Ilsun You (Ed.)

LNCS 12583

Information Security Applications

21st International Conference, WISA 2020
Jeju Island, South Korea, August 26–28, 2020
Revised Selected Papers

Springer

Toward a Fine-Grained Evaluation of the Pwnable CTF

Sung-Kyung Kim, Eun-Tae Jang, and Ki-Woong Park(✉)

Department of Information Security, Sejong University, Seoul, South Korea
jotun9935@gmail.com, euntaejang@gmail.com, woongbak@sejong.ac.kr

Abstract. In the untacted era of the recent COVID-19 virus outbreak, the pedagogic value of Capture the Flag (CTF) has grown even more as an effective means for students to learn knowledge about the overall computer system and information security through active participation without facing the teacher. However, in the process of successfully introducing CTF into the classroom, educators may suffer a high burden due to factors such as time and economy in the process of crafting problems and operating CTFs. Accordingly, various studies have been conducted to reduce this burden. On the other hand, in introducing CTF to the classroom, the burden of educators also exists in the aspect of an in-depth evaluation of students' academic achievement. This means that educators need to evaluate students' academic abilities in-depth so that educators can provide clear feedback on the factors that caused students to fail. Through this, educators can effectively increase student learning efficiency by helping students correct their own weaknesses. The need for such detailed evaluation can be said to be quite high in the pwnable field, one of the representative fields of CTF. This is because pwnable requires participants to have a comprehensive understanding of overall program analysis, vulnerability, mitigation bypassing techniques, systems, and so on. However, the evaluation manner of the existing CTF is not suitable for an in-depth evaluation of students' academic ability because they simply measure whether or not they solve problems in a pass and/or non-pass manner. Therefore, we designed a fine-grained evaluation CTF platform that aims to help educators provide precise evaluation and feedback on learners' failure factors in an attempt by educators to introduce CTF into the classroom to educate pwnable to reduce the burden on educators in properly evaluating student's Academic achievement.

Keywords: Capture the flag · CTF · Pwnable · Control flow hijack · Exploit

1 Introduction

Recently, various studies are attempting to increase the effectiveness of information security education [3,14,16]. Accordingly, pedagogics using the CTF (Cap-

Supported by the Institute for Information Communications Technology Promotion (IITP) of the Korea government (MSIT) [Grant No. 2018-0-00420, 2019-0-00273].

© Springer Nature Switzerland AG 2020
I. You (Ed.): WISA 2020, LNCS 12583, pp. 179–190, 2020.
https://doi.org/10.1007/978-3-030-65299-9_14

ture the Flag) manner in information security education has been recognized as a new paradigm. Besides, in the present era of the recent outbreak of COVID-19 virus, CTF is more valuable in that it is a pedagogical way of an untacted manner that enables active participation of students in a non-face-to-face. As the pedagogical value of CTF increases, various research and education platforms are being developed to help educators increase student access to CTF and increase learning efficiency [2,4,5,7,15,16].

Representatively, some recent studies on CTFs suggest a manner such as automated problem creation to reduce the burden required for educators to introduce CTFs into the classroom [1,8,17]. However, the burden of educators in introducing CTF into the classroom also exists in the process of carefully evaluating students. This means that if an educator can evaluate a student's academic ability in detail, the educator can provide clear feedback on the factors that caused the student to fail in learning, and effectively improves the student's learning efficiency in a manner that corrects the student's weaknesses [14]. However, the evaluation manner of the CTF platform, which is used for the competition, evaluates the competency of participants in a pass and/or non-pass manner. For example, in CTF competitions, CTF organizers use some kind of computer science and information security knowledge to make problems. When the organizer makes a problem system. The flag (generally in the form of a string) is hidden so that it cannot be read without specific knowledge of programs and files. The participant successfully acquires the flag hidden by the organizer using the knowledge required by the problem and then submits it to the flag certification server of the CTF competition. At this time, whether or not the participant solves the problem is determined as whether or not the corresponding flag is successfully acquired.

