International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2022, 12(3): 331–350, doi: 10.21655/ijsi.1673-7288.00290

Research Article

# Exploit-oriented Automated Information Leakage

Songtao Yang (杨松涛)[1], Kaixiang Chen (陈凯翔)[2], Zhun Wang (王准)[2], Chao Zhang (张超)[2]

[1] (Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)
[2] (Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China)
Corresponding author: Chao Zhang, chaoz@tsinghua.edu.cn

**Abstract** Automatic Exploit Generation (AEG) has become one of the most important ways to demonstrate the exploitability of vulnerabilities. However, state-of-the-art AEG solutions in general assume the target system has no mitigations deployed, which is not true in modern operating systems since they often deploy mitigations like Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR). This paper presents the automatic solution EoLeak that can exploit heap vulnerabilities to leak sensitive data and bypass ASLR and DEP at the same time. At a high level, EoLeak analyzes the program execution trace of the Proof-Of-Concept (POC) input that triggers the heap vulnerabilities, characterizes the memory profile from the trace, locates sensitive data (e.g., code pointers), constructs leakage primitives that disclose sensitive data, and generates exploits for the entire process when possible. We have implemented a prototype of EoLeak and evaluated it on a set of Capture The Flag (CTF) binary programs and several real-world applications. Evaluation results reveal that EoLeak is effective at leaking data and generating exploits.

**Keywords** information leakage; automatic exploit generation; dynamic analysis; taint analysis; memory profiling

**Citation** Yang ST, Chen KX, Wang Z, Zhang C. Exploit-oriented automated information leakage, *International Journal of Software and Informatics*, 2022, 12(3): 331–350. http://www.ijsi.org/1673-7288/290.htm

## 1 Introduction

Automatic Exploit Generation (AEG)[1, 2] has become one of the most important ways to assess vulnerabilities. Given a vulnerable binary program and the Proof-Of-Concept (POC) input that triggers vulnerabilities, an AEG system can automatically analyze a target binary program and generate exploits. AEG can not only support the generation of attacks but also assist in the defense. For instance, software vendors can use AEG tools to assess the threat level of software vulnerabilities and thus identify the urgency of vulnerability repair.

Researchers have proposed many AEG solutions in recent years. The early research[3–8] mainly focused on analyzing the vulnerabilities in stacks and format strings, and the exploit

This is the English version of Chinese article "面向缓解机制评估的自动化信息泄露方法. 软件学报, 2022, 33(6): 2082–2096. doi: 10.13328/j.cnki.jos.006570"

Funding items: National Key Research & Development Program of China (2021YFB2701000); National Natural Science Foundation of China (61972224 and U1736209).
Received 2021-09-05; Revised 2021-10-15; Accepted 2022-01-10; IJSI published online 2022-09-23

patterns were relatively fixed and effective. However, research in recent years[9–15] is more dedicated to the complex types of vulnerabilities, such as heap vulnerabilities, which call for more sophisticated exploit techniques. For example, heap memory layout manipulation is used to construct feasible exploits. Nevertheless, existing AEG solutions pay little regard to the scenarios where defense mechanisms such as vulnerability mitigation are deployed in the target environment, which pose significant challenges to exploits. As a matter of fact, with the widespread deployment of various vulnerability mitigation mechanisms in modern operating systems, it is required for the attackers to break through the defense mechanism in the target environment if they plan to launch practical attacks in modern production and service environments.

The modern operating systems see the extensive deployment of three famous defense mechanisms, i.e., data execution prevention (NX/DEP)[16], stack protection variables (Canary/Cookie)[17], and Address Space Layout Randomization (ASLR)[18]. Specifically, DEP is designed to prevent the data written in memory from being executed as code. After the setting of Canary, the destructive behavior can be detected when the stack buffer overflow vulnerability overwrites the function return address in the stack frame. ASLR is used to randomize the base addresses such as code segments, data segments, and stack and heap in the memory, which restricts attackers' access to the addresses of available important data and code and makes attacks more difficult. In addition, there are other measures for exploit mitigation, including the control-flow integrity solution[19–21], which can also effectively mitigate exploits. Due to various issues with performance and compatibility, however, manufacturers have not yet deployed these protective measures extensively. Therefore, the AEG solution must bypass defense mechanisms such as DEP, ASLR, and Canary.

In current AEG research, some solutions[4, 5, 22, 23] assumed that the target environment and programs do not invoke the defense mechanism; some[8, 12–14] can bypass DEP but cannot resist ASLR, and others[3, 6, 7, 24–26] can bypass ASLR through stack vulnerabilities or bypass ASLR through heap vulnerabilities in the absence of DEP. Currently, no study is conducted on bypassing both DEP and ASLR through heap vulnerabilities.

The key to bypassing ASLR is the leakage of the randomized memory addresses. The ASLR defense deployed in modern operating systems is based on large-size memory segments, including stack, heap, and shared libraries, with relatively coarse granularity. The offset within each segment is a fixed value not subject to randomization. The most effective way to bypass ASLR is to leak the address of a segment, from which we can infer all other addresses in the same segment the attacker needs, which are located at a fixed offset. Therefore, it is necessary to identify a pointer containing the randomized address for disclosure, the common methods for which involve reusing the semantics of the target binary program or building a new output function to trigger the output function to print the address. The attacker can infer other code addresses by this leaked random address and complete the exploit accordingly.

This paper proposes an automatic information leakage system (EoLeak) for mitigation mechanism evaluation, which can simultaneously bypass DEP and ASLR through heap vulnerabilities to achieve effective exploitation. To be specific, EoLeak first makes a dynamic analysis of the runtime memory execution process of the POC that triggers the vulnerability, locates the variables of sensitive information, then constructs automatic information leakage for the variables of sensitive information, and generates exploits based on the leaked information. The solution records all relevant information and searches for possible leakage paths by constructing the memory profile to locate essential variables in the memory. The solution achieves the initial attack on heap vulnerabilities through the heap vulnerability analysis model for the leakage capability building, thus expanding the memory utilization capability. Moreover, it also implements a lightweight dynamic taint strategy, which searches for library function calls with user-controllable parameters by monitoring the transfer operation in the memory buffer, and thus, the cost of instrumentation is reduced. For the final exploit, it follows a similar strategy to build user-controllable library function calls and properly handles the exceptional cases of leakage failure.

We have implemented the system on the QEMU-based record and replay platform PANDA[27] and assessed it through 17 CTF heap vulnerability binary programs and five real-world applications. According to the results, the system has successfully generated 15 automatic leakages and 14 final exploits. The system can automatically analyze and locate the runtime memory address and relevant pointers of sensitive information in real-world applications.

## 2 Case Study

The AEG solution using heap vulnerabilities to bypass DEP and ASLR faces the same problem as the manual construction of exploits by analysts. In this section, we study a real case of heap vulnerability exploit to outline the challenges and the solutions for the leakage system proposed in this paper.

