Cybersecurity

RESEARCH

Open Access

Revealing the exploitability of heap overflow through PoC analysis

Qintao Shen[1,2*], Guozhu Meng[1,2] and Kai Chen[1,2]

Abstract

The exploitable heap layouts are used to determine the exploitability of heap vulnerabilities in general-purpose applications. Prior studies have focused on using fuzzing-based methods to generate more exploitable heap layouts. However, the exploitable heap layout cannot fully demonstrate the exploitability of a vulnerability, as it is uncertain whether the attacker can control the data covered by the overflow. In this paper, we propose the Heap Overflow Exploitability Evaluator (HOEE), a new approach to automatically reveal the exploitability of heap buffer overflow vulnerabilities by evaluating proof-of-concepts (PoCs) generated by fuzzers. HOEE leverages several techniques to collect dynamic information at runtime and recover heap object layouts in a fine-grained manner. The overflow context is carefully analyzed to determine whether the sensitive pointer is corrupted, tainted, or critically used. We evaluate HOEE on 34 real-world CVE vulnerabilities from 16 general-purpose programs. The results demonstrate that HOEE accurately identifies the key factors for developing exploits in vulnerable contexts and correctly recognizes the behavior of overflow.

Keywords Exploitability evaluation, Proof-of-concept, Static analysis, Taint analysis, Heap object recovery

Introduction

Heap-based overflow (heap overflow) vulnerabilities have posed great threats to the modern systems, by which attackers can steal confidential data, execute arbitrary code, elevate privilege and so on (Lin et al. 2022; Chen et al. 2020; Kiriansky and Waldspurger 2018; Daniel et al. 2008). Attributed to the evolving fuzzing techniques and tools (e.g., AFL (https://lcamtuf.coredump.cx/afl/) and OSS-fuzz Serebryany 2017), many heap-based vulnerabilities are constantly discovered and reported every year. However, the Proof-of-Concepts (PoCs) discovered by fuzzers, which are usually crafted inputs leading to system crashes, can only prove the existence of vulnerabilities. Little is known about the impact and potentials for exploitation, i.e., whether these vulnerabilities can be exploited to further consequences. That motivates a line of research on the exploitability analysis (Wang et al. 2021; Wu et al. 2019; Chen and Xing 2019; Chen et al. 2020; Wang et al. 2018; Wu et al. 2018; Heelan et al. 2018, 2019; Yun et al. 2020; Zou et al. 2022; Lin et al. 2022).

It is challenging to evaluate the exploitability of a vulnerability. First of all, it is non-trivial to construct an *exploitable heap layout*, a special layout that leads to the corruption of at least one security-sensitive object in vulnerable programs. This is often one of essential steps for exploitation. Automatic exploit generation (AEG) (Wang et al. 2021; Chen and Xing 2019; Chen et al. 2020; Wang et al. 2018; Wu et al. 2018; Heelan et al. 2018, 2019; Yun et al. 2020) manipulate heap layouts using heap primitives specific to programs like the Linux kernel, PHP/Python interpreters, or CTF programs. However, these heap primitives are explicit, powerful and easy-to-trigger, that are difficult to be utilized easily in general-purpose programs (Zhang et al. 2023).

*Correspondence:
Qintao Shen
shenqintao@iie.ac.cn
[1] SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Springer Open

© The Author(s) 2024. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Prior studies propose advanced fuzzing techniques, aiming to create exploitable PoCs, i.e., *ePoC*. For instance, SCATTER (Zhang et al. 2023) is an approach to generate more high-risk exploitable proof-of-concept (ePoC) inputs, makes it primitive-free by using a distance-guide fuzzer to mutate the crashing inputs directly.

Even with ePoCs of heap vulnerabilities, it still remains challenging for evaluating the exploitability for general-purpose programs considering the *reachability* of exploitable heap layouts. On the one hand, it is unclear whether the critical heap objects can be reached and affected by user inputs. That necessitates a data flow from user inputs to these heap objects, allowing attackers to arbitrarily read or write data. Moreover, the length of affected memory can limit the consequence of exploitation. On the other hand, the vulnerable code snippet that contains heap vulnerabilities should be also reached where attackers can turn the program into a vulnerable state and bound with malicious inputs for exploitation.

In this paper, we propose we present Heap Overflow Exploitability Evaluator (Hoee), a new approach to address the aforementioned challenges. A set of techniques are leveraged for fine-grained analysis of vulnerable context to identify the key factors for developing exploits. First a runtime tracer is designed to collect the dynamic information including executed instruction flow, memory read/write operations and global dynamic sections. Second the type information of the heap objects is extracted from the source code. Basing on the source analysis, we analyze the dereferences to these objects in the binary code. Then, Hoee implements taint memory analyzer for taint analysis and dynamic heap analyzer for heap object recovery. Data from user inputs is labeled as taint sources, the propagation is analyzed along with the concrete heap objects and real execution paths to determine the relationships between user inputs and victim objects. Through an analysis of the instruction flow after vulnerable locations, we can determine whether the corrupted objects could be used in security-sensitive code snippets. After that, the overflow are evaluated in fine-grained in the overflow context detector. Our Hoee can pinpoint whether critical pointers are corrupted and used in sensitive operations at the runtime of a PoC, and enhance the basis for vulnerability exploitability determination.

We implement a prototype of Hoee and evaluate it on 34 real-world heap overflow from 16 open source programs. We analyze the exploitability of a set of PoCs generated by AFL (https://lcamtuf.coredump.cx/afl/) by identifying the key factors to develop exploits in vulnerable context. The coverage similarity between the analyzed PoCs is evaluated to illustrate the effectiveness of exploitability evaluation of the vulnerabilities. Then we evaluate the correctness of our object context analysis. The results show that our object context analysis can correctly identify the behavior of heap overflow. Lastly we evaluate the efficiency of Hoee. Comparing to the execution without instrumentation, the average of performance loss is about 2155 times during trace phase. On average, each gigabyte (GB) of logs takes 770 s to parse, and the size of log size is 1.34 GB, taking about 17.20 min. In summary, we make the following contributions.

- We present the design of Hoee, a novel tool for general-purpose programs to evaluate the exploitability of given heap overflow PoCs. Compared with fuzzing-based work, Hoee can more directly tell analysts the relationship between PoC input and overflow data, and the detailed semantics of overflow objects in the heap layout.
- We propose a new method to analyze the overflow context to determine whether sensitive pointers are corrupted, tainted, and used in sensitive operations. Combining source code static and dynamic analysis, the vulnerable context can be analyzed fine-grained, providing a detailed heap layout for exploitation development.
- We implement Hoee and evaluate it on 34 heap overflow vulnerabilities from 16 real-world open-source programs. The results show that our tool can precisely identify the key factors of exploitation development and correctly describe the details in vulnerable context.

Background and related work

Heap overflow and exploitation

Heap overflow is a system weakness when a program writes more data into heap-allocated memory than expected. Oftentimes, it is attributed to dangerous memory operations (e.g., *memcpy*) and inadequately validation of user-supplied data. The direct consequence of heap vulnerabilities is the corruption of the program and the input that causes crashes accordingly is regarded as a PoC.

Even worse, the type of vulnerabilities offers attackers an opportunity to overwrite data in adjacent memory and further corrupt data. The action that attackers leverage the vulnerabilities to gain unauthorized access is called *exploitation*, and the exploitability of one vulnerability refers to the likelihood of being exploited. Nevertheless, vulnerability is not necessarily equivalent to exploitability. In particular, an attacker may not have plenty of information to exploit a vulnerability, or an exploit cannot circumvent the defensive measures during program runtime. Therefore, the exploitability analysis is significant,

helping to prioritize the found vulnerabilities in terms of their severity.

