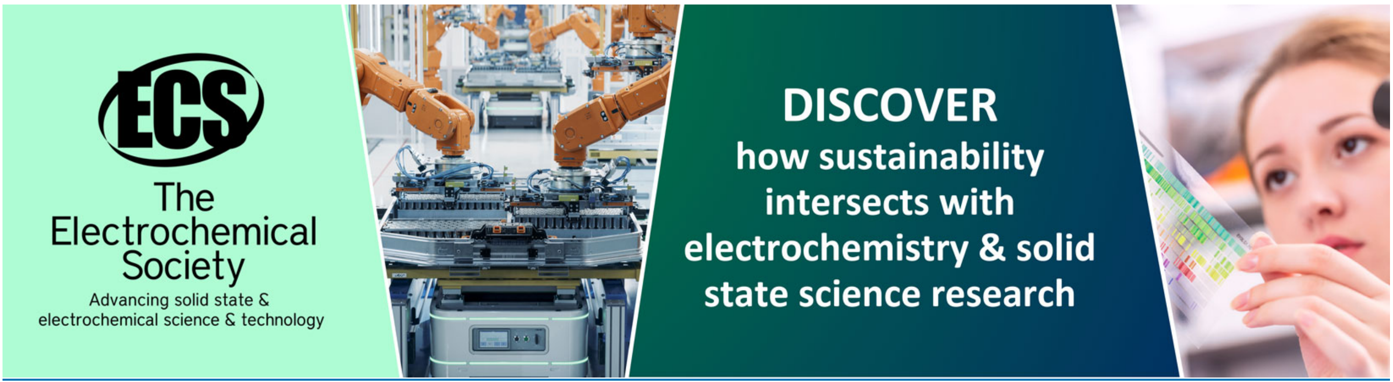
IOP Conference Series: Earth and
Environmental Science

PAPER • **OPEN ACCESS**

Analysis to Heap Overflow Exploit in Linux with Symbolic Execution

To cite this article: Ning Huang *et al* 2019 *IOP Conf. Ser.: Earth Environ. Sci.* **252** 042100


ECS
The Electrochemical Society
Advancing solid state & electrochemical science & technology
DISCOVER
how sustainability intersects with electrochemistry & solid state science research

Analysis to Heap Overflow Exploit in Linux with Symbolic Execution

Ning Huang[a], Shuguang Huang[b] and Chao Chang[c]

School of Electronic Engineering, National University of Defense Technology, Hefei 230037, China

[a]trukimurarin@163.com, [b]809848161@qq.com, [c]1063311751@qq.com

Abstract. Heap overflow is a common error of buffer overflow in Linux. The control flow of a program may be hijacked when the program satisfies several specific conditions. The existing automatic exploit generation technologies for buffer overflow find vulnerability trigger point and generate exploit by checking the control flow state. However, the heap overflow data rarely lead to a control flow hijacking as well as protection mechanisms limit the trigger condition. It is difficult to analyze the exploitability of heap overflow automatically through the existing analysis technology. For the heap overflow errors in Linux, we summarize the features of exploit on the basis of analyzing the instances, building the detection model of the exploitability of heap overflow, and proposing a method for analyzing the exploitability of heap overflow based on the model. The proposed method monitors the input data and insecurity functions of the program by using taint analysis; builds the path constraints and data constraints which satisfy the conditions of heap overflow exploit through selective symbolic execution; solves the abovementioned constraints and generates the test case automatically. All the steps of our method can be finished automatically by using the symbolic execution tool S2E. The experiments show that this method can automatically analyze and detect the exploitability of heap overflow errors.

1. Introduction

The development of information technology has highlighted the discovery and exploit of software vulnerability. Many vulnerabilities can be mined effectively by mining technologies, but only a part of these vulnerabilities can be exploited, thereby causing serious consequences. The rapid and accurate analysis of the exploitability of vulnerability has come to a key problem of vulnerability analysis and detection [1] [2].

Heap overflow error is a common buffer overflow weakness. The exploit of heap overflow vulnerabilities may lead to the hijacking of a program control flow and arbitrary code execution. Linux has set several protection mechanisms to prevent control flow hijack through heap overflow attack. Several well-known mechanisms include double free, double linked conflict detection, and chunk size detection. However, the exploit instances have shown that heap overflow remains an effective attack on several instances in recent years.

Many related research and results are available for automatic detection and exploit generation of control flow hijacking vulnerability [3] [4], such as AEG [5], CRAX [6] [17], MAYHEM [7] and

PolyAEG [8]. These methods mainly use taint analysis and symbolic execution to determine the hijacking point of the program, solve the path constraints from a source point of data input to a hijacking point, and generate exploit. However, the following restrictions apply to analyze the exploitability of heap overflow: (1) In normal cases, the heap overflow data will indirectly cover key data that may cause IP register hijacking. (2) The trigger of a heap overflow exploit depends on the allocation and free operations to the chunks by the operating system. Existing detection techniques lack the analysis of these problems.

