International Journal of Machine Learning and Computing, Vol. 11, No. 2, March 2021

Real-time Attack-Scheme Visualization for Complex Exploit Technique Comprehension

Seima Kose, Yumi Suenaga, and Kazumasa Oida

***Abstract*—Recent exploit techniques are highly complex, and it is not easy for cybersecurity learners to understand the attack strategies quickly and clearly. For efficient and comprehensive learning, this paper proposes an attack-scheme visualization system that fulfills three requirements: attack progress visualization in real-time, memory and register-level description, and concise description of the attack schemes. This paper exemplifies two cases: stack buffer overflow and ROP attacks, and demonstrates how the system operates and how users can learn that existing defense technologies are effective or ineffective depending on the execution environments.**

***Index Terms*—Exploit code, visualization, ROP, cybersecurity learning.**

I. INTRODUCTION

Nowadays, new vulnerabilities in software and hardware are discovered every day, and new attack techniques that exploit vulnerabilities have also been developed. Software and hardware vendors have devised a variety of countermeasures against those attack techniques; nevertheless, attackers have come up with ways to circumvent the countermeasures. Advances in attack technologies are being highly accelerated by various bug bounty programs (HackerOne, iDefence, etc.) and numbers of hacking competitions (Pwn2Own, Mobile Pwn2Own, DEFCON, etc.).

Because of this arm race between attackers and defenders, highly sophisticated cyber-attack techniques, such as control-flow hijack attacks [1], have been developed. One of them is return-oriented programming (ROP) [2], which is an exploit technique that allows attackers to achieve control flow hijacking through executing machine instruction sequences called a gadget, which is present in the machine's memory and ends with a return instruction. By chaining gadgets together, it is reported that attackers can perform arbitrary operations [3].

Meanwhile, the growing security market requires more security professionals. The need for skilled practitioners is projected to grow at a rate of 32% [4]. In our opinion, training systems for security specialists should provide the following three requirements for *efficient* and *comprehensive* learning:

1) The system should provide an environment in which exploit codes can run (e.g., it should not be a simulator or emulator), and should visualize what the code is performing in real-time because learners can gain a lot of knowledge through modifying and executing the codes.
2) The system should present enough detail explanation for exploit techniques; the system should describe “how the exploit codes work” rather than “what it can do.” The assembly language level explanation is preferred to the script level such as Metasploit [5] etc.
3) The system should present only essential information related to the attack. Current exploit codes are highly complex and often include unnecessary instructions. Analysis tools [6] and debuggers provide sufficient detail but at the same time too much unrelated information. Filtering out irrelevant information in advance can enhance the efficiency of learning.

There are companies, such as Palo Alto Networks, CISCO, IBM, etc. and open-source frameworks, such as FBCTF [7], CyTrONE [8], etc. that provide cyber ranges, virtual environments for cyberwarfare training and cyber technology development. These focus on teaching the best practice on how to respond to network cyber-crime rather than teaching how attack codes work. Another way to practice and learn hacking tools is to create a personal hacking lab, an isolated sandbox environment. A hacking lab typically uses open source software, such as Kali Linux [9] and Metasploitable [5]. A hacking lab explains what the script-level attack commands can do rather than how the attack codes work. For learning more deeply, learners must spend lots of time reading source codes.

The prototype system in this paper is designed to visualize in real-time the detail mechanisms related to the essence of attack schemes. As far as we know, there are no studies that discussed this type of learning system. The learning system currently has three functions. (1) The system displays the detailed status of a running exploit process on web pages. (2) The system can explain to learners why some defense techniques against the attack are effective/ineffective. Lastly, (3) the system tests learners’ comprehension, for example, by asking them to make up an attack code applicable to a modified vulnerable code.

The paper is organized as follows. Section II presents the work related to this paper. Section III describes how the system visualizes a running attack code in real-time. Although there are numbers of complex control flow hijacking techniques, our prototype system currently