When heap overflow occurs, the corrupted heap objects are also called ***victim*** objects. Generally speaking, the attackers intend to place the ***victim*** objects with security sensitive bytes adjacent to the vulnerable objects. To achieve this goal, some *heap primitives* in the program code space may be used. ***Heap primitives*** are such code snippets containing one or more heap operations, that can help the attackers to change the heap to the desired layouts.

Modern fuzzers are designed to discover more erroneous behaviors with different heap layouts of the vulnerabilities. An exploitable layouts is a special heap layout that at least one victim object is locating in the overflow region. This makes the first step for vulnerability exploitation. If a PoC can generate an exploitable heap layout, it is regarded as an ***exploitable PoC (ePoC)*** (Zhang et al. 2023). According to state-of-the-art studies (Zou et al. 2022; Lin et al. 2022), the high-risk heap vulnerabilities exhibit three features, i.e., they can lead to (1) *function pointer dereference primitive*, (2) *write primitive* and (3) *any invalid free, including double free and a invalid pointer to be freed*.

Automatic exploit generation

Automatic Exploit Generation (AEG) can provide the most sufficient proof of the exploitability of vulnerabilities. One of the most important tasks in realizing AEG is identifying the heap primitives to construct exploitable heap layouts. SHRIKE (Heelan et al. 2018) proposed an algorithm for resolving the layout manipulation for heap memory corruption exploits, based on pseudo-random black-box search. It can search for one that achieves a desired heap layout and allows an analyst to focus on the higher level concepts in an exploit. Gollum Heelan et al. (2019) proposed a genetic algorithm by searching the test cases for heap primitives to solve heap layout problems of PHP and Python language interpreters. ARCHEAP Yun et al. (2020) designed a method towards the *ptmalloc2* to discover new heap primitives from the heap managers. MAZE Wang et al. (2021) presented a symbolic-execution-based solution for analyzing the heap layouts to identify primitives of use-after-free (UAF) vulnerabilities in the PHP, Python and Perl interpreters. It can generate expected heap layouts. Revery Wang et al. (2018) used the layout-contributor digraph to characterise a vulnerability's memory layout and then uses a layout-oriented fuzzing to explore diverging paths. It designs a control-flow stitching solution to stitch crashing paths and diverging paths with the goal of synthesising EXP inputs.

Another technique to help construct heap layouts or discover heap primitives is fuzzing. FUZE Wu et al. (2018) utilized kernel fuzzing along with symbolic execution to identify exploitation primitives for kernel UAF exploitation. KOOBE Chen et al. (2020) presented a technique to summarise the heap overflow capabilities of PoCs and use fuzzing to generate more candidate exploitation strategies on top of Syzkaller (https://github.com/google/syzkaller), S2E (Chipounov et al. 2011) and Angr (Shoshitaishvili et al. 2016). SLAKE (Chen and Xing 2019) published the common exploitation methods of kernel slab objects by extending the LLVM and Syzkaller.

Recent AEG approaches work well on special software programs such as the Linux kernel, language interpreter, or CTF programs. However, there are limitations when analyzing general-purpose programs without empirically exploiting primitives. The major reason is that it is hard to determine suitable easy-trigger heap primitives due to the more multiplex and sophisticated environment of general-purpose programs. In the absence of heap primitives, automated construction of heap layouts is difficult to complete.

Exploitability analysis of vulnerabilities

In most situations, the exploits of such vulnerabilities are difficult to develop. The exploitability of vulnerabilities can also be evaluated by assessing the key factors to exploitation. From a certain perspective, the more PoCs mean the higher risk of vulnerabilities. Fuzzers like (Böhme et al. 2017; Chen et al. 2018; Lee et al. 2021; Zong et al. 2020) improved the existing fuzzing tools to generate more PoCs towards the specific bug position.

Recent studies focus on the research of the capacities of the PoCs. Evocatio Jiang et al. (2022) leveraged a capacity-guided fuzzer to uncover new bug capacities from one crash test case. SyzScope Zou et al. (2022) defined several high-risk bug impacts from the perspective of whether the vulnerability exploitation primitives can be implemented. In the Linux kernel, the high risk impacts are defined as the following: any UAF and heap out-of-bound bugs that lead to *function pointer dereference primitive* or *write primitive*, or any invalid free bugs. GREBE Lin et al. (2022) aimed to unveil the exploitation potential by detecting various error behaviors through an object-driven kernel fuzzing technique. More error behaviors mean different status of critical variables and memory heap layouts, providing more opportunities to implement the exploitation primitives. With the multiple manifested error behaviors, some more exploitable ones may indicate a higher potential for a kernel bug. SCATTER Zhang et al. (2023) proposed a new primitive-free approach to generate more exploitable Proof-of-Concepts (ePoCs) for heap overflow vulnerabilities in general-purpose programs. It designed a distance-guided fuzzing method to place the security-sensitive victim objects near the

overflow corrupted regions to construct the exploitable heap layouts.

Although these fuzzing-based approaches make it possible to generate heap layouts that are more exploitable, it is still not sufficient to evaluate the real exploitability of vulnerabilities due to a lack of knowledge about whether critical objects can be controlled by user-supplied inputs. These difficulties motivate us to explore a more reliable and accurate approach to determine whether there are enough exploitation primitives.

Motivation

To evaluate the exploitability of a fuzzer-generated PoC, a common method is to debug the PoC for: 1) determining how to construct specific inputs that can affect the target object in program, and 2) identifying suitable vulnerable objects for subsequent exploitation. However, manually debugging the PoCs generated by fuzzers one by one would consume significant time and cost. Therefore, we propose a new method to automatically evaluate these PoCs' exploitability, aiming to determine whether user inputs can affect the victim objects and analyze the victim for exploitable factors at a fine-grained level.

In this section, we first use an example to illustrate the factors that need to be determined for the exploitability analysis of a heap overflow vulnerability and then present the challenges that need to be solved.

A motivation example

Not all heap overflow vulnerabilities can lead to control-flow hijacking. Although the corruption of pointers can have security impacts, there are still gaps in achieving an exploit. The code snippet in Listing 1 shows an example from a real-world program, *CVE-2017-12955* in Exiv2, one of our evaluated programs. The overflow occurs at line 1281 when copying memory using *memcpy*.

Listing 1 Code example from basicio.cpp of exiv2, CVE-2017-12955

```
1277 long MemIo::read(byte* buf, long rcount)
1278 {
1279   long avail = EXV_MAX(p_->size_ - p_->idx_, 0);
1280   long allow = EXV_MIN(rcount, avail);
1281   std::memcpy(buf, &p_->data_[p_->idx_], allow);
1282   p_->idx_ += allow;
1283   if (rcount > avail) p_->eof_ = true;
1284   return allow;
1285 }
```

Several PoCs are provided in the bug report from GitHub issues (Exiv2 cc). These PoCs are generated by fuzzers with memory sanitizer that cause the process to crash when an overflow is detected. The PoC given in the case can be considered an ePoC as a victim object located in the overflow region. However, although the *ePoCs* can generate exploitable heap layouts, which may cause high-risk impacts, there are still gaps in developing the exploits. As described above, an exploit must be able to control the program's runtime behavior precisely to achieve its intention. Actually, the fuzzer-generated *ePoCs* cannot determine which bytes can be hijacked or which objects can be exploited. To achieve a successful exploit, three factors must be confirmed.

