# Git-based CTF: A Simple and Effective Approach to Organizing In-Course Attack-and-Defense Security Competition

SeongIl Wi
*KAIST*

Jaeseung Choi
*KAIST*

Sang Kil Cha
*KAIST*

## Abstract

Security competitions, a.k.a., CTFs, have never been easy to run for a classroom teacher despite there being considerable body of research on these events. It is often frustrating for teachers to organize and administer such an event as doing so requires significant time and human resource investments. Creating new problems for every CTF is challenging as there are many factors to consider while developing a CTF problem such as the difficulty level of each challenge. In this paper, we propose a simple, but effective approach that we refer to as Git-based CTF to hosting an in-class attack-and-defense CTF contest while minimizing the operational costs for teachers. We share our experience and lessons learned by organizing a Git-based CTF in KAIST.

## 1 Introduction

Capture The Flag (CTF), a competition that provides a legal way for the participants to demonstrate hands-on hacking skills, is gaining its momentum as a pedagogical tool in security education due to its substantial learning outcomes [7, 14]. Since 2013, PicoCTF [11] has been encouraging high school students to study computer systems in an adversarial perspective. The University of Maryland recently ran a MOOC (Massive Online Open Courseware) course [31], which aims to teach students how to build a secure software system [25].

Unfortunately, however, organizing an in-course CTF competition has *never* been an easy task for teachers due to the following practical challenges.

C1 (Interactivity Challenge): To be realistic and educationally effective, CTF events should involve continuous interaction between attackers and defenders. However, many public CTF frameworks do not support interactivity in their designs.

C2 (Configuration Challenge): Configuring a CTF environment requires a significant amount of time and considerable skills. Especially, less experienced educators such as high school teachers can suffer from this issue.

C3 (Monitoring Challenge): There should be someone, potentially a teaching assistant, who can continually monitor and administer the CTF while it is running. Typically, a CTF lasts for days, and administering the entire event requires substantial human resources.

C4 (Contents Creation Challenge): Teachers need to invent new problems every time they run a competition as it can be trivial for students to reuse the exploitation scripts or flags used in previous competitions. Note there are typically ten to fifty problems in a CTF depending on the size of the competition. Furthermore, teachers should consider the difficulty level as well as the diversity of each problem when they create a new one.

Although there is surging research interest in tackling the above mentioned challenges, we remain unaware of any practical solution that meets all of the requirements. Table 1 presents a comparison between our approach and other existing CTF platforms. For instance, there are several CTF frameworks [5, 13, 23, 29] that aim to ease the configuration challenge (C2), but none of the works addressed neither C3 nor C4. AutoCTF [17] specifically tackles C4 by employing an automatic bug injection system called LAVA [16]. However, their approach does not consider the interactivity challenge (C1), and the vulnerability pattern is fixed across problems.

Notably, BIBIFI [25] mitigates the contents creation challenge (C4) by employing a so-called "build-it" phase during a competition. It divides participants into two types: (1) build-it players; who write an application given a specification written by the teacher, and (2) break-it players; who try to attack the newly created applications. Therefore, the teacher's load is reduced from creating dozens of CTF problems to simply writing a specification for an application. However, their design choice does not allow students to engage in real-time attack and defense exercises (C1). Furthermore, as noted by the authors, the teacher should manually check every submitted fix to determine whether or not it addresses a single defect at the end of the competition. That is, BIBIFI is also associated with a monitoring challenge (C3).

In this paper, we propose a simple and powerful pedagogical framework that we call Git-based CTF with which to organize an in-course attack-and-defense CTF competition. Our approach is mainly inspired by the design of BIBIFI, but it enables regular teachers easily to host an attack-and-defense CTF as class homework by minimizing the configuration and monitoring burden. Furthermore, the content creation in Git-based CTF is at least as easy as it is with BIBIFI. That is, Git-based CTF tackles each of the aforementioned challenges (C1–C4).

First, students of Git-based CTF play the role of both attacker and defender unlike in BIBIFI. Therefore, attackers and defenders can interact with each other in real time as in typical attack-and-defense CTFs. Given that real-time attack and defense exercises can be overwhelming to students who have less experience in security [4, 24], we devise a means of providing direct guidance to novices for a subset of the problems in Git-based CTF (§3.2).

Second, setting up a competition environment in Git-based CTF is inexpensive and fully distributed. Students use Docker containers to run the service applications of their opponents and to develop an exploit. Although they work on their own machines, they can communicate with each other through Git and GitHub [2]. Specifically, an attack in Git-based CTF involves submitting an issue to the target team's repository. The content of the issue is encrypted and only visible to the target team and the instructor(s). Since the entire attack history is stored in GitHub, the instructors can later fetch the history and verify the submitted attacks on their own machines.

Third, Git-based CTF significantly reduces the monitoring effort of the instructors. Unlike BIBIFI, we ask every student to inject vulnerabilities into a network application they prepared. By having the injection phase, we can easily check which of the injected vulnerabilities can be exploited by an attack: we can simply run the attack against both a patched and a vulnerable version of the target application. As a result, teachers can run the CTF for weeks as in a typical class homework assignment without monitoring the CTF servers.

