Methods and Means of Searching Errors When Working With Dynamic Memory

Andrey Dergachev[1][0000-0002-1754-7120], Daniil Sadyrin[2][0000-0001-5002-3639], Aglaia Ilina[3][0000-0003-1866-7914], Ivan Loginov[4][0000-0002-6254-6098], and Iurii Korenkov[5][0000-0002-8948-2776]

[1] ITMO University, Kronverkskiy prospekt, 49, St. Petersburg, 197101, Russia
`http://www.ifmo.ru/, dam600@gmail.com`
[2] `cyberguru007@yandex.ru`
[3] `agilina@itmo.ru`
[4] `ivan.p.loginov@gmail.com`
[5] `idkorenkov@itmo.ru`

Abstract. **The subject** The work discusses methods and means of finding errors when working with dynamic memory that arise as a result of exploiting vulnerabilities in implementations of dynamic memory allocation algorithms in the C language - allocators. Such vulnerabilities are common for software systems of various levels and purposes, including system software. Techniques for their operation are easily implemented, their descriptions are publicly available on the Internet, which explains their widespread use. **The purpose of the work** The goal is to develop an integrated approach and software that can detect vulnerabilities of dynamic memory allocators both at the compilation stage and during the operation of the software, issue appropriate warnings and recommendations, and also at the compilation stage, edit the code so that exploitation of vulnerabilities is impossible. One of the important quality criteria of the developed approach is the minimization of the overhead during the verification of software products. **The method of the work** Based on the studies and analysis of the exploitation techniques of the vulnerabilities Poisoned Null-byte, Overlapped Chunks, Fastbin Attack, Unsafe Unlink, House of Einherjar, House of Force, House of Spirit, House of Lore, Unsorted Bin Attack, a conclusion was drawn on the need for checking during the verification process software applications of the following conditions:

- the possibilities of manipulating the fields of data structures that store service information, up to the creation of fake sections of the main memory on the heap or on the stack;
- possibilities of accessing arbitrary sections of the computer memory due to intentional violation of the logic of the memory allocation algorithms.

The analysis of modern verification methods and existing software products aimed at detecting vulnerabilities in the operation of memory allocation algorithms is carried out. Their advantages and disadvantages are revealed. **The results** As a result of the work done, a software solution to the problem of detecting the potential exploitation of vulnerabilities of dynamic memory allocators in the form of a low-level debugger based on the method of symbolic execution of program code is proposed.

Keywords: Verification · Bugs in software · Symbolic execution · Model checking · Dynamic memory · C language.

1 Introduction

The presence of vulnerabilities in the operation of dynamic memory allocation algorithms can lead to loss and/or intentional data corruption and, among other negative consequences, to property damage. Obviously, the detection of appropriate errors in the program code is necessary to minimize unwanted effects and their misuse. Therefore, the development of strategies for the automatic detection of such errors within the life cycle of software and products based on them is an urgent task.

2 Vulnerabilities in the realization of the algorithms of dynamic memory allocation

Table 1. Vulnerabilities in different versions of *glibc*.

	Poison ed Null -byte	Overla pped Chunks	Fastbin Attack	Unsafe Unlink	House of Einh erjar	House of For ce	House of Spir it	House of Lore	Unsort ed bin Attack
glibc 2.25	+	+	+	+	+	+	+	+	+
glibc 2.26	–	+	+	+	+	–	–	+	+

The basic concepts of dynamic memory allocation in C [1] are the following. Each thread is assigned with some memory area – "arena" in which the memory chunks will be allocated and freed upon application requests. Each arena is owned by one or more heaps consist of chunks, and a new heap is allocated with full use of the previous heap. "Malloc_state" - (the arena header, is a structure from which the memory for storing the initial heap for this arena is usually taken) stores information about bins (linked lists of chunks in the heap), top chunk (a chunk on the upper memory boundary requested from the OS) and so on. The heap is described by a structure called "heap_info", which stores pointers to its arena, previous heap, etc. Chunk (the "malloc_chunk" structure) is the range of memory on the heap allocated to the application. It can be combined with other chunks to get a larger chunk if needed. The metadata for the allocated and free chunks are different. Free chunks are stored in singly connected or doubly connected lists – i.e. bins. There are:

- 10 of fastbin bins. They store chunks ranging in size from 32 to 160 bytes. The fastbin list works on the principle of LIFO (Last Input First Output).
- 64 of smallbin bins. They store chunks smaller than 1024 bytes. Each bin gets chunks of the appropriate size. Smallbins are based on the FIFO principle (First Input First Output).
- 63 of largebin bins. They store chunks larger than 1024 bytes. Largebin chunks are stored in descending order of size.
- 1 unsorted bin - all released chunks get into it.