---

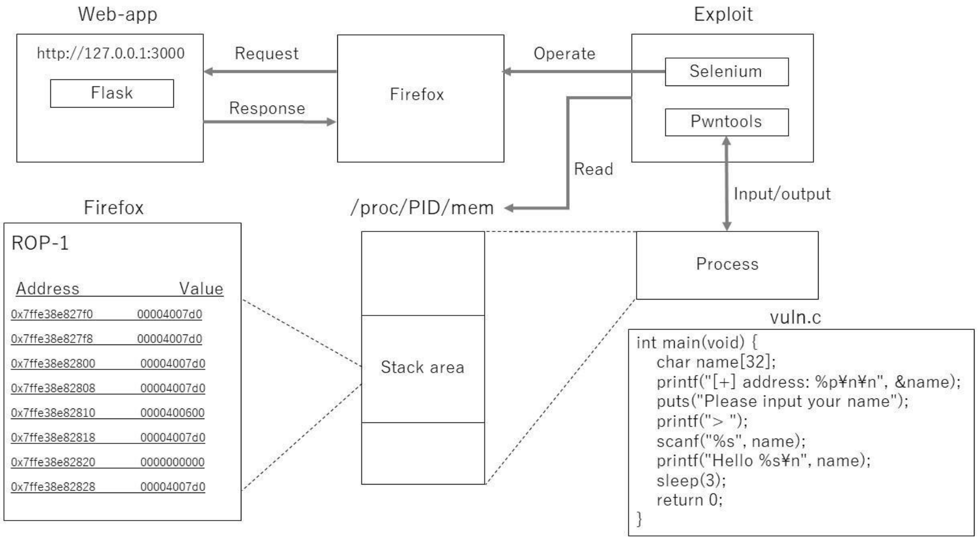
Manuscript received February 5, 2020; revised September 11, 2020.

S. Kose is with the Department of Computer Science and Engineering, Fukuoka Institute of Technology, Fukuoka, 811-0295, Japan (e-mail: s16a2025@bene.fit.ac.jp).

Y. Suenaga and K. Oida are with the Graduate School of Course of Computer Science and Engineering, Fukuoka Institute of Technology, Fukuoka, 811-0295, Japan (e-mail: mfm19102@bene.fit.ac.jp, oida@fit.ac.jp).

doi: 10.18178/ijmlc.2021.11.2.1030

supports stack buffer overflow and ROP attacks. Section IV exemplifies the visualization of these two attacks. Section V discusses how to deepen the knowledge about the attacks, and Section VI concludes the paper.



ROP-1

Address	Value
0x7ffe38e827f0	00004007d0
0x7ffe38e827f8	00004007d0
0x7ffe38e82800	00004007d0
0x7ffe38e82808	00004007d0
0x7ffe38e82810	0000400600
0x7ffe38e82818	00004007d0
0x7ffe38e82820	0000000000
0x7ffe38e82828	00004007d0

```
int main(void) {
    char name[32];
    printf("[+] address: %p¥n¥n", &name);
    puts("Please input your name");
    printf("> ");
    scanf("%s", name);
    printf("Hello %s¥n", name);
    sleep(3);
    return 0;
}
```

Fig. 1. The system consists of the web-app and exploit parts. The exploit interacts with the process of vulnerable code `vuln.c`, file `/proc/PID/mem`, Firefox browser, and users of the system. The web-app interacts with Firefox.

II. RELATED WORK

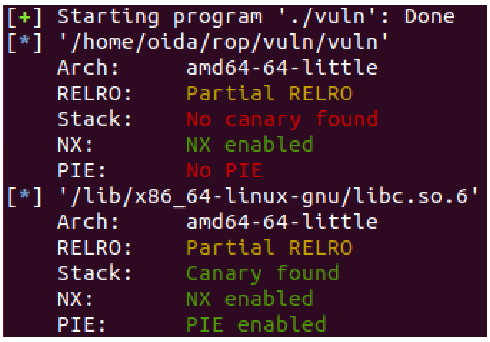
Recent cyber-attack techniques, especially control flow hijacking, are highly complex and numbers of variants of the techniques have been developed [10]. Furthermore, there are studies that automatically produce exploit codes for buffer overflows [11], ROP chains [12], heap overflows [13], etc.

To catch up with the development speed of attack tools, various mitigation technologies have been developed. They are Address Space Layout Randomization (ASLR) [14], No eXecute bit (NX bit) [15], Stack Smashing Protection (SSP) [16], Position-Independent Executable (PIE) [17], RELocation Read-Only (RELRO) [18] etc. The state-of-the-art defense technologies, whose implementation are currently research prototypes, are control-flow integrity (CFI) [19] and code-pointer integrity (CPI) [20]. All of them, however, cannot completely defeat the exploit techniques.