Finally, Git-based CTF shifts the contents creation burden to students as in BIBIFI [25]. However, students in Git-based CTF are naturally motivated to check the quality of the fixes of other teams in order to break them again, as each student experiences the role of both an attacker and a defender. We also note that injected vulnerabilities from students typically reflect the skills and experiences of each of them. Therefore, the difficulty levels of CTF challenges are likely to be well-distributed (§4).

Table 1: Comparison between Git-based CTF and other existing CTF frameworks.

| **Framework Name** | **C1** | **C2** | **C3** | **C4** |
|---|---|---|---|---|
| **Git-based CTF** (ours) | ✓ | ✓ | ✓ | ✓ |
| BIBIFI [25] | ✗ | ✓ | ✗ | ✓ |
| iCTF [30, 32] | ✓ | ✓ | ✗ | ✗ |
| NIZKCTF [20] | ✗ | ✗ | ✓ | ✗ |
| PicoCTF [8, 11] | ✗ | ✗ | ✗ | ✓ |
| InCTF [24] | ✓ | ✗ | ✗ | ✗ |
| CTFd [13] | ✗ | ✓ | ✗ | ✗ |
| SecGen [27] | ✗ | ✓ | ✗ | ✓ |
| Catalyst [29] | ✓ | ✓ | ✗ | ✗ |
| Backman [4] | ✓ | ✓ | ✗ | ✗ |
| CCTF [21, 22] | ✗ | ✗ | ✗ | ✗ |
| CyTrOne [5, 23] | ✗ | ✓ | ✗ | ✗ |
| AutoCTF [17] | ✗ | ✗ | ✗ | ✓ |
| VM-based Framework [12] | ✗ | ✓ | ✓ | ✗ |
| KYPO [34] | ✗ | ✓ | ✗ | ✗ |

We ran a preliminary in-course CTF with the proposed idea as part of a course activity during the Spring 2018 semester at KAIST in Korea. We ran the CTF for about two weeks, and found that the administrative cost for running the event is significantly lower than that associated with a classic attack-and-defense CTF that we ran in 2017 for the same course. We observed various kinds of vulnerabilities with a range of difficulty levels introduced by students. The students discovered 14 unintended vulnerabilities during the competition. Such unintended vulnerabilities provided the motivation for highly experienced students to become more engaged in the CTF.

Overall, this paper makes the following contributions.

1. We summarize the practical challenges in hosting in-course attack-and-defense CTFs.

2. We present a novel way to organize an attack-and-defense CTF as a class homework or an activity.

3. We discuss several lessons learned from organizing a Git-based CTF.

4. We make the source code of our CTF framework public [28], which includes a series of scripts to organize and play Git-based CTF.

## 2 Background

CTF is a competition that involves capturing a flag as a proof of solving challenges. There are mainly two categories: (1) jeopardy style and (2) attack-and-defense style. Jeopardy-style CTFs consist of a set of problems to solve. For each problem, there is a specifically configured server that stores a flag, which is a secret string used by participants as a proof of successfully attacking the corresponding server. Jeopardy-style CTFs typically do not involve any defensive exercises: they are mostly attack-only. On the other hand, in an attack-and-defense CTF, each participant manages a server that runs a set of service applications. The participants should find vulnerabilities in the applications in order to obtain flags from the other servers, and should patch their own applications to protect their flags. Since every team should be connected to each other, it is more difficult to host an attack-and-defense CTF than a jeopardy-style one. CTFs provide participants a chance to exercise hands-on security skills, and its pedagogical value is gaining more attention recently [9, 10, 11, 14, 15, 30].

Table 1 presents the comparison between existing CTF platforms with regards to the challenges we addressed in §1. Note that CTF platforms that focus on C1 tend to miss C3, which shows the difficulty of administering attack-and-defense CTFs over a long period as in a regular class homework or project. This paper presents a novel way to host an attack-and-defense-style CTF as a course activity while not suffering from such challenges.

GitHub [2] has become an essential platform for managing open-source software. Zagalsky *et al.* [36] show GitHub's collaborative features such as issue tracker and pull requests can benefit software education. There have been several attempts in security education that leverage GitHub in a CTF competition [20, 25]. Git-based CTF follows the similar approach, but we use PGP encryption [19] to securely store exploits.

## 3 Git-based CTF

In this section, we describe the design of Git-based CTF. Git-based CTF consists of three phases: (1) preparation, (2) injection and (3) exercise. Figure 1 illustrates the overall flow of Git-based CTF. First, each team should prepare a network service application (§3.1). Next, players inject vulnerabilities into their team's service (§3.2). Lastly, players analyze other teams' services and exploit them in the exercise phase (§3.3). In each phase, we adequately utilize Git [1] and GitHub [2] to fulfill the goal of minimizing instructors' manual effort.