- **Whether security-sensitive bytes are corrupted during the overflow** If only some non-pointer data is corrupted, there may be no security impacts on the running process. However, this is not absolute, as some bytes may be mistakenly used as pointers. The security risk is higher if pointers are corrupted. In many papers mentioned in the related work, heap layouts with sensitive pointers corruption are also called exploitable heap layouts. In the running example, a memory block allocated with a size of 20 bytes at 0x9f2740 was found to have been written with data that exceeded its capacity. There are two victim objects in the overflow region. The first one is the chunk metadata of memory blocks used by *ptmalloc*, and the second one is an object defined in the program, with a size of 40 bytes and an address of 0x9f2760, which contains a pointer. The pointer is the source pointer in line 1281, making it security sensitive.
- **Whether more precise memory manipulation can be achieved by inputting full or partial control over the contents of the overwritten data** In Listing 1, there are three arguments *(destination, source, size)* when calling *memcpy*. When overflow occurs, the value of *size* is greater than the length of buffer of *destination*. The content of memory pointed by *source* can be partially affected by the PoC input, making it possible to rewrite some of the addresses in the overflow region precisely.

- **Exploiting corrupted bytes requires an opportunity to be used in at least one security-sensitive operation** The example PoC provided in the report crashes the program because of illegal address access. However, a possible exploitation may be executed if the program can continue running until the corrupted heap block is freed. The result of manual debugging indicates that the source pointer in the *memcpy* can be affected by inputs, so attackers may carefully control the program execution to avoid crashing by crafting suitable input. The adjacent object in the overflow region is used when calling the function *MemIO::read* in next time as the source pointer in *memcpy*, which has been corrupted by the overflow data. As a result, the corrupted pointer crashed the program due to an invalid address visit. We found that the value of the illegal pointer can be directly modified by the input bytes, which means that we can further make the pointer legal to make it possible to realize an arbitrary write or change the program counter.

In summary, during manual debugging to develop an exploit, the main goals are to build exploitable heap layouts, rewrite security-sensitive objects, and dereference the corrupted objects by crafting the inputs. The exploitable objects must be carefully chosen based on their semantics. Although the fuzzer-generated PoCs can construct exploitable heap layouts, it is important to determine whether the corrupted data could be controlled precisely by inputs and whether the corrupted objects can be used again in a security-sensitive context. Unfortunately, it demands a manual effort which is costly and inefficient. Therefore, we aim to propose a new method to achieve these goals automatically.

Challenges

From the motivation example, it shows that a detailed analysis of fuzzer-generated PoCs can greatly assist in evaluating their exploitability. On one hand, fuzzers are widely deployed and generate numerous PoCs continually. Therefore, we need an automated method to comprehend the actual security impacts of these PoCs. On the other hand, if any PoC for a vulnerability is proven exploitable, then the vulnerability is worth further in-depth research. To gather more evidence from the dynamic executions of the PoCs, three challenges need to be addressed.

How to recover the heap layouts and identify the victim objects? To recover the heap layout we should know the boundary information of heap blocks. User-level applications request memory from the system by using heap manager API functions. Generally, most of the general-purpose programs in the real world take the default heap manager supported by *glibc*. However, some of them, such as *FFmpeg* may taken different or self-implemented heap managers for better security or performance.

Not only are the boundaries of allocated memory blocks considered, but also those of user-defined objects. Both kinds of objects can be resolved with enough symbols from the source code. We leverage static analysis and dynamic tracing records to recover the symbols. For common heap management functions like *malloc*, *calloc*, *realloc* and *free*, we recover their symbols based on the known semantics.

How to determine the relationship between the sensitive pointers and user inputs? The handling of user input data in real-world programs may be very complicated. To determine how the user input data is processed in the program, we leverage taint analysis with concrete registers and memory values. It can help to determine whether the variables or pointers are tainted by user inputs, thereby being controlled by attackers.

Whether are the corrupted pointers used in sensitive operations? These selected PoCs can certainly cause some errors at runtime. We only focus on the code path from the location of vulnerable code to the corrupted points of the program in the selected PoC traces. The corrupted bytes are considered security-sensitive if they satisfy any of the following features: (1) the corrupted bytes are the read/write pointers to the memory; (2) the corrupted bytes are used as argument pointers to be passed into functions like *free*; (3) the corrupted bytes are used as function pointers in *call* instructions.

Design

The overview of our approach HOEE is shown in Fig. 1. HOEE contains four main components: *Dynamic Information Collection, Object Type Analysis, Overflow Context Analysis* and *Exploitability Evaluation*. The component *Dynamic Information Collection*, of which the input is a PoC file, collects dynamic information during runtime by the *runtime tracer*, for example, instruction flow, memory read/write, and dynamic sections. The component *Object Type Analysis* takes the vulnerable program as input, aiming to parse the information of object types from source code and generate a type reference map statically. The collected dynamic information and type reference map are used in *Overflow Context Analysis*. We perform taint memory and dynamic heap analysis to build the overflow context detector. In the last component *Exploitability Evaluation of the PoC*, a detailed analysis is conducted including corrupted

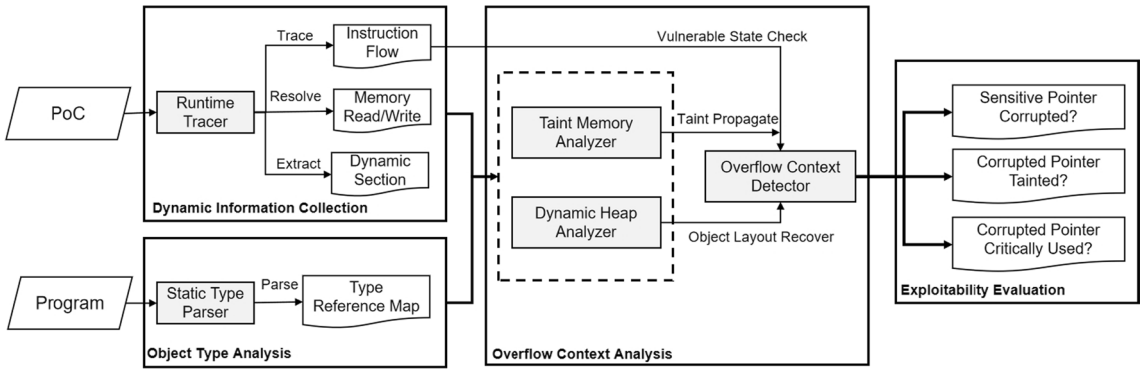


Fig. 1 Overview of HOEE. In exploitability evaluation, the sensitive pointer corrupted means to determine whether the overflow can corrupt sensitive pointer, the corrupted pointer tainted means to determine whether user input can effect the corrupted pointer, the corrupted pointer critically used means to determine whether the corrupted pointer can be dereferenced in security-sensitive code snippets

sensitive pointers, whether these pointers are tainted by user input and and able to be triggered by attackers.

Dynamic information collection

Given one PoC, the overflowed address and the maximal overflow length are unclear. Therefore, we have to first detail the overflow caused by the vulnerability. Generally, the memory of these heap objects is managed by heap managers. The semantics of an object are determined by the code that points to it. If obtaining the process of object generation, usage and deallocation, and the corresponding memory operations, we can figure out the occurrence of overflow and its possible influencing region. Otherwise, there are some dynamically loaded sections for global offset tables (*.got*) and procedure linkage tables (*.plt*), which may be corrupted by the overflow. To collect the required information, we design a runtime tracer module based on the dynamic instrumentation framework supported by *PIN* (Luk et al. 2005). The trace logs is recorded during the re-running of instrumented binary with PoC.

- **Instruction Flow** During program execution, the process states are changed along with the instruction flow. We resolve the values of the referenced registers, the locations in memory, and the related positions in the source code. At the call sites, we collect the values of six registers *rdi, rsi, rdx, rcx, r8* and *r9* used for passing arguments according to the calling convention. At each *ret* instruction, we resolve the value of register *rax* which stores the return value.
- **Memory Read/Write** When the values are loaded from or stored in the main memory, the referenced memory locations and values are collected. So we can recover the heap with concrete values upon the tracing records.
- **Dynamic Section** All dynamically loaded images will be scanned to collect information about the addresses of its internal sections, including the names, start addresses, and sizes.