Another way to prevent or mitigate cyber-attacks is to practice hands-on training in a cyber range, where trainees experience attacks to find the best solutions to the attacks. There are researches that simulate attack situations for understanding basic concepts [21]-[23]. Realistic cybersecurity training is currently conducted in military environments, and the proprietary systems that are available publicly are expensive [8]. Some open-source training frameworks [7], [8] are recently available. They are, however, not suited for efficiently leaning how attack codes work.

III. SYSTEM CONFIGURATION

Fig. 1 illustrates the structure of our prototype system that consists of two modules: exploit and web-app. The exploit module attacks a vulnerable binary code `vuln` (whose C language source code is `vuln.c`) using pwntools, where pwntools is an exploit development library that helps attackers to create attack codes in the following three steps. First, it indicates what kinds of defense mechanisms the vulnerable code and the operating system have (Fig. 2). Second, it searches for vulnerabilities in the code. Third, it assists in creating attack codes to exploit the vulnerabilities.



```
[+] Starting program './vuln': Done
[*] '/home/oida/rop/vuln/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Fig. 2. Pwntools framework reveals defense mechanisms in the target file `vuln` and the standard C library `libc.so.6`.

An attack code is not automatically created but it is assembled by attackers. To understand the scheme of the attackers, our system displays the memory data of a running vulnerable code in real time. This is feasible because the proc filesystem (procfs) [24] creates `/proc/PID/mem` file in memory, which contains the memory information of the running process whose process id is PID. The exploit retrieves an important part of the stack data from the file and then transfers the data to the Firefox browser in JSON format. The Selenium framework is used to adjust the timing of displaying the retrieved data on the browser.

When the browser is ordered to open the URL of `http://127.0.0.1.3000` using the HTTP GET method, the web-app module returns the web page, which is constructed by Flask, a web application framework. In Fig. 1, only an essential portion of the process memory is displayed on the browser and easy-to-understand comments are attached.

The system can work properly by adding two executable statements to the vulnerable code. The first is a function that outputs the buffer address used in the attack, whereby the system can recognize the place where in the stack area the system should focus on (the address can be automatically retrieved from `/proc/PID/mem` file if ASLR is not enabled). In Fig. 1, `printf("[+] address: %p\n\n", &name)` in `vuln.c` corresponds to the statement. The second is function `sleep(3)`, which requires the next statement of the function to be executed after three seconds. The function must be inserted just before return or exit statement; otherwise, the system may not be able to read the data in the memory file due to the termination of the process.

IV. CASE STUDIES

This section illustrates the feasibility of our approach. The prototype software running on an Ubuntu 18.04.3 LTS machine visualizes two attacks: stack buffer overflow and ROP attacks.

A. *Stack Buffer Overflow*