**Configuration.** To setup a Git-based CTF, instructors first need to create and assign one GitHub repository for each team. The instructors then create a PGP key pair [19] and securely share the private key among them. Each team maintains their service application in the assigned repository. This repository also serves as an interface in which attacks are submitted. Each team should create their own PGP key pair. All the public keys along with the instructor's public key will be shared across the teams. The instructors can run a script to fetch the submitted attacks and evaluate them automatically (§3.3). Note that instructors in Git-based CTF do not have to prepare a separate server to run a CTF.

### 3.1 Preparation Phase

In this step, each team is required to prepare a network service application that binds to a port number specified by the instructor. When the application is running, a client should be able to connect to the service through a network. Each team prepares their own application in one of the following two scenarios.

**Hands-on Development.** In this scenario, we follow the same strategy as in the build-it phase of BIBIFI [25]. We first provide a specification about the service, and each team develops their own service according to the specification. We also prepare a set of test cases that the service application should pass in order to prove its functionality. With this approach, students have a chance to exercise secure coding practices.

**Importing Open-source Software.** Instead of making every team develop the same kind of application, instructors can give freedom to each team to import an arbitrary open-source software of their choice. Note that teams can take a non-network application and convert it to a network service if they want. Also, the instructors may encourage students to investigate the imported source code and check whether their service application is really safe to run. In this scenario, students can learn source code auditing methods and skills.

When a service application is ready, each team pushes their code to their own GitHub repository. In this phase, each repository is private and only visible to the team members and the instructors. By the end of this phase, the repository should contain a `Dockerfile` that will automatically install packages required by the application as well as a `Makefile` that automatically builds the application. The resulting binary should successfully run within a Docker container configured with the provided `Dockerfile`. To support easier development, we provide a template `Dockerfile`, which is built upon a Debian base image. It takes a `flag.txt` file as input, which

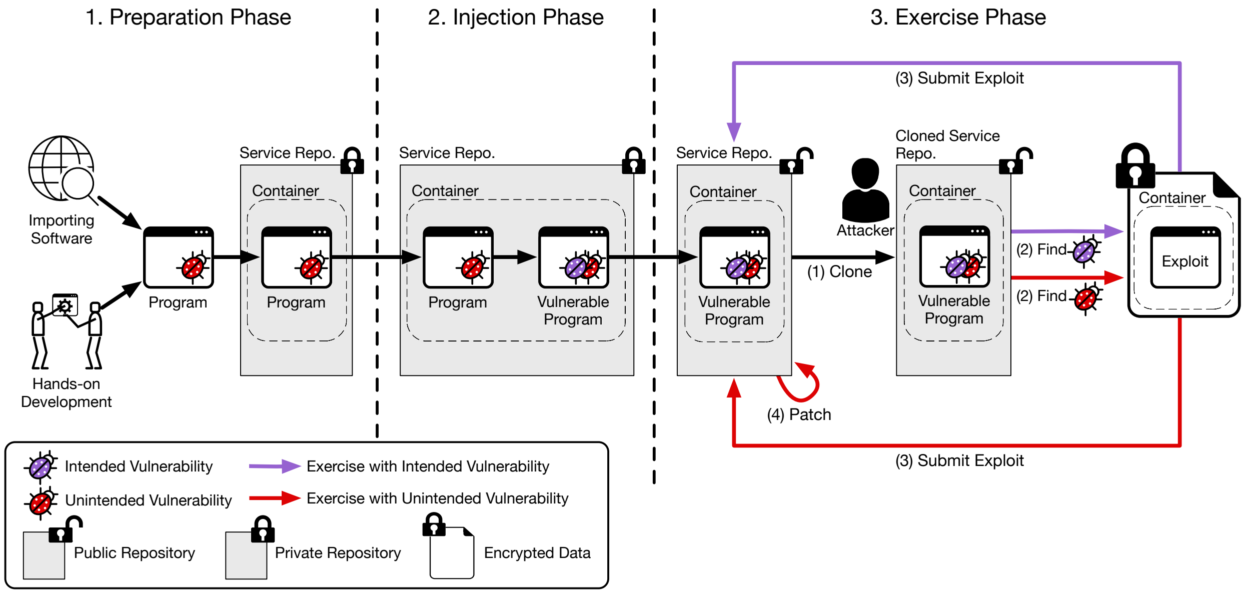

Figure 1: Git-based CTF workflow.

contains a random string generated by the instructors, and copies the flag file into `/var/ctf/` directory inside the container. Since the instructors may want to confirm that each repository successfully builds the service application, we provide a simple script [28] that can automatically check the health of each repository by running a series of commands such as `git clone`, `docker build` and `make`.

We note that the applications prepared in this phase may contain unintended vulnerabilities. Such vulnerabilities can exist regardless of whether the students developed the services by themselves, or they imported an open-source software. Although open-source software is likely to have less vulnerabilities than the software developed by the students within a short period, they can still be vulnerable to zero-day attacks. In the rest of the paper, we will refer to such vulnerabilities as *unintended vulnerabilities*.