As such, the CTF for existing competition purposes only evaluates the participant's problem-solving capacity in a pass and/or non-pass manner. Therefore, for problems that require comprehensive knowledge to solve a specific problem, it is difficult to identify the participant's failure point in this evaluation manner. These features can be burdensome for educators attempting to introduce CTF as an educational tool in the classroom. This is because if students are evaluated only in a pass and/or non-pass manner, educators must invest additional time and money to analyze the causes of learners' failures and provide appropriate feedback. A representative example of a field where the burden of educators is prominent is the pwnable field that requires a comprehensive understanding of binary and system knowledge and exploit technology and so on. For example, the control flow hijack type of problem frequently asked in pwnable is solved through the following complex process. First, the vulnerability of the program must be identified through static and dynamic analysis of the problem provided in the form of source code or binary file. Next, if you have successfully identified the vulnerability, you need to create the input data of the program that allows the program to trigger the vulnerability. Also, depending on the type of problem, a single and/or multiple vulnerabilities might be utilized to hijack the control flow of a program, and an appropriate payload must be configured to allow an

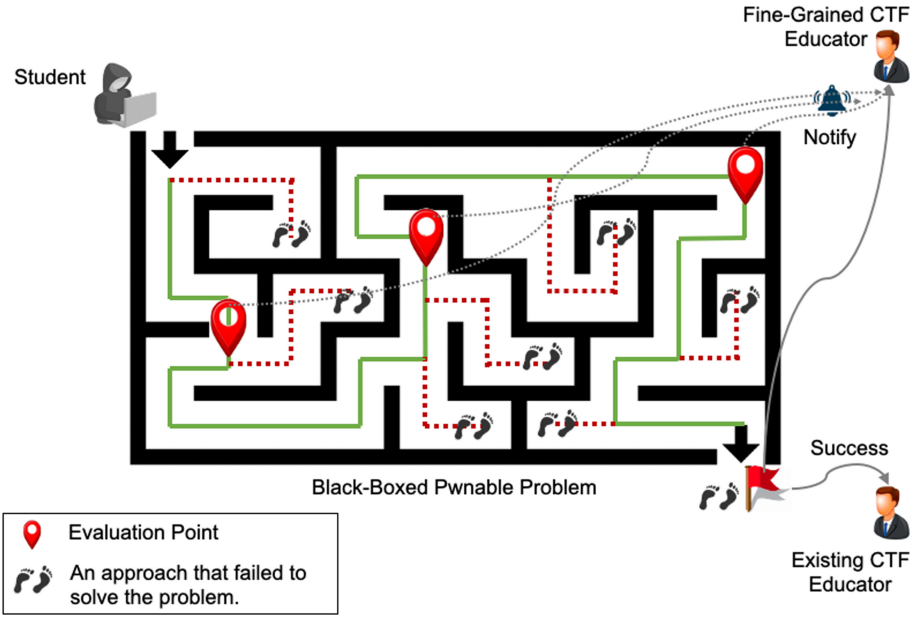

Fig. 1. Comparison of existing CTF and fine-grained CTF.

attacker to craft or execute existing code in the application address space. In some cases, when the mitigation policy is applied to the system and binary, a bypassing technique is used to bypass it. As such, pwnable problems generally require comprehensive knowledge of coding, system, attack, and defense, and program analysis skills, so learners need a comprehensive understanding of the overall process to solve single pwnable problem (see Table 1).

The existing CTF's pass and/or non-pass evaluation manner has limitations in accurately judging the learner's failure factors because it is judged that the problem itself has not been solved when a learner fails at some points in these processes. Therefore, this paper helps educators to accurately evaluate learners' failure factors in introducing CTF to the classroom, thereby reducing the burden on educators in appropriately evaluating student capabilities. To achieve this, we have defined a general problem solving process for control flow hijack type problems, and based on this, we designed a pwnable CTF platform that aims to enable precise evaluation and feedback on learners' failure factors.

The rest of this paper is organized as follows. Chapter 2 describes the general knowledge required to solve pwnable problems. Chapter 3 draws detailed evaluation points for the overall composition of a pwnable problem. Chapter 4 describe the design and implementation of fine-grained pwnable CTF. Finally, Chap. 4 presents the conclusions of this study.