Hao et al [16] proposed a method AHEG for automatic heap exploit generation on the basis of AEG. This method is implemented on the precondition that there is a path to hijack the EIP caused by heap error. However, in the actual cases, the discovery of this path is restricted by some special protection or check mechanisms of Linux.

Based on the abovementioned starting point, we propose an automatic exploit generation method for heap overflow in Linux. This method marks the input that may trigger a heap overflow error as the tainted source, takes functions as analysis units, extracts the features of the tainted data changes during the heap overflow exploit, designs a progressive vulnerability detection model, and filters out vulnerabilities that could lead to control flow hijack. Furthermore, we propose certain new solutions for several problems encountered in the field of automatic exploit generation for heap overflow. The validity of the method is verified by testing several test sets and projects in Linux and provides a new idea for improving the accuracy of the analysis for exploitability of heap overflow errors.

2. Background

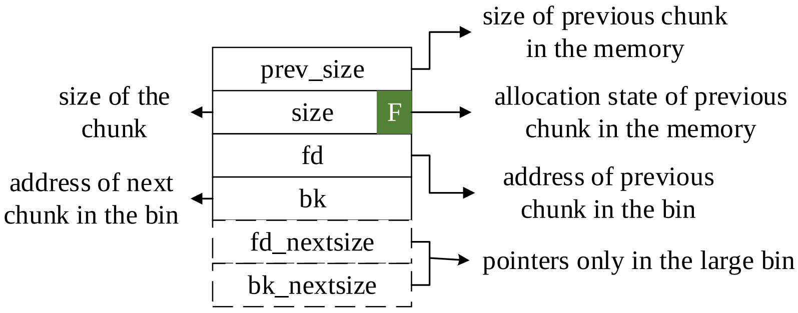

size of previous chunk in the memory
prev_size
size of the chunk
size
F
allocation state of previous chunk in the memory
fd
address of next chunk in the bin
bk
address of previous chunk in the bin
fd_nextsize
bk_nextsize
pointers only in the large bin

Figure 1. Structure of chunk header.

The Linux system uses a dynamic library, glibc, to manage the allocation, release, merger, or several other operations of chunks. Glibc manages the freed chunk through a link list structure called a bin. Every kind of chunk will be inserted to its corresponding bin after it is freed. Each chunk has its specific header to record the information of the chunk. Fig. 1 depicts the structure of a chunk header. According to the analysis on heap overflow exploits, we conclude that heap overflow exploit has three basic types.

(1) Overwrite the chunk pointers in chunk headers, such as an unlink attack. This exploit type requires the allocation and release of the chunks. In these processes, the pointers and flag bit in the header must be modified to be able to use the fake information to cheat the system, bypass the protection mechanisms, and direct them to an illegal controllable memory to achieve the exploit.

(2) Overwrite the chunk size in chunk headers, such as a house of force. We can use overflow data to override the size of adjacent chunk and modify it as an arbitrary large. Afterward, write in the modified chunk again to achieve the anywhere-write-anything.

(3) Overwrite the key data in the adjacent chunk. In certain cases, the key data may be stored in the chunk. If the overflow data, which can be controlled by input, cover this area, then we can also achieve the same goal as abovementioned.

We find that the program with heap overflow, which satisfies the following three conditions, will lead to control flow hijacking by studying the types of heap overflow exploits above: chunk overflow, exploitability of the chunk overflow, and exploitability of the key data. We define the following features of the program to indicate whether the program satisfies the abovementioned conditions.

isOverflow (C): We use this feature to describe the chunk C is overflow.

isChunkExploitability (C): We use this feature to describe whether the overflow chunk C can be controlled by tainted data.

isKeyExplotability (A): We use this feature to describe the exploitability of a key. Parameter A is the attribute of the key data. Parameter A will be a variable pointer, function pointer, or several other key data in different exploit modes.

isExploitable: We use this feature to describe whether the program satisfies all the attack conditions. If all these features are satisfied, then the value of **isExploitable** will be true; else, it will be false. Therefore, the value of **isExploitable** can also be expressed as Eq. 1:

$$isExploitable = isOverflow(C) \wedge isChunkExploitable(C) \wedge isKeyExploitable(A) \qquad (1)$$

3. Implementation

3.1. System Overview

We implement an automatic exploit generation method for heap overflow in Linux called HADE. Fig. 2 demonstrates the framework of the HADE. First, we send a crash file to the program that runs in the virtual machine as seed inputs. The HADE works in the host and uses S2E [9] [10] as its symbolic execution engine. The S2E will mark and then symbolize the seed inputs as tainted data. Afterward, the optimized symbolic execution with path-guided algorithm [11] [12] enables the program to run along a determined path and finally reach the code area that contains heap overflow errors. Fig. 3 shows the process of path selection in the optimized symbolic execution.