## 3.2 Injection Phase

Now that all the teams have prepared an application in their own repository, they inject $N$ distinct vulnerabilities into their service application[1]. Here, $N$ is chosen by the instructor depending on the classroom circumstances, e.g., based on the number of students. Each team creates $N$ separate branches in their service repository, and introduces a vulnerability per each branch. We will refer to these injected vulnerabilities as *intended vulnerabilities*, to distinguish them from *unintended vulnerabilities* that may exist even before the injection.

After injecting vulnerabilities into the application, each team should create an exploit for each of the vulnerabilities in order to prove their exploitability. An exploit in Git-based CTF is a program that runs in a Docker container, which takes in a target IP address as input, and prints out the content of the flag stored in the service Docker container to the standard output assuming that the target service is running at the given IP address. An exploit as well as the corresponding `Dockerfile` is signed and encrypted with PGP keys [19] before they get pushed to the repository. To make the exploit content *only* visible to the instructors, it should be encrypted with the public key of the instructors. Note that exploits created in this phase can be considered as a solution for each intended vulnerability, and other team members should not be able to access them when the repository becomes public in the next phase.

**Automated Exploit Verification.** At this point, we can check the exploitability of each intended vulnerability by simply running the corresponding exploit against the service application. We first launch the vulnerable application within a Docker container with a randomly generated `flag.txt` file. We then run another Docker container that runs the corresponding exploit, and verify whether it returns a valid flag string, i.e., the random string stored in the flag file. If and only if the exploit is successful, it will show the valid flag string. The same

[1]For simplicity, we assume here that a single vulnerability is enough to hijack the control flow of the target service. In reality, however, students may inject a series of related vulnerabilities altogether. For instance, an exploit may involve a memory leak vulnerability to bypass ASLR, as well as a buffer overflow vulnerability to initiate a ROP.

automated technique is used to evaluate submitted attacks (§3.3).

**Contents Creation.** In the injection phase, instructors obtain vulnerable services that can be used as a CTF problem for free. Furthermore, attacks against such problems can be automatically checked and evaluated. One potential concern, however, is whether we can obtain a set of challenges that are diverse and interesting enough. According to our experimental study (§4), we found that students tend to inject vulnerabilities based on their skill level. As a result, CTF challenges produced by students were diverse in terms of both vulnerability kinds and difficulty levels.

**Low Barriers to Entry.** Since commits in each branch show what changes had made to the original program, it will serve as a useful hint for CTF beginners in the next phase, where players have to analyze and exploit vulnerabilities of the applications. In Git-based CTF, even an unexperienced student can easily spot a vulnerable point of a target program by simply invoking `diff`. We note that this is one of the key aspect of Git-based CTF because we assume that students can have various different backgrounds, and an attack-and-defense CTF should be playable for them too.

## 3.3 Exercise Phase

The primary goal of the final phase is to exercise attacks and defenses. Each team analyzes other teams' service applications, finds vulnerabilities, and exploits them. In this phase, we make all the service repositories public so that the participants can access the source code. There are two ways to play the competition: (1) analyzing the commits made in the injection phase and figuring out the intended vulnerabilities, or (2) seeking for unintended vulnerabilities in the service. As we discussed earlier, CTF novices can choose to follow the first way by consulting the commits made for each vulnerable branch to spot vulnerable points.

When players identify a vulnerability from one of the services run by other teams, they should write an exploit for it. Recall from §3.2, an exploit in Git-based CTF is a program that runs in a Docker container that can connect to a victim service running in another Docker container. An exploit should be able to retrieve a flag stored in the container. For exploits in this phase, every player should encrypt their exploits with public keys of both the instructors and the team that maintains the vulnerable service. Players will create GitHub *issues* in target application's repository to submit their encrypted exploits. We provide each student a simple command-line tool that helps create an issue in a target repository. We also provide instructors a command-line tool for fetching the submitted issues and verifying their exploitability as in the injection phase.

**Defense in Git-based CTF.** To defend against attacks from others, each team can fix *unintended* vulnerabilities in their own service application. Since intended vulnerabilities already have a fix, i.e., the original code, the defense in Git-based CTF should always be about unintended vulnerabilities, but not about intended ones. Whenever an unintended vulnerability is found in a service, the team that owns the application can fix it and push the modification to the master branch of the repository. Every patch can be monitored by other teams because the repository is public. This means if a patch is wrong or incomplete, other teams can attack the same unintended vulnerability again. We also periodically award points to participants who have successfully exploited an unintended vulnerability until it is fixed by the defending team. Therefore, every participant should monitor patches unlike BIBIFI [25] where each fix should be verified by the instructor.