When malloc is called, chunks are retrieved from the unsorted bin and transferred to the commensurate bins, or returned to the user. All beans are stored in the malloc_state structure.

The *glibc* library implements algorithms for efficient work with memory. The requested chunks, depending on the size, are extracted from different bins in the order established by the corresponding algorithms of the *glibc* library. At present, there is a set of known vulnerabilities in the realization of the algorithms of dynamic memory allocation in the *glibc* library, which are considered on the Internet sites [2,3] and are illustrated with detailed examples [4]. As can be seen from table 1, *glibc* developers introduced patches that make it impossible to conduct a number of attacks, including a patch was proposed in 2017 to combat overwriting metadata in a single byte of the heap: https://sourceware.org/ml/libc-alpha/2017-10/msg00773.html. Table 1 shows a comparison between versions 2.25 and 2.26 of the *glibc* library, depending on the presence of vulnerabilities listed in the first row of the table.

According to the CWE (Common Weakness Enumeration) classification [5], the list of attacks in table 1 exploits vulnerabilities that fall into the following categories:

1. CWE-122 Heap-based Buffer Overflow
2. CWE-415 Double Free
3. CWE-416 Use after free
4. CWE-476 NULL Pointer Dereference

If we classify the presented vulnerabilities according to the technique of use, the main list can be presented in the following edition (table 2) depending on:

- attacker's capabilities to overwrite top chunk (see heap organization in *glibc*) and fields: prev_size, size, fd, bk;
- the ability to create a special "fake" chunk on the stack or heap;
- from the presence of "double free" vulnerability;
- from the ability to free a pointer to an arbitrary address ("arbitrary free").

Let us illustrate the undesirable behavior of programs(table 3) due to the use for example of the Overlapped chunks and Fastbin Attack techniques, based on some of the vulnerabilities like CWE-415.

In the first column of table 3 the code of the first example is placed. In this example after deliberately recording an invalid value of the size of chunk p2 and releasing this chunk if a new memory chunk (p6) is allocated in its

Table 2. Comparative characteristics of attacks.

Tech niques	Overwrite area						Demands of “fake” chunk		Dema nds of Doub le Free	Dema nds of Arbitra ry Free
	Top chunk	prev _size	PREV INUSE (1byte)	size	fd	bk	On the stack	On the heap		
Pois oned null-byte			+							
Over lapped chunks				+						
Fast bin Attack					+				+	
Unsafe Unlink		+	+					+		
House of Ein herjar		+	+	+						
House of For ce	+									
House of Spi rit							+			+
House of Lo re						+	+			
Unsor ted bin Attack		+	+	+		+				

place, the effect of overlapping two chunks (p3 and p6) appears, which clearly demonstrates the output of the program placed under the code. The second example demonstrates the usage of the double free vulnerability: the chunk p1 is released twice and entered in a fastbin. After that the rewriting in its field fd of the position of ”stack_var” variable allows one after the new calling of malloc to return the chunk (pp4) to an arbitrary address, in this case, at address ”8 + (char *) & stack_var”, which is reflected in the output of the program.

It is obvious the occurrence of similar situations during the software operation is unacceptable and leads to the inevitable search for solutions to the problem

Table 3. Code examples

Code example 1	Code example 2

intptr_t *p1,*p2,*p3,*p4,*p5,*p6;
int prev_in_use = 0x1;
p1 = malloc(100);
p2 = malloc(100);
p3 = malloc(100);
p4 = malloc(100);
p5 = malloc(100);
 	
int evil_chunk_size = malloc_usable_
size(p2) + malloc_usable_size(p3)
+ prev_in_use + sizeof(size_t) * 2;
 	
(unsigned int *)((unsigned char *)p1 +
malloc_usable_size(p1)) = evil_chunk_size;
 	
free(p4);
free(p2);
p6 = malloc(200);
memset(p3, ’3’, 100);
memset(p6, ’6’, 150);
fprintf(stderr, ”p6 = %s”, (char *)p6);
fprintf(stderr, ”p3 = %s”, (char *)p3);
	unsigned long long stack_var;
	
	fprintf(stderr, ”The address we want
	malloc() to return is %p.”,
	8+(char *)&stack_var);
	
	unsigned long long *p1 = malloc(8);
	unsigned long long *p2 = malloc(8);
	free(p1);
	free(p2);
	free(p1);
	unsigned long long *pp1 = malloc(8);
	unsigned long long *pp2 = malloc(8);
	stack$_v ar = 0x20$;
	
	pp1 = (unsigned long long)
	(((char*)&stack_var) - sizeof(pp1));
	
	unsigned long long *pp3 = malloc(8);
	unsigned long long *pp4 = malloc(8);
	
	fprintf(stderr, ”allocated chunk:
	%p, %p, %p, %p”, pp1, pp2, pp3, pp4);

Program result 1	Program result 2
p6 = 66666666666666666666666666666666 6666666666666666666666666666666666666 6666666666666666666666666666666666666 6666666666666666666666666666666666666 6666666633333333333333333333333333333 333333333333333333333333333333333 p3 = 6666666666666666666666666666666 666666633333333333333333333333333333 333333333333333333333333333333333	The address we want malloc() to return is 0x7ffc4099d018. allocated chunk: 0x1edb010, 0x1edb030, 0x1edb010, 0x7ffc4099d018

of automatization of the fight against the considered phenomena at all stages of the software life cycle.