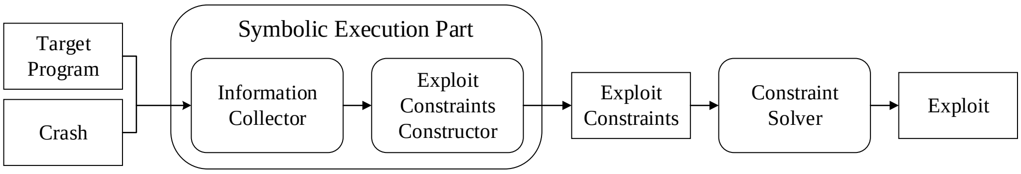


Figure 2. Framework of HADE.

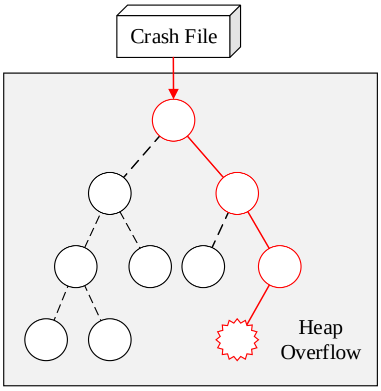


Figure 3. Path selection to the heap overflow point.

As is shown in Fig. 4, the exploit constraints constructor contains three sub-modules: heap overflow detection, exploit mode matching and controllability detection of key data.

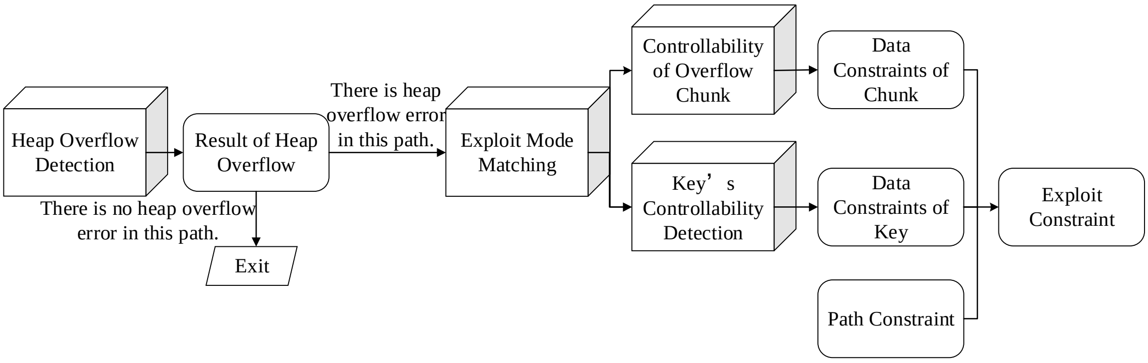

Heap Overflow Detection
Result of Heap Overflow
There is heap overflow error in this path.
There is no heap overflow error in this path.
Exit
Exploit Mode Matching
Controllability of Overflow Chunk
Key’s Controllability Detection
Data Constraints of Chunk
Data Constraints of Key
Path Constraint
Exploit Constraint

Figure 4. Implementation of Exploit Constraints Constructor.

Various modes of heap overflow exploits are available, and the exploit of every mode requires different conditions. In the present study, we will introduce the following three exploit models that have been constructed in the HADE: unlink, house of force, and house of lore. In addition to the three models, people can construct other models of heap overflow exploit and add them to the HADE as plugins.

The relationship between exploit constraint with data constraints and path constraint can be described as Eq. 2:

$$exploitConstraint = dataConstraint \wedge pathConstraint \tag{2}$$

By contrast, the data constraint consists of two parts: chunkConstraint, which indicates the data constraint of the chunk for exploit; keyDataConstraint, which indicates the other data constraints of other key data. Their relationship is as Eq. 3:

$$dataConstraint = chunkConstraint \wedge keyDataConstraint \tag{3}$$

By analyzing the chunks that involved in heap overflow, we summarized three parts of chunk data which is needed for exploit. They are: Data of overflow chunk, whose data constraints is overflowChunkConstraint; Data of overflow data, whose constraints is overflowDataConstraint; Data of fake chunk, whose constraint is fakeChunkConstraint. Their relationship is shown as Eq. 4:

$$chunkConstraint = overflowChunkConstraint \wedge overflowDataConstraint \wedge fakeChunkConstraint \tag{4}$$

To analyze the exploitability of heap overflow, we need to collect some necessary information of program states. We hook the function of the program that runs in QEMU [13] and retrieve the information of each chunk. In accordance with the different operations on chunks, each chunk has four kinds of states: initial (only after its allocation), written (while the chunk is written), freed (after the chunk is free), and null (after the chunk pointer is null). The state of the chunk will be updated when the hooked function is detected. The update rules are summarized in Table 1.