The stack buffer overflow attacks are classical and straightforward attacks, and at least five countermeasures have been implemented in the current Ubuntu system: RELRO, SSP, NX bit, PIE, and ASLR. These are explained later when necessary. Fig. 2 shows the status of them. ASLR, which is a system-wide property, is enabled in our environment. Under the environment shown in Fig. 2, our system exemplifies how an overflow attack can divert the flow of execution into any function or codes using binary code vuln (whose source code `vuln.c` is in Fig. 1).

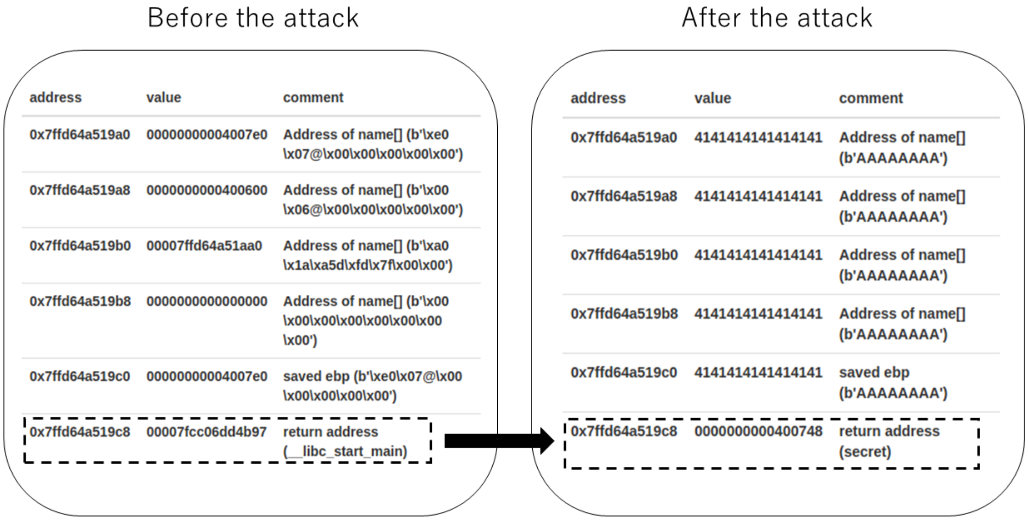
If a function, say `secret()`, is also defined in the `vuln.c` file and the name of the function is *a priori* known, then pwntools can derive the memory address of the function from symbol name "`secret`." When `vuln` asks to input your name (see `puts("Please input your name")` in `vuln.c`), the exploit sends 49-byte data (called a payload from now on), which consists of 40 characters of '`A`,' the address of function `secret()`, and a line feed code. The intent of the exploit can be articulated by visualization.

Fig. 3 shows the web pages output by the system. It can be easily recognized that the overflow attack replaces not only the buffer area with characters '`A`' but also the return address of `__libc_start_main` with the address of function `secret()`, which implies that the exploit module has controlled the execution flow.

B. *Return-Oriented Programming*

ROP further develops the potential for buffer overflow attacks. The overflow attack often inserts malicious codes into the data storage area. Even if the NX bit [15] marks the storage area non-executable, ROP attacks can circumvent this mechanism by using the existing code in static or dynamic libraries. Therefore, ROP is one of the code reuse attacks. In the ROP attacks, attackers often make up complex payloads that consist of a variety of "ROP gadgets," which are short sequences of assembly instructions that end with `ret`, and put them in the stack area.

In this case study, the system demonstrates how an attacker can invoke shell `/bin/sh` using vulnerable code `vuln` under the same condition shown in Fig.2. Note in general that the ability of adversaries to operate the shell without formal login authentication implies that they can remotely control the target machines. The exploit executes the function `main()` in `vuln.c` twice for coping with another defense mechanism ASLR [14], which randomly arranges the address space positions of the stack, heap, libraries, etc.



Before the attack

address	value	comment
0x7ffd64a519a0	00000000004007e0	Address of name[] (b'\xe0 \x07@\x00\x00\x00\x00\x00')
0x7ffd64a519a8	0000000000400600	Address of name[] (b'\x00 \x06@\x00\x00\x00\x00\x00')
0x7ffd64a519b0	00007ffd64a51aa0	Address of name[] (b'\xa0 \x1a\xa5d\xfd\x7f\x00\x00')
0x7ffd64a519b8	0000000000000000	Address of name[] (b'\x00 \x00\x00\x00\x00\x00\x00 \x00')
0x7ffd64a519c0	00000000004007e0	saved ebp (b'\xe0\x07@\x00 \x00\x00\x00\x00')
0x7ffd64a519c8	00007fcc06dd4b97	return address (__libc_start_main)

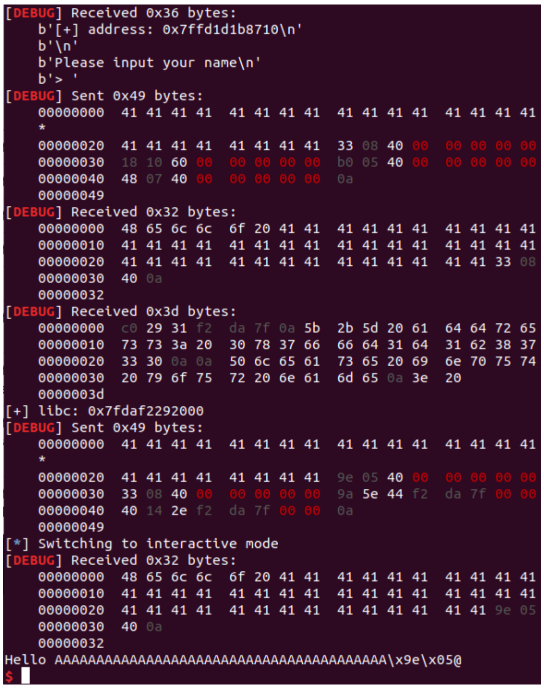
After the attack

address	value	comment
0x7ffd64a519a0	4141414141414141	Address of name[] (b'AAAAAAAA')
0x7ffd64a519a8	4141414141414141	Address of name[] (b'AAAAAAAA')
0x7ffd64a519b0	4141414141414141	Address of name[] (b'AAAAAAAA')
0x7ffd64a519b8	4141414141414141	Address of name[] (b'AAAAAAAA')
0x7ffd64a519c0	4141414141414141	saved ebp (b'AAAAAAAA')
0x7ffd64a519c8	0000000000400748	return address (secret)

