Hindawi
Security and Communication Networks
Volume 2022, Article ID 1251987, 9 pages
https://doi.org/10.1155/2022/1251987

Research Article

BofAEG: Automated Stack Buffer Overflow Vulnerability Detection and Exploit Generation Based on Symbolic Execution and Dynamic Analysis

Shenglin Xu and Yongjun Wang

School of Computer, National University of Defense Technology, Changsha, China

Correspondence should be addressed to Shenglin Xu; xushenglin@nudt.edu.cn

Received 27 March 2022; Revised 4 May 2022; Accepted 26 May 2022; Published 22 June 2022

Academic Editor: Luigi Coppolino

Stack buffer overflow vulnerability is a common software vulnerability that can overwrite function return addresses and hijack program control flow, causing serious system problems. Existing automated exploit generation (AEG) solutions cannot bypass position-independent executable (PIE) exploit mitigation and cannot cope with the situation where the standard output function is not introduced into the program. In this paper, we propose a solution to alleviate the above difficulties: BofAEG, which is based on symbolic execution and dynamic analysis to automatically detect stack buffer overflow vulnerability and generate exploit. We used to capture the flag (CTF) and common vulnerabilities and exposures (CVE) programs for experiments. Results show that BofAEG can not only detect and generate exploits effectively but also implement more exploit techniques and is faster than existing AEG solutions.

1. Introduction

As a pervasive vulnerability in various applications and operating systems, buffer overflow is easily exploited by attackers because languages such as C/C++ do not have instructions to automatically detect buffer overflow, and the time cost of writing code to detect whether a buffer overflow will occur in real time is too great. Stack is one of the most common buffers. First described in detail in 1996, Aleph One [1] described the architecture of the Linux stack and proposed how to use stack-based buffer overflow to implant a piece of code into a process to obtain a shell. Based on this idea, an attacker can create malicious code anywhere in the program's memory, causing the operating system to crash and reject the service, or even control the program's entire execution process, thus obtaining complete control of the target host machine. There were 188 CVEs related to stack buffer overflow in the last 5 years [2].

Since the return address of a function is stored in the stack, an attacker can modify the return address to hijack the control flow of the program by exploiting the stack buffer overflow vulnerability. Therefore, program control flow hijacking is the characteristic of stack overflow vulnerability. CANARY [3] is a stack buffer overflow attack mitigation. When it first enters the function, it places a random number canary on the stack and then determines whether the random number has been changed at the end of the function. If it has been changed, it indicates that an attack has occurred. CANARY effectively prevents attacks that only depend on the stack buffer overflow vulnerability.

Traditional stack buffer overflow attack first arranges malicious code in the controllable area and then hijacks the program control flow to the area to execute malicious code. However, with the implementation of the NX (Non-Executable Memory) mitigation [4], the method is invalid. Because NX makes these controllable areas unexecutable. However, ROP (Return-oriented Programming) [5] attack can effectively bypass NX. It is a new type of attack based on code reuse technology, in which attackers extract instruction fragments (gadgets) from existing libraries or executable files to build malicious code. These instructions themselves are located in executable text segments, and they end with ret instructions to realize the convergence of instruction segment execution flows. However, the ASLR (Address Space Layout Randomization) [6] system mitigation can effectively prevent the attacker from getting the address of these instruction segments by randomizing the layout of linear areas such as heap, stack, and shared library mapping. Further, the PIE (position-independent executable) mitigation makes the program change the load base address every time it is loaded so that the gadgets located in the program itself are also invalid. For programs without PIE protection, the base address of each load is fixed, usually 0×400000 on 64 bits. On the contrary, the base address of programs protected by PIE is different every time they are loaded.

The existing automated vulnerability detection methods, such as fuzzing [7–11], can detect a large number of software vulnerabilities, but cannot verify the exploitability of the generated vulnerabilities. Therefore, automated exploit generation of software vulnerabilities has become a research hotspot in the field of software security. In the field of automated exploit generation (AEG), the accuracy and reliability of symbolic execution techniques make it an important tool for automated program analysis. Symbol execution [12] is a program analysis technology, which can get the input for specific code areas to be executed by analyzing programs. When using symbolic execution to analyze a program, the program will use symbolic values as input, instead of the specific values used in general program execution. When the target code is reached, the analyzer can get the corresponding path constraints, and then get the specific value that can trigger the target code through the constraint solver. Through this method, we can get the input that triggers the stack buffer overflow vulnerability.

In this paper, we propose an effective method BofAEG to automatically detect and generate exploit of ELF x64 binaries with stack buffer overflow vulnerabilities. Because the stack buffer overflow vulnerability is characterized by overwriting the return address and hijacking the program control flow, BofAEG first checks whether there is a win function (backdoor) in the program. Then if the program is protected by PIE, BofAEG attempts to trigger the win function directly or look for address leakage. Next BofAEG reaches the exact location of the stack buffer overflow vulnerability through symbolic execution and obtains all controllable symbolic addresses. Finally, BofAEG applies different exploit techniques according to whether the program contains win functions and is protected by PIE. Because there is no real memory information in symbol execution, dynamic analysis is needed to determine whether the program memory state meets the conditions when applying the exploit technology.

The contributions of this paper are summarized as follows:

(1) We study and summarize the characteristics and difficulties of automated exploit generation of stack buffer overflow vulnerability, and propose a solution, BofAEG.

(2) We use CTF and CVE programs for experiments, and the experiments show that BofAEG can effectively perform automated detection and exploit generation for stack buffer overflow vulnerability. Compared with the existing solutions, BofAEG can deal with more situations and generate exploits more quickly.

(3) We implement six exploit techniques for the stack buffer overflow vulnerability and prove that the stack buffer overflow vulnerability is highly harmful and can bypass most of the existing exploit mitigations.