Table 1. Update rule of chunk.

Operation	States of Chunks	Rules of the State Transition
Allocate	Allocated	Init (C); add_to_chunklist (C)
Write	Written	byte ← bytewrite
Free	Freed	del_from_chunklist (C), add_to_bin (C)
Null	Null	del_from_bin (C)

3.2. Automatic Analysis Steps

The whole module of exploit constraints constructor will work as the following steps.

(1) Heap overflow detection. We can easily recognize the memory area tainted by input because the S2E will mark the input as tainted symbolized data. Thus, the chunk is checked each time after it is written, and its size is compared with those of the symbolized blocks to determine the error point of the program and determine the chunk that overflows.

We judge that chunk C is overflow by comparing the initial length of chunk C with the size of the symbolic block S. Fig. 5 displays the change of the C state.

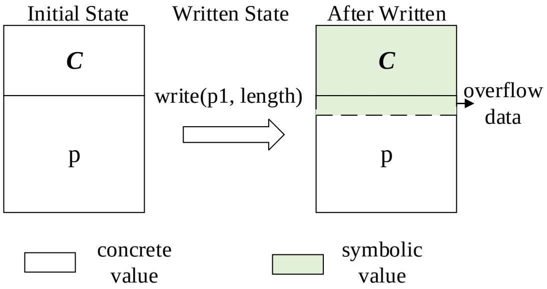


Figure 5. Change in the state of the chunk.

We assume that the set ***Symb*** includes all the symbolic blocks in memory and the set ***Chunk*** includes all the chunks which are allocated legally.

If chunk ***C*** is overflow, and the overflow data are tainted by symbolic data, then we add it to the chunk set $\boldsymbol{Chunk}_{overflow}$. We use Algorithm 1 to judge if the ***C*** is overflow.

```
Algorithm 1 Heap Overflow Detection
Input: Chunk, Symb
Output: $Chunk_{overflow}$
foreach(C in Chunk)
    foreach(S in Symb)
        if($C.addr + C.size \in [S.addr, S.addr + S.size]$):
            $Chunk_{overflow} \leftarrow C$
            break
    end if
return $Chunk_{overflow}$
```

(2) Heap overflow exploit mode matching. Various exploit models are built to match different exploit modes, and the models are input to the HADE as its plugins. The HADE will hook the marked function and match the execution states with our model when the program is running. Through exploit mode matching, HADE builds the data constraints that can trigger the attack.

We implement that the HADE can recognize the following three kinds of exploit mode by matching the exploit mode: unlink, house of force, and house of lore.

In order to describe the states of chunks more easily, we define the following function to show the location states of two memory blocks:

$$Neighbour(block1, block2)$$

If the return value of **Neighbour** is true, it indicates the two memory blocks, block1 and block2, are adjacent and the start address of block1 is smaller than the block2.

During the analyzing of the exploitability of heap overflow, we usually meet a problem that the program may start to execute exception handling functions because of the failure operation to the pointers of chunk. The root cause of this problem is that program cannot recognize the key values correctly if they are tainted by symbolic values. To solve this problem and prevent the target program exiting incorrectly, we have to concrete some specific symbolic values according to the memory state. Therefore, we use the following function to express the concrete rules:

$$\text{Concrete}(addr, size).$$

The definitions of the parameters in Concrete are:
addr: the start address of the concrete block;
size: the length of the concrete block.
The implementation of exploit mode matching is shown in Code 1.

Code 1 Exploit Mode Matching.

```
$T_A$:
   /* Execute Algorithm 1 */
    isOverflow(c) ← Overflow_chunk_search( Chunk, Symb );
    assert( ! isOverflow(c) );
$T_B$:
    memCondition ← Memory_layout_condition( );
    (Rule, exploitType ) ← Judge_exploit_type( memCondition );
$T_C$:
    Concrete_symbolic( Rule, exploitType );
$T_D$:
    expTrigger ← Judge_trigger( memCondition, exploitType );
    chunkConstraint ← Constraint_construction( Rule, expTrigger );
return chunkConstraint;
```

In Code 1, T_A, T_B, T_C and T_D indicate the different moments of program execution. All of these moments may be different for programs. In some case, some of these moments may indicate the same moment. HADE monitors the dynamically running programs at all times and does the corresponding analysis and operation according to the execution states of programs. Table 2 shows the conditions for the operation moments above in different modes of exploit.

Table 2. Conditions for the operation moments in different modes of exploit.