Table 1. Example of required knowledge for pwnable learning according to Evaluation Points(EP)

EP	Required Skills	Example	Description
Ep1	Program Analysis	Static Analysis	It is a technology that analyzes a program without actually executing the program When the CTF competition provides solvers with source code for vulnerable programs, solvers need to understand the programming language of the programs provided to analyze them. Also, if the competition only provides binaries instead of source code, the solver needs knowledge of reverse engineering skills and system architecture to analyze the program
		Dynamic Analysis	This is a method to dynamically analyze a program by executing the program in a real or virtual environment. Participants in CTF competitions use a debugger to analyze the program during the execution process, and may also use Fuzzing and Symbolic Execution techniques to identify vulnerabilities in the program
	Vulnerability Identification & Vulnerability trigger	Stack Overflow	Stack overflow is a vulnerability that can inject data across the boundaries of variables allocated to the process' stack memory area. In the stack memory area, information including the return address of the function is stored, which can result in manipulating arbitrary indirect calls
		Integer Overflow	In certain languages, including C/C++, when the expression range of an integer data type is exceeds, undefined behavior such as a change in the sign of the data may occur. If the variable is used in conditional expressions or memory allocation size, it may cause fatal results
		Use After Free (UAF)	Most operating systems use their own memory management policy to reduce fragmentation of heap area. The UAF vulnerability can lead to information leaks, code execution, etc., depending on conditions when reusing freed memory
EP2	Control-Flow Hijacking	Indirect Call Overwrite	A skill that handles the program's control flow by manipulating data associated with the program's indirect call. The return address, function pointer, global offset table, etc. are subject to tampering
		Shellcoding	The skill of creating a small-sized program that executes specific instructions in the system, usually in machine code
EP3	Mitigation Bypassing	Return to Library	A method that bypasses protection by modulating the execution flow into a library code area that has execution authority. It is mainly used in situations where there is no write permission for the stack or heap area due to the protection techniques such as NX
		Return Oriented Programming	This method is used to bypass protection techniques such as NX, DEP, and ASLR. This skill uses a gadget in the program code area to control the call stack

2 Pwnable CTF Problem-Solving Workflow

In this study, we divide the required knowledge of the general pwnable problem into four stages based on the overall stage for exploitation: program analysis, vulnerability identification, control flow hijacking, and mitigation bypassing. This chapter describes the typical required knowledge for each step of solving pwnable problems (see Table 1).

2.1 Program Analysis

In general, pwnable systems are configured to acquire a flag by hijacking system permissions using a security vulnerability in a program running on a remote server. Accordingly, the attacker analyzes the program to identify the vulnerabilities of the program. Therefore, the knowledge required to solve the problem varies depending on the architecture of the server on which the vulnerable program is running. The process of analyzing the program is provided in competition problems. This process requires skills such as reverse engineering depending on whether the source code is provided and whether symbols and obfuscation are present [6,9,11,12].

2.2 Vulnerability Identification

Once the solver has successfully analyzed the program, the single and/or multiple vulnerabilities that exist within the given program are then identified to create the appropriate program inputs to trigger it. Therefore, at this stage, the solver must understand the various security vulnerabilities and sufficient programming knowledge to trigger the vulnerability [13].

2.3 Control Flow Hijacking

Most pwnable problems aim to hijack the control flow of the program as a final goal. To achieve this goal, attackers usually use the skill of manipulating areas where arbitrary manipulation is possible because write permission remains in the memory of application. For example, the return address of the function, function pointer, vtable, Global Offset Table (GOT) area, etc. can be a target. Therefore, the learner should understand the memory space and various techniques for handling control flow.

2.4 Mitigation Bypassing

The final step for the exploit is to take control of the program on the remote server. However, owing to mitigation policies developed over a long period of time, many CTF competitions require participants to understand the methods for bypassing these protection techniques. Accordingly, after control flow hijacking, it is necessary to understand the protection techniques applied to systems and binaries and various skills to bypass them.

3 Evaluation Point Derivation

The main idea of this study is as follows. If it is possible to automate and measure the main steps for solving typical pwnable problems, the cause of failure can be analyzed also through the learner's failure point. Therefore, in this study, four evaluation points were derived based on the general process of a control flow hijacking attack, which uses the memory corruption exploit to derive clear points of failure for learners: crash, control flow handling, mitigation bypassing, and full exploit.