The remainder of this paper is structured as follows. Section 2 covers related work. Section 3 describes methods for automated stack buffer overflow vulnerability detection and exploit generation, including stack buffer overflow vulnerability and its characteristics, two ways of ROP attack, and the method flow of BofAEG. Section 4 provides comparative experiments between BofAEG and existing AEG solutions. Section 5 discusses the limitations of BofAEG and our future work. Section 6 concludes the paper.

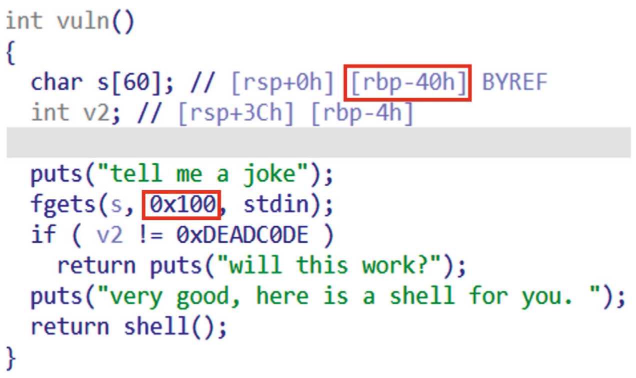
2. Related Work

In recent years, various AEG solutions for different objectives have emerged. The solutions related to heap vulnerability are [13–17]. And [18–21] are solutions for format string vulnerability. In addition to the software level, [22–25] go deep into the Linux kernel. Similarly, AEG solutions exist not only in Linux system, but also in Android system [26–29].

In this paper, we take ELF x64 binary file under Linux system as the target to automatically detect and generate exploit for stack buffer overflow vulnerability. Heelan et al. [30] use binary instrumentation to do taint propagation and gather runtime information and generate exploits by checking whether the EIP register is affected by the taint. AEG [31] first preprocesses the source code to generate bytecode. Based on bytecode, AEG uses conditional symbol execution to find vulnerable functions, objects covered by overflow, and paths that trigger bugs. At the same time, dynamic binary analysis is used to extract runtime information, and finally, exploits are generated. They extended this method to Mayhem [32] to support binaries. CRAX [33] also starts at the crash point, symbolically executing the program to find exploitable states and generate exploits. Padaryan et al. [34] overcome ASLR on this basis. Xu et al. [35] then extended the method to overcome NX. Although the solution considers NX, it cannot solve the problem that ASLR and NX are turned on at the same time and relies on the program to contain the "jmp esp" instruction to complete the exploit. Under the condition that ASLR and NX are turned on at the same time, Zeratool [36] achieves the purpose of hijacking the program control flow to the win function and using ROP attacks to execute commands. However, it does not consider that the program is protected by PIE and relies on the existence of standard output functions (puts, printf, etc.) in the program when using ROP attacks.

3. Automated Stack Buffer Overflow Vulnerability Detection and Exploit Generation

3.1. Stack Buffer Overflow Vulnerability. Stack buffer overflow means that the number of bytes written by the program to a variable in the stack exceeds the number of bytes applied by the variable itself, resulting in the change of variables in its adjacent stack. For the ELF x64 program compiled by C/C++ language, the execution process of the program is the calling process of the function. There is a nested call relationship between functions, and each unfinished function occupies a section of stack space, which is called a stack frame. The stack frame is inserted and popped out of the stack with the call of the function. Local variables and return addresses are stored in the stack frame. A typical stack buffer overflow vulnerability is shown in Figure 1. The local variable *s* only applies for the size of 0 × 40 bytes, but the program calls fgets to read it up to 0 × 100 bytes, resulting in a stack buffer overflow. Its stack frame is shown in Figure 2. Reading 0 × 100 bytes from *s* will overwrite the local variable v2, the caller's stack frame address, and the return address of vuln. This makes the program control flow that should have returned the main function hijacked.



```
int vuln()
{
  char s[60]; // [rsp+0h] [rbp-40h] BYREF
  int v2; // [rsp+3Ch] [rbp-4h]

  puts("tell me a joke");
  fgets(s, 0x100, stdin);
  if ( v2 != 0xDEADC0DE )
    return puts("will this work?");
  puts("very good, here is a shell for you. ");
  return shell();
}
```

Figure 1: vuln function of dctf2021_sanity.

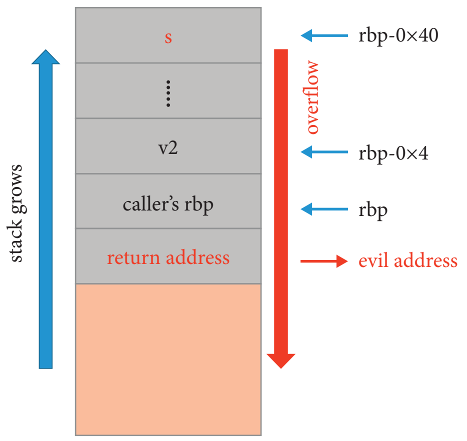
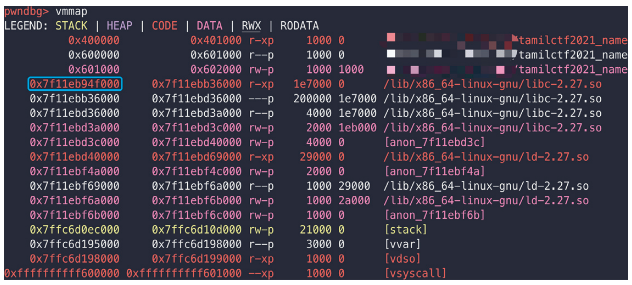