Fig. 3. The web page before and after the overflow attack. The ASCII code of character '`A`' is 41in hexadecimal notation.

Fig. 4 shows a log file of pwntools, which records all interactions with other functions such that "Sent" ("Received") in the log file indicates data sent (received) by the exploit module. As shown in the figure, the exploit sends 0x49-byte (73-byte) data twice and received an address (libc: 0x7fb895320000), which is the base address of library libc randomly selected by ASLR. Note that the exploit successfully invokes `/bin/sh`; the last line of the log file contains "`$`," which works as the prompt of the shell.

The log file explains almost nothing about why the shell prompt appears; whereas our system clearly answers the essence of the attacker's tactics in real-time by outputting the web pages in Fig. 5 and Fig. 6. As shown in Fig. 5, using the stack buffer overflow, the first payload rewrites the return address with the address of a gadget, which executes only two instructions: `pop rdi` and `ret`. When the gadget is executed, the stack pointer register (RSP) points to the next address of the replaced address, in which the address of `puts@got` is written. Since the gadget executes `pop rdi`, the address of `puts@got` is moved to RDI register and the gadget returns the execution flow to the address where the address of `puts@plt` exists. Therefore, function `puts()` outputs the address of `puts@got` and returns to the next address where the address of `main()` exists. In short, the aim of the exploit is to execute "`puts(puts@got)`" and go back to `main()`.



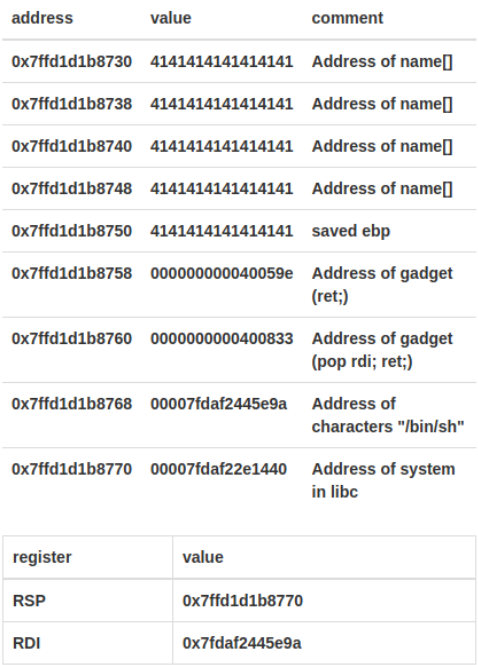
```
[DEBUG] Received 0x36 bytes:
    b'[+] address: 0x7ffd1d1b8710\n'
    b'\n'
    b'Please input your name\n'
    b'> '
[DEBUG] Sent 0x49 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41
    *
    00000020  41 41 41 41  41 41 41 41  33 08 40 00  00 00 00 00
    00000030  18 10 60 00  00 00 00 00  b0 05 40 00  00 00 00 00
    00000040  48 07 40 00  00 00 00 00  0a
    00000049
[DEBUG] Received 0x32 bytes:
    00000000  48 65 6c 6c  6f 20 41 41  41 41 41 41  41 41 41 41
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41
    00000020  41 41 41 41  41 41 41 41  41 41 41 41  41 41 33 08
    00000030  40 0a
    00000032
[DEBUG] Received 0x3d bytes:
    00000000  c0 29 31 f2  da 7f 0a 5b  2b 5d 20 61  64 64 72 65
    00000010  73 73 3a 20  30 78 37 66  66 64 31 64  31 62 38 37
    00000020  33 30 0a 0a  50 6c 65 61  73 65 20 69  6e 70 75 74
    00000030  20 79 6f 75  72 20 6e 61  6d 65 0a 3e  20
    0000003d
[+] libc: 0x7fdaf2292000
[DEBUG] Sent 0x49 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41
    *
    00000020  41 41 41 41  41 41 41 41  9e 05 40 00  00 00 00 00
    00000030  33 08 40 00  00 00 00 00  9a 5e 44 f2  da 7f 00 00
    00000040  40 14 2e f2  da 7f 00 00  0a
    00000049
[*] Switching to interactive mode
[DEBUG] Received 0x32 bytes:
    00000000  48 65 6c 6c  6f 20 41 41  41 41 41 41  41 41 41 41
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41
    00000020  41 41 41 41  41 41 41 41  41 41 41 41  41 41 9e 05
    00000030  40 0a
    00000032
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x9e\x05@
$ 
```

Fig. 4. A log file that contains data sent and received by pwntools.

address	value	comment
0x7ffd1d1b8710	4141414141414141	Address of name[]
0x7ffd1d1b8718	4141414141414141	Address of name[]
0x7ffd1d1b8720	4141414141414141	Address of name[]