### 2.1 Exploit case for heap vulnerabilities

As shown in Fig. 1, the simplified logic of the target program can be presented as the given code (legitimacy is ensured for the `level` parameter), in which there is a use-after-free heap vulnerability. Neither does the program check the legitimacy of the `buf` pointer to the target heap chunk when calling the heap-chunk `free` function on Line 20, nor does it set the `buf` pointer to `NULL` after `free`ing the heap chunk.

```
1  int sizes[3] = {0x28, 0xfa0, 0x61a80};
2  char* buf[3];
3  bool exist[3];
4
5  void create(int level) {
6    if (!exist[level]) {
7      buf[level] = calloc(1, sizes[level]);
8      read(buf[level], sizes[level]);
9      exist[level] = true;
10   }
11 }
12
13 void renew(int level) {
14   if (exist[level]) {
15     read(buf[level], sizes[level]);
16   }
17 }
18
19 void remove(int level) {
20   free(buf[level]);
21   exist[level] = false;
22 }
```

**Figure 1** Code example of heap vulnerability

POC triggers the vulnerability by constructing the overlapping memory layout of heap chunks. As shown in Fig. 2, the rectangles represent the header and the body of the heap chunks, and the rectangular bars on the side represent the current memory region to which the corresponding heap chunk pointer `buf` points. After executing in sequence (a) creating a small heap chunk, (b) freeing a small heap chunk, (c) creating a large heap chunk, (d) freeing a small heap chunk, (e) creating a small heap chunk, and (f) creating a medium heap chunk, the existence indications matching all heap chunks are set as "`true`", and the pointers of the large heap chunk and small heap chunk overlap, both pointing to the starting address of the small heap chunk in (f). In addition, this makes the memory region that the pointer of the large heap chunk points to overwrite the memory regions, including the small heap chunk, as well as the header and body of the medium heap chunk. Accordingly, writing to the large heap chunk can lead to heap overflow at the header of the medium heap chunk.

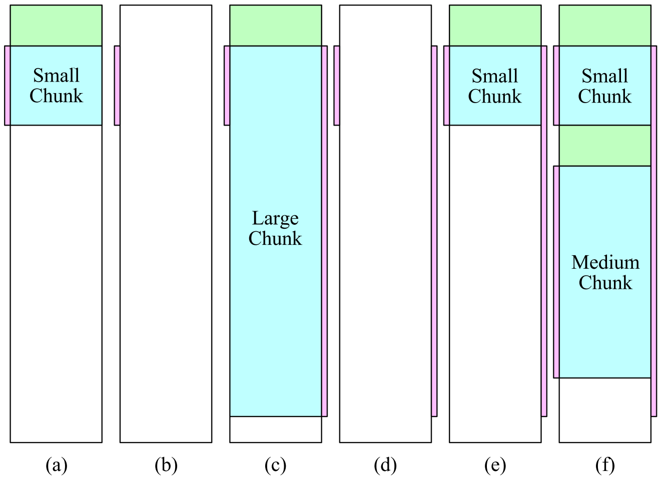


**Figure 2** Memory layout change in the heap during program execution when inputting POC that triggers vulnerabilities

When an attacker attempts to exploit such a vulnerability with the protection of ASLR, the leakage of the randomized address of the libc library would be the first attempt. After a fake heap chunk is created in a small heap chunk, and the safe unlink attack is executed, the pointer to the small heap chunk will be tampered with to a value slightly lower than the address of the pointer itself, with the small heap chunk containing the pointer to the large heap chunk. At this point, it is possible to modify the contents of the small and large heap chunks in turn and then tamper with the value of the pointer of the large heap chunk and its data pointing to the memory region to achieve arbitrary address write. The attacker can use the arbitrary address write primitive to replace a libc library function address in the Global Offset Table (GOT) with the Procedure Linkage Table (PLT) address of the output library function and use any libc library function pointer as the first parameter when triggering the function call, to leak the libc library function address. With this leaked address, the attacker can calculate the actual runtime address of the system function and the `/bin/sh` string in the libc library, write them in the heap chunk, and trigger the system function call to complete the exploit.

## 2.2 Attack model

This paper assumes that three widely deployed defense mechanisms are invoked in the target environment, namely, DEP[16], stack protection variables[17], and ASLR[18]. It also assumes the existence of a common heap vulnerability in the target binary program that can be exploited, e.g., use-after-free or heap overflow.

In addition, this paper assumes that the attacker has POC input that can trigger heap vulnerabilities, which the widely developed vulnerability detection tools can meet. As modern AEG solutions generally allow users to provide the matching exploit templates[28], this paper allows attackers to divide the POC input that triggers vulnerabilities to assist in tracking analysis.

Moreover, this paper also assumes that attackers can command the heap Fengshui capability based on the recent research of heap layout manipulation[9–11] to ensure a relatively convenient and fixed heap layout through POC input that triggers the vulnerability and thus facilitate further analysis.

## 2.3 Research challenges

In the exploitation, an attacker needs to identify a randomized address of the libc library function for leakage and then generate exploits according to the leaked address. Generating both leakage and exploits requires the call of certain library functions (including print data and system functions) with the parameters controlled by the attacker. To this end, the following challenges should be addressed.

(1) Challenge 1: What important sensitive information is worth leaking, and how can it be located in memory? For instance, a randomized address should be acquired through information disclosure to bypass ASLR. When only the target binary and POC are given, it is necessary to build a profile of the memory structure in the execution process.

(2) Challenge 2: How can the located sensitive information be leaked? Users can only influence the execution of a binary program by providing input for the program. To achieve information leakage, an attacker needs to identify the correct input to trigger the function-printing data at a specific location.

(3) Challenge 3: How can exploits be generated according to the leaked information? Even though the information is leaked, there is still a gap before exploit generation; thus, additional efforts are required to overcome such obstacles.

## 2.4 Automatic leakage solution

To address the above challenges, this paper proposes a novel solution, EoLeak, to automatically execute the leakage of sensitive information for exploits and the corresponding exploit generation. In general, the solution involves the dynamic analysis of the execution trace of a binary program, constructs a memory profile of the runtime memory structure, locates valuable data variables, builds user-controllable read-write capabilities, and generates heap exploits that can bypass ASLR and DEP.

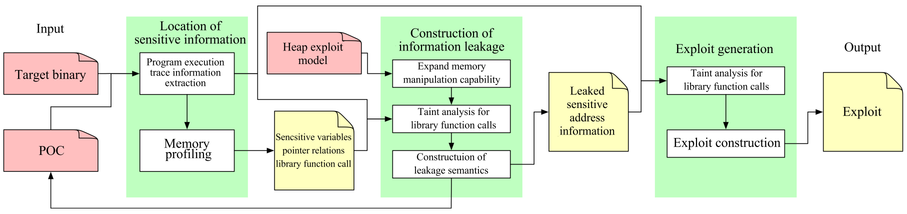
The solution uses the given POC to analyze the running process of the binary and recognizes the pointers and sensitive variables involved in the calculation. It records the relevant memory information and timestamps in the diagram and identifies a nesting pointer chain leading to the specified memory location.

EoLeak first uses heap vulnerabilities based on a heap memory model to build the leakage capability to acquire broader memory manipulation capabilities. Tracking the user input bytes by performing lightweight dynamic taint analysis, the solution analyzes the user-controllable memory region, extracts abstract heap operation information from the execution path of the binary program, and checks whether the corresponding conditions for heap exploits are satisfied in the pre-set list of heap exploit templates. To reduce the expenses on the performance of taint analysis and directly obtain the link between user input and memory data, this paper merely considers the memory transfer operation as the propagation strategy in the taint analysis.

Exploit construction follows a strategy similar to that for leakage capability building. This solution uses arbitrary or controllable memory write to call the target library function with the assigned parameters and execute exploits. When automatic leakage fails or the binary is protected by other advanced defense mechanisms, the solution also attempts to use the corresponding heap exploit technique to generate exploits directly.

## 3 System Design

This section introduces the design details of the EoLeak solution. As shown in Fig. 3, there are three major steps.



Input
Target binary
POC
Location of sensitive information
Program execution trace information extraction
Memory profiling
Heap exploit model
Sencsitive variables pointer relations library function call
Construction of information leakage
Expand memory manipulation capability
Taint analysis for library function calls
Constructuion of leakage semantics
Leaked sensitive address information
Exploit generation
Taint analysis for library function calls
Exploit construction
Output
Exploit

**Figure 3** Overview of the automatic information leakage system

(1) Location of sensitive information. Given a vulnerable binary program and a POC that triggers heap vulnerabilities, the solution first analyzes program execution traces, extracts instruction semantics, and locates the sensitive information by restoring pointers and valuable memory objects. The variable addresses are recorded through memory profiling, and a pointer graph is maintained to search for the leakage paths leading to specific addresses.