Figure 2: Stack frame of vuln function in dctf2021_sanity.



```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
          0x400000           0x401000 r-xp     1000 0      [redacted]tamilctf2021_name
          0x600000           0x601000 r--p     1000 0      [redacted]tamilctf2021_name
          0x601000           0x602000 rw-p     1000 1000   [redacted]tamilctf2021_name
    0x7f11eb94f000     0x7f11ebb36000 r-xp   1e7000 0      /lib/x86_64-linux-gnu/libc-2.27.so
    0x7f11ebb36000     0x7f11ebd36000 ---p   200000 1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7f11ebd36000     0x7f11ebd3a000 r--p     4000 1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7f11ebd3a000     0x7f11ebd3c000 rw-p     2000 1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7f11ebd3c000     0x7f11ebd40000 rw-p     4000 0      [anon_7f11ebd3c]
    0x7f11ebd40000     0x7f11ebd69000 r-xp    29000 0      /lib/x86_64-linux-gnu/ld-2.27.so
    0x7f11ebf4a000     0x7f11ebf4c000 rw-p     2000 0      [anon_7f11ebf4a]
    0x7f11ebf69000     0x7f11ebf6a000 r--p     1000 29000  /lib/x86_64-linux-gnu/ld-2.27.so
    0x7f11ebf6a000     0x7f11ebf6b000 rw-p     1000 2a000  /lib/x86_64-linux-gnu/ld-2.27.so
    0x7f11ebf6b000     0x7f11ebf6c000 rw-p     1000 0      [anon_7f11ebf6b]
    0x7ffc6d0ec000     0x7ffc6d10d000 rw-p    21000 0      [stack]
    0x7ffc6d195000     0x7ffc6d198000 r--p     3000 0      [vvar]
    0x7ffc6d198000     0x7ffc6d199000 r-xp     1000 0      [vdso]
0xffffffffff600000 0xffffffffff601000 --xp     1000 0      [vsyscall]
```

Figure 3: The memory address mapping of tamilctf2021_name.

3.2. Ret to Libc. Ret to libc is a exploit technique mainly aimed at dynamically linked programs. Because the program is dynamically linked, it will load libc.so at runtime. Libc.so is a dynamically linked version of the runtime glibc in the C language library under Linux. And it contains a large number of functions that can be used, including system, execve. Therefore, the attacker can gain control of the target program by finding the addresses of these functions in memory and using the stack buffer vulnerability to overwrite the return addresse to these functions.

With ASLR and NX turned on, the load base address of libc.so is random, and the attacker cannot directly execute malicious code by using the controllable memory address in the program. Take tamilctf2021_name as an example, the memory address mapping of its runtime is shown in Figure 3. The address selected by the blue box (0x7f11eb94f000) is the load base address of libc.so, which is random every time the program runs. Therefore, like the system function and "/bin/sh" string in libc.so, their addresses are also random. The key to ret to libc is to first obtain the load base address, and then hijack the control flow to the system function located in libc.so.

Therefore, the stack frame layout of applying ret to libc to tamilctf2021_name is shown in Figure 4. The left is the first time to exploit the stack buffer overflow vulnerability. It sets plt_puts as the parameter and jumps to got_puts for execution through pop rdi; ret, a gadget located in the program text segment. The address of the libc.so function introduced by the program is stored in the GOT table, and the PLT table transfers the program control flow to the actual function by referencing the function address in the GOT table. The right shows that after leaking the address, it places the "/bin/sh" string and the address of system in the stack.

The control flow graph of ret to libc is shown in Figure 5. Red addresses represent known program code addresses, and purple addresses represent random addresses in libc.so. It first leaks the address of puts in libc.so by using the puts function introduced in the program, so that the address of libc.so loaded in the program can be obtained. Then it transfers control flow back to vuln_func to trigger the stack buffer overflow vulnerability again. Finally, after getting the load address of libc.so, it sets the function parameter to the address of "/bin/sh", and jumps to the system for execution, where "/bin/sh" and system are located in libc.so. That is, the program control flow is hijacked from the program code segment to libc.so for execution.

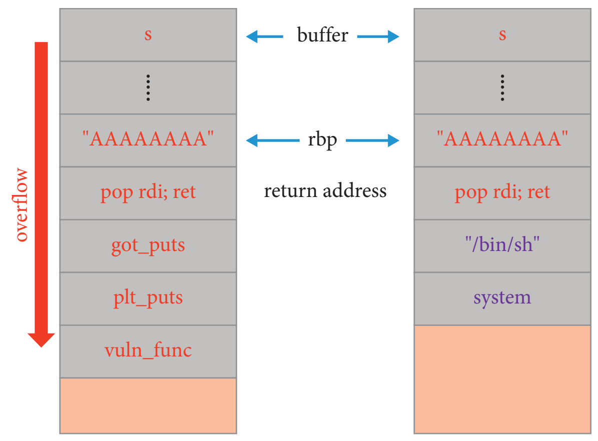


Figure 4: Stack frame layout of ret to libc.

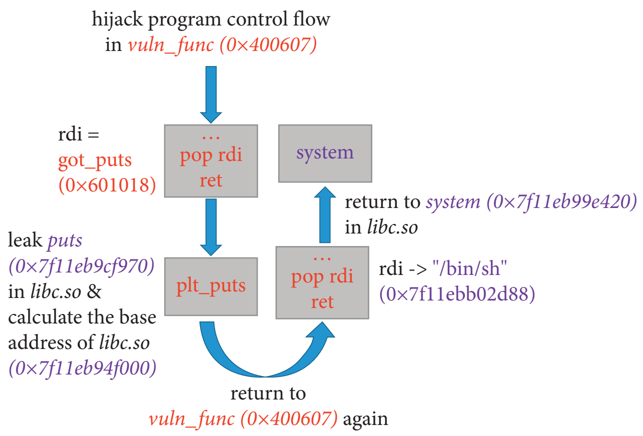

hijack program control flow in vuln_func (0×400607)
rdi = got_puts (0×601018)
… pop rdi ret
system
return to system (0×7f11eb99e420) in libc.so
leak puts (0×7f11eb9cf970) in libc.so & calculate the base address of libc.so (0×7f11eb94f000)
plt_puts
… pop rdi ret
rdi -> "/bin/sh" (0×7f11ebb02d88)
return to vuln_func (0×400607) again

Figure 5: Control flow graph of ret to libc.