Exploit Mode	T_A	T_B	T_C	T_D
unlink	p1:Allocated p2:Allocated Neighbour (p1, p2)	p1:Written	isOverflow (p1)	p2:Freed
house of force	p1:Allocated Neighbour (p1, topChunk)	p1:Written isOverflow (p1)	p2:Allocated isKeyExploitable (p2.size)	p2:Allocated isKeyExploitable (p2.size)
house of lore	p1:Allocated Neighbour (p1, p2)	isOverflow (p1) p2:Freed	isKeyExploitable (fake)	p2:Allocated fake:Allocated

When we find a chunk that is overflow by Algorithm 1 at moment T_A, HADE begins to collect the execution states of program.

When HADE collects enough execution states, the chunk's layout condition of memCondition is constructed. Based on the memCondition, we can judge what type of exploit the heap overflow may match and get the exploit mode exploitType and concrete rule Rule at moment T_B. The mapping from memCondition to exploitType is shown as Table 3.

Table 3. Mapping from memCondition to exploitType.

memCondition	exploitType
$(p1, p2 \in \text{Chunk}) \wedge (p1 \in \text{Chunk}_{\text{overflow}}) \wedge$ Neighbour(p1, p2)	unlink
$(p1 \in \text{Chunk}) \wedge (p1 \in \text{Chunk}_{\text{overflow}}) \wedge$ Neighbour(p1, topChunk)	house of force
$(p1 \in \text{Chunk}) \wedge (p1 \in \text{Chunkoverflow}) \wedge (p2 \notin \text{Chunk}) \wedge$ Neighbour(p1, p2)	house of lore

The concrete rule for each mode of exploit is shown in Table 4. With the executing of program, we may find a problem that due to the checking mechanisms, some operation to the chunks may lead to exception handling. Therefore, HADE need to concrete some key symbolic value to keep the program executing dynamically and correctly at moment T_C according to the exploit mode exploitType.

Table 4. Concrete rule Rule for each exploitType.

exploitType	Rule
unlink	Concrete(p1.addr, sizeof (p1.header.fd))
house of force	Concrete (p1.addr+p1.size+sizeof (p1.size), sizeof (p1.size)) Concrete ([esp]+4, sizeof (p2.size))
house of lore	Concrete (fake.addr, sizeof (fake.header)) Concrete (p2.addr, sizeof (p2.header))

At moment T_D, we judge the final condition of heap overflow exploitability for different exploitType. We use expTrigger to record the layout conditions memCondition and exploit mode exploitType. Then, HADE builds suitable chunkConstraint on the basis of expTrigger. The construction of chunkConstraint must follow the specific rules Rule. The data constraints built by HADE for each exploit mode is shown as Table 5.

Table 5. Data constraints for each exploitType.

exploitType	overflowChunkConstraint	overflowDataConstraint	fakeChunkConstraint	keyDataConstraint

unlink	p1.fd ← k_addr - 12	p2.F ← 0
		p2.prev_size ← p1.size
			--	fake.bk ← X

house of force	--	top_size ← -1	--	p2.size ← (X – top_addr – 16)

house of lore	--	p2.bk ← fake.addr	fake ← X	fake.fd ← p2.addr
				fake.size ← p2.size

(3) Overflow chunk's controllability detection. This module is used to check the exploitability of overflow chunk. Occasionally, specific data must be placed in an accurate position in the overflow chunk for exploit. The data constraint of overflow chunk will also be constructed in the symbolic blocks because our detection method is implemented on the basis of symbolic execution and taint analysis. An exploitable overflow chunk's structure is presented in Fig. 6.

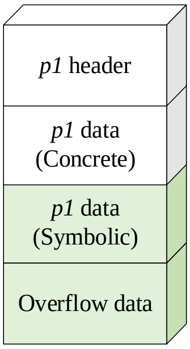


Figure 6. Structure of the overflow chunk which can be controlled by input data.

In this step, the constraint of overflow chunk chunkConstraint consists of over flow Chunk Constraint and overflowDataConstraint. Their relationship is shown as Eq. 5:

$$chunkConstraint = overflowChunkConstraint \wedge overflowDataConstraint \tag{5}$$

If and only if all byte constraints of the bytes in overflow chunk and overflow data are compatible with their corresponding conditions, we call this overflow chunk is controllable and return the data constraint of overflow chunk constraint.

(4) Key's controllability detection. This module is used to monitor the key data in the program memory. HADE detects the controllability of key data by checking the compatibility between keyDataConstraint which is built in exploit mode matching and the byte constraints of key data. If they are not compatible, it means the key data cannot overwritten by the keyDataConstraint that may lead to the failure of the heap overflow exploit. In this case, we call the key data is uncontrollable.