(2) Construction of information leakage. With the heap model, the solution achieves vulnerability inference through heap exploit templates to expand the memory manipulation capability and perform lightweight dynamic taint analysis to study the relationship between user input and the parameters for library function calls. The read and write function is established to achieve parameter control, and the solution can construct leakage primitives to print sensitive information.

(3) Exploit generation. The solution generates the final exploits of the target binary according to the leakage information through a strategy similar to building leakage capability. If the library function call of exploits of the selected parameters cannot be triggered, the solution will also use the corresponding template to handle the specific case of heap exploits as a supplement.

### 3.1 Location of sensitive information

The first step of EoLeak is to run a given vulnerable binary program by POC and explore sensitive information during the execution.

#### 3.1.1 *Important data*

Many pointers and memory objects are generated to participate in the calculation during program execution. In particular, the following three kinds of information should be noted:

(1) the code pointers used in program execution, especially those in the writable memory region;

(2) memory object pointers, including heap chunk pointers returned by the heap allocation function;

(3) variables frequently accessed during program execution or those used as the parameters for function calls.

Writable code pointers are precious in exploits. An attacker can hijack the control flow by tampering with these pointers, and writable code pointers can provide a reference for the inference of the base addresses of the shared library in the randomized code space. Sensitive data is usually stored as member variables in memory objects whose address space is under the control of the heap manager. By heap chunk pointers, the runtime heap states can be obtained; thus, the distribution of memory objects in the memory space can be inferred. Some variables, such as the starting pointer of the data structure, usually participate in the operation of subsequent data structures or are often called by the application program interface as parameters. EoLeak considers these variables crucial to the generation of exploits and records the access frequencies of these variables.

### 3.1.2 *Memory profiling*

To identify and locate the above sensitive information, EoLeak constructs a memory profile when analyzing the dynamic program execution trace: it disassembles each instruction to obtain its operation code and operand information and extracts the runtime variable value of the corresponding operand from the execution trace. By analyzing the addressing patterns of registers and memory operands, the solution identifies the pointers, counts the visits, and then adds the data to the global memory pointing mapping that preserves the memory locations and timestamps of variable lifecycles of all records.

To locate the pointer that preserves the addresses of library functions such as libc, EoLeak first searches for all the variable values of runtime addresses within the memory segment of the library. After obtaining the address region information of the memory segment through a separate process monitor, EoLeak records all runtime call and jump targets, checks, and filters for those addresses within the memory segment, and executes a separate verification process to comparatively confirm that each library address corresponds to a library function symbol.

As shown in Fig. 4, the pointer memory profile of the sample program contains the nesting of two pointers: the `pS` pointer at 0x6020b0 points to the memory region 0x602098 containing pointers `pM` and `pH`, which point to separate memory regions. When the memory region the `pH` points to with 0x175e010 as the starting address contains valuable, sensitive data, the leakage of this important data can be constructed by use of two consecutive leakage readings: the value of the `pH` pointer at `*pS`+0x10 is first read, and then the sensitive information variable at `*pH`+0x8 is read. Since the offset in the memory is relatively fixed, it is possible to print each node along the whole chain as long as the address of the first `pS` pointer at 0x6020b0 is known.

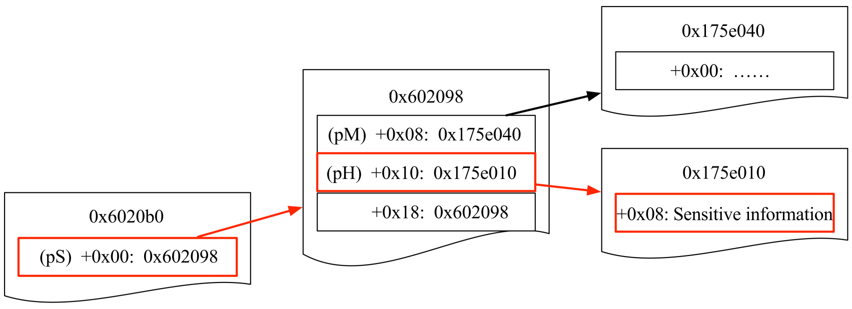


**Figure 4** Runtime pointer memory profiling of the sample vulnerable program

In the collected memory analysis graph, we describe memory as a series of nesting memory regions to which the pointers point, with each pointer located at an offset of a specific memory region. Given a readable pointer with a known address and a time frame of a vulnerability, EoLeak can automatically search for the pointer leakage chain of the specified target variable to achieve the application of this leakage path through the function of continuous arbitrary readings.

## 3.2 Construction of information leakage

After obtaining the location of sensitive information, EoLeak builds the leakage capability to print sensitive information. As explained in the previous subsection, the leakage of a specific variable can be broken down into consecutive arbitrary readings without conflicts. Therefore, the solution builds the leakage function of single arbitrary readings and then stitches it together to form an arbitrary reading chain. More specifically, it analyzes the semantics of the target binary program, builds the leakage capability in combination with heap vulnerabilities, and calls the printout library function with the address of the target variable as a parameter before finally triggering the function call to finish the leakage process.

### 3.2.1 *Expansion of memory manipulation capability*

Programs written by developers usually have strict verification on the data input by the user to avoid the vulnerability attack caused by corrupting the memory. As a result, the target binary program often has insufficient execution semantics to achieve controllable library function calls. Therefore, a necessary step is to provide more exploitable primitives to expand the available range of memory manipulation in combination with heap vulnerability attacks. Attackers can manually adjust the POC input to realize the initial attack on heap vulnerabilities, and EoLeak also provides a solution to automatically derive and build such heap exploit attacks through simple heap models and exploit templates.

The existing heap vulnerability attacks generally come from mature heap exploit techniques, and the attack schemes against different heap managers also become specialized in the development. `ptmalloc2`, adopted by glibc, is one of the most commonly used heap managers, which can provide efficient management of dynamically allocated memory on the heap through the `arena` structure and its `bin` array. In `ptmalloc2`, the memory allocated by each `malloc` has a unified `chunk` structure, which contains the header data for heap structure management that stores the size of the previous heap chunk, i.e., `prev_size`, and the size of this heap chunk, i.e., `size`. The freed heap chunk reuses the first byte of the main part as `FD` and `BK` pointers (pointing to the next and previous freed heap chunks, respectively). Each allocated memory heap chunk is classified as one of the following four categories according to its size and usage state: 1) fast bin, 2) small bin, 3) large bin, and 4) unsorted bin.

For the heap manager `ptmalloc2`, EoLeak abstracts the templates of typical heap exploit schemes, including the conditions required to trigger the vulnerability attack and the exploit effect upon the execution of the attack. To be specific, EoLeak provides a list of heap exploit templates, each of which corresponds to a heap exploit attack scheme, including the conditions required (such as the limit on the size of allocated heap chunks and overflow possibility of the heap chunk header) and effect of attack (such as heap allocation of arbitrary addresses).

Table 1 lists the main exploit templates supported by EoLeak. In the case of the safe unlink attack executed in the sample program, no pointer can be written by the attacker in the memory of the target binary program before the execution, and thus further leakage cannot be achieved. The safe unlink attack, a use-after-free attack, incorrectly frees an already freed small bin or unsorted bin through controllable `FD` and `BK` pointers. In the heap exploit template, EoLeak requires the following conditions to be met for attack execution:

(1) there is a pointer to a heap chunk at a known address;

(2) the heap chunk can be arranged as a freed one that can pass checks, and its FD and BK corresponding offsets have controllable contents;

(3) the neighboring chunks of this chunk can trigger the free operation.