3.3. Ret to Dl-Resolve. In the case that the standard output function is not introduced into the program, how can the attacker obtain the load address of libc.so? The answer is that there is no need to obtain it. Under dynamic linking, there are a lot of function references between program modules, and it will take a lot of time to dynamically link all functions before the program starts to execute. Therefore, Linux adopts the delayed binding mechanism, the basic idea of which is that the function is bound (symbol search, relocation, etc.) when it is first used, and it is not bound if it is not used. The address in the GOT table is obtained by relocation through _dl_runtime_resolve when the function is used for the first time. Therefore, the main idea of ret to dl-resolve is to forge the relocation structure of the target function (including function name, relocation table) in the readable and writable memory. Then it uses _dl_runtime_resolve to resolve the target function (such as system) and hijack the program control flow to the target function. This technique is closely related to the ELF file structure [37]. Fortunately, the ret2dl-resolve module of pwntools [38], a well-known exploit framework, makes it easy for us to generate such payloads. In summary, while this technique does not rely on the standard output function, it relies on the standard input function to store fake data into readable and writable memory.

3.4. BofAEG. We use angr [39] for symbolic execution and getting input that triggers the stack buffer overflow vulnerability, and radare2 [40] for dynamic execution and analysis of binary programs. The flow chart of the entire method is shown in Figure 6. First, BofAEG takes a binary program as input and then calls the find_win function to check if there is a win function in the program. The win function refers to a hacker method that bypasses the security control of software and obtains access rights to a program or system from a secret channel. For CTF programs, there are mainly two types of win functions. One is to read the contents of the “flag” file into memory and print it through standard output, as shown in Figure 7; the other is to execute the system function and return an interactive shell, such as Figure 8.

Secondly, BofAEG determines whether the binary is protected by PIE, and makes a choice based on whether it contains the win function. If a win function is included then explore_to_win is called to use symbolic execution to get the input to the win function. Figure 9 shows a program protected by PIE with a flag win function. The local variable s1 has a stack buffer overflow vulnerability, but due to PIE, the attacker cannot obtain the program load address and modify the return address to the flag win function. Fortunately, the flag win function can be executed when the value of the local variable v6 is 0xDEADBEEF. Finally, explore_to_win can get the payload that uses the s1 buffer overflow to modify the v6. However, if the program is protected by PIE and there is no win function, BofAEG calls find_leak to try to obtain the program load address (text_addr) and libc.so load address (libc_addr). Figure 10 shows the leakage of text_addr and libc_addr in the program. Find_leak uses dynamic analysis to identify the features of text_addr and libc_addr in the standard output stream of the program (text_addr: 0 × 555555, libc_addr: 0x7fff). Then, it is judged whether the identified address is in the program or libc.so loading memory address range. With this method, find_leak can get the loading address of the program or libc.so to bypass PIE/ASLR.

Thirdly, BofAEG calls the find_stack_bof function to detect stack buffer overflow vulnerability. find_stack_bof uses symbolic execution to explore paths, and because the shorter the path that triggers the vulnerability, the easier it is to analyze, it uses a breadth-first search strategy during exploration. Since the stack buffer overflow vulnerability is characterized by hijacking the program control flow (that is, the rip register is modified), find_stack_bof checks whether the rip is symbolized after each symbolic state transition. If rip is symbolicated, the program triggers a stack buffer overflow vulnerability. At the same time, find_stack_bof records the controllable symbolic memory address, in preparation for the final use of symbolic constraints to generate exploit.

Finally, the goal of binary exploit is to get shell interaction and execute arbitrary commands (usually to make the program execute system (”/bin/sh”)). Therefore, the get_-shell function implements six exploit techniques based on the stack buffer overflow vulnerability characteristics:

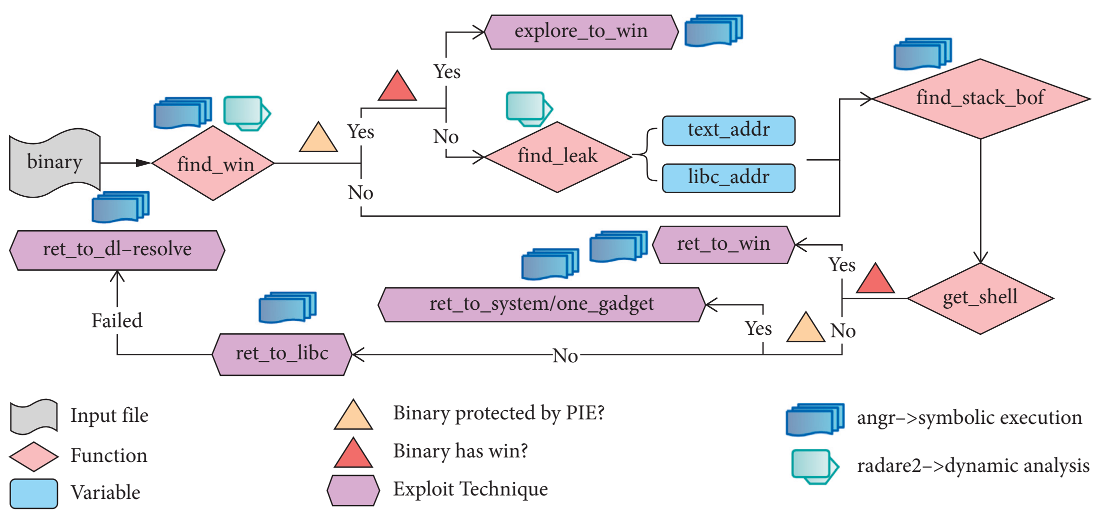
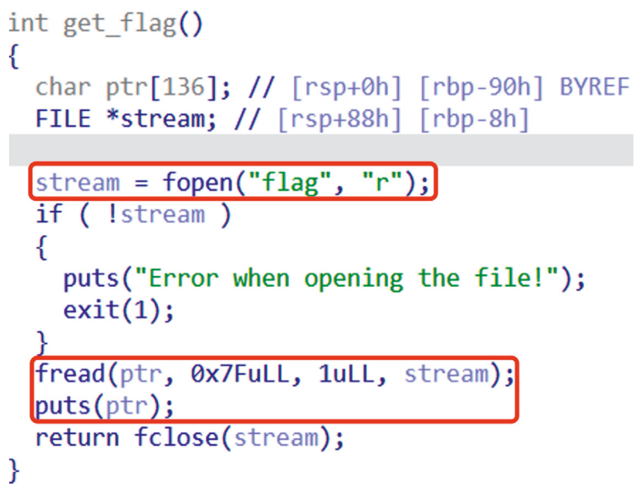
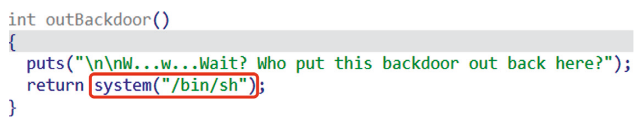