**Automated Scoring System.** Our design of Git-based CTF enables automated scoring. Suppose students have injected $k$ intended vulnerabilities $v_1, v_2, \cdots, v_k$ into a program $p$, and let $p_i$ be a modified version of the program, which has only one intended vulnerability $v_i$. To evaluate an attack against intended vulnerabilities, we run the attack against $p_1, p_2, \cdots, p_k$ as well as the original program $p$. We then observe in which version the attack returns a valid flag. If the exploit works only on one of the modified programs, we can immediately identify which vulnerability has been attacked by the exploit. If the exploit works only on the original program, we consider it as an attack against an unintended vulnerability. Whenever there is an attack for an unintended vulnerability, we deduct points of the victim for every $M$ minutes, where $M$ is a configurable parameter. This is to encourage students to patch their programs. In order to preserve the interactivity between attackers and defenders, we do not differentiate between two distinct unintended vulnerabilities of the same application unless one of the vulnerabilities is fixed in the master branch. For every attack, i.e., a GitHub issue, we always fetch the latest version committed before the attack, and consider the version of the program as $p$. This way, participants can experience real-time interactions as both an attacker and a defender as in typical attack-and-defense CTFs.

We note that Git-based CTF has the characteristics of both jeopardy and attack-and-defense style: it is similar to jeopardy-style CTFs in terms of the attack-only nature of intended vulnerabilities, but it is also an attack-and-defense CTF with respect to unintended vulnerabilities.

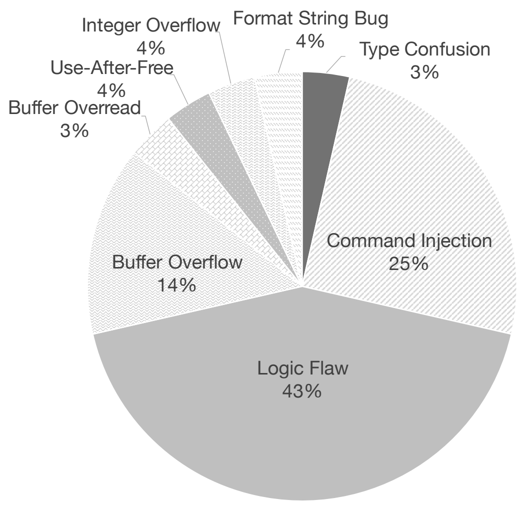

Figure 2: The distribution of different kinds of injected vulnerabilities.

**Scoreboard.** Git-based CTF is a fully distributed CTF framework, which does not have a dedicated web server for displaying scores. Instead, students as well as instructors can run our script [28] to see the current scoreboard. The script fetches the attack log and automatically populates an HTML file that shows a graph representing score over time for each team as in typical CTF events.

## 4 Evaluation

We ran a Git-based CTF in 2018 as part of a graduate-level course in KAIST, Korea [18]. Most students of the course were information security major who have various different backgrounds including cryptography and mathematics. Among 21 students, 11 of them had no experience in a security competition. We first divided them into 6 teams and asked them to develop a simple secure messaging application in about three weeks. We gave a precise specification that describes the behavior of the messaging application, and we forced them to use either C or C++ in order to increase the chance of having more interesting vulnerabilities. After the first phase, we asked each student to inject at least one vulnerability into their own application. In total, the students introduced 28 vulnerabilities in the 6 distinct applications developed in the previous phase. Although it was a preliminary event, we obtained several meaningful lessons and results from our postmortem analysis. We enumerate them in the rest of this section.

**Diversity of Injected Vulnerabilities.** We observed various types of injected vulnerabilities including logic errors and classic memory corruption errors. The most common one was a logic error that involves an incorrect program logic. For example, about a half of the logic errors were due to protocol design flaws. We also saw a variety of memory corruption vulnerabilities such as buffer overflow, buffer overread, use-after-free as well as type-confusion vulnerabilities. Figure 2 summarizes the result. The result shows that Git-based CTF helps the instructors prepare a diverse set of CTF challenges without the burden of creating contents.

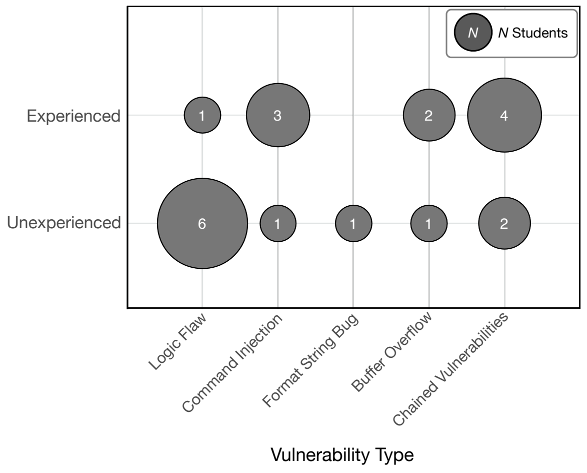

Figure 3: The number of students by vulnerability type and level of experience.