The instrumentation is designed to work at instruction-level. All instructions, including program space and dynamic libraries, are analyzed and instrumented. The dynamic values of registers and memory accesses are parsed using the *PIN* framework.

Object type analysis

The source code contains the richest code semantic information. With the structure definition in the source code, we can perform a fine-grained analysis of an object byte by byte. At the same time, in binary code, even if debugging symbols exist, more complex analysis is required to determine the type information of an object. For example in Fig. 2-B, the assembly language can be challenging to understand. Since our goal is to know the type of an allocated memory block, the most direct way is to analyze the code flow in source code.

In order to describe the heap object layout in a fine-grained manner, we analyze the source code to learn the object types referenced by the binary code. The input of this component is the source code of the program, and the output records both the data layout of the objects and the relationship between the data types and the source line positions where they are referenced. The first task is to extract the data structures from the source code. We parse each of the data structures and get their data layouts. At the same time, the sensitive data pointers and function pointers are identified. Another task is to determine what data types are referenced by the code.

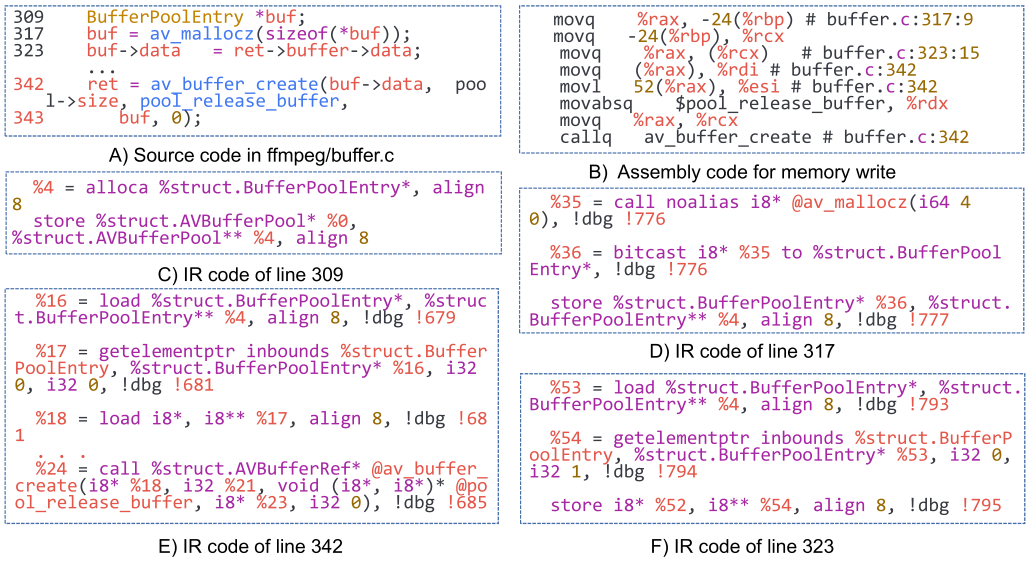


Fig. 2 Source code to the LLVM IR and assembly code. **A** The snippet of example source code. **B** The key assembly code. **C–F** The IR code snippets for each line

As mentioned above, the program may generate objects with their self-implemented heap managers. In order to get more accurate type information of the used data structures, we perform a lightweight value flow analysis on source code. Figure 2 shows an example code snippet from the real-world program *FFmpeg*.

Figure 2A) is the source code snippet and (B) is the corresponding binary code snippet. At line 317, the function *av_mallocz* returns a pointer with type *void* * (i8* in IR). Figure 2F) shows the assignment of an element in a complex data structure, Line 323, and the store type is *i8**. At line 342, the variables are forced to be converted to simple data types before passed as arguments. For the three scenes above, we perform a backward analysis at the *store* and *call* instructions to find their sources and finally get the the source types.

Overflow context analysis

After obtaining dynamic information and type reference map in the previous steps, we conduct an overflow context analysis for the identification of key factors to develop exploits. It proceeds with three phrases as follows.

Taint Memory Analyzer Our taint analysis is also designed to determine whether the overflow corrupted data is related to the user inputs. As we mainly focus on the exploitability of the selected PoC, the taint analysis is performed on the real executed instruction flow of its binary code. First of all, only the code of the program itself will be loaded into the taint analysis engine, excluding the code from third-party libraries such as *libc* and system calls. Secondly, the memory locations of user or external input data are directly identified by resolving the argument and return pointers of taint source functions such as *read*, and then labeled as a taint source. In taint propagation analysis, sink functions such as **memcpy**, *strcpy* will be appropriately handled.

Dynamic Heap Analyzer We rebuild the heap layouts by resolving the calls to the heap functions, and further identify the size and the start address of a memory block. The start addresses can be obtained from the return values of these functions. We calculate the size of the allocated blocks according to the arguments of allocators. We define different processing methods for common heap functions such as *malloc* in *glibc*. For a self-implemented object allocator, we can specify the real memory operating functions to determine their heap semantic information. What we most care about is the allocated block size. For example, the function *av_mallocz(size_t size)* means to allocate *size* bytes memory with all zero values. The function *av_mallocz_array(size_t nmemb, size_t size)* means to allocate *nmemb* blocks of size *size*. The real size of the returned block is *nmemb*size*.

As Fig. 2B shows, we combine the binary code snippets to the source lines by resolving the debug information. To recover the symbols of objects, we mainly focus on the function calls and memory write instructions that use pointers. At the callsites, the pointers are loaded and assigned to the parameter passing registers. Since these registers have already been resolved and logged during the dynamic tracing before, it is easy to map the types and the pointer addresses. However, when multiple

write operations are emitted at the same source lines, it may be troublesome to map the variables to the memory addresses precisely. To simplify the problem, these source lines resulting multiple writes are ignored. In summary, when a pointer is proved with explicit address and type information, the corresponding object is added to our object layout.

Overflow Context Analyzer It is uncertain when a heap overflow will occur. So it is necessary to identify the memory reads and writes as well as sensitive function calls at the vulnerable code locations. Some overflows are caused by out-of-bounds reads or writes. For example, one code snippet, *var = array[i]* or *array[i] = val*, has the index variable *i*, which may be greater than the real length of *array*. Incorrect pointer calculation may also lead to overflow, e.g., *ptr = address; *ptr = val*, in which the loaded address are not correctly checked. To this end, we collect and compare the real types of both the read pointers and the write pointers. According to programming conventions, for writing and reading objects byte-by-byte, the types of the read and written objects should be the same. On the other hand, many heap overflows can be attributed to unchecked parameters when using the *memcpy* function, which memory access locations are indicated by the argument values.

Now we can precisely know the read source, the write destination and the data length at a vulnerable code context. We will compare the actual type of the source pointer to that of the destination pointer. If one of them is out-of-bounds or has been changed to a different object, it is highly likely that an overflow has occurred.

Exploitability evaluation

As described in Sect. 3.2, we aim to achieve the following three goals in the exploitability evaluation: (1) validating whether the overflow has corrupted sensitive bytes in the victim objects. (2) Determining whether corrupted sensitive bytes are tainted by user input. (3) Evaluating critical uses on corrupted pointers after overflow.

The vulnerable code of overflow can be executed multiple times without crashes even security impacts have been issued. To check the vulnerable state, our overflow context analyzer will resolve the source addresses of read and the destination addresses of write. The types of both source objects and destination objects, resolved at object layout recovery, will compare whether they match. The type mismatch is considered as overflow. Then the real boundaries of objects will be calculated based on the pointer addresses and the size of the operations. Heap overflow occurs when the writes are out-of-bounds. Then the over-written regions are carefully analyzed to confirm whether there are sensitive bytes, including the element pointers inner complex structures, and the chunk data of an allocated memory block.