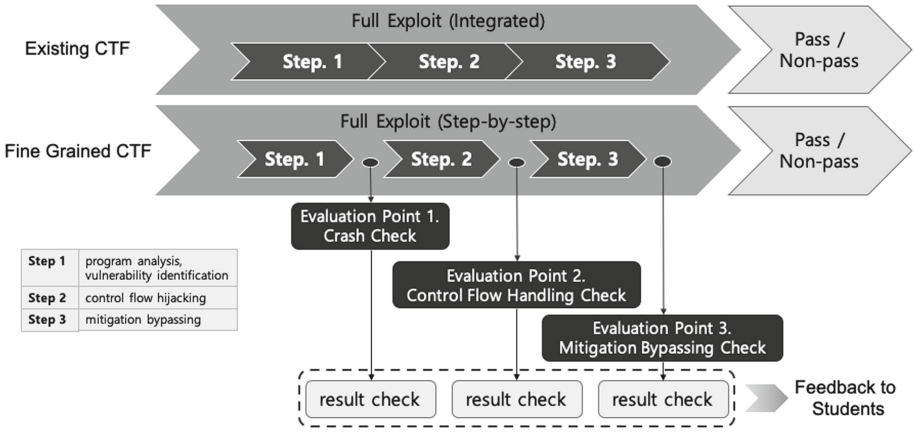

Fig. 2. Design concept comparison of existing CTF and fine-grained CTF.

3.1 Evaluation Point 1 – Crash Check

In the pwnable CTF competition, a program containing a vulnerable is generally provided to the solver by default, and in some cases, the source code of the corresponding program is also provided. Therefore, the student first goes through the static and dynamic analysis process in the problem solving process to identify the vulnerability of the program. Evaluation Point 1 evaluates students' ability to analyze programs and identify vulnerabilities. To cause a crash associated with a vulnerability in a running program for the solver, they must analyze the given program, find the vulnerability in that program, and craft an appropriate input payload that can trigger the vulnerability through programming. Accordingly, in this study, a student who can cause a crash related to a vulnerability in a running program is considered capable of analyzing basic problems. That is, if a student successfully passes Evaluation Point 1, the educator can judge that the student has the ability to analyze the program that contains the vulnerability, identify the program's vulnerability, and craft the input value that can trigger it through programming. At this stage, the student can identify bugs in the program by performing a source code auditing or reversing process to precisely analyze the program. In addition, bugs in the program can be identified by using dynamic testing techniques such as fuzzing and symbloic execution. Meanwhile, students who do not pass Evaluation Point 1 can be judged to have insufficient knowledge. Thus, educators can provide appropriate feedback to users to help students overcome this learning hurdle.

3.2 Evaluation Point 2 – Control Flow Handling Check

The pwnable problem usually requires the solver the ability to craft an exploit by exploiting single or multiple vulnerabilities in the program. To measure this, evaluation point 2 checks students' ability to exploit the program's vulnerability

to manipulate the program's control flow. In other words, the evaluation point is a step of measuring the user's exploit capability in an environment where mitigation techniques are not applied. At this point, the student that successfully handles the instruction pointer of the program as an arbitrary value has successfully passed Evaluation Point 2. Such a student is judged to not only has knowledge of the preceding steps (program analysis), but also skills that can trigger potential vulnerabilities related to the instrument pointer by combining the vulnerabilities that exist within the program. Meanwhile, a student who passed Stage 1 but failed to pass the Evaluation Point 2 has the knowledge required in the previous stage, but he or she has insufficient knowledge for manipulating the instruction pointer as needed.

3.3 Evaluation Point 3 – Mitigation Bypassing Check

In the control flow hijacking scenario, a difference is observed between modulating the instruction pointer and seizing the complete control flow. This difference depends on whether the program and system are mitigated, so passing this stage requires the ability to bypass various mitigation techniques. For example, if a stack canary protection technique is applied to a program, the solver may need to utilize an information disclosure vulnerability such as leaking canary data inserted in the program stack to avoid the exploit code failure. In addition, when the program is executed in a system environment to which ASLR mitigation is applied, the solver can utilize an attack technique that can craft an exploit by using a code gadget with a fixed address such as ROP. To check this, evaluation point 3 reconstructs a given problem by partially applying various mitigation techniques applied to the problem. To sum up, evaluation point 3 measures whether a user has the ability to bypass exploit mitigation configured in various ways. Therefore, educators can judge that students who do not pass Evaluation Point 3 have insufficient understanding of protection techniques and the techniques to bypass them. Also, as with the previous evaluation point 2, if the student successfully passes the level, the student is considered to have the knowledge required for the previous evaluation point. Students who pass the evaluation point 2 but do not pass evaluation point 3 may be considered to have the necessary knowledge in previous steps, but not enough knowledge to bypass certain mitigation techniques.