For the target binary program, Condition (1) is the static memory state, which can be obtained through simple inspection via memory profiling; Condition (2) requires that the heap chunk should have the write capability that meets the constraint conditions; Condition (3) requires the free of the neighboring chunks, which can be obtained by the analysis of the program execution trace.

**Table 1** List of heap exploit templates currently supported by EoLeak

| Type of heap vulnerability attack | Attack conditions | Exploit effect | Expandable primitive |
|---|---|---|---|
| Fastbin attack | (1) There are two fastbin heap chunks that can be freed in sequence<br>(2) `FD` can be modified after the free of the first chunk<br>(3) fastbin chunks can be allocated again in sequence | The fastbin chunk freed secondly can be allocated to the overlapping heap address | (1) Expand memory region the pointer points to<br>(2) Create overlapping heap chunks |
| Unlink attack | (1) There is a pointer to a heap chunk at a known address<br>(2) This chunk can be marked as freed chunk, with `FD` and `BK` controllable<br>(3) The neighboring heap chunks of this one can trigger the free operation | The known pointer is set to a specified value | (1) Expand memory region the pointer points to<br>(2) Create controllable pointers |
| Safe unlink attack | (1) There is a pointer to a heap chunk at a known address<br>(2) This chunk can be marked as freed chunk, with `FD` and `BK` controllable<br>(3) The neighboring heap chunks of this one can trigger the free operation | The known pointer is set to the address slightly lower than that of the pointer | (1) Expand memory region the pointer points to<br>(2) Craete controllable pointers |
| Unsorted bin leakage | (1) There is a heap chunk that can be freed into the unsorted bin<br>(2) This heap chunk is located at the tail of the unsorted bin link after its free<br>(3) `FD` can be accessed after the free of this chunk | `FD` pointer of unsorted bin chunk points to an offset of the main arena | Create pointers to the libc address |
| Falsification of memory write | (1) There are two pointers that can overwrite the region they point to<br>(2) One of the pointers can be modified to point to the address lower than that of the other, with the offset not exceeding the length of writable memory | The value of the other pointer can be overwritten once | Create controllable pointers of arbitrary write |

The attack effect of the template can be summarized as follows: the known pointer is set to point to the address slightly lower than that of the pointer.

In the analysis stage, EoLeak has recorded the pointers and their corresponding memory regions. EoLeak maintains a simple heap model that represents the runtime heap layout structure by analyzing the metadata of the heap chunks, including the header information and the doubly linked list pointer structure. The solution extracts abstract heap operation information by analyzing the program execution trace and collects the modification semantics of heap states in the program path. Then, these modifications are combined with the heap model for comparison to check whether they meet the conditions of any heap vulnerability attack template. If all the conditions of a heap vulnerability attack template are met, the solution will execute a heap attack according to the template, verify the effectiveness of the attack, and update the status of POC input and program tracking. If the subsequent analysis suggests the failure of this attack, the solution will return to the state of the previous attack and try other attack templates.

Upon the execution of POC, the sample vulnerable program is executed to the heap layout state as shown in Fig. 2(f), and the solution starts to check whether it meets the conditions of the safe unlink heap attack template. For Condition (1), the solution identifies several heap chunk pointers with a fixed address in the `bss` segment. For Condition 2, it finds the program path, where a `pS` pointer is shown in Figure 4, and the small heap chunk it points to can be filled with the user's fake chunk. The length of the fake chunk can be longer than the allowable value of the small heap chunk, and they are allowed to overwrite the size field and existence bit of the headers of the subsequent neighboring heap chunks through heap overflow. Although both small and large heap chunks can be filled with user-controllable contents, only `pS` meets the requirement that the writable length starting from the pointed-to memory region should be greater than the remaining chunks. Thus, the `prev_size` field and `prev_in_use` bit can be overwritten as the specified content as required (marking the previous fake chunk as "freed" and passing the size matching checks). As to the third condition, another program path meets what it takes to trigger heap chunk free. As all three conditions are satisfied, EoLeak constructs the corresponding attack input according to the template and executes the corresponding program paths, in turn, to successfully update the status of the current program.

### 3.2.2 *Taint analysis for controllable library function calls*

After collecting more memory manipulation primitives, EoLeak attempts to build a single leakage capability. Before obtaining the real address of the library function, we still rely on the target binary program to complete the leakage process. Therefore, it is required to identify the existing output capability in the target binary program and trigger it with the data address to be leaked as a parameter.

First, EoLeak searches for the library functions with printout semantics in the target binary program, which have been collected in memory profiling. Then, it analyzes the program execution trace to determine whether there is a suitable program path to call the printout library function with controllable parameters. This involves two cases of controllable parameters: the parameters directly come from some bytes in the current POC input or from variable transmission in some controllable memory regions in the current program path. The former can undergo simple adjustment by modifying the current input, while the latter requires an additional writing path to write the required address data into the corresponding memory region. If there is no suitable program path to call the library function, EoLeak must first hijack other library function symbols to construct the leakage primitive and then call the function with controllable parameters. In brief, EoLeak should learn which memory regions the user input can influence and which memory regions the function parameters come from.

To track the data flow from user input, in addition to disassembling binary instructions and obtaining runtime data, EoLeak also performs a lightweight taint analysis according to instruction semantics to track the user input. The result of the user input library functions are marked as taint sources, and the parameters for target library function calls are denoted as taint sinks. As dynamic taint transmission is an onerous process, and the heavy analysis workload cannot guarantee accuracy, EoLeak only uses memory transfer operations of no more than first-order arithmetic or bit operations as the taint propagation strategy. In this way, the performance expense can be reduced, focusing on the memory region directly controlled by the user-input bytes. If the attacker controls the target, it is highly probable that the consecutive bytes constituting this address will be transmitted in the memory or execute offset operations. In this case, the taint analysis merely tracks the original user input, reducing the maintenance cost and simplifying the subsequent exploit generation analysis.

### 3.2.3 *Construction of leakage semantics*

There are many possibilities in the results of the above analysis by EoLeak. For instance, multiple program paths of reading semantics can fulfill the leakage capability; the parameters of the print library function may come from multiple controllable memory regions, and each memory region could also support multiple write modes.

EoLeak selects from these candidate components the ones to stitch together to get a complete leakage primitive. As the leakage capability of continuous arbitrary read through the pointer chain requires no conflict between each arbitrary reading leakage, the solution analyzes the execution trace after stitching the leakage path and calculates the side effects it involves, including the writings to other memory locations and additional operations on the heap layout. Moreover, it gives a side-effect score to each generated library function call to measure the impact of side effects, with the leakage path under lighter side effects preferred. Specifically, the solution extracts the memory read-write and allocate-free operations in each execution trace, which are divided into two parts for processing. The memory read-write in the execution trace scores three points for each memory read at different addresses and one point for each group of memory read-write in the execution trace in reverse order (i.e., there is an address execution trace that is read first and then written). For the memory allocation and free operations in the execution trace, the solution scores one point for each memory allocation and free and one point for each group of mismatched allocation and free (matching means a pair of allocation and free operations share the same target address). When selecting a leakage path, the solution prefers the leakage path with a lower score according to memory allocation-free and read-write scores.

The memory profile of the target program records all pointer positions related to the library function address after randomization. EoLeak automatically calculates the possible leakage chain for each library function pointer and attempts to leak any pointer to obtain the runtime library function address after ASLR. The analyst can also manually mark the target sensitive information in the memory profile, and EoLeak tries to build the matching leakage path for this variable.