Heap overflow may lead to *write primitive* due to the pointer corruption. When one overwriting occurs, the target address, the length of write primitive, and the data to be written can be analyzed and measured. If the pointer variable pointed to the target address is tainted, data may be written to input-affected address or even arbitrary address. If the pointer variable pointed to the read source address or the pointed memory is tainted, the data to be written may be constrained value or even arbitrary value. For example, the three registers *rdi, rsi, rdx* are referenced as arguments when calling to *memcpy*. If *rdi* is tainted, the write target may be assigned to an arbitrary address. If *rsi* is tainted, the read source may be changed to any address in the process's memory space, including user input data regions. If *rdx* is tainted, the length of copy may be controlled flexibly.

Not all overflow crashes the process immediately. The corrupted sensitive bytes may result in security impacts only when they are used in security sensitive code snippets. If the chunk data is corrupted and later used in heap free operations by heap managers, it may result in an *unlink* exploit. If the function pointer is corrupted and then used to be a target of call instruction, it can lead to a control-flow hijack. If the corrupted data pointer is used to load values from memory, it may leak the data in sensitive regions or crash the process due to address error. If the corrupted data pointer is going to be freed, it will create an opportunity to carry out a double free attack.

Implementation

We have implemented Hoee for the x86_64 programs on the Linux Operating System. Our Hoee consists of the following four major components.

Dynamic Information Collection We implement the tracing module with the PIN framework (Luk et al. 2005). The tracing is began at the calling of function *main*. At the instruction-level instrumentation, we carefully identify the calls and returns, resolve the concrete values of registers. Only the instructions in the program code space are recorded to reduce the storage. We distinguish between threads that belong to different instruction flow for multi-threads processes. No matter which code segment it runs in, the memory accesses will be recorded. It can help to recover all of the concrete values of accessed memory locations.

Object Type Analysis The programs are compiled using clang with option *–save-temps* to generate LLVM bitcode files for the source code. We realize a LLVM-based tool to parse the bitcode files.

Overflow Context Analysis The Taint Memory Analyzer is realized based on *TRITON* (Saudel and Salwan

2015), an open-source dynamic binary analysis engine. We write python scripts to parse the trace logs and recover the heap object layouts. The known heap API functions are easily handled just by adding several lines of code to parse the arguments and the return values.

Exploitability Evaluation A map is built to record each of the allocated objects. At the vulnerable context, the memory access are carefully analyzed along the instruction flow, to identify the victim objects in the overflow region, and whether sensitive pointers are overwrite or tainted. The addresses of corrupted pointers are marked to determine whether they are critically used after overflow triggered.

Evaluation

In this section, we evaluate our HOEE with the vulnerabilities from the real world programs.

Setup

The experiments were performed on a Ubuntu 18.04 server with 128 GB RAM and an Intel(R) Xeon(R) Silver 4110 CPU (2.10 GHz) with 32 cores. The kernel version is 4.15.0, and the system ASLR (Address Space Layout Randomization) is disabled. The version of libc is 2.27.0, in which the default heap manager is *ptmalloc* (posix thread malloc).

The goal of HOEE is to identify more advantageous available factors in the development of exploits, which indicate higher possibility to achieve a successful exploit. We selected 34 CVE vulnerabilities of heap overflow from 16 open-source C/C++ projects. Those vulnerabilities are all reported and classified into heap overflows in their vulnerability reports with at least one PoC given out. During the fuzzing phase, address sanitizer is enabled to detect the overflow. All of the vulnerabilities are fuzzed for at least 24 h to generate more crashes. Then we performed simple filtering on the generated seed data to obtain PoCs that can trigger the vulnerable code.

All of the programs are configured with options like *"–disable-shared"* to generate statically linked executable files and re-compiled using Clang 11.0.0 with additional CFLAGS *"-g –save-temp"* to generate bitcode files and save the debug symbols. The positions of vulnerable code are directly obtained from the vulnerability reports. We measured the similarity between the PoCs generated by the fuzzer using cosine distance based on basic block coverage. Then these PoCs were sorted by similarity and divided into 20 parts on average. We randomly select PoCs from each parts for further tracing and analysis. If the size of a log file is greater then 10 GB, it is discarded considering that the time consumption for object context analysis is much longer. Additionally, a POC with a similar level of similarity will be selected instead.

Overall results

The overall results of the exploitability evaluation are shown in Table 1. Overflow context analysis is only performed on the reported bug locations, even though they may not be the actual root cause of the vulnerabilities. This is because in most cases, we can only obtain vulnerability reports from fuzzers, which include the crash locations and error paths, without actual vulnerability root cause analysis reports.

It should be noted that different PoCs may result in different execution paths, and not all available factors may appear in the same PoC. Therefore, the data presented in this table is based on the PoCs that contain the most available factors, which are considered to have better exploitability (**bPOC**), as determined by statistical analysis. We have established a rule with priorities to select PoCs that have better exploitability. The rules are described below in order of priority. (1) ***VTW***, Vulnerable Tainted Write pointer, refers to that there are cases where a write pointer is tainted in the vulnerable code context. (2) ***VTR***, Vulnerable Tainted Read sources, refers to that there are cases where a read pointer is tainted in the vulnerable code context. (3) ***VSU***, Vulnerable Sensitive Uses, refers to that there are cases where a corrupted pointer or byte, which is overwritten in the vulnerable context, is used in sensitive operations, such as indexing memory access, function calls, etc. (4) ***TP***, Tainted Pointers, refers to that there are tainted pointers in the program's final state. If the first two conditions can be satisfied simultaneously, its priority will be the highest.

The program may be repeatedly in a vulnerable context if the vulnerable code is executed multiple times during program runtime. Therefore, when analyzing corrupted objects, we focus on analyzing the data type and pointer type of the corrupted objects, in addition to overflow length. The data and pointer types represent the semantic information of the corrupted objects, and can often better assist in manually searching for operation primitives in the code. Considering that we can still uncover potential targets for exploitation in other PoCs, we provide the total number of union types of corrupted objects and pointer types to represent the maximum potential overflow that can corrupt the observed data.

In Table 1, the first two columns indicate the names of the evaluated programs and vulnerabilities. The third and fourth columns of data represent the number of corrupted object types in the vulnerable context. The fifth and the sixth columns of data represent the number of corrupted pointer types in the vulnerable context. The ***bPoC*** represents the number of types in Better PoC, and the ***Total*** represents the number of types in all the analyzed PoCs. The data on the number of corrupted object