3.4 Evaluation Point 4 – Full Exploit Check

Evaluation Point 4 verifies whether the student has succeeded in obtaining a flag of the remote system through a control flow hijacking exploit. This step is the same as the scoring method in the general CTF platform. Students who have completely passed the final evaluation point can be judged to have all the knowledge required for the problem.

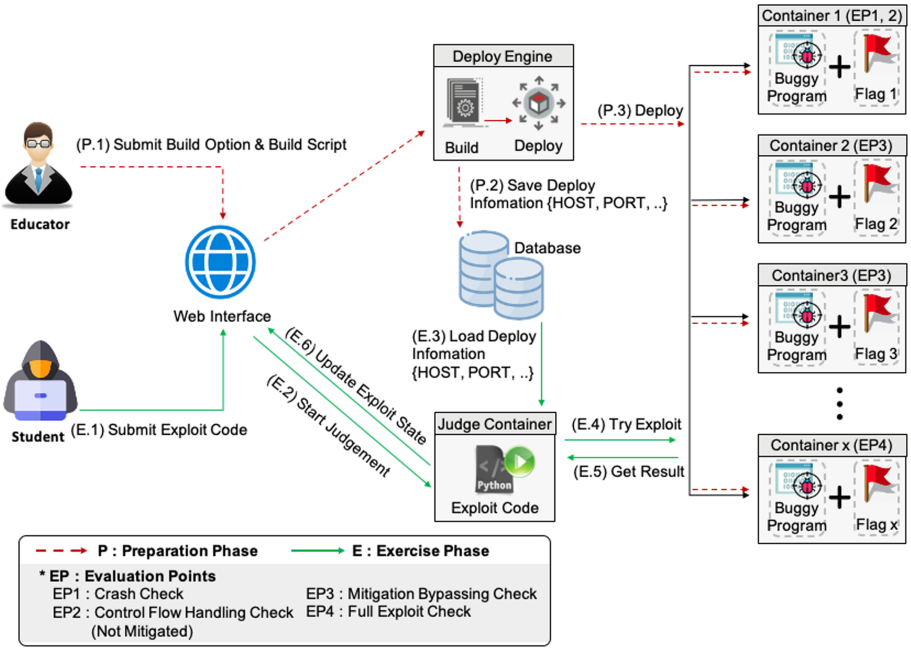

Fig. 3. Design of the fine-grained CTF.

4 Design and Implementation of the Fine-Grained CTF

In this section, we describe the design for implementing fine-grained pwnable CTF. Figure 3 shows the overall design overview of our fine-grained CTF architecture. The fine-grained CTF aims to automatically transform the evaluation of the pass/non-pass manner of the existing jeopardy-style pwnable CTF into a more fine-grained evaluation method. To achieve this, we used a method to build a separate evaluation container environment for each evaluation point derived in Sect. 3. For example, we configure a separate evaluation container environment that measures whether control flow has been tampered with in order to evaluate the user's control flow handling capabilities. We also constructed each evaluation container environment for all subsets of the mitigation technique applied to the pwnable problem, to verify the user's ability to bypass the various protection techniques used in the problem. Our evaluation system is largely composed of a preparation phase in which educators distribute problems and an exercise phase in which students solve problems. The rest of this section describes the process of deploying the pwnable problem by the educator in the preparation phase, and the process by which the user's exploit code is evaluated in our fine-grained CTF during the exercise phase.

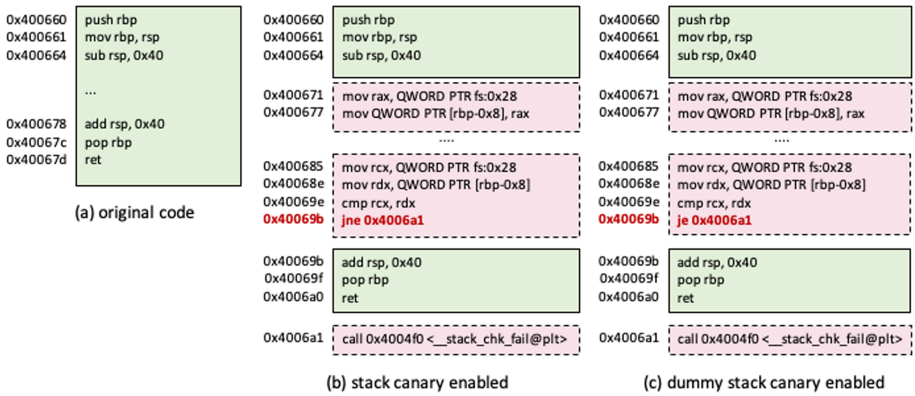

Fig. 4. Change of code address offset according to the application of the code instrumentation protection technique.