## 3.3 Exploit generation

After the completion of information leakage, the leaked address of the libc library function and the previously expanded controllable memory range through the template are employed to fulfill the utilization of the target binary program. By using and constructing a strategy similar to that for the construction of the leakage semantics, the library function call in the exploit semantics is built, such as system (`/bin/sh`), which will be used as an example below. There are also other library functions, e.g., `execve`, which only call for different combinations of parameters. The process of automatic exploit construction can also involve two steps: the first step is to equip a callable code pointer with the randomized address of the runtime system function, and the second step is to ensure that the first or corresponding parameter is filled with the address of `/bin/sh`.

Since two symbols in the same library always share the same offset in different randomized base addresses, the runtime address of the system function can be calculated through the leaked address of the libc library function. The called code pointer can adopt a typical representative, such as GOT, which preserves the real address of the library function used in the program. The libc library contains the string `/bin/sh`, and its address can also be calculated through the leaked address. In some cases, the first parameter called by the system library function cannot be directly controlled by the user input value. Therefore, EoLeak seeks an alternative: the parameter is set to a pointer pointing to a writable memory region, and the writing process is triggered by appending or modifying bytes in the original POC to write the `/bin/sh` string in the target memory region rather than its runtime address. When the address and parameters of the system function are ready, it is possible to tamper with the symbol of the libc library function to trigger the library function call of the system function and utilize the target binary.

EoLeak bypasses the protection of ASLR based on function address leakage. In the leakage and exploit generation stage, shellcode injection is not required. Instead, the initial heap exploit template expands the available memory range. The hijacking library function calls are performed via the analysis of the target address and parameters of controllable library function calls, and attacks are achieved through the reuse of code which can resist DEP. During the heap exploit, it generally does not depend on the stack overflow function, and it can build the corresponding leakage path by specifying the stack protection variable through memory profiling, which enables the resistance to the guard from stack protection variables.

## 4 Solution Implementation

This paper implements the EoLeak system based on the PANDA project. Specifically, we have implemented some of the analysis code in C++ and packaged the whole system as a Python-based server/client architecture.

### 4.1 Record and replay

The analysis of randomization protection has always been an arduous task. The randomized value is only generated in practical execution, before which the relevant randomized program behavior cannot be observed. Although dynamic analysis has solved this problem, the random results vary in each execution, which poses new challenges to researchers.

EoLeak employs the technology that involves recording the program execution trace and replaying it fixedly. All runtime environment variables are generated as usual and recorded in a log file during the recording. When the execution trace is replayed, each instruction shares the same execution behavior as it was originally recorded. A consistent running environment helps researchers easily compare and analyze the execution process of the same program execution trace between multiple replays and identify the randomized library function address and the corresponding memory structure accurately.

### 4.2 PANDA plug-in

Based on the QEMU simulator, PANDA has multiple function callbacks in the TCG lifecycle. For instance, PANDA_CB_INSN_TRANSLATE is called before the basic block is converted to TCG IR for the first time, and PANDA_CB_AFTER_INSN_EXE is triggered after the execution of an instruction. By selectively inserting the analysis logic code through these callback functions, EoLeak can record and analyze the execution process of the target binary program under specific user input. For example, the instructions are disassembled before the execution with the Capstone disassembler, and the recognized pointers are recorded in the global pointer mapping structure, where taint and propagation analysis are carried out.

### 4.3 Server/client architecture

As an analysis platform based on QEMU, PANDA faces major performance problems and high time costs when starting the new virtual machine of the mirror instance of a target operating system. EoLeak needs to dynamically determine the following user input of the target binary program and cannot afford the time cost of frequent startup and shutdown of the virtual machine. To this end, we design EoLeak as a server/client architecture, where the client is a Python controller that receives commands from the server and manipulates the PANDA virtual machine. It wraps common PANDA operations, such as recording the execution process of a given target binary program and the corresponding new program input, replaying the program execution trace and running the analysis plug-in for analysis, and injecting/ejecting the ISO file from the server into/out of the virtual machine. Upon the completion of the analysis operation, the client sends the analysis results back to the server. For a lower time cost in the virtual machine startup, the client retains a running client operating system instance in the background to receive and execute the commands from the server and automatically restart the virtual machine as required.

The server of EoLeak, another Python project, is also the core analyzer of the system. It sends PANDA operations and analysis instructions to the client, receives the analysis results from the client, and drives the automatic vulnerability construction and exploit generation process. During the generation of new program input, the server sends the input with the binary file to the client for another execution trace analysis. By splitting the exploit analysis module and the PANDA execution module into the server and client architectures, EoLeak can be accelerated in parallel by connecting one server with multiple executor clients.

## 5 Assessment and Verification

This section introduces the experiment assessment and verification of the EoLeak system and mainly answers the following research questions:

(1) Question 1: Can EoLeak successfully locate sensitive information?

(2) Question 2: Is the heap attack template adopted by EoLeak effective?

(3) Question 3: Can EoLeak generate automatic leakages of sensitive information and automatic exploits for heap vulnerability programs?

### 5.1 Location of sensitive information

As shown in Table 2, we collected 17 heap vulnerability programs from famous CTF events and websites, and the POCs provided in all cases are not available. The system adopts libc version 2.23, with the invoked defense types represented by SNP in the table (S represents the stack protection variable; N represents DEP, and P represents PIE; the letters indicate the deployed defense, and "–" means that none of the three defenses exist). The full randomization ASLR is employed in the system.

**Table 2** CTF heap vulnerability binaries tested by EoLeak

| Binary program | CTF event | Vulnerability type | Defense deployed | Binary program | CTF event | Vulnerability type | Defense deployed |
|---|---|---|---|---|---|---|---|
| aiRcraft | RCTF'17 | Use-after-free | SNP | RNote2 | RCTF'17 | Heap buffer overflow | SNP |
| b00ks | ASIS'16 | Single byte overflow | NP | SecretHolder | HITCON'16 | Use-after-free | SN |
| babyheap | 0CTF'17 | Heap buffer overflow | SNP | secret-of-my-heart | Pwnable.tw | Single byte overflow | SNP |
| freenote | 0CTF'15 | Release twice | SN | secure_keymanager | SECCON'17 | Heap buffer overflow | SN |
| mario | Defcon'18 | Use-after-free | SNP | SleepHolder | HITCON'16 | Use-after-free | SN |
| message_me | ASIS'18 | Use-after-free | SN | stkof | HITCON'14 | Heap buffer overflow | SN |
| minesweeper | CSAW'17 | Heap buffer overflow | — | vote | N1CTF'18 | Use-after-free | SN |
| note3 | ZCTF'16 | Heap buffer overflow | SN | zone | CSAW'17 | Single byte overflow | SN |
| RNote | ZCTF'17 | Single byte overflow | N | | | | |

To answer Question 1, we collected the pointers of the addresses of the libc library function recorded by EoLeak during dynamic analysis and took them as the representatives of sensitive information. As shown in Table 3, EoLeak has successfully located the addresses of the libc library function for leakage in all CTF binaries. The number of libc addresses identified in the left column represents the number of memory locations of the address values related to the libc library function after address randomization, which was identified under the original POC and was provided by the dynamic analysis module. The number of libc addresses identified after the attack in the right column represents the number of the corresponding memory locations identified after the implementation of the initial heap attack, which reflects the improvement in the memory manipulation capability due to the heap attack. The libc library correlation refers to the addresses with a fixed offset from the base address of the libc library, including libc function symbols and some memory structure fields within the libc data segment that also help leak the libc runtime address.