Figure 6: Method flow chart.



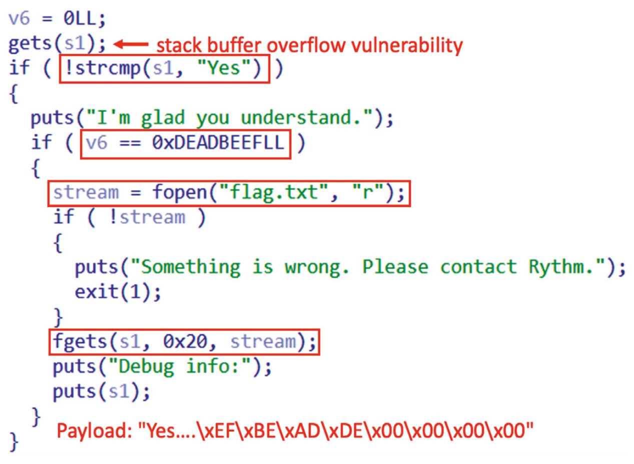
```
int get_flag()
{
  char ptr[136]; // [rsp+0h] [rbp-90h] BYREF
  FILE *stream; // [rsp+88h] [rbp-8h]

  stream = fopen("flag", "r");
  if ( !stream )
  {
    puts("Error when opening the file!");
    exit(1);
  }
  fread(ptr, 0x7FuLL, 1uLL, stream);
  puts(ptr);
  return fclose(stream);
}
```

Figure 7: A flag win function in umdctf2021_jne.



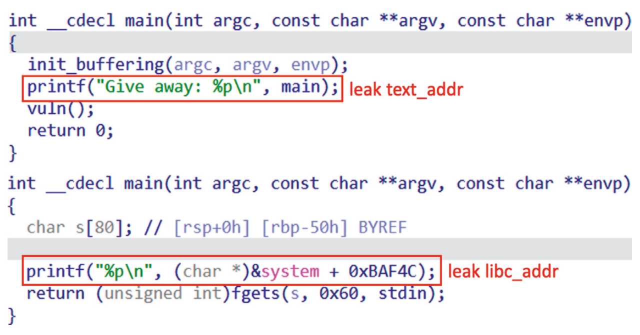
```
int outBackdoor()
{
  puts("\n\nW...w...Wait? Who put this backdoor out back here?");
  return system("/bin/sh");
}
```

Figure 8: A system win function in downunderctf2021_out.



```
v6 = 0LL;
gets(s1); ← stack buffer overflow vulnerability
if ( !strcmp(s1, "Yes") )
{
  puts("I'm glad you understand.");
  if ( v6 == 0xDEADBEEFLL )
  {
    stream = fopen("flag.txt", "r");
    if ( !stream )
    {
      puts("Something is wrong. Please contact Rythm.");
      exit(1);
    }
    fgets(s1, 0x20, stream);
    puts("Debug info:");
    puts(s1);
  }
}  Payload: "Yes....\xEF\xBE\xAD\xDE\x00\x00\x00\x00"
```

Figure 9: main function fragment of lexingtonctf2021_gets.



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  init_buffering(argc, argv, envp);
  printf("Give away: %p\n", main); leak text_addr
  vuln();
  return 0;
}
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s[80]; // [rsp+0h] [rbp-50h] BYREF