4.1 Preparation Phase

In the preparation phase, the educator first submits the source code of the pwnable problem, the build script which builds the source code, and the mitigation type to be applied to the problem through the web interface. Next, the Deploy Engine checks the available port information of the system and constructs an evaluation container corresponding to each port number. The container created in the process consists of a power set for mitigation specified by the user. For example, when the mitigation set for a specific container is $\{\emptyset\}$ (least mitigated), the container corresponds to Evaluation Point 2, which evaluates a user's control flow handling capability. In addition, when the mitigation is the same as the mitigation set specified by the user (most mitigated), the container refers to Evaluation Point 4. It means evaluating whether the user can bypass all mitigation techniques applied to the problem.

In each fine-grained evaluation container configuration, we reconstruct the binary file so that the built binary file always has the same code address offset and memory layout. A lot of memory corruption exploit techniques use code address offset and memory layout information of binary files in exploit code construction. The ROP is a representative exploit technique that uses a code gadget which is in the binary file. However, the binary file applied with mitigation technique through binary instrumentation such as stack canary has a difference in code address offset in the program as shown in (a) and (b) of Fig. 4. Also, in many exploit techniques such as buffer overflow and UAF, the memory layout of the program has an important effect on exploit reliability. For this reason, in a fine-grained evaluation system, it is necessary to reconstruct the problem binary files executed in each evaluation container to have the same code offset and memory layout. To achieve this, we implemented the dummy StackProtector Pass by modifying the code that generates the canary check instruction of the StackProtector Pass in the LLVM project [10]. As shown in Fig. 4, LLVM's

StackProtector pass inserts an arbitrary stack canary during the source code build process, and inserts code that checks it during execution. Based on this, we used the method to modify the instruction that the StackProtector pass checks in the function epilogue for the stack canary inserted in the function prologue. The assembly code of the binary file generated through this is as shown in Fig. 4 (b) and (c).

The host ip address, port number and evaluation type used in the process of deploying the evaluation container are stored in the database. This data will be used in the execution and judgment process of the exploit code submitted by the user in the future exercise phase.

Also, during the deployment of our fine-grained CTF evaluation container, a randomly generated flag is stored in each container. If the same flag is used in each evaluation container, the user may maliciously bypass the high stage problem by simply printing the flag obtained through solving the low stage problem. For example, consider the case where a malicious user submits exploit code that causes a program crash to the judgment server. Then the user successfully passes the evaluation point 1. Subsequently, a malicious scenario in which a malicious user simply prints the flag data string of evaluation point 1 obtained through the exploit code targeting evaluation point 1 to evaluation points other than evaluation point 1 may exist. Because of the existence of this malicious scenario, the flags existing in the containers constituting each evaluation point should not only be difficult for the user to infer, but also must use different flag values for each container. The flag strings of each evaluation container are also stored into the database for the user's exploit code judgment at a future exercise phase.

4.2 Exercise Phase

In the exercise phase, students submit exploit code through a web interface in the form of an online judge system. Unlike the usual jeopardy-style CTF method, which transmits an exploit payload over the network to remote servers where the vulnerable binaries are operating, our proposed fine-grained CTF gets an exploit code from users. This is because our fine-grained CTF is a system designed with educators as the main target. Our fine-grained CTF system allows educators to provide detailed feedback by investigating the exploit code written by the student, as well as the point of failure of the student derived through a series of evaluation processes.

Next, the exploit code submitted by the user is executed in an isolated container environment. This is to restrict malicious behavior that can occur when the user's code is executed directly in the host environment of the system where the fine-grained CTF is hosted. For example, if the exploit code uploaded by the user is not isolated and operates directly in the host environment, the user can directly perform various malicious actions such as reading flag information stored in the database directly on the host computer. Because of the high risk of executing code directly in the host computing environment, our fine-grained CTF design forces user-submitted code to run only in an isolated environment.

ALGORITHM 1: Evaluation process of the judgment container.