**Table 3** Identification of addresses related to libc library in CTF binaries

| Binary program | POC length | Number of instructions executed by POC | Number of libc symbols cited | Number of identified pointers of libc address | Number of identified pointers of libc address after attack | Number of heap addresses identified | Number of stack addresses identified | Number of program addresses identified |
|---|---|---|---|---|---|---|---|---|
| aiRcraft | 32 | 1,729 | 17 | 14 | 17 | 2 | 75 | 37 |
| b00ks | 154 | 2,786 | 12 | 12 | 12 | 3 | 82 | 23 |
| babyheap | 75 | 1,342 | 19 | 19 | 22 | 1 | 68 | 2 |
| freenote | 68 | 1,435 | 13 | 11 | 13 | 7 | 72 | 2 |
| mario | 847 | 7,604 | 40 | 44 | 50 | 44 | 79 | 46 |
| message_me | 213 | 1,861 | 15 | 10 | 13 | 2 | 88 | 1 |
| minesweeper | 12 | 2,568 | 28 | 13 | 13 | 5 | 123 | 28 |
| note3 | 138 | 2,873 | 15 | 10 | 10 | 4 | 67 | 1 |
| RNote | 164 | 1,490 | 16 | 10 | 13 | 3 | 60 | 2 |
| RNote2 | 1,374 | 1,677 | 18 | 18 | 22 | 8 | 64 | 2 |
| SecretHolder | 68 | 2,104 | 13 | 8 | 8 | 3 | 52 | 1 |
| secret-of-my-heart | 43 | 2,554 | 19 | 19 | 22 | 1 | 72 | 27 |
| secure_keymanager | 378 | 1,992 | 13 | 8 | 11 | 3 | 68 | 2 |
| SleepHolder | 41 | 2,381 | 16 | 11 | 11 | 3 | 52 | 3 |
| stkof | 18 | 1,636 | 16 | 7 | 10 | 2 | 55 | 2 |
| vote | 36 | 2,442 | 21 | 10 | 10 | 3 | 64 | 3 |
| zone | 86 | 3,783 | 15 | 11 | 11 | 43 | 79 | 1 |

We also assessed the capability of five real-world programs to locate sensitive information. As shown by the results given in Table 4, EoLeak has successfully located the libc address in all applications. It is worth noting that the number of libc library-related addresses identified under the original POC bears no apparent relationship with the size of the target binary file but is roughly the same as the number of libc symbols cited in the target binary. This suggests that almost all possible libc-related addresses used to bypass ASLR are only from GOT. In addition, the libc-related addresses obtained via the expansion of the initial heap attack are not subject to this restriction and are stable options for leakage sources.

**Table 4** Identification of addresses related to libc library in real software

| Real software | Defense deployed | Number of libc symbols cited | Number of identified pointers of libc address | Number of heap addresses identified | Number of stack addresses identified | Number of program addresses identified |
|---|---|---|---|---|---|---|
| ProFTPd | SN | 198 | 122 | 13,451 | 88 | 1,186 |
| nginx | N | 123 | 86 | 6,746 | 138 | 4,675 |
| nullhttpd | SN | 78 | 59 | 27 | 122 | 4 |
| apache | SN | 149 | 92 | 583 | 300 | 1,364 |
| smbclient | SNP | 238 | 171 | 4,836 | 376 | 20 |

## 5.2 Heap attack template

To answer Question 2, we verified the effectiveness of the attack template list adopted by EoLeak on 17 CTF heap vulnerability programs. According to the results presented in Table 5, five of the seventeen programs can well match the Fastbin attack template; eight can match the Unlink attack template; two can achieve memory write by forging data structures; and two cannot match the corresponding attack template, namely that for 88.2% of the target programs, EoLeak can automatically explore the initial attack from the not excessively complex vulnerability trigger points of the heap layout to the heap vulnerabilities that follow the template. The newly identified addresses related to the libc library after the preliminary attack shown in Table 3 reflect the improvement in memory availability due to this attack. We also investigated two program cases that failed to match the preliminary attack template and the situations in real software, the specific reasons for which are introduced in the following subsection.

**Table 5** Heap attack templates applicable to CTF heap vulnerability binaries

| Binary program | Vulnerability type | Applicable attack template | Binary program | Vulnerability type | Applicable attack template |
|---|---|---|---|---|---|
| aiRcraft | Use-after-free | Fastbin attack | RNote2 | Heap buffer overflow | Unlink attack |
| b00ks | Single byte overflow | Fake memory write | SecretHolder | Use-after-free | Unlink attack |
| babyheap | Heap buffer overflow | Fastbin attack | secret-of-my-heart | Single byte overflow | Fastbin attack |
| freenote | Release twice | Unlink attack | secure_keymanager | Heap buffer overflow | Unlink attack |
| mario | Use-after-free | Fastbin attack | SleepHolder | Use-after-free | Unlink attack |
| message_me | Use-after-free | Unlink attack | stkof | Heap buffer overflow | Unlink attack |
| minesweeper | Heap buffer overflow | — | vote | Use-after-free | — |
| note3 | Heap buffer overflow | Unlink attack | zone | Single byte overflow | Fake memory write |
| RNote | Single byte overflow | Fastbin attack | | | |

## 5.3 Leakage and exploit generation

As for Question 3, we used 17 CTF heap vulnerability programs to assess the EoLeak system and attempted to generate leakages and exploits. For each heap vulnerability program, we prepared a POC that can trigger heap vulnerabilities but fails to exploit them.

As shown in Table 6, EoLeak has successfully built leakages for 15 of the 17 heap vulnerability programs, of which 14 have generated final exploits. In other words, EoLeak reports a success rate of up to 88.2% in building the leakage of the libc library function address, with 82.4% of the target programs exploited.

**Table 6** Effect of information leakage construction and exploit generation of CTF heap vulnerability binaries

| Binary program | Information leakage construction | Exploit generation | Binary program | Information leakage construction | Exploit generation |
|---|---|---|---|---|---|
| aiRcraft | ✓ | ✓ | RNote2 | ✓ | ✓ |
| b00ks | ✓ | ✓ | SecretHolder | ✓ | ✓ |
| babyheap | ✓ | ✓ | secret-of-my-heart | ✓ | ✓ |
| freenote | ✓ | ✓ | secure_keymanager | ✓ | ✓ |
| mario | ✓ | | SleepHolder | ✓ | ✓ |
| message_me | ✓ | ✓ | stkof | ✓ | ✓ |
| minesweeper | | | vote | | |
| note3 | ✓ | ✓ | zone | ✓ | ✓ |
| RNote | ✓ | ✓ | | | |

We further investigated three cases that failed to build leakages and one that failed to generate exploits and analyzed the reasons for the failures.

(1) Multi-threading. As a multi-thread program, vote is too complex for the current analysis module. At the moment, EoLeak only supports the analysis of single-thread vulnerability programs. For the same reason, EoLeak is unable to understand the race condition vulnerabilities for the moment.

(2) Custom heap structures. The minesweeper program implements a custom heap structure and manager. As the heap model inference by EoLeak relies on the standard `ptmalloc2`, it cannot process custom heaps, nor can it successfully generate leakages or final exploits for programs.

(3) Heap exploit technique not supported yet. The mario program needs to use the `_IO_file` structure to enable a heap exploit attack, but such exploit technique has not yet received support from the current heap model inference mechanism. Therefore, although `FD` pointers can be used to construct leakages normally, they encounter failures during exploit generation.

(4) Exploit window. In practice, real software paths are complex, and most vulnerabilities only allow a limited attack window and only can write once, making it difficult to reuse. Even if some vulnerabilities meet the conditions for exploit templates, it is hard to construct a conflict-free continuous reading chain due to complex changes in the heap layout, and thus automatic

leakages can hardly be realized.