3 Existed approaches and solutions

At presence there are a number of approaches to solving the problem, among which, as the most promising, we can distinguish the following: fuzzing method, dynamic analysis approach, model checking approach, symbolic execution approach, Binary Decision Diagrams application and SAT task solution.

It should be noted that the use of each of the methods implies the overhead of the execution of the program, which in some cases can be up to 500%. Let us consider in more detail some of them.

In [6] and [7], the use of "smart" fuzzing to search for buffer overflow vulnerabilities was proposed. Here, fuzzing is understood as an approach in which, instead of the expected input data, random or specially generated data is transmitted to the program. The objects of interest are crashes and freezes, violations of internal logic and checks in the application code, memory leaks.

Dynamic analysis is one of the code verification methods and is performed during program execution. The disadvantages of this approach are the impossibility of checking all the ways the program can be executed and the slowdown of the program due to the parallel execution of dynamic analysis.

In [8], a verification method is considered using the created models of programs and algorithms (model checking). The specification for the program is written in the language of temporal logic, and then special algorithms automatically check whether the created model matches the specification.

An interesting approach is the symbolic execution of a program [9]. This technique of simulation execution of a program allows to represent some of the input variables used in the program in symbolic form. Such a symbol denotes the set of values of the input variable of the program from the scope of its definition. Each symbolic execution is equivalent to executing a program on a set of specific test values of input variables, which reduces the power of the set of necessary tests. Via symbolic execution approach, it is required to select input data on which an error will occur.

Since the task is relevant, today there are several products, such as CBMC, HAIT, Heap Hopper, ArcHeap and MOPS, dedicated to solving the problem.

CBMC is a verifier that provides the possibility of limited model checking (Bounded Model Checker) for the languages ANSI-C and C ++. It allows to verify overflow of the array (buffer overflow), pointers safety, exceptions, and user-specified assertions.

A tool called HAIT (Heap Analyzer with Input Tracing) [10] implements the approach of automatic collecting of the information about the state of the heap and the operations that are performed on it. The prototype is based on the Triton framework created for dynamic binary analysis of programs. HAIT logs memory operations and tagged values. Symbolic expressions are stored in the form of abstract syntax trees (AST), and the analysis of parsed tagged data is used to track memory operations affected by user input.

Heap Hopper applies the principles of symbolic execution to search for buffer overflow vulnerabilities in the heap [11] . Heap Hopper is based on the Angr framework. At each step of the program execution, an object of the SimState class is created, which stores the state of the registers and memory of the program at the moment. Registers and memory can have a specific or symbolic value. Each symbolic variable is represented as a class of BitVectorSymbol. It is also possible to manually mark the necessary input data as symbolic - it can be symbolic memory, represented as a SimSymbolicMemory class, or a symbolic file, represented by a SimFile class. When the conditional branch instruction is reached, a constraint on the symbolic variable is added. When calling the "malloc" operator with a symbolic parameter for the size of allocated memory, a memory chunk with symbolic metadata is created. At each step in the SimState class, the heap state is saved. The heap model presented in the Heap Hopper is considered approximate.

ArcHeap [12] is an automatic tool for detecting unexplored heap exploitation techniques, regardless of their realization. For its operation it is necessary to describe the parameters of the memory allocator as well as set of possible actions on the heap. During the study ArcHeap checks to see if combinations of these actions can potentially be used to perform maintenance techniques, such as random storage or overlapping chunks. As proof ArcHeap generates a PoC that demonstrates a discovered exploitation technique.

MOPS Modelchecking Programs for Security properties [13] - verifier of models extracted from the code of programs written in C. The correctness requirements are specified in a special form and correspond to the statements of the so-called “defensive” programming. During compilation, all possible execution paths are analyzed without regard to conditions. All possible tracks are collected. Of these, operators important to safety are highlighted. Having a context-free grammar of the C language, the program is presented as a pushdown automaton. The security model is represented as a finite-state machine that accepts a sequence of security operations. The sequence of security operators is “suitable” if it is received at the input by a state machine and puts it in a “safe” state.