4. Experimental Results and Discussion

4.1. Running Time of HADE

We use programs of Juliet Test Suite v1.2 as test programs to verify the detection ability of heap overflow of HADE. All the experiments are run in the Ubuntu-32bit system. The hardware environment includes: Intel Core i7 7th Gen CPU; 16GB memory; SSD with 256GB. As a contrast, we also choose the CRAX for the same experiments. Both HADE and CRAX use S2E for the symbolic execution.

There are 11 sets of programs in CWE122_Heap_Based_Buffer_Overflow for different types of heap overflow errors. We divide our test programs into 11 groups according to the sets of CWE122_Heap_Based_Buffer_Overflow. We firstly run all programs with crash files by HADE and CRAX. The time for symbolic execution is shown in Fig. 7.

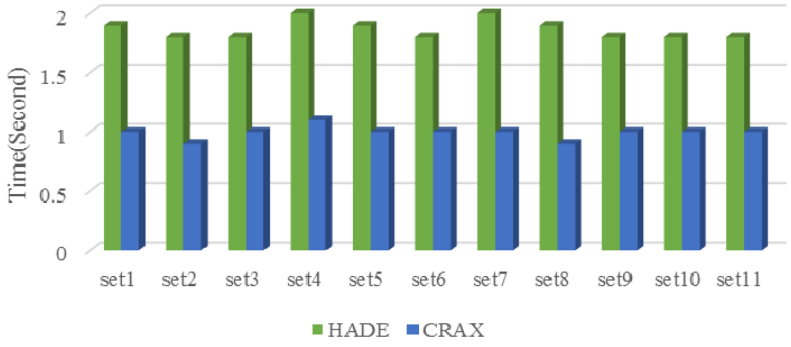


Figure 7. Average time of heap overflow detection with crash files.

Programs run along a certain execution path by optimized algorithm with crash files. Therefore, there are not path selection when programs run with crash files. However, HADE still need more time than CRAX. This result means that HADE pays more time to run the same execution path than CRAX.

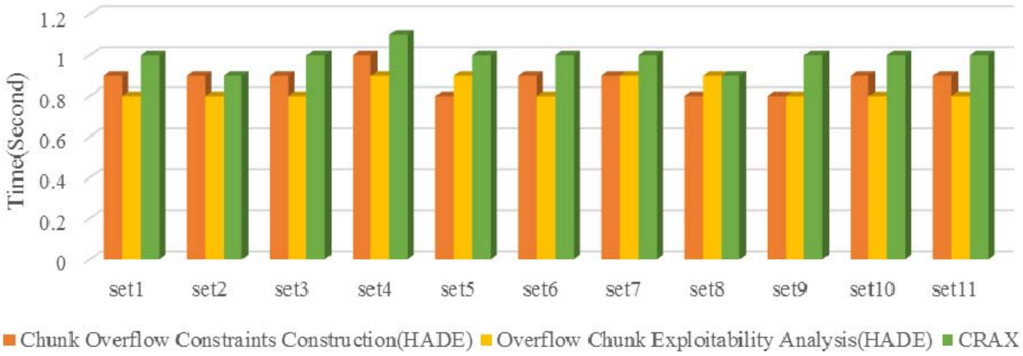


Figure 8. Time comparison between HADE and CRAX.

To find out the reasons for the difference in time, we record the time that each step of HADE needs and compare them with CRAX. The result is shown in Fig. 8.

There are mainly two steps HADE must finish that CRAX has not. The first step is to detect the heap overflow errors; the second step is to analysis the exploitability of overflow chunk. The heap overflow errors detection result that analyzed by HADE and CRAX is shown in Table 6. HADE is able to generate the *overflowChunkConstraint* and *overflowDataConstraint* for all programs. By comparison, CRAX can generate neither of them.

Table 6. Heap overflow detection results.

Tool	***overflowChunkConstraint***	***overflowDataConstraint***
HADE	√	√
CRAX	×	×

Therefore, we can conclude that each step of HADE needs less time of symbolic execution than CRAX, but the sum of them is more than CRAX.

4.2. Exploit Mode Match Ability of Heap Overflow

Table 7. Experimental results by the HADE.

Program Name	Input Source	Symbolize Input Length
CWE131_memmove_31	arg.	400
CWE805_wchar_t_memcpy_01	arg.	40
unsafe_unlink [15]	file	400
Goahead [14]	stdio	2000
shellman (unlink)	stdio	400
shellman (House_of_force)	stdio	400
shellman (House_of_lore)	stdio	400
House_of_force [15]	file	380
House_of_lore [15]	file	420
Calc	stdio	100

We collect 10 programs that contain heap overflow errors for our experiment. All these programs are written by using the C language. Protection mechanisms that defend against exploit have two kinds:

one mechanism is the protection of the Linux system, such as N/X and PIE; another mechanism is the checking mechanisms in glibc, such as double free checking and double linked list conflict detection. All these mechanisms will be turned off during our analysis. The input information and experiment results are displayed in Table 7.