## 6 Related Work

### 6.1 AEG

Early solutions for AEG did not consider defense mechanisms in general. APEG[22], as the first AEG solution, compares the conditional checks that differ before and after the patch to the target binary and uses symbolic execution to automatically build inputs that fail to pass the conditional checks before the patch. AEG[4] carries out symbolic execution of the source code by control-flow hijacking on the stack to exploit stack overflow and format string vulnerabilities. Mayhem[5] endeavors to generate exploits for stack overflow and format string vulnerabilities in binaries and executes ret2stack-shellcode and ret2libc attacks through the symbolic execution of control-flow hijacking. PolyAEG[23] generates multiple exploits for binaries with abnormal input by analyzing the user-controllable control-flow direction through taints. All these AEG solutions cannot bypass DEP or ASLR.

Some research broke through defense mechanisms but still cannot automatically bypass ASLR. HI-CFG[8] converts benign input into vulnerability points between buffers, generates control flow graphs with buffer information, and searches for buffer transmission through read-write operations. Its core objective is to generate POC input that can cause abnormal states. Manual guidance is required if one wants to generate exploits that can bypass the ASLR defense mechanism. Revery[12], KOOBE[13], and FUZE[14] study how to use fuzzing techniques to generate attacks against heap vulnerabilities. Revery searches the user-mode program for a memory layout similar to the crash execution path, trying to stitch the replacement program path. KOOBE is committed to the out-of-bound write vulnerability in the Linux kernel. FUZE analyzes timestamps used to create and dereference dangling pointers. These solutions dismiss bypassing ASLR as beyond the scope of studies.

Other studies attempt to bypass ASLR, but they can only achieve this through stack exploits rather than heap exploits. Heelan[3] proposed a solution involving symbolic execution to generate known stack overflow vulnerabilities, bypassing ASLR by jumping registers to unrandomized addresses. Q[6] uses a small number of unrandomized code to harden existing exploits and bypass ASLR. It builds return-oriented programming gadgets through the binary analysis platform BAP[29] and matches the proposed QooL language for stack exploits. CRAX[7] generates exploits for large-scale real-world binaries by tainting user-controlled program counters. Although it can use ret2libc and jmp2reg techniques to bypass ASLR on the stack, it cannot do so on heap vulnerabilities. ShellSwap[24] transplants the existing old exploits to an effective new one by symbolic tracking and the combination of code layout repair and path stitching of exploits. It bypasses ASLR by overwriting the instruction pointer with a specific `jmp` instruction with controllable registers; thus, it cannot bypass DEP simultaneously. Using symbolic execution, R2dlAEG[26] explores the use of the return2dl-resolve attack technique to generate exploits that bypass data execution and ASLR for stack overflow vulnerabilities. By detecting multiple exploit chains, KEPLER[25] generates the exploits of return-oriented programming that bypass modern mitigation techniques, and it needs to hijack the control flow and use the control-flow hijacking primitive to build the POC for exploits.

Data flow analysis can naturally resist ASLR protection. FLOWSTITCH[30] automatically generates data-oriented exploits to stitch information leakages or escalate privileges without violating the control-flow integrity. DOP[31] further executes data-oriented programming by building data flow accessories, but they do not bypass ASLR on heap vulnerabilities. BOPC[32] assumes that all the control-flow mitigation is invoked: it searches for the memory layout at a given entry point and sets the memory layout through multiple arbitrary memory write primitives (if ASLR is bypassed, it is required to read the primitives) to make subsequent data flows to meet the required SPL logic.

The AEG research on the interpreter has also gained attention. SHRIKE[9] performs regression tests on PHP to extract heap operations. In addition, it searches for a specific memory layout where the overflow block is adjacent to the target block. Gollum[10] automatically solves the problem of identifying data structure overflow and using it in a certain way. It first checks the available heap layout and then searches for the corresponding heap operation process in a lazy way. As the exact heap allocation address is required, it cannot bypass ASLR.

### 6.2 Taint analysis

As an important method for analyzing program data flow, taint analysis can directly determine whether there is any correlation between the input source and target points. EoLeak makes lightweight dynamic taint analysis to study the relationship between user input and the parameters of the library function call. As static taint analysis can hardly ensure the correctness of the results, current taint analysis systems often adopt dynamic analysis combined with dynamic symbolic execution and fuzzing techniques.

Dytan[33] provides a general framework for taint analysis and adopts dynamic instrumentation for taint analysis, which, however, reports great limitations in taint marking and instrumentation efficiency. Libdft[34], by the shadow memory method, can improve the efficiency of dynamic data flow tracking and build up the practicability of dynamic taint analysis. The existing dynamic taint analysis methods are plagued by the problems of over-taint (too many taints and the wide spread of taint variables) or under-taint (improper taint propagation analysis and that the variables supposed to have been marked have not yet been marked)[35]. Moreover, taint analysis needs to specify the taint propagation rules for each instruction, which heavily depend on the running platform of a target program, and it takes a great engineering cost to migrate them to different platforms. There is no dynamic taint analysis system by far that can analyze 32-bit and 64-bit programs on x86 and x86-64 platforms. To alleviate the over-taint and under-taint problems and enhance the reliability and completeness of dynamic taint analysis, TaintInduce[36] can automatically analyze and obtain propagation rules by using the least semantic information of the corresponding structure, demonstrating a good cross-platform migration capability. Some dynamic taint analysis tools combine binary analysis, such as TEMU[37] and Triton[38].

Given the heavy workload and low efficiency of traditional taint analysis methods, some state-of-the-art research attempt to employ machine learning methods. Specifically, neural networks are used to simulate the input and output of instructions to automatically establish propagation rules for instructions to reduce workload and achieve cross-platform. For instance, Neuraint[39] uses neural networks to directly estimate the correlation between input sources and target points.

## 7 Conclusion

For AEG solutions, it is a realistic challenge to take into account the mitigation strategies deployed in the target system. We propose an effective automatic solution, EoLeak, to bypass ASLR and DEP by heap vulnerabilities. Through dynamic analysis of program execution traces by the POC input that triggers vulnerabilities, EoLeak uses lightweight taint analysis of memory transmission, and a simple heap exploit model to infer possible library function pointers and exploit targets. We have tested EoLeak on 17 CTF heap vulnerability binary programs, 15 of which can automatically build leakages, and 14 can generate final exploits. Additionally, EoLeak can successfully analyze sensitive information and memory controllability in real-world programs.

## References

[1] Liu J, Su PR, Yang M, He L, Zhang Y, Zhu XY, Lin HM. Software and cyber security—A survey. Ruan Jian Xue Bao/Journal of Software, 2018, 29(1): 42–68 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5320.htm [doi: 10.13328/j.cnki.jos.005320]

[2] Zhao SR, Li XJ, Fang Y, Yu YP, Huang WH, Chen K, Su PR, Zhang YQ. A survey on automated exploit generation. Computer Research and Development, 2019, 56(10): 2097–2111 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2019.20190655]

[3] Heelan S. Automatic generation of control flow hijacking exploits for software vulnerabilities [MS. Thesis]. Oxford: University of Oxford, 2009.

[4] Avgerinos T, Cha SK, Rebert A, Schwartz EJ, Woo M, Brumley D. Automatic exploit generation. Communications of the ACM, 2014, 57(2): 74–84. [doi: 10.1145/2560217.2560219]

[5] Cha SK, Avgerinos T, Rebert A, Brumley D. Unleashing mayhem on binary code. Proc. of the 2012 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2012. 380–394. [doi: 10.1109/SP.2012.31]

[6] Schwartz EJ, Avgerinos T, Brumley D. Q: Exploit hardening made easy. Proc. of the 20th USENIX Security Symp. San Francisco: USENIX Association, 2011.