Input : N – problem number
E – exploit code submitted by the student
D – database
C – crash identifier ("SIGABRT", "SIGSEGV")
Output: S – exploit status

```
1  P ← getPortNums (D, N)        /* get the port numbers of evaluation containers.  */
2  H ← getHost (D, N)                 /* get the host IP of evaluation containers.  */
3  for p ← P do
4      F← getFlags (p)          /* read the flag stored in the evaluation container. */
5      s ← tryExploit (E, H, p)              /* try exploit and save output stream */
6      if isContain (s, F) then
7          S← updateExploitState (S, D, p)           /* Update Exploit status.  */
8      end
9      if isContain (s, C) then
10         S← updateExploitState (S, D, p)   /* Update Exploit status(Crashed).  */
11     end
12 end
```

The exploit code submitted by the user in the judgment container is executed with the host ip address and port number stored in the database as arguments. Therefore, the exploit code submitted by the student must be crafted with the host ip address and port number as system arguments in our fine-grained CTF system.

The process in which the exploit code submitted by the student in the judgment container is evaluated in detail in a fine-grained manner is described in detail in Algorithm 1. At this stage, our fine-grained system was built in the ubuntu environment, so the characters "SIGABRT" and "SIGSEGV" are used as identifier strings to identify crash in the linux system. Also, deploying a separate container for crash check, which is the purpose of evaluation point 1, can cause unnecessary system overhead, so we have inserted a string matching process for crash check into the evaluation process without constructing a separate container.

5 Conclusions

Recently, CTF, which was mainly used for hackers to exchange technical expertise and engage in competition, has now been widely implemented as an educational platform in the field of information security. Accordingly, various research approaches have been applied to improve learning efficiency for beginners. In addition, research has been conducted to reduce the costs and educators' burdens for operation of a CTF. This study subdivides the pwnable CTF, which requires a comprehensive understanding of the entire system, into distinct evaluation points to improve the ability of educators to identify the failure factors of

learners. However, for this approach to successfully relieve the educators' burden, it is necessary not only to propose evaluation points but also to automate the detection of these points. Therefore, we design a CTF platform that can automate the detection of learner failure points based on these evaluation points.

References

1. Burket, J., Chapman, P., Becker, T., Ganas, C., Brumley, D.: Automatic problem generation for capture-the-flag competitions. In: 2015 {USENIX} Summit on Gaming, Games, and Gamification in Security Education (3GSE 15) (2015)
2. Chapman, P., Burket, J., Brumley, D.: Picoctf: A game-based computer security competition for high school students. In: 2014 {USENIX} Summit on Gaming, Games, and Gamification in Security Education (3GSE 14) (2014)
3. Chothia, T., Novakovic, C.: An offline capture the flag-style virtual machine and an assessment of its value for cybersecurity education. In: 2015 {USENIX} Summit on Gaming, Games, and Gamification in Security Education (3GSE 15) (2015)
4. ctfd: Ctfd. https://ctfd.io. Accessed 29 May 2020
5. daehee: pwnable.kr. http://pwnable.kr/. Accessed 29 May 2020
6. gdb: gdb. https://www.gnu.org/software/gdb/
7. hackthebox: hack the box. https://www.hackthebox.eu/. Accessed 29 May 2020
8. Hulin, P., et al.: Autoctf: creating diverse pwnables via automated bug injection. In: 11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17) (2017)
9. ida: ida. https://www.hex-rays.com/products/ida/. Accessed 29 May 2020
10. llvm: Llvm project. https://llvm.org/docs/index.html. Accessed 29 May 2020
11. microsoft: debugging tools for windows. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/. Accessed 29 May 2020
12. pwndbg: pwndbg. https://github.com/pwndbg/pwndbg. Accessed 29 May 2020
13. pwntools: pwntools. http://docs.pwntools.com/en/stable/. Accessed 29 May 2020
14. Rege, A.: Multidisciplinary experiential learning for holistic cybersecurity education, research and evaluation. In: 2015 {USENIX} Summit on Gaming, Games, and Gamification in Security Education (3GSE 15) (2015)
15. rootme: root me. https://www.root-me.org/. Accessed 29 May 2020
16. Vykopal, J., Barták, M.: On the design of security games: From frustrating to engaging learning. In: 2016 {USENIX} Workshop on Advances in Security Education ({ASE} 16) (2016)
17. Wi, S., Choi, J., Cha, S.K.: Git-based {CTF}: A simple and effective approach to organizing in-course attack-and-defense security competition. In: 2018 {USENIX} Workshop on Advances in Security Education ({ASE} 18) (2018)