0x7ffd1d1b8728	4141414141414141	Address of name[]
0x7ffd1d1b8730	4141414141414141	saved ebp
0x7ffd1d1b8738	0000000000400833	Address of gadget (pop rdi; ret;)
0x7ffd1d1b8740	0000000000601018	Address of puts@got
0x7ffd1d1b8748	00000000004005b0	Address of puts@plt
0x7ffd1d1b8750	0000000000400748	Address of main()

register	value
RSP	0x7ffd1d1b8750
RDI	0x601018

Fig. 5. The web page after the first payload is sent.



address	value	comment
0x7ffd1d1b8730	4141414141414141	Address of name[]
0x7ffd1d1b8738	4141414141414141	Address of name[]
0x7ffd1d1b8740	4141414141414141	Address of name[]
0x7ffd1d1b8748	4141414141414141	Address of name[]
0x7ffd1d1b8750	4141414141414141	saved ebp
0x7ffd1d1b8758	000000000040059e	Address of gadget (ret;)
0x7ffd1d1b8760	0000000000400833	Address of gadget (pop rdi; ret;)
0x7ffd1d1b8768	00007fdaf2445e9a	Address of characters "/bin/sh"
0x7ffd1d1b8770	00007fdaf22e1440	Address of system in libc

register	value
RSP	0x7ffd1d1b8770
RDI	0x7fdaf2445e9a

Fig. 6. The web page after the second payload is sent.

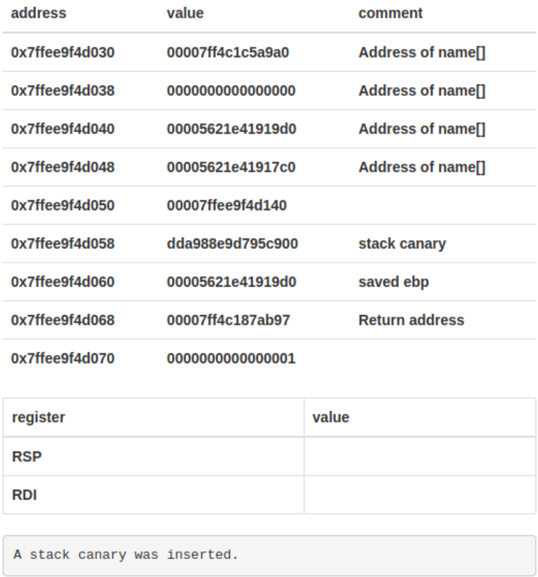
The values of registers change with time. In Fig. 5, RDI has the address of `puts@got` and RSP points to the address in which the address of `main()` exists. Therefore, the figure expresses the state of the memory and registers just before the main function is executed again.

The address of `puts@got` is used to calculate the address of function `system()` that executes `/bin/sh`. The address is obtained by adding the base address of library libc to the relative address of symbol '`system`' in the library. Since ASLR works, the base address of library libc is randomly selected; nevertheless the exploit can obtain the base address by subtracting the relative address of symbol '`puts`' from the address of `puts@got` (the current address of `puts()`).

In Fig. 6, there are two ROP gadgets. The first gadget is not meaningless; it is used for `movaps` instruction to work properly. The second puts the address of characters "`/bin/sh`" in RDI so that `system()` invokes /bin/sh. Now that the address of `system()` is resolved, the address is included in the second payload.

V. FURTHER LEARNING

Learners can observe more clearly the behavior of payloads and the defense systems by modifying the vulnerable codes or execution environments. Let us consider the case where a learner changes an option of compiler `gcc` so that the SSP mechanism [16] is enabled. Fig. 7 shows that SSP inserted a stack canary between the buffer `name[]` and the return address just after `scanf("%s", name)` was called. After the ROP attack, as shown in Fig. 8, the canary was overwritten by `0x4141414141414141`. The change in the canary value when the function returns indicates an occurrence of buffer overflow. The memo in the figure indicates termination of the process due to stack smashing detection. The termination prevents the exploit from taking control of the process.



address	value	comment
0x7ffee9f4d030	00007ff4c1c5a9a0	Address of name[]
0x7ffee9f4d038	0000000000000000	Address of name[]
0x7ffee9f4d040	00005621e41919d0	Address of name[]
0x7ffee9f4d048	00005621e41917c0	Address of name[]
0x7ffee9f4d050	00007ffee9f4d140	
0x7ffee9f4d058	dda988e9d795c900	stack canary
0x7ffee9f4d060	00005621e41919d0	saved ebp
0x7ffee9f4d068	00007ff4c187ab97	Return address
0x7ffee9f4d070	0000000000000001	

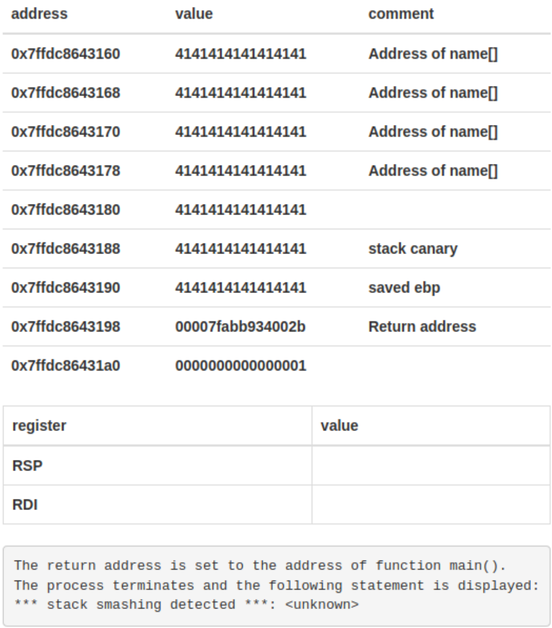
register	value
RSP	
RDI	