Most of the crash analysis tools detect the corruption state of the program by monitoring the symbolic state of the IP register. CRAX is a typical crash analysis tool. We use the CRAX to analyze the programs to judge whether they can generate test cases to compare the effectiveness and difference between HADE and existing tools. The comparison results are summarized in Table 8. In this table, t1 is the average time for systems to analyze the target programs, and t2 is the average time for systems to generate the test cases of target programs.

Table 8. Comparison of the test case generation results between CRAX and HADE.

Program Name	CRAX		HADE	
	t1/s	t2/s	t1/s	t2/s
CWE131_memmove_31	1.0	-	1.9	-
CWE805_wchar_t_memcpy_01	1.0	-	1.8	-
unsafe_unlink	0.9	-	4.2	4.8
Goahead	3.4	-	5.7	-
shellman (unlink)	-	-	-	3.8
shellman (House_of_force)	-	-	-	3.7
shellman (House_of_lore)	-	-	-	3.8
House_of_force	1.0	-	4.1	3.9
House_of_lore	0.9	-	4.1	3.9
Calc	2.3	-	5.2	5.1

The results in Table 8 indicate that CRAX uses less average time than HADE to analyze target programs. By contrast, no test cases are generated by the CRAX for all these programs, but the HADE generates test cases for seven of these programs. We can conclude that the HADE can analyze the exploitability efficiently by increasing the execution time.

However, the HADE does not generate test cases for another three programs. We perform additional analysis in accordance with the experimental records to analyze the reason. Table 9 presents the records of paths to the heap overflow error point and concreted symbolic blocks based on different crashes provided by the fuzzing module.

Table 9. Constraint construction results by the HADE.

Program Name	Paths to the Heap Overflow Point	Concreted Symbolic Blocks
CWE131_memmove_31	1	1
CWE805_wchar_t_memcpy_01	1	1
unsafe_unlink	1	1
Goahead	0	1
shellman (unlink)	>1	1
shellman (House_of_force)	>1	2
shellman (House_of_lore)	>1	2
House_of_force	1	2
House_of_lore	1	2
Calc	>1	2

In Table 9, the HADE constructs a path constraint to the heap overflow point for nine of the programs but not for Goahead.c. We find that the reason for the heap overflowing of Goahead.c is written to the chunk through a nonlinear change in the tainted data by analyzing the source code of

Goahead.c. We use KLEE and S2E as the basic symbolic execution engines for the HADE. Most of the symbolic execution tools, including KLEE and S2E, have limitations in analyzing the nonlinear program path; thus, the tainted data may lose their symbolic attribution. Hence, the HADE cannot construct the path constraint to the heap overflow point for Goahead.

Alternatively, shellman and Calc run along the path based on different inputs, which enable them to have more than one path to the heap overflow point. Therefore, the HADE builds different path constraints with various crashes.

Before the experiments, we modify a part of the source code of unsafe_unlink, House_of_force, and House_of_lore in shellphish/how2heap for all these programs to read the input through files. The path to the heap overflow point is 1 because the three programs have a single path. In the dynamic analysis, the HADE finds that the concreted symbolic blocks are 1, 2, and 2 given their different exploit models.

CWE131_memmove_31 and CWE805_wchar_t _memcpy_01 are single-path programs. The number of their concreted symbolic blocks is 1.

Table 10 displays the analysis results of the program feature detected by the HADE. Heap overflow errors exist in CWE131_memmove_31 and CWE805_ wchar_t_memcpy_01, but these errors do not match any exploit mode that we have set in the HADE. Moreover, the two programs satisfy the conditions of the heap overflow and overflow chunk's exploitability but do not satisfy the condition of key's exploitability.

Table 10. Program feature detection results by the HADE

Program Name	Exploit Mode	Heap Overflow	Overflow Chunk's Exploitability	Key's Exploitability
CWE131_memmove_31	-	✓	✓	×
CWE805_wchar_t_memcpy_01	-	✓	✓	×
unsafe_unlink	unlink	✓	✓	×
Goahead	unlink	×	×	×
Goahead (modified)	unlink	✓	✓	✓
shellman (unlink)	unlink	✓	✓	✓
shellman (House_of_force)	House_of_force	✓	✓	✓
shellman (House_of_lore)	House_of_lore	✓	✓	✓
House_of_force	House_of_force	✓	✓	✓
House_of_lore	House_of_lore	✓	✓	✓
Calc	House_of_force	✓	✓	✓

Unsafe_unlink, House_of_force, House_of_lore, and Calc satisfy all the conditions that heap overflow exploit requires. Moreover, these programs can match at least one of our exploit modes.