**Difficulty Levels.** We further analyzed each of the injected vulnerabilities, and found that there is a meaningful correlation between the experience level of students and the difficulty level of injected vulnerabilities. That is, students tend to inject vulnerabilities based on their skill level. Figure 3 presents such a correlation. For simplicity, if a student had participated in at least one CTF event, we considered that the student is "experienced". Note that unexperienced students tend to focus on logic errors because they are not comfortable about memory exploitation techniques. Indeed all the cryptographers in the course injected logic errors. However, students who have advanced knowledge about hacking techniques injected more complex vulnerabilities. Four of them injected multiple vulnerabilities, e.g., memory leaks as well as corruption, that should be chained altogether in order to successfully bypass defenses, e.g., ASLR and DEP, and spawn a remote shell. This naturally leads us to have a set of CTF challenges of moderately distributed difficulty levels for the students.

**Unintended Vulnerabilities.** The students participated in Git-based CTF found unintended vulnerabilities as well as intended ones. In total, 14 vulnerabilities and

18 functionality bugs were reported during the activity. It turns out that each team had at least one unintended vulnerability. The students fixed 12 bugs in total, and in took about 10 hours on average to fix a bug. The longest time took for fixing a bug was 24 hours. We note that unintended vulnerabilities are found mostly by students who have a strong background on security and hacking. Therefore, we believe it is important to have mixed groups with different levels of experience. Since Git-based CTF provides an obvious hint for each intended challenges, less experienced students in a team may work on them while more experienced students try to find unintended vulnerabilities.

## 5 Discussion

**Diverse Challenges.** Recall from §3.1, teams in Git-based CTF can prepare their applications either by developing their own, or by importing an existing project. In our preliminary study, we only performed the former. We believe by employing the second approach, we may observe more various challenges for a competition. For example, each team may have different kinds of applications, and if some teams prepared a web application, we could have observed web-based attacks such as SQL injection, XSS, and CSRF attacks.

**Unintended Scoring.** Recall from §3.3, Git-based CTF can automatically identify which intended vulnerability is exploited by an attack. However, suppose an attacker who found an *unintended* vulnerability crafts an exploit that can identify the version of the target program, and outputs a flag only when a specific version of the program is detected. For example, suppose a program $p$ has one intended vulnerability $v_1$, and an attacker crafted an exploit that prints out a flag only when the target program is $p_1$. In this case, the attack is considered as a valid exploit for $v_1$, and our system will give a point to the attacker even though she did not exploit $v_1$.

We believe such unintended scoring is not a concern for two reasons. First, exploiting unintended vulnerabilities is more difficult than exploiting intended ones. Therefore, unintended scoring may be considered as extra points for the participants who managed to exploit harder challenges. Second, in Git-based CTF, participants can have only a fixed amount of score for intended vulnerabilities, whereas they can get unlimited score for unintended ones unless they are fixed. Therefore, the amount of score they can obtain from unintended vulnerabilities is much larger than the one from intended vulnerabilities anyways.

**Cheating.** Collusion is possible in Git-based CTF as students may want to share their exploit code. However, instructors in Git-based CTF can always download all the exploit code and run a traditional plagiarism detection tool such as MOSS [26]. This is different from traditional CTFs where the organizers do not possess any exploit code. We leave it as future work to combine existing anti-cheating CTF solutions such as [12].

## 6 Related Work

There have been numerous attempts to improve CTF-based education. This section summarize some of them.

**Addressing Administration Challenge.** CTFd [13] lowers the cost of configuring and running a CTF by providing a ready-made CTF web page, which includes a convenient and customizable administrative panel for organizers as well as a graphical scoreboard. Although CTFd reduces the administration costs of running a CTF, it only supports a jeopardy style CTF. The iCTF framework [32] provides a customizable attack-and-defense-style CTF framework. Each team in iCTF has its own Virtual Machine (VM), which must be set up by itself, and the VMs are connected to each other to form a virtual private network. However, configuring and managing the network environment requires significant human effort as well as HW resource. Raj *et al.* [24] propose a container-based CTF framework, called InCTF, to reduce the HW burden. Theoretically, InCTF allows an organizer to run a CTF event with a single server without having a complex network setup. The ShellWePlayAGame (SWPAG) framework [30] further reduces the configuration costs by providing an easy-to-use website. With the SWPAG, one can easily create an environment for a CTF competition in a few clicks. However, because of its interactive characteristics, teachers need to continuously monitor and care about whether the competition is running normally. We note that Git-based CTF inherently does not require any environmental setup and monitoring effort, and it utilizes free-of-charge services such as GitHub [2] and BitBucket [3].