Since only certain aspects of the task are implemented in all the presented products, the authors of this article consider it possible and expedient to search for a more comprehensive approach that would allow creating a product whose capabilities would include methods to detect the most complete list of vulnerabilities presented in program codes combined with possible overhead minimization for its implementation.

4 The proposed solution

It is known when verifying C / C ++ programs several scenarios are possible:

- emulation of code execution in a virtual machine.
- instrumentation of the executable file after the building and its execution on a real processor.
- instrumentation of the program source code during compilation and subsequent execution of the file [14].
- performing changes to the source code of the program separately before compilation, for example, by facilities of the TXL language [15].
- the symbolic execution of the internal representation of the code during the compilation [16].
- execution using the debugger after the building.

In addition, an important step is to obtain an abstract representation of register information. The most used are the internal representations of VEX and REIL. VEX uses an intermediate representation of SSA (Static single assignment), in which each variable is assigned a value only once.

The following comprehensive approach to solving the problem is proposed: dynamic symbolic (concolic - concrete and symbolic) execution of the executable file which combines the real execution of the program with symbolic execution should be carried out. Dynamic symbolic execution will allow to applicate techniques of program execution paths investigation and by adding security predicates to path constraints to check the potentially dangerous operations for real errors included in a program. To create a more precise representation of the heap during symbolic execution, it is proposed to work directly with the "malloc_state" structure.

All the abilities and vulnerabilities from the table 1 should to be checked for each state of symbolic execution. This approach will allow to understand the applicability of attacks to errors while working with the dynamic memory.

In addition, in order to reduce overhead, instead of emulating, one should run the tested application by using a special debugger. In the future, it is also planned to investigate methods for finding errors in programs with interactive input.

5 Conclusion

The article examined the known techniques of attacks on dynamic memory, approaches to software verification, briefly characterized the existing solutions for finding errors in program code. A new approach to finding errors in memory allocators has been proposed. In the future, this approach is planned for implementation with subsequent comparison with existing solutions.

References

1. Understanding glibc malloc, `https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/comment-page-1/`. Last accessed 25 May 2019
2. Yet another free() exploitation technique, `http://phrack.org/issues/66/6.html`. Last accessed 25 May 2019
3. Malloc Des-Maleficarum, `http://www.phrack.org/issues/66/10.html`. Last accessed 25 May 2019
4. A repository for learning various heap exploitation techniques, `https://github.com/shellphish/how2heap`. Last accessed 08 Nov 2019
5. Common Weakness Enumeration, `https://cwe.mitre.org/index.html`. Last accessed 25 May 2019
6. Bhardwaj, M., Bawa, S.: Fuzz testing in stack-based buffer overflow. Advances in Intelligent Systems and Computing **759**, 23–36 (2019)
7. Mouzarani, M., Sadeghiyan, B., Zolfaghari, M. A.: Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes.Proceedings - 2015 IEEE 21st Pacific Rim International Symposium on Dependable (2016)

8. Karna, A.K., Chen, Y., Yu, H., Zhong, H., Zhao, J.: The role of model checking in software engineering. Frontiers of Computer Science **12**(4), 642–668 (2018)
9. Dudina, I.A., Belevantsev, A.A.: Using static symbolic execution to detect buffer overflows. Programming and Computer Software.**43**(5), 277–288 (2017)
10. Atzeni, A., Marcelli, A., Muroni, F., Squillero, G.: HAIT: Heap analyzer with input tracing. ICETE 2017 - Proceedings of the 14th International Joint Conference on e-Business and Telecommunications, 4, pp. 327–334. (2017).
11. Eckert, M., Bianchi, A., Wang R, Shoshitaishvili Y, Kruegel C, Vigna G.: Heap Hopper: Bringing Bounded Model Checking to Heap Implementation Security, 27th USENIX Security Symposium, 2018
12. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives, `https://arxiv.org/pdf/1903.00503.pdf`. Last accessed 25 May 2019
13. MOPS: an Infrastructure for Examining Security Properties of Software, `http://web.cs.iastate.edu/~hridesh/teaching/610/06/02/papers/mops-ccs02.pdf`. Last accessed 25 May 2019
14. Jang, Y.-S., Choi, J.-Y.: Automatic prevention of buffer overflow vulnerability using candidate code generation. IEICE Transactions on Information and Systems.**E101D**(12), 3005–3018 (2018)
15. Dahn, C., Mancoridis, S.: Using program transformation to secure C programs against buffer overflows. Proceedings - Working Conference on Reverse Engineering, WCRE. 2003-January
16. Loding, H., Peleska, J. Symbolic and Abstract Interpretation for C/C++ Programs / Electronic Notes in Theoretical Computer Science 217 (2008), `https://www.sciencedirect.com/science/article/pii/S1571066108003885`. Last accessed 25 May 2019