Table 1 Overall exploitability evaluation of real-world heap overflow PoCs

Program	CVE-ID	Corrupted objects		Corrupted pointers		VTR in bPoC	VTW in bPoC	TP in bPoC	VSU in bPoC
		bPoC	Total	bPoC	Total				
ffmpeg	2020-22034	2	5	17	32	0	647	60	322
	2020-22017	2	3	3	4	0	264	93	15
exiv2	2017-12955	2	3	1	2	27	1	3	1
	2018-17229	5	5	4	5	1	0	0	10
	2017-14860	0	0	0	0	0	2	11	0
cflow	2019-16166	1	1	0	0	3	401	14746	0
libpng	2021-4214	1	1	0	0	17	0	0	0
tcpreplay	2022-37049	0	0	0	0	0	0	0	0
tcpdump	2016-7984	0	0	0	0	0	135	0	0
	2016-7985	0	0	0	0	2	0	0	0
	2017-11108	0	0	0	0	0	0	0	0
ngiflib	2019-16346	3	3	3	3	0	0	0	83
libxml2	2017-8872	2	2	2	4	0	0	0	28
	2021-3517	0	0	0	0	95	0	0	0
swftools	2023-27249	0	0	0	0	0	0	76	0
	2021-42199	1	1	1	1	0	0	380	0
	2021-42204	1	6	1	4	42	0	2	12
	2022-35081	0	0	0	0	263168	219	0	0
ncurse	2021-39537	0	0	0	0	0	0	1	0
xpdf	2022-38228	2	5	6	10	786	2908	16551	434051
	2022-38229	0	0	0	0	2	0	89	0
	2022-38231	4	5	2	2	0	0	14	36413
	2022-38236	0	0	0	0	0	0	25	0
	2022-38238	3	5	0	0	18	0	0	0
libtiff	2023-30086	0	0	0	0	0	0	0	0
	2023-30075	2	2	1	1	0	0	0	2328
	2022-48281	3	3	2	2	1	18	146	465589
	2018-8905	2	2	2	2	0	0	1	1352
libjpeg	2021-39518	0	0	0	0	844	244	8	0
mp4v2	2023-29584	0	0	0	0	33	0	502	0
libreDWG	2023-25222	1	1	1	1	620	0	1008	0
	2022-33026	0	0	0	0	0	0	0	0
	2021-42585	0	0	0	0	0	0	18	0
gpac	2021-36417	0	0	0	0	502	0	44	0

types can demonstrate the potential for these objects or pointers to be exploited. For example, it can help determine which exploitable objects can be placed at adjacent heap addresses.

Column 7 shows the number of tainted read sources in vulnerable context of ***bPoC***, revealing there is an opportunity for input control in the content of the overflow data. Column 8 represents the number of tainted write target pointers in the vulnerable context of ***bPoC***, revealing that there is an opportunity for input control in the write target of the overflow. Column 9 shows the number of total labeled tainted pointers at the end states of the ***bPoC***. The last column indicates whether the overflowed bytes are used in sensitive operations just in the ***bPoC***, showing the significant security impacts of the ***bPoC***.

A PoC that satisfies multiple conditions at the same time is considered to have a higher exploitability. For example, in the PoCs of *CVE-2017-12955*, *CVE-2022-38228* and *CVE-2022-48281*, not only ***VTR*** and ***VTW*** can be detected, but also these corrupted bytes are found to be used in sensitive operations (***VSU***).

Some vulnerabilities do not over-write any objects because they are actually caused by out-of-bounds reads, such as *CVE-2022-37049* in *tcpreplay*, *CVE-2016-7984, CVE-2016-7985, CVE-2017-11108* in *tcpdump*, *CVE-2023-30086* in *libtiff*, *CVE-2023-27249, CVE-2021-42204* in *swftools*, *CVE-2022-38229, CVE-2022-38236* in *xpdf* and *CVE-2022-33026, CVE-2021-42585* in *libreDWG*. But they still worth analyzing because the over-read data may be used in sensitive operations. For example, in *CVE-2021-42204*, the data of out-of-bounds read is used in *fprintf* with string format operations, which may lead to a format string exploitation.

Cases such as *CVE-2022-35081* and *CVE-2022-48281* demonstrate a large number of ***VTRs*** and ***VSUs*** due to vulnerable code snippets being invoked in loops. More than half of the *bPoCs* still have multiple tainted pointers in memory at the end of the program, meaning they have some potential to develop primitives from other sensitive pointers.

Case such as *CVE-2019-16346* and *CVE-2017-8872* security-sensitive objects and pointers affected by overflow corruption were identified. However, they do not exhibit any taint characteristics, which means they cannot be tampered with by user input. The PoCs which can overflow the security-sensitive objects are considered to be ePoCs as the definition. But during actual exploitation, attackers have very limited ability to tamper with sensitive data through input.

Effectiveness of selected PoCs

The PoCs are selected from fuzzers' generation, which means their execution paths are similar or different. For the same vulnerability, not all PoCs can develop exploits. Some vulnerabilities can only be exploited under very specific execution paths. The greater the difference in the execution paths of PoCs for the same vulnerability, the more likely to reveal new erroneous behaviors. With a greater variety of misbehaviors, the exploitability of vulnerabilities can be better proven.

We validate the diversity of PoCs used in exploitability evaluations by comparing the basic block coverage. If the basic block coverage of two inputs is very close, their similarity will also be high. Besides, if there is a large difference between them, their similarity will be low. The similarity metric was calculated using the cosine similarity formula, which considers the coverage of each basic block across all test cases, and the results are shown in Table 2.

Table 2 shows the total number of generated PoCs that can trigger the buggy code for all vulnerabilities and the minimum and maximum similarity in the generated PoCs compared to the given initial PoCs. In our experiments, the similarity data was only used to indicate some differences between the PoCs we used. We can see that the basic block similarity between PoCs of different vulnerabilities can vary significantly and may even exhibit substantial differences. The fact that some PoCs for vulnerabilities have a similarity higher than 0.9 indicates that the PoCs used for analysis are also very similar. The test case with the lowest similarity is *CVE-2017-8872*, with a minimum value of only 0.17.

However, Our main work is to evaluate the exploitability a given PoC, which better fuzzing tools can generate. Diversified PoC generation is not our research objective.

Table 2 Basic block coverage similarity of PoCs

CVE ID	# of Generated PoCs	Similarity	
		Min	Max
2020-22034	763	0.56	0.74
2020-22017	3858	0.72	1.0
2017-12955	149	0.94	1.0
2017-14860	133	0.93	1.0
2018-17229	844	0.80	0.97
2019-16166	1967	0.78	1.0
2021-4214	420	0.92	1.0
2022-37049	276	0.81	0.91
2016-7984	3678	0.74	1.0
2016-7985	1774	0.75	1.0
2017-11108	21361	0.47	0.93
2019-16346	1571	0.66	1.0
2017-8872	453	0.17	1.0
2021-3517	12176	0.78	0.99
2022-38228	1440	0.83	1.0
2022-38229	10487	0.21	1.0
2022-38231	6563	0.81	1.0
2022-38236	243	0.67	1.0
2022-38238	5590	0.82	1.0
2023-27249	158	0.49	1.0
2021-42199	2260	0.54	1.0
2021-42204	3808	0.41	1.0
2022-35081	223	0.55	1.0
2021-39537	2712	0.69	0.99
2023-30086	575	0.82	1.0
2023-30775	5122	0.69	1.0
2022-48281	1397	0.69	1.0
2018-8905	4546	0.77	1.0
2021-39518	2152	0.74	1.0
2023-29584	1630	0.96	1.0
2021-36417	2050	0.72	0.99
2023-25222	1011	0.53	1.0
2022-33026	733	0.67	1.0
2021-42585	215	0.79	1.0

Fortunately, existing works like SCATTER (Zhang et al. 2023) can increasingly address this problem well.

Evaluation of object context analysis

The main task of HOEE is to identify the key factors in the development of exploits in the context of heap overflow. It relies on the accuracy of object context analyzer, which in turn depends on both the taint memory analyzer and the dynamic heap analyzer. Finally, the effectiveness can be determined by its ability to practically verify the vulnerability context.

Our taint memory analyzer is developed based on *TRITON* (Saudel and Salwan 2015), an open-source, high-performance taint analysis engine. Those APIs that cause changes in taint propagation due to input, copying, and other factors are handled in a special manner to ensure the accuracy of the result. The dynamic heap analyzer directly resolves the trace logs, representing the real execution of PoCs. Both of the two components take the concrete execution state as input. So, we focus on evaluating the effectiveness of the whole object context analysis by checking whether the labeled tainted or corrupted bytes result in crashes.