**Security of CTF Platforms.** Since CTF platforms themselves can be vulnerable, several studies have attempted to mitigate this concern. In BIBIFI [25], the authors wrote their web application in Haskell in order to prevent potential vulnerabilities such as memory corruption, XSS, CSRF, and SQL injection with the help of strong type system of Haskell. NIZKCTF [20] removes the flags from the scoring server in order to handle this challenge. Specifically, teams in NIZKCTF do *not* submit a flag, but a non-interactive zero-knowledge proof [6] to the organizer. Each team has its own unique proof that is public to anyone. Therefore, it is impossible for team $A$ to impersonate team $B$ to submit a flag on behalf of $B$ to harm another team $C$. The idea is specifically useful in attack-and-defense-style CTFs as attacked teams can have points deducted. We note that Git-based CTF also supports the same level of security against the impersonation attack with PGP [19]. On the other hand, NIZKCTF suffers from the contents creation challenge as they do not use Git repositories for building and maintaining software. Our approach can naturally help students learn open-source software development process as well as secure coding practice.

**Security of Flags.** Burket *et al.* [8] present an Automatic Program Generator (APG) where CTF platforms can automatically generate problem instances for a given problem. This way, participants who observe different problem instances cannot share their flags. Similarly, Chothia *et al.* [12] propose an offline-style CTF competition where each participant runs a VM locally on their own machine. Each VM has a unique flag, thus it is unlikely for participants to share their flags. Although this is a powerful mitigation against collusion, their approach does not support hosting attack-and-defense-style CTFs. Both the ideas are complementary to our approach, and they can be adapted to Git-based CTF to prevent collusion between students.

**Entry Barrier.** There are several approaches considering the entry barrier of CTF competitions. The PicoCTF [11] is a CTF platform that employs a story-based game making it easier for pre-collegiate students to get interested in computer security. The KYPO cyber range [33, 34, 35] is also designed to help encourage novices by dividing a CTF challenge into a sequence of small phases of different difficulty levels. Burns *et al.* [9] collect a set of challenges from 160 CTF competitions, and summarizes what kind of security knowledge is required for beginners to solve the challenges. Mirkovic *et al.* [22] discuss several lessons learned by organizing CCTF which has participants from various different backgrounds. Backman *et al.* [4] present a small-scale attack-and-defense CTF for undergraduate students. Specifically, it keeps the number of teams to be as small as possible while making the duration of each CTF round to be one hour. By giving enough time for each round, it tries to lower the bar for beginners.

**Content Creation Challenge.** There are several works on automatically generating CTF challenges and scenarios. SecGen [27] is a framework that can automatically build a security scenario by randomly selecting sequence of modules, which include CTF challenges, and insecure OS, and so forth. Each module in the scenario corresponds to a sole CTF problem. However, each participant can have a unique experience by combining different problems together. Hulin *et al.* present AutoCTF, which uses automatically generated vulnerable programs as CTF challenges. The authors extend the automatic bug injection tool called LAVA, and use it to inject exploitable bugs into a program. However, the types and patterns of the injected vulnerabilities are fixed.

## 7 Conclusion

In this paper, we present Git-based CTF, a novel attack-and-defense CTF platform that can be easily hosted as an in-course activity. We addressed four major challenges that current CTF platforms have, and showed that Git-based CTF can effectively handle all the addressed challenges. The source code for running Git-based CTF is publicly available at GitHub [28].

## 8 Acknowledgements

We thank GitHub for providing unlimited free plan for organizing classes. We also thank HyungSeok Han and anonymous reviewers for their constructive feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.B0717-16-0109, Building a Platform for Automated Reverse Engineering and Vulnerability Detection with Binary Code Analysis).

## References

[1] Git. `https://git-scm.com/`.

[2] GitHub. `https://github.com`.

[3] Atlassian. Bitbucket. `https://bitbucket.org/`.

[4] Backman, N. Facilitating a battle between hackers: Computer security outside of the classroom. In *Proceedings of the ACM Technical Symposium on Computing Science Education* (2016), pp. 603–608.

[5] Beuran, R., Pham, C., Tang, D., ichi Chinen, K., Tan, Y., and Shinoda, Y. CyTrONE: An integrated cybersecurity training framework. In *Proceedings of the International Conference on Information Systems Security and Privacy* (2017), pp. 157–166.

[6] Blum, M., Feldman, P., and Micali, S. Non-interactive zero-knowledge and its applications. In *Proceedings of the Annual ACM Symposium on Theory of Computing* (1988), pp. 103–112.

[7] Bratus, S. What hackers learn that the rest of us don't: Notes on hacker curriculum. *IEEE Security Privacy 5*, 4 (2007), 72–75.

[8] Burket, J., Chapman, P., Becker, T., Ganas, C., and Brumley, D. Automatic problem generation for capture-the-flag competitions. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education* (2015).

[9] Burns, T. J., Rios, S. C., Jordan, T. K., Gu, Q., and Underwood, T. Analysis and exercises for engaging beginners in online CTF competitions for security education. In *Proceedings of the USENIX Workshop on Advances in Security Education* (2017).

[10] Carlisle, M., Chiaramonte, M., and Caswell, D. Using ctfs for an undergraduate cyber education. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education* (2015).

[11] Chapman, P., Burket, J., and Brumley, D. PicoCTF: A game-based computer security competition for high school students. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education* (2014).