  printf("%p\n", (char *)&system + 0xBAF4C); leak libc_addr
  return (unsigned int)fgets(s, 0x60, stdin);
}
```

Figure 10: The leakage of ext_addr and libc_addr in the program.

(1) Explore to win. This exploit technique uses symbol execution technology to try to get the input that triggers the win function when the program is protected by PIE and there is a win function.

(2) Ret to win. This exploit technique achieves the purpose of hijacking the program control flow to the win function by overwriting the return address with the address of the win function when the program has a win function and the address of the win function is known.

(3) Ret to system. This exploit technique is implemented when the program is protected by PIE and there is a libc.so address leak (that is, libc_addr is obtained). Because the program loading address is unknown, the functions introduced in the program cannot be used, so it is necessary to directly use the gadgets and functions in libc.so. Because libc.so contains the "/bin/sh" string and the system function, ret to system can directly use the ROP attack to execute system ("/bin/sh").

(4) Ret to one_gadget. one_gadget [41] is a tool used to find the address in libc.so that can lead to execve

("/bin/sh", NULL, NULL) when the program registers and memory state meet certain conditions. Figure 11 shows one_gadget. Ret to one_gadget uses dynamic analysis to obtain the register and real memory information of the program when the stack buffer overflow vulnerability is triggered, and then obtains the applicable one_gadget address according to this information. Similar to ret to win, ret to one_gadget modifies the return address to the address of the applicable one_gadget.

(5) Ret to libc. As described in 3.2, this technique is one of the most common ways of ROP attacks. When the program load address is known (not protected by PIE or text_addr is obtained) and the standard output function is introduced, this technique first calls the standard output function to leak the libc.so load address. It then triggers the stack buffer overflow vulnerability again by hijacking the control flow to the vulnerable function. Finally, the program control flow is hijacked to the system function in libc.so.

(6) Ret to dl-resolve. As described in 3.3, this technique does not rely on the standard output function, but on the standard input function to store fake data into readable and writable memory. It uses _dl_runtime_resolve to resolve the fake target function into the program and call the target function.

4. Evaluation

Based on the main idea of Section 3, we implemented this method in 700 lines of python, specifically using angr v9.1.11752 for symbolic execution and radare2 v5.6.4 for dynamic analysis. The experiments were carried out on an Ubuntu 18.04 64 bit machine with Intel(R) Core(TM) i7-8700 CPU @ 3.20 GHz, 16G RAM, and 5.4.0 kernel version.

We use CTF and CVE programs with stack buffer overflow vulnerabilities to perform experiments, most of them can be found in CTFTIME [42]. A CTF program can be thought of as a simplified version of a real-world program, used to more concisely and demonstrate the principles of the vulnerability. The difference is that the CTF program is used to test the relevant abilities of the players in the competition, so the vulnerability can be exploited, while the real-world program has higher complexity, and even if there is a vulnerability, it may not be exploitable.

According to the method introduced in Figure 6, we use whether the program is protected by PIE and whether it contains win functions as the selection criteria to demonstrate the effectiveness of BofAEG as much as possible, which is divided into the following four types: 1. Not protected by PIE, but containing win functions. 2. Not protected by PIE and does not contain win functions. 3. It is protected by PIE but does not contain win functions. 4. It is protected by PIE and has win functions. Besides, We assume that the system has ASLR turned on and these programs are protected by NX. Table 1 shows the results of Zeratool [36] and BofAEG on 24 CTF programs and 5 CVE program. BofAEG

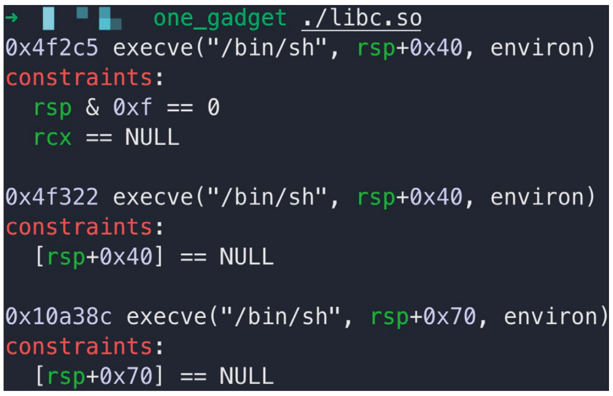

```
→  one_gadget ./libc.so
0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rsp & 0xf == 0
  rcx == NULL

0x4f322 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a38c execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

Figure 11: one_gadget.

can automatically detect stack buffer overflow vulnerabilities and generate exploits for 22 of them, while Zeratool can only successfully exploit 7 of them. And the total time for BofAEG to complete detection and exploitation is less than Zeratool. Among the 7 programs that BofAEG cannot exploit, 2 programs (blue) successfully detect stack buffer overflow vulnerability, and the remaining 5 program (red) cannot detect the vulnerability.

For programs containing win functions, Zeratool only supports function-level win function calls, while BofAEG supports function-level and block-level win function calls. Figure 8 is a function-level win function, while Figure 12 is a block-level win function. If the program control flow is hijacked directly to the main function containing win function, the conditional check of local variable v5 will not be automatically bypassed. Therefore, BofAEG chooses a more fine-grained block-level win function call and directly hijacks the program control flow to address $0 \times 4011EF$ to bypass the condition check.

For programs that do not contain win functions but introduce standard output functions, both Zeratool and BofAEG can better implement the ret to libc exploit technique. However, BofAEG can implement the ret to dl-resolve exploit technique even if the program does not introduce a standard output function. In addition, for programs protected by PIE, BofAEG can effectively find address leaks and apply exploit techniques.

5. Discussion

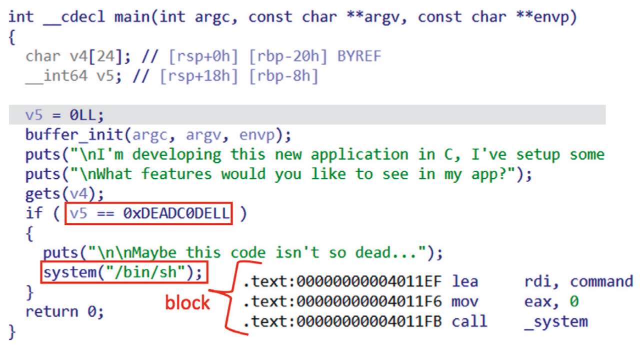
BofAEG overcomes some of the difficulties faced by current research on automated detection and exploit generation of stack buffer overflow vulnerability. However, BofAEG still has some limitations and issues to overcome, so we discuss these limitations and future work.

5.1. Limitations. It can be known from the experiments that BofAEG still has programs that cannot be successfully detected and exploited, and these programs illustrate the limitations of BofAEG.

(1) Limitations of symbolic execution. Since BofAEG uses symbolic execution to automatically detect stack

Table 1: Results of Zeratool and BofAEG on CTF and CVE programs.