Because the address sanitizer is configured at the fuzzing phase, some test cases will terminate without crashes when compiled without sanitizer. Therefore, we first select vulnerabilities that can cause crashes due to segment faults as evaluation targets. Secondly, the trace logs, one for each vulnerability selected randomly from the analyzed PoCs above, are manually analyzed to determine the crash positions and error pointers. Then, we check whether those pointers are labeled as corrupted, tainted, or both corrupted and tainted in the reports of Hoee.

The evaluation is shown in Table 3. Column 1 is the name of the vulnerability. Column 2 shows the positions of the crashes. Column 3 is the reason why they crashed. The last column shows whether the error pointers are identified as *Corrupted* or *Tainted*, or not identified at all (*NO*). The label *Corrupted* means that the pointer is changed at the vulnerable heap overflow context. The label *Tainted* indicates that the input data will affect the pointer.

There is only one case, *CVE-2021-39518* from *libjpeg*, in which the error pointer is not identified. This is because the source pointer used in *memcpy* is a NULL pointer as an argument passed from the caller function indicating that there may be another null pointer vulnerability resulting in this crash.

In the other cases, the error pointers in the four vulnerabilities *CVE-2018-17229, CVE-2020-22017, CVE-2020-22034* and *CVE-2022-48281* are identified as *Corrupted*, they may be written to limited values during overflow. The pointer in *CVE-2017-14860* is identified as *Tainted*. There may be other vulnerabilities that caused the pointer to be illegally modified.

The pointer in *CVE-2017-12955* is identified as both *Corrupted* and *Tainted*, indicating that the vulnerable pointer may be controlled fully or partly to achieve exploit primitives.

Our object context analysis identified overflow vulnerability behavior in 100% of cases that could cause security impacts. For all of the above cases, the object context analysis can pinpoint the details of where the vulnerability occurred. This can be attributed to the concrete execution log generated by dynamic information collection module.

Efficiency

Our method has two main parts where time consumption occurs when performing analysis on a large scale on PoCs. The first one is the tracing to collect the dynamic information. Because the origin binaries are instrumented to obtain the debug symbols at runtime, it can slow down the program. The second one is the time of parsing a trace log file. The larger the log, the more instructions need to be parsed, and it also means that the memory environment of the PoC is more complex and the taint analysis is more time-consuming. Table 4 shows the efficiency of dynamic information collection and parsing of these evaluated PoCs.

These programs were compiled separately based on the commit of each vulnerability. The largest program in the test cases is *ffmpeg_g* from *FFmpeg*, which has a size of 64.97 MB. The smallest program is *png2swf* from *swftools*, which the size is less than 0.01 MB. The the average size of all the binaries is 9.73 MB.

The average performance loss during the trace phase is about 2155 times comparing to the time of origin execution. The program with the highest performance loss is the case *CVE-2018-17229*, program *exiv2*, reaching

Table 3 The evaluation of object context analysis

CVE ID	Crash position (location and code)	Crash reason	Identified
2017-12955	basicio.cpp:1134 delete p_;	free with invalid chunk	Corrupted & Tainted
2017-14860	jp2image:277 memcpy();	oversize of copy size	Tainted
2018-17229	stl_tree.h: std::_Rb_tree();	Invalid String Pointer	Corrupted
2020-22017	buffer.c:111 b = (*dst) →buffer;	Invalid Address Access	Corrupted
2020-22034	vf_floodfill.c:61 c0 = frame→ data[0][y * frame→linesize[0] + x];	Invalid Address Access	Corrupted
2021-39518	linebuffer.cpp:322 memcpy();	Source of memcpy is NULL	NO
2022-48281	tif_unix.c:350 free()	free invalid pointer	Corrupted

Table 4 The efficiency of dynamic information collection and parsing

CVE ID	Binary size (MB)	Origin time (s)	Trace time (s)	Log size (GB)	Parse time (s)
2020-22034	64.97	0.041	448.56	6.92	3952.57
2020-22017	53.14	0.08	118.07	2.95	1750.57
2017-12955	9.90	0.01	16.39	0.20	98.61
2017-14860	9.88	0.01	12.22	0.05	24.92
2018-17229	10.08	0.002	31.51	0.58	215.52
2019-16166	0.36	0.03	23.42	0.55	663.41
2021-4214	0.59	0.015	11.99	0.24	45.90
2022-37049	0.40	0.06	6.0	0.09	38.95
2016-7984	2.47	0.22	9.03	0.10	61.38
2016-7985	2.47	0.05	5.30	0.05	26.13
2017-11108	2.50	0.33	3.95	0.01	14.61
2019-16346	0.04	0.05	1.80	0.003	1.80
2017-8872	3.54	0.04	62.81	1.57	1627.42
2021-3517	3.51	0.04	12.83	0.17	144.12
2022-38228	3.69	0.04	50.83	3.93	4751.53
2022-38229	3.69	0.06	73.16	1.81	1628.43
2022-38231	3.69	0.035	126.45	5.88	3351.20
2022-38236	3.69	0.05	98.75	2.47	1580.60
2022-38238	3.69	0.07	264.69	3.08	2188.46
2023-27249	1.07	0.07	14.91	0.27	97.64
2021-42199	1.07	0.34	82.39	1.68	1244.01
2021-42204	2.11	0.16	17.31	0.25	95.33
2022-35081	0.0004	0.03	37.73	0.97	1791.55
2021-39537	0.59	0.05	13.80	0.31	210.51
2023-30086	1.00	0.58	14.53	0.12	89.43
2023-30775	1.70	0.05	36.52	0.81	584.41
2022-48281	1.39	0.08	828.16	1.02	1216.27
2018-8905	1.07	0.02	164.92	1.59	654.87
2021-39518	0.05	0.04	121.77	3.61	2695.63
2023-29584	5.99	0.04	81.65	2.16	823.90
2021-36417	20.56	0.06	4.70	0.07	151.37
2023-25222	35.05	0.12	78.12	0.85	316.87
2022-33026	34.45	0.05	42.92	0.95	201.24
2021-42585	42.28	0.04	9.82	0.17	73.56

15,755 times. The program with the smallest performance loss is the case *CVE-2017-11108*, program *tcpdump*, only 12 times. This is because it is necessary to determine whether the code execution location is within the program space, and real-time recording of the running status takes time. It can be noticed that the performance loss of different programs is not the same. On the one hand, programs with different disk I/O frequencies and frequent use of library and system functions will spend more time waiting without generating more logs. On the other hand, different programs have different processing logic, and programs that use many looped read/write memory data will bring a greater workload to our trace.

On average, a gigabyte (GB) of logs contains around 12 million lines, and it takes about 770 s (approximately 13 min) to parse. And the average size of trace logs is 1.34GB, taking about 17.20 min to parse. The time spent in the parsing phase is higher if the log file size is bigger. For parsing PoC state log lines, only one thread was used to ensure the correctness of program status interpretation.

Discussion and limitations

Accuracy of taint analysis

We develop our taint module based on TRITON. In taint analysis, if a variable is tainted by user inputs, it usually means that the value of this variable may be controlled by malicious users. However, it cannot be absolutely determined that the variable is definitely controlled by user input. This is because the variable's value may be modified in multiple places or constrained by some other conditions during program execution, and the control flow may be very complex. Taint analysis may also have false positive or false negative results, although we have set the concrete values from the real dynamic states.

Time consumption of tracing

PIN tracing offers many events, including memory accesses, system calls, and executed instructions. However, the performance of PIN tracing depends on several factors. The tracing module in our HOEE would become rarely slow when the process frequently reads and writes memory.

The longer tracing time means the larger tracing log files, which take up more storage space and longer time for analysis. Fortunately, this problem can be relieved by using multi-processes. If the log file is too large to parse, it will be discarded based on the following three points. (1) Many other PoCs have similar execution paths. (2) There are enough PoCs providing different execution paths. (3) Compared to other PoCs, complex executions are not conducive to human researchers' understanding, which would raise the difficulty of developing exploits.