[12] Chothia, T., and Novakovic, C. An offline capture the flag-style virtual machine and an assessment of its value for cybersecurity education. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education* (2015).

[13] Chung, K. Lowering the barriers to capture the flag administration and participation. In *Proceedings of the USENIX Workshop on Advances in Security Education* (2017).

[14] Conti, G., Babbitt, T., and Nelson, J. Hacking competitions and their untapped potential for security education. *IEEE Security Privacy 9*, 3 (2011), 56–59.

[15] Davis, A., Leek, T., Zhivich, M., Gwinnup, K., and Leonard, W. The fun and future of CTF. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education* (2014).

[16] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., and Whelan, R. LAVA: Large-scale automated vulnerability addition. In *Proceedings of the IEEE Symposium on Security and Privacy* (2016), pp. 110–121.

[17] Hulin, P., Davis, A., Sridhar, R., Fasano, A., Gallagher, C., Sedlacek, A., Leek, T., and Dolan-Gavitt, B. AutoCTF: Creating diverse pwnables via automated bug injection. In *Proceedings of the USENIX Workshop on Offensive Technologies* (2017).

[18] KAIST. IS521: Information security laboratory. `https://kaist-is521.github.io/`.

[19] Lucas, M. W. *PGP & GPG: Email for the Practical Paranoid*. No Starch Press, 2006.

[20] Matias, P., Barbosa, P., Cardoso, T., Mariano, D., and Aranha, D. NIZKCTF: A non-interactive zero-knowledge capture the flag platform. *CoRR abs/1708.05844* (2017).

[21] Mirkovic, J., and Peterson, P. A. H. Class capture-the-flag exercises. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education* (2014).

[22] Mirkovic, J., Tabor, A., Woo, S., and Pusey, P. Engaging novices in cybersecurity competitions: A vision and lessons learned at ACM Tapia 2015. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education* (2015).

[23] Pham, C., Tang, D., Chinen, K.-i., and Beuran, R. CyRIS: A cyber range instantiation system for facilitating security training. In *Proceedings of the Symposium on Information and Communication Technology* (2016), pp. 251–258.

[24] Raj, A. S., Alangot, B., Prabhu, S., and Achuthan, K. Scalable and lightweight CTF infrastructures using application containers. In *Proceedings of the USENIX Workshop on Advances in Security Education* (2016).

[25] Ruef, A., Hicks, M., Parker, J., Levin, D., Mazurek, M. L., and Mardziel, P. Build it, break it, fix it: Contesting secure development. In *Proceedings of the ACM Conference on Computer and Communications Security* (2016), pp. 690–703.

[26] Schleimer, S., Wilkerson, D. S., and Aiken, A. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2003), pp. 76–85.

[27] Schreuders, Z. C., Shaw, T., Shan-A-Khuda, M., Ravichandran, G., Keighley, J., and Ordean, M. Security scenario generator (secgen): A framework for generating randomly vulnerable rich-scenario VMs for learning computer security and hosting CTF events. In *Proceedings of the USENIX Workshop on Advances in Security Education* (2017).

[28] SoftSec KAIST. Git-based CTF. `https://github.com/SoftSec-KAIST/GitCTF`.

[29] Taylor, C., Arias, P., Klopchic, J., Matarazzo, C., and Dube, E. CTF: State-of-the-art and building the next generation. In *Proceedings of the USENIX Workshop on Advances in Security Education* (2017).

[30] Trickel, E., Disperati, F., Gustafson, E., Kalantari, F., Mabey, M., Tiwari, N., Safaei, Y., Doupé, A., and Vigna, G. Shell we play a game? CTF-as-a-service for security education. In *Proceedings of the USENIX Workshop on Advances in Security Education* (2017).

[31] University of Maryland. Cybersecurity specialization. `https://www.coursera.org/specializations/cyber-security`.

[32] Vigna, G., Borgolte, K., Corbetta, J., Doupé, A., Fratantonio, Y., Invernizzi, L., Kirat, D., and Shoshitaishvili, Y. Ten years of iCTF: The good, the bad, and the ugly. In *Proceedings of the USENIX Summit on Gaming, Games, and Gamification in Security Education* (2014).

[33] Vykopal, J., and Barták, M. On the design of security games: From frustrating to engaging learning. In *Proceedings of the USENIX Workshop on Advances in Security Education* (2016).

[34] Vykopal, J., Ošlejšek, R., Čeleda, P., Vizvary, M., and Tovarňák, D. KYPO cyber range: Design and use cases. In *Proceedings of the International Conference on Software Technologies* (2017).

[35] Vykopal, J., Vizváry, M., Oslejsek, R., Celeda, P., and Tovarnak, D. Lessons learned from complex hands-on defence exercises in a cyber range. In *Proceedings of the IEEE Frontiers in Education Conference* (2017).

[36] Zagalsky, A., Feliciano, J., Storey, M.-A., Zhao, Y., and Wang, W. The emergence of github as a collaborative platform for education. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work & Social Computing* (2015), pp. 1906–1917.