```
A stack canary was inserted.
```

Fig. 7. A stack canary was used as a buffer overflow indicator.

Learners can further deepen their knowledge by creating a payload that solves the problems given by the system. For example, the system askes learners to invoke `/bin/sh` when the buffer size of `name[]` in `vuln.c` is reduced from 32 to 16 bytes.

Since our system can visualize the memory content of processes in real time, we can easily extend the system to support any kind of control-flow hijacking attacks, which include heap overflow and format string attacks.



address	value	comment
0x7ffdc8643160	4141414141414141	Address of name[]
0x7ffdc8643168	4141414141414141	Address of name[]
0x7ffdc8643170	4141414141414141	Address of name[]
0x7ffdc8643178	4141414141414141	Address of name[]
0x7ffdc8643180	4141414141414141	
0x7ffdc8643188	4141414141414141	stack canary
0x7ffdc8643190	4141414141414141	saved ebp
0x7ffdc8643198	00007fabb934002b	Return address
0x7ffdc86431a0	0000000000000001	

register	value
RSP	
RDI	

```
The return address is set to the address of function main().
The process terminates and the following statement is displayed:
*** stack smashing detected ***: <unknown>
```

Fig. 8. A stack canary was overwritten by `0x4141414141414141`.

VI. CONCLUSIONS AND FUTURE WORK

Current exploit techniques are highly sophisticated and complex. For efficient and comprehensive learning of the techniques, we proposed a new approach that achieves real-time attack progress visualization, assembly language-level detailed description, and concise description of the attack schemes. Our idea was to display attack code behavior in the stack area in cooperation with the proc filesystem.

A prototype system that visualizes stack buffer overflow and return-oriented programming attacks demonstrated the feasibility of our approach. The system enables learners to further deepen their knowledge by executing a vulnerable code after modifying the code or execution conditions.

We are currently planning two research projects. The first is to implement the system as a web application so that users can learn from a distance. The second is to visualize more complex control-flow hijack attacks such as heap overflow.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

AUTHOR CONTRIBUTIONS

Seima Kose conducted all the research and developed the prototype system; Yumi Suenaga and Kazumasa Oida discussed the user interface of the system and contributed to writing the paper. All authors had approved the final version.

REFERENCES

[1] M. Payer, "Control-flow hijacking: Are we making progress?" in *Proc. the 2017 ACM on Asia Conference on Computer and Communications Security* ACM, 2017, p. 4.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[3] H. Shacham *et al*., "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proc. ACM Conference on Computer and Communications Security*, New York, 2007, pp. 552–561.

[4] U.S. Department of Labor. Occupational outlook handbook: Information security analysists. [Online]. Available: https://www.bls.gov/ooh/computer-and-information-technology/information-security-analysts.htm

[5] Metasploit: Penetration Testing Software. [Online]. Available: https://www.metasploit.com/

[6] M. Graziano, D. Balzarotti, and A. Zidouemba, "Ropmemu: A framework for the analysis of complex code-reuse attacks," in *Proc. the 11th ACM on Asia Conference on Computer and Communications Security*, ACM, 2016, pp. 47–58.

[7] Facebook, Inc. (2017). Platform to host capture the flag competitions. [Online]. Available: https://github.com/facebook/fbctf/

[8] R. Beuran, D. Tang, C. Pham, K.-i. Chinen, Y. Tan, and Y. Shinoda, "Integrated framework for hands-on cybersecurity training: Cytrone," *Computers & Security*, vol. 78, pp. 43–59, 2018.

