encyclopedia of Communication Research Methods*, volume 4. SAGE Publications, Thousand Oaks, California, USA, 2017.

[10] Florian Fainelli. The OpenWrt embedded development framework, February 2008. Invited talk at the 2008 Free and Open Source Software Developers European Meeting.

[11] Ming Fang and Munawar Hafiz. Discovering buffer overflow vulnerabilities in the wild: An empirical study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, New York, New York, USA, 2014. ACM.

[12] Andrew Fasano, Tim Leek, Brendan Dolan-Gavitt, and Josh Bundt. The rode0day to less-buggy programs. *IEEE Security & Privacy*, 17(6):84–88, November 2019.

[13] Ivan Fratric. 365 days later: Finding and exploiting Safari bugs using publicly available tools, October 2018. `https://googleprojectzero.blogspot.com/2018/10/365-days-later-finding-and-exploiting.html` [Accessed March 30, 2019].

[14] GitLab. Issue boards. `https://about.gitlab.com/product/issueboard/` [Accessed December 17, 2019].

[15] Google. Fuzzer test suite. `https://github.com/google/fuzzer-test-suite` [Accessed December 18, 2019].

[16] Allen D. Householder, Garret Wassermann, Art Manion, and Chris King. The CERT(C) guide to coordinated vulnerability disclosure. `https://resources.sei.cmu.edu/asset_files/SpecialReport/2017_003_001_503340.pdf` [Accessed March 31, 2019].

[17] Schuyler W. Huck and Robert A. McLean. Using a repeated measures ANOVA to analyze the data from a pretest-posttest design: a potentially confusing task. *Psychological Bulletin*, 82(4):511–518, 1975.

[18] Robert Joyce. Come get your free NSA reverse engineering tool!, March 2019. Presentation at the 2019 RSA Conference.

[19] Brian Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1984.

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. *CoRR*, abs/1808.09700, 2018.

[21] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, October 2019. Early Access.

[22] MITRE. Cwe-125: Out-of-bounds read. `https://cwe.mitre.org/data/definitions/125.html` [Accessed February 3, 2020].

[23] MITRE. Cwe-787: Out-of-bounds write. `https://cwe.mitre.org/data/definitions/787.html` [Accessed February 3, 2020].

[24] Nadim Nachar. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in quantitative Methods for Psychology*, 4(1):13–20, 2008.

[25] Cal Newport. *Deep work : rules for focused success in a distracted world*. Grand Central Publishing, New York ; Boston, 1st ed. edition, January 2016.

[26] Vegard Nossum. Fuzzing the OpenSSH dæmon using AFL. `http://www.vegardno.net/2017/03/fuzzing-openssh-daemon-using-afl.html` [Accessed March 30, 2019].

[27] Anne Oeldorf-Hirsch. Between-subjects design. In Mike Allen, editor, *The SAGE Encyclopedia of Communication Research Methods*, volume 4. SAGE Publications, Thousand Oaks, California, USA, 2017.

[28] Hacker One. The 2019 hacker report. `https://www.hackerone.com/resources/reporting/the-2019-hacker-report` [Accessed December 4, 2019].

[29] OpenWrt. ubus (OpenWrt micro bus architecture). `https://openwrt.org/docs/techref/ubus` [Accessed Januarary 22, 2020].

[30] Athanasios Papoulis and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. Tata McGraw-Hill Education, New York, New York, 2 edition, 2002.

[31] Reginald E. Sawilla and Xinming Ou. Identifying critical attack assets in dependency attack graphs. In Sushil Jajodia and Javier López, editors, *13th European Symposium on Research in Computer Security*, volume 5283 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2008.

[32] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software, August 2017. Invited talk at the 26th USENIX Security Symposium.

[33] Michael R Sheldon, Michael J Fillyaw, and W Douglas Thompson. The use and interpretation of the friedman test in the analysis of ordinal-scale data in repeated measures designs. *Physiotherapy Research International*, 1(4):221–228, 1996.

[34] R. Shirrey. RFC 4949: Internet security glossary, version 2. `https://tools.ietf.org/rfc/rfc4949.txt` [Accessed March 31, 2019], August 2007.

[35] Paul Spooren. Running OpenWrt inside Docker. `https://forum.openwrt.org/t/running-openwrt-inside-docker-sbin-init-stuck/13774/8` [Accessed December 17, 2019].

[36] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, volume 16, pages 1–16, 2016.

[37] Robert Swiecki. Honggfuzz. `http://honggfuzz.com` [Accessed December 18, 2019].

[38] trailofbits. Challenge sets. `https://www.trailofbits.com/research-and-development/challenge-sets/` [Accessed December 17, 2019].

[39] Sai Vamsi, Venkata Balamurali, K Surya Teja, and Praveen Mallela. Classifying difficulty levels of programming questions on HackerRank. In *International Conference on E-Business and Telecommunications*, volume 3, pages 301–308. Springer, 2019.

[40] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy*, pages 374–391, May 2018.

[41] R. F. Woolson. *Wilcoxon Signed-Rank Test*, pages 1–3. American Cancer Society, 2008.

[42] Michal Zalewski. American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl/` [Accessed March 30, 2019].