[7] Huang SK, Huang MH, Huang PY, Lai CW, Lu HL, Leong WM. CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. Proc. of the 2012 IEEE 6th International Conference on Software Security and Reliability. Gaithersburg: IEEE, 2012. 78-87. [doi: 10.1109/SERE.2012.20]

[8] Caselden D, Bazhanyuk A, Payer M, Szekeres L, McCamant S, Song D. Transformation-aware exploit generation using a HI-CFG. University Berkeley, Department of Electrical Engineering and Computer Science, 2013. [doi: 10.21236/ADA587051]

[9] Heelan S, Melham T, Kroening D. Automatic heap layout manipulation for exploitation. Proc. of the 27th USENIX Security Symp. Baltimore: USENIX Association, 2018. 763–779.

[10] Heelan S, Melham T, Kroening D. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. Proc. of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London: ACM, 2019. 1689–1706. [doi: 10.1145/3319535.3354224]

[11] Wang Y, Zhang C, Zhao Z, Zhang B, Gong X, Zou W. MAZE: Towards automated heap feng shui. Proc. of the 30th USENIX Security Symp. Virtual: USENIX Association, 2021. 1647–1664.

[12] Wang Y, Zhang C, Xiang X, Zhao Z, Li W, Gong X, Liu B, Chen K, Zou W. Revery: From proof-of-concept to exploitable. Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto: ACM, 2018. 1914–1927. [doi: 10.1145/3243734.3243847]

[13] Chen W, Zou X, Li G, Qian Z. KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. Proc. of the 29th USENIX Security Symp. Virtual: USENIX Association, 2020. 1093–1110.

[14] Wu W, Chen Y, Xu J, Xing X, Gong X, Zou W. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. Proc. of the 27th USENIX Security Symp. Baltimore: USENIX Association, 2018. 781–797.

[15] Ning G, Zhang T, Wen W, Mei R. Study of non-heapspray IE's vulnerability exploitation technique. Netinfo Security, 2014(6): 39–42 (in Chinese with English abstract). [doi: 10.3969/j.issn.1671-1122.2014.06.007]

[16] van de Ven A. New Security Enhancements in Red Hat Enterprise linux v. 3, update 3. Raleigh: Red Hat, 2004.

[17] Cowan C, Pu C, Maier D, Walpole J, Bakke P. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. Proc. of the 7th USENIX Security Symp. San Antonio: USENIX Association, 1998. 63–78.

[18] PaX T. PaX address space layout randomization (ASLR). 2003. http://pax.grsecurity.net/docs/aslr.txt

[19] Abadi M, Budiu M, Erlingsson U, Ligatti J. Control-flow integrity principles, implementations, and

applications. ACM Trans. on Information and System Security, 2009, 13(1): 1–40. [doi: 10.1145/1609956.1609960]

[20] Burow N, Carr SA, Nash J, Larsen P, Franz M, Brunthaler S, Payer M. Control-flow integrity: Precision, security, and performance. ACM Computing Serveys, 2017, 50(1): 1–33. [doi: 10.1145/3054924]

[21] Tice C, Roeder T, Collingbourne P, Checkoway S, Erlingsson U, Lozano L, Pike G. Enforcing forward-edge control-flow integrity in GCC & LLVM. Proc. of the 23rd USENIX Security Symp. San Diego: USENIX Association, 2014. 941–955.

[22] Brumley D, Poosankam P, Song D, Zheng J. Automatic patch-based exploit generation is possible: Techniques and implications. Proc. of the 2008 IEEE Symp. on Security and Privacy. Oakland: IEEE, 2008. 143–157. [doi: 10.1109/SP.2008.17]

[23] Wang M, Su P, Li Q, Ying L, Yang Y, Feng D. Automatic polymorphic exploit generation for software vulnerabilities. In: Zia T, ed. Security and Privacy in Communication Networks. Cham: Springer International Publishing, 2013. 216–233.

[24] Bao T, Wang R, Shoshitaishvili Y, Brumley D. Your exploit is mine: Automatic shellcode transplant for remote exploits. Proc. of the 2017 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2017. 824–839. [doi: 10.1109/SP.2017.67]

[25] Wu W, Chen Y, Xing X, Zou W. KEPLER: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX Association, 2019. 1187–1204.

[26] [26] Fang H, Wu L, Wu Z. Automatic return-to-dl-resolve exploit generation method based on symbolic execution. Computer Science, 2019, 46(2): 127–132 (in Chinese with English abstract). [doi: 10.11896/j.issn.1002-137X.2019.02.020]

[27] Dolan-Gavitt B, Hodosh J, Hulin P, Leek T, Whelan R. Repeatable reverse engineering with PANDA. Proc. of the 5th Program Protection and Reverse Engineering Workshop. Los Angeles: ACM, 2015. [doi: 10.1145/2843859.2843867]

[28] Chen K, Zhang C, Yin T, Chen X, Zhao L. VScape: Assessing and escaping virtual call protections. Proc. of the 30th USENIX Security Symp. Virtual: USENIX Association, 2021. 1719-1736.

[29] Brumley D, Jager I, Avgerinos T, Schwartz EJ. BAP: A binary analysis platform. In: Gopalakrishnan G, eds. Computer Aided Verification. Berlin: Springer, 2011. 463–469. [doi: 10.1007/978-3-642-22110-1_37]

[30] Hu H, Chua ZL, Adrian S, Saxena P, Liang Z. Automatic generation of data-oriented exploits. Proc. of the 24th USENIX Security Symp. Washington: USENIX Association, 2015. 177–192.

[31] Hu H, Shinde S, Adrian S, Chua ZL, Saxena P, Liang Z. Data-oriented programming: On the expressiveness of non-control data attacks. Proc. of the 2016 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2016. 969–986. [doi: 10.1109/SP.2016.62]

[32] Ispoglou KK, AlBassam B, Jaeger T, Payer M. Block oriented programming: Automating data-only attacks. Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto: ACM, 2018. 1868–1882.

[33] Clause J, Li W, Orso A. Dytan: A generic dynamic taint analysis framework. Proc. of the 2007 International Symp. on Software Testing and Analysis. London: ACM, 2007. 196–206. [doi: 10.1145/1273463.1273490]

[34] Kemerlis VP, Portokalidis G, Jee K, Keromytis AD. libdft: Practical dynamic data flow tracking for commodity systems. Proc. of the 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments. New York: ACM, 2012. 121–132.

[35] Schwartz EJ, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (bug might have been afraid to ask). In: Proc. of the 2010 IEEE Symp. on Security and Privacy. Oakland: IEEE, 2010. 317–331. [doi: 10.1109/SP.2010.26]

[36] Chua ZL, Wang Y, Baluta T, Saxena P, Liang Z, Su P. One engine to serve'em all: Inferring taint rules without architectural semantics. In: Proc. of the Network and Distributed System Security Symp. San Diego: Internet Society, 2019. [doi: 10.14722/ndss.2019.23339]

[37] Song D, Brumley D, Yin H, Caballero J, Jager I, Kang MG, Liang Z, Newsome J, Poosankam P, Saxena

P. BitBlaze: A new approach to computer security via binary analysis. In: Sekar R, ed. Information Systems Security. Berlin, Springer, 2008. 1–25. [doi: 10.1007/978-3-540-89862-7_1]

[38] Saudel F, Salwan J. Triton: Concolic execution framework. In: Proc. of the Rennes: Symp. sur lasécurité des Technologies de l' Information et des Communications, 2015.

[39] She D, Chen Y, Shah A, Ray B, Jana S. Neutaint: Efficient dynamic taint analysis with neural networks. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1527–1543. [doi: 10.1109/SP40000.2020.00022]

**Songtao Yang**, doctoral candidate. His research interests include system software security and binary attack and defense.

**Zhun Wang**, engineer. His research interests include software security and binary attack and defense.

**Kaixiang Chen**, doctoral candidate. His research interests include security attack and defense technology.

**Chao Zhang**, Ph.D., long-term associate professor, doctoral supervisor, senior member of CCF. His research interest is software and system security.