[9] N.-G. Gilberto, and J. A. Ansari, *Web Penetration Testing with Kali Linux: Explore the Methods and Tools of Ethical Hacking with Kali Linux*, Packt Publishing Ltd, 2018.

[10] C. Wang, "Advanced code reuse attacks against modern defenses," Ph.D. dissertation, 2019.

[11] L. Xu, W. Jia, W. Dong, and Y. Li, "Automatic exploit generation for buffer overflow vulnerabilities," in *Proc. 2018 IEEE International Conference on Software Quality, Reliability and Security Companion* (QRS-C). IEEE, 2018, pp. 463–468.

[12] Y. Wei, S. Luo, J. Zhuge, J. Gao, E. Zheng, B. Li, and L. Pan, "Arg: Automatic Rop chains generation," *IEEE Access*, vol. 7, no. 120, pp. 152–120, 2019.

[13] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proc. the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1689–1706.

[14] PaX Team. (2003). Address Space Layout Randomization. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[15] L. D. Paulson, "New chips stop buffer overflow attacks," *Computer*, vol. 37, no. 10, pp. 28-30, 2004.

[16] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive

detection and prevention of buffer-overflow attacks," in *Proc. USENIX Security Symposium*, 1997, pp. 63–78.
[17] J. Jelinek, "Position Independent Executable (PIE)," http://gcc.gnu.org/ml/gcc-patches/2003-06/msg00140.html, June 2003.
[18] Hardening ELF binaries using Relocation Read-Only (RELRO). (Jan. 2019). [Online]. Available: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro
[19] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 16, 2017.
[20] V. Kuznetsov, L. Szekeres, M. Paea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. 11th Symposium on Operating Systems Design and Implementation*, 2014, pp. 147–163.
[21] M. Liljenstam, J. Liu, D. M. Nicol, Y. Yuan, G. Yan, and C. Grier, "Rinse: The real-time immersive network simulation environment for network security exercises (extended version)," *Simulation*, vol. 82, no. 1, pp. 43–59, 2006.
[22] M. E. Kuhl, M. Sudit, J. Kistner, and K. Costantini, "Cyber attack modeling and simulation for network security analysis," in *Proc. 2007 Winter Simulation Conference*, 2007, pp. 1180–1188.
[23] A. Futoransky, F. Miranda, J. Orlicki, and C. Sarraute, "Simulating cyber-attacks for fun and profit," arXiv preprint arXiv: 1006.1919, 2010.
[24] Linux kernel documentation for procfs. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/proc.txt

Makoto Kose was born in Kagoshima, Japan in 1997. He received a bachelor of computer science and engineering from Fukuoka Institute of Technology, Fukuoka, Japan in 2020. He is now working for Basic Inc. His interest includes binary exploitation and web application security.

Yumi Suenaga was born in Fukuoka, Japan in 1997. She received a bachelor of computer science and engineering from Fukuoka Institute of Technology, Fukuoka, Japan in 2019. She is currently enrolled in the Graduate School of Course of Computer Science and Engineering, Fukuoka Institute of Technology. She is interested in blockchain technologies.

Kazumasa Oida received a bachelor of information science, master of engineering, and doctor of informatics degrees from the University of Tsukuba in 1983, Hokkaido University in 1985, and Kyoto University in 2002, respectively. He worked for the Nippon Telegraph and Telephone Corporation for twenty years as an engineer, where he participated in the development of private network systems. He is currently a professor in the Department of Computer Science and Engineering, Fukuoka Institute of Technology, Japan. His main interests include cybersecurity, social network analysis, and blockchain.