Program	NX	CANARY	PIE	Win	Zeratool	BofAEG	Exp Tech
redpwnctf2020_coffer	✓	✕	✕	✓	N/A	6 s	Ret to system win
csictf2020_pwn0x1	✓	✕	✕	✓	N/A	5 s	Ret to system win
csictf2020_pwn0x2	✓	✕	✕	✓	N/A	6 s	Ret to system win
csictf2020_pwn0x3	✓	✕	✕	✓	7 s	6 s	Ret to system win
dctf2021_sanity	✓	✕	✕	✓	N/A	4 s	Ret to system win
umdctf2021_jne	✓	✕	✕	✓	7 s	6 s	Ret to flag win
csawctf2021_password	✓	✕	✕	✓	N/A	60 s	Ret to system win
h@cktivityctf2021_retcheck	✓	✕	✕	✓	N/A	8 s	Ret to flag win
downunderctf2021_deadcode	✓	✕	✕	✓	N/A	6 s	Ret to system win
downunderctf2021_out	✓	✕	✕	✓	6 s	5 s	Ret to system win
csawctf2020_roppity	✓	✕	✕	✕	30 s	6 s	Ret to libc
downunderctf2020_return	✓	✕	✕	✕	30 s	7 s	Ret to libc
dctf2021_babybof	✓	✕	✕	✕	37 s	5 s	Ret to libc
umdctf2021_jnw	✓	✕	✕	✕	27 s	6 s	Ret to libc
tamilctf2021_name	✓	✕	✕	✕	N/A	4 s	Ret to libc
dicectf2021_babyrop	✓	✕	✕	✕	N/A	6 s	Ret to dl-resolve
utctf2021_resolve	✓	✕	✕	✕	N/A	6 s	Ret to dl-resolve
nahamconctf2021_smol	✓	✕	✕	✕	N/A	4 s	Ret to dl-resolve
sharkyctf2020_give	✓	✕	✓	✕	N/A	4 s	text_addr and ret to libc
wpictf2020_dorsia1	✓	✕	✓	✕	N/A	4 s	libc_addr and ret to one_gadget
dctf2021_hotelrop	✓	✕	✓	✕	N/A	5 s	text_addr and ret to libc
lexingtonctf2021_gets	✓	✕	✓	✓	N/A	11 s	Explore to flag win
lexingtonctf2021_madlibs	✓	✕	✕	✓	N/A	N/A	N/A
cyberctf2021_harvester	✓	✓	✓	✕	N/A	N/A	N/A
Cve-2004-1257_abc2mtex	✓	✕	✕	✕	N/A	N/A	N/A
Cve-2011-1938_php	✓	✕	✕	✕	N/A	N/A	N/A
Cve-2012-4409_mcrypt	✓	✕	✕	✕	N/A	N/A	N/A
Cve-2013-2028_nginx	✓	✕	✕	✕	N/A	N/A	N/A
Cve-2017-13089_wget	✓	✕	✕	✕	N/A	N/A	N/A



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4[24]; // [rsp+0h] [rbp-20h] BYREF
  __int64 v5; // [rsp+18h] [rbp-8h]

  v5 = 0LL;
  buffer_init(argc, argv, envp);
  puts("\nI'm developing this new application in C, I've setup some
  puts("\nWhat features would you like to see in my app?");
  gets(v4);
  if ( v5 == 0xDEADC0DELL )
  {
    puts("\n\nMaybe this code isn't so dead...");
    system("/bin/sh");
  }
  return 0;
}
```