The symbolic execution with optimized path-selecting algorithm makes the shellman run along different paths in accordance with the crashes that input to the program. We find that shellman can satisfy all the three exploit modes by running along different paths. Meanwhile, shellman also satisfies the conditions of heap overflow, overflow chunk's exploitability, and key's exploitability.

Owing to the limitation of the symbolic execution, the HADE cannot build path constraints to the heap overflow point of Goahead. We re-symbolize the data that will result in the heap overflow to test the effectiveness of our exploit models. The experimental results show that the modified Goahead satisfies all the features of the heap overflow exploit.

5. Conclusion

This study analyzes the features of heap overflow exploit, builds analysis models and algorithms, and implements the analysis method HADE for the exploitability of heap overflow in Linux on the basis of symbolic execution and taint analysis. The HADE judges whether a program with heap overflow error can be exploited by analyzing the heap overflow feature, overflow chunk's exploitability, key data's exploitability, and matching exploit mode. The abovementioned works can provide a reference for the protection from heap overflow exploit.

We select the automatic exploit generation tool CRAX which is also implemented on the basis of S2E for our comparison experiments. 11 sets of programs with heap overflow errors are tested to check the running time of HADE and CRAX. Experimental results show that CRAX pays less time on the symbolic execution for tested programs, but it is not able to analyze the necessary conditions for heap overflow exploit in Linux as HADE.

What's more, ten programs with heap overflow errors are chosen for the experiments to verify the effectiveness of exploit mode matching of HADE. The results show that HADE can distinguish all three kinds of exploit in most tested programs.

However, our method still has limitations in the following aspects: (1) the program detected by the HADE must run in a system without several protection mechanisms, including N/X, PIE, and certain other glibc checking mechanisms; and (2) the weakness of the symbolic execution for nonlinear data limit the analysis ability of tainted data that may lead to heap overflow exploit.

References

[1] C. Miller, J. Caballero, U. Berkeley, et al. “Crash analysis with BitBlaze”. Revista Mexicana De Sociología, Vol.44, pp.81-117.

[2] X. Jia, C. Zhang, et al. “Towards Efficient Heap Overflow Discovery”. 26th USENIX Security Symposium 2017, pp.989-1006.

[3] L. He, P. Su. “Research Progress on automatic Exploit of Software Vulnerabilities”. China Education Network 2016 (z1), pp.46-48.

[4] D Brumley, P Poosankam, D Song, et al. “Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications”. 2008 IEEE Symposium on Security and Privacy, pp.143-157.

[5] T. Avgerinos, K. C. Sang, B. Hao, et al. “AEG: Automatic Exploit Generation”. Network and Distributed System Security Symposium, NDSS 2011 (Vol.57).

[6] S. K. Huang, M. H. Huang, P. Y. Huang, et al. “CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations”. IEEE Sixth International Conference on Software Security and Reliability 2012, Vol.2, pp.78-87.

[7] T. Avgerinos, A. Rebert, D. Brumley, et al. “Unleashing Mayhem on Binary Code”. 2012 IEEE Symposium on Security and Privacy, Vol.19, pp.380-394.

[8] M. Wang, P. Su, Q. Li, et al. “Automatic Polymorphic Exploit Generation for Software Vulnerabilities”. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol.127, pp.216-233.

[9] V. Chipounov, V. Kuznetsov, G. Candea. “S2E: a platform for in-vivo multi-path analysis of software systems”. ACM SIGPLAN Notices - ASPLOS '11, Vol.46, pp.265-278.

[10] V. Chipounov, V. Kuznetsov, G. Candea. “The S2E Platform: Design, Implementation, and Applications”. ACM Transactions on Computer Systems (TOCS) - Special Issue APLOS 2011, Vol.30, pp.1-49.

[11] H. Huang, Y. Lu, L. Liu, et al. “A research on Control-flow taint information directed symbolic execution”. Journal of University of Science and Technology of China, Vol.46, no.1, pp.21-27.

[12] Q. Xiao, Y. Chen , L. Qi, et al. “Detection and analysis of size controlled heap allocation”. Journal of Tsinghua University, Vol.55, no.5, pp.572-578.

[13] F. Bellard. “QEMU, a fast and portable dynamic translator”. 2005 USENIX Annual Technical Conference, pp.41-46.

[14] CVE-2014-9707. http: //cve.mitre.org/cgi-bin/cvename.cg-i?name=CVE-2014-9707.

[15] Shellphish/how2heap. https://github.com/shellphish/how2- heap.

[16] L. T. Hao, D. Brumley. “Automatic Heap Exploit Generation”. Communications of the ACM, Vol.57, pp.74-84.

[17] S. K. Huang, M. H. Huang, P. Y. Huang, et al. “Software Crash Analysis for Automatic Exploit Generation on Binary Programs”. IEEE Transactions on Reliability, Vol.63, no.1, pp.270-289.