Benefits of exploitability evaluation

Although the key factors of the given context of a PoC can be identified precisely with HOEE, there are still gaps in developing exploits, the exploitability of the vulnerability cannot be exactly evaluated. If the key factors can be identified, the probability of being exploited is higher; if not, it cannot be exploited. The more PoCs analyzed, the more reliable the evaluation of vulnerability exploitability becomes. Therefore, it can reveal the possibility of a vulnerability being exploited by extensively analyzing diversity of PoCs.

However, it is difficult for current fuzzers to explore the entire execution path of the vulnerable code. HOEE can still help the analysts a lot. On one hand, the easier one PoC generated by fuzzers can trigger an exploitable execution path, the easier it is to develop and deploy an exploit. The more PoCs with differences analyzed, the more reliable the conclusions drawn. The evaluation of HOEE is a qualitative analysis, not a quantitative analysis.

Limitations

Due to the exceptional complexity of real-world applications, their code and data flow during runtime can be very high. For example, web server applications, running continuously without interruption, can generate massive amounts of log data. The current version of HOEE is not able to handle these situations. The kernel idea of our method is to detect errors through detailed operation analysis of objects. On one hand, the log size can be reduced by only retaining the necessary and critical parts. On the other hand, it may also work well if suitable methods can be developed at runtime without log files.

Conclusion

In this paper, we present HOEE, a novel approach to evaluate the exploitability of the PoCs of heap overflow vulnerabilities. HOEE can identify the key factors of exploitation development. HOEE implements an overflow context analyzer to obtain detailed heap object layouts and a taint memory analyzer to determine whether user inputs in the vulnerable context can affect the corrupted bytes. Based on the recovered heap object layouts and vulnerable context, we cannot only identify the corrupted objects, but also determine whether the critical variables are tainted in the vulnerable context, and whether the corrupted bytes have been used in security-sensitive operations. Finally, HOEE evaluates the generated PoCs to reveal the exploitability of the heap overflow vulnerabilities. The experiments are conducted on 34 CVE vulnerabilities from 16 real-world programs, demonstrating that HOEE is capable of evaluating complex real-world projects with good performance. Three of the vulnerabilities are considered to have higher exploitability. Furthermore, the effectiveness evaluation shows that HOEE can assist in identifying the pointers that lead to crashes.

Acknowledgements

We would like to thank the anonymous reviewers for detailed comments and useful feedback.

Author contributions

QS: investigation, conceptualization, methodology, materials, writing, editing, experiment, validation, review, resources. GM: methodology, discussion, writing, review, supervision. KC: discussion, review, supervision.

Availability of data and materials

We will publish our benchmark and evaluation framework after the work has been accepted.

Declarations

Competing interests

The authors declare that they have no competing interests.

Received: 20 October 2023 Accepted: 10 April 2024
Published online: 18 July 2024

References

AFL. https://lcamtuf.coredump.cx/afl/

Böhme M, Pham V-T, Nguyen M-D, Roychoudhury A (2017) Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2329–2344

Chen Y, Lin Z, Xing X (2020) 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

Chen Y, Xing X (2019) 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

Chen H, Xue Y, Li Y, Chen B, Xie X, Wu X, Liu Y (2018) Hawkeye: towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pp 2095–2108

Chen W, Zou X, Li G, Qian Z (2020) KOOBE: towards facilitating exploit generation of kernel Out-Of-Bounds write vulnerabilities. In: 29th USENIX security symposium (USENIX security 20), pp 1093–1110

Chipounov V, Kuznetsov V, Candea G (2011) S2e: a platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Not 46(3):265–278

Daniel M, Honoroff J, Miller C (2008)Engineering heap overflow exploits with JavaScript. In: Boneh D, Garfinkel T, Song D (eds) Proceedings of the 2nd USENIX workshop on offensive technologies, WOOT'08, San Jose, CA, USA, July 28, 2008 (2008)

Exiv2: CVE-2017-12955. https://github.com/Exiv2/exiv2/issues/58

Google: Syzkaller. https://github.com/google/syzkaller

Heelan S, Melham T, Kroening D (2018)Automatic heap layout manipulation for exploitation. In: 27th USENIX security symposium (USENIX security 18), pp 763–779

Heelan S, Melham T, Kroening D (2019) Gollum: modular and greybox exploit generation for heap overflows in interpreters. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, pp 1689–1706

Jiang Z, Gan S, Herrera A, Toffalini F, Romerio L, Tang C, Egele M, Zhang C, Payer M (2022) Evocatio: conjuring bug capabilities from a single POC. In: Proceedings of the 2022 ACM SIGSAC conference on computer and communications security, pp 1599–1613

Kiriansky V, Waldspurger C (2018) Speculative buffer overflows: attacks and defenses. arXiv preprint arXiv:1807.03757

Lee G, Shim W, Lee B (2021) Constraint-guided directed greybox fuzzing. In: 30th USENIX security symposium (USENIX security 21), pp 3559–3576

Lin Z, Chen Y, Wu Y, Mu D, Yu C, Xing X, Li K (2022) Grebe: unveiling exploitation potential for Linux kernel bugs. In: 2022 IEEE symposium on security and privacy (SP). IEEE, pp 2078–2095

Lin Z, Wu Y, Xing X (2022) Dirtycred: escalating privilege in linux kernel. In: Proceedings of the 2022 ACM SIGSAC conference on computer and communications security, pp 1963–1976

Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Not 40(6):190–200

Saudel F, Salwan J (2015) Triton: Concolic execution framework. In: Symposium sur la Sécurité des Technologies de L'information et des communications (SSTIC)

Serebryany K (2017) OSS-Fuzz-Google's continuous fuzzing service for open source software

Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Kruegel C, Vigna G (2016) SoK: (state of) the art of war: offensive techniques in binary analysis. In: IEEE symposium on security and privacy

Wang Y, Zhang C, Xiang X, Zhao Z, Li W, Gong X, Liu B, Chen K, Zou W (2018) Revery: from proof-of-concept to exploitable. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. CCS '18. Association for Computing Machinery, New York, pp 1914–1927 (2018)

Wang Y, Zhang C, Zhao Z, Zhang B, Gong X, Zou W (2021) MAZE: towards automated heap Feng Shui. In: 30th USENIX security symposium (USENIX security 21), pp 1647–1664

Wu W, Chen Y, Xing X, Zou W (2019) KEPLER: facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. In: 28th USENIX security symposium (USENIX security 19), pp 1187–1204

Wu W, Chen Y, Xu J, Xing X, Gong X, Zou W (2018) FUZE: towards facilitating exploit generation for kernel Use-After-Free vulnerabilities. In: 27th USENIX security symposium (USENIX security 18). USENIX Association, Baltimore, pp 781–797

Yun I, Kapil D, Kim T (2020) Automatic techniques to systematically discover new heap exploitation primitives. In: 29th USENIX security symposium (USENIX security 20), pp 1111–1128

Zhang B, Chen J, Li R, Feng C, Li R, Tang C (2023) Automated exploitable heap layout generation for heap overflows through manipulation Distance-Guided fuzzing. In: 32nd USENIX security symposium (USENIX security 23), pp 4499–4515

Zong P, Lv T, Wang D, Deng Z, Liang R, Chen K (2020) FuzzGuard: filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: 29th USENIX security symposium (USENIX security 20), pp 2255–2269

Zou X, Li G, Chen W, Zhang H, Qian Z (2022)SyzScope: revealing high-risk security impacts of Fuzzer-exposed bugs in Linux kernel. In: 31st USENIX security symposium (USENIX security 22), pp 3201–3217

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.