```
.text:00000000004011EF lea     rdi, command
.text:00000000004011F6 mov     eax, 0
.text:00000000004011FB call    _system
```

Figure 12: A block-level system win function in downunderctf2021_deadcode.

buffer overflow vulnerability, obtain input that accurately reaches the vulnerability point, and generate the final exploit, it is impossible to avoid path explosion and constraint solving problems faced by symbolic execution. For example, because the input in lexingtonctf2021_madlibs is formatted and spliced by multiple "%s" in sprintf, the symbol constraints cannot be expressed correctly; the symbolic execution fails to explore the stack buffer overflow vulnerability in the more complex cve-2004-1257_abc2mtex due to the path explosion problem. Therefore, it is difficult for BofAEG to detect stack buffer overflow vulnerabilities for CVE programs with high complexity.

(2) Limitations of a single vulnerability type. Although the stack buffer overflow vulnerability is highly harmful, CANARY can effectively mitigate the harmfulness of this vulnerability. However, since CANARY has been randomized and stored in the program memory when the program is started if the value of CANARY can be leaked through other vulnerabilities, the stack buffer overflow exploits can also be completed. cyberctf2021_harvester contains both stack buffer overflow and format string vulnerabilities. The attacker needs to leak CANARY through the format string vulnerability, and then use the stack buffer overflow vulnerability to complete the exploitation.

5.2. Future Work. In the future, we will conduct further research on the above limitations to try to overcome them.

(1) Optimize symbolic execution. Use static analysis techniques to guide symbolic execution to mitigate the path explosion problem, while trying to optimize the symbolic processing function so that it can correctly generate symbolic constraints.

(2) Combine multiple vulnerability types. Attempt to exploit in combination with other vulnerability

types, such as format string vulnerability. This requires detecting all of these vulnerabilities and using some technology to manipulate them, which is not simple.

6. Conclusion

In this paper, we introduced the stack buffer overflow vulnerability, which is caused by a program writing more bytes to the buffer variable on the stack than it requested for the buffer size. The stack buffer overflow vulnerability can overwrite the return address of the function to achieve the purpose of hijacking the control flow of the program. Based on this feature, we implemented BofAEG, which uses symbolic execution and dynamic analysis to automatically detect stack buffer overflow vulnerability and generate exploit. The results show that BofAEG can not only detect and generate exploits effectively but also implements more exploit techniques and is faster than existing AEG solutions. The source code of BofAEG and the test cases used in the experiments are available on github [43].

Data Availability

The source code of BofAEG and the test cases used in the experiments are available on github (https://github.com/Kirito0/bof_aeg).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by National Key Research and Development Program of China (no. 2018YFB0204301) and the National Natural Science Foundation of China (no. 61472439).

References

[1] A. One, “Smashing the stack for fun and profit,” *Phrack magazine*, vol. 7, pp. 14–16, 1996.
[2] CVEs, “Related to Stack Buffer Overflow Vulnerabilities,” 2022, https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer%20overflow.
[3] C. Cowan, C. Pu, D. Maier et al., “Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks,” in *USENIX Security Symposium*pp. 63–78, San Antonio, TX, 1998.
[4] Y. Gao, A. Zhou, and L. Liu, *Data-execution Prevention Tech-Nology in Windows System*, Information Security & Communications Privacy, 2013.
[5] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming,” *ACM Transactions on Information and System Security*, vol. 15, no. 1, pp. 1–34, 2012.
[6] P. Team, “Pax Address Space Layout Randomization (Aslr),” 2003, https://pax.grsecurity.net/docs/aslr.txt.
[7] M. Zalewski, “American Fuzzy Lop,” 2017, https://lcamtuf.coredump.cx/afl/.
[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, *Vuzzer: Application-Aware Evolutionary Fuzzing*, pp. 1–14, NDSS, 2017.
[9] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “{QSYM}: a practical concolic execution engine tailored for hybrid fuzzing,” in *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pp. 745–761, Baltimore, MD, May 2018.
[10] P. Chen and H. Chen, “Angora: efficient fuzzing by principled search,” in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711–725, IEEE, San Francisco, CA, USA, May 2018.
[11] Y. Chen, Y. Jiang, F. Ma et al., “{EnFuzz}: ensemble fuzzing with seed synchronization among diverse fuzzers,” in *28th USENIX Security Symposium*vol. 19, pp. 1967–1983, USENIX Security, 2019b.
[12] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[13] D. Repel, J. Kinder, and L. Cavallaro, “Modular synthesis of heapexploits,” in *Proceedings of the 2017 WorkshoponProgramming Languages and Analysis for Security*, pp. 25–35, Dallas, TX, USA, October 2017.
[14] D. Liu, J. Wang, Z. Rong et al., “Pangr: a behavior-based automatic vulnerability detection and exploitation framework,” in *Proceedings of the 2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)*, pp. 705–712, IEEE, New York, NY, USA, August 2018.
[15] Y. Wang, C. Zhang, X. Xiang et al., “Revery: from proof-of-concept to exploitable,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1914–1927, Toronto, Canada, October 2018.
[16] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “{HeapHopper}: bringing bounded model checking to heap implementation security,” in *Proceedings of the 27th USENIX Security Sympo-Sium (USENIX Security 18)*, pp. 99–116, Baltimore, MD, May 2018.
[17] Z. Zhao, Y. Wang, and X. Gong, “Haepg: an automatic multi-hop exploitation generation framework,” in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 89–109, Springer, July 2020.
[18] Y. Wan, Y. Deng, D. Shi, L. Cheng, and Y. Zhang, “Automatic exploit generation system based on symbolic execution,” *Computer Systems & Applications*, vol. 26, pp. 44–52, 2017.
[19] H. Zhao, S. Huang, Z. Deng, and H. Huang, “Automatic detection and test cases generation of format string vulnerability based on symbol execution,” in *Application Research of Computers*, China Academic Journal Electronic Publishing House, Chengdu, China, 2019.
[20] R. Wang, Z. Pan, F. Shi, and M. Zhang, “Aemb: an automated exploit mitigation bypassing solution,” *Applied Sciences*, vol. 11, no. 20, p. 9727, 2021.
[21] R. Wang, M. Zhang, H. Huang, and S. Yi, “Research on automatic exploit generation method of format string vulnerability based on symbolic execution,” in *Journal Of Air Force Engineering University (Natural Science Edition)*China Academic Journal Electronic Publishing House, Xi’an, China, 2021.
[22] Y. Chen and X. Xing, “Slake: facilitating slab manipulation for exploiting vulnerabilities in the linux kernel,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and*

Communications Security, pp. 1707–1722, London, UK, November 2019.

[23] W. Wu, Y. Chen, X. Xing, and W. Zou, "{KEPLER}: facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pp. 1187–1204, Clara, CA, May 2019.

[24] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1165–1184, New York, USA, October 2020.

[25] W. Chen, X. Zou, G. Li, and Z. Qian, "{KOOBE}: towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, pp. 1093–1110, Boston, MA, USA, August 2020.

[26] L. Luo, Q. Zeng, C. Cao et al., "System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 225–238, USA, June 2017.

[27] M. Zhou, F. Zeng, Y. Zhang, C. Lv, Z. Chen, and G. Chen, "Automatic generation of capability leaks' exploits for android applications," in *Proceedings of the 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 291–295, IEEE, Xi'an, China, April 2019.

[28] L. Luo, Q. Zeng, C. Cao et al., "Tainting-assisted and context-migrated symbolic execution of android framework for vulnerability discovery and exploit generation," *IEEE Transactions on Mobile Computing*, vol. 19, no. 12, pp. 2946–2964, 2020.

[29] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu, "Gui-squatting attack: automated generation of android phishing apps," *IEEE Transactions on Dependable and Secure Computing*, p. 1, 2019.

[30] S. Heelan, *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*, Ph.D. thesis. University of, Oxford, 2009.

[31] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[32] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pp. 380–394, IEEE, San Francisco, CA, USA, May 2012.

[33] S. K. Huang, M. H. Huang, P. Y. Huang, C. W. Lai, H. L. Lu, and W. M. Leong, "Crax: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations," in *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, pp. 78–87, IEEE, Gaithersburg, MD, USA, June 2012.

[34] V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov, "Automated exploit generation for stack buffer overflow vulnerabilities," *Programming and Computer Software*, vol. 41, no. 6, pp. 373–380, 2015.

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

[36] Zeratool, "Github Repository," https://github.com/ChrisTheCoolHut/Zeratool.

[37] E. L. F. # File_Structure, "Wiki," https://wiki.osdev.org/ELF#File_Structure.

[38] pwntools, "Github repository," https://github.com/Gallopsled/pwntools.

[39] Y. Shoshitaishvili, R. Wang, C. Salls et al., "Sok:(state of) the art of war: offensive techniques in binary analysis," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, IEEE, San Jose, CA, USA, May 2016.

[40] radare2, "Github repository," https://github.com/radareorg/radare2.

[41] one_gadget, "Github Repository," https://github.com/david942j/one_gadget.

[42] CTFTIME: https://ctftime.org/.

[43] bof_aeg, "Github repository," https://github.com/Kirito0/bof_aeg.