Journal of Information Security, 2019, 10, 177-187
http://www.scirp.org/journal/jis
ISSN Online: 2153-1242
ISSN Print: 2153-1234

Unlink Attack Defense Method Based on New Chunk Structure

Yuanzhi Huo[1], Gang Wang[1], Fachang Yang[2]

[1]Department of Information Engineering, Inner Mongolia University of Technology, Hohhot, China
[2]School of Software and Microelectronics, Peking University, Beijing, China
Email: zmshyz1995@126.com

How to cite this paper: Huo, Y.Z., Wang, G. and Yang, F.C. (2019) Unlink Attack Defense Method Based on New Chunk Structure. *Journal of Information Security*, **10**, 177-187.
https://doi.org/10.4236/jis.2019.103010

Received: May 7, 2019
Accepted: July 16, 2019
Published: July 19, 2019

Abstract

The Unlink attack is a way of attacking the heap overflow vulnerability under the Linux platform. However, because the heap overflow data seldom directly leads to program control flow hijacking and related protection mechanism limitations, the existing detection technology is difficult to judge whether the program meets the heap overflow attack condition. There are certain inspection measures in the existing unlink mechanism, but with carefully constructing the contents of the heap, you can bypass the inspection measures. The unlink mechanism must be triggered with the free function, and this principle is similar to function-exit of stacks. The paper obtains the inspiration through the canary protection mechanism in the stack, adds it to the chunk structure, encrypts the canary value, and defends the unlink attack from the fundamental structure. The experimental results show that this method can effectively prevent the occurrence of unlink attacks and has the ability to detect common heap overflows.

Keywords

Heap Overflow, Canary, Overflow

1. Introduction

The heap can dynamically allocate memory, allowing programs to request memory of unknown size. The heap is a continuous linear area in memory, growing from a low address to a high address. Early heap allocation and recycling in Linux were implemented by Doug Lea. After a series of development, the heap allocator, currently used in the Linux standard distribution heap allocator in glibc is called ptmalloc 2 [1]. Ptmalloc 2 mainly allocates and frees memory blocks through the malloc/free function. The micro-structure of the

heap structure is called chunk in memory. The user successfully allocates the heap memory through the malloc function and the system returns the heap pointer to the user. The user can read, write, and free the chunk through the heap pointer. In 1999, the Conover of the w00w00 security team introduced the principle and utilization of the heap overflow. Since then, the goal of exploiting has gradually expanded to the heap [2]. When the number of bytes written to a chunk data segment in the program exceeds the number of bytes that the chunk can use, it causes data overflow and leads to overwriting the next heap to the physically adjacent high address. And this problem is caused by the lack of strict boundary checking [3] [4]. Although there is no return address on the heap memory that can directly control the execution flow of the program, the attacker can still change the inherent execution flow of the program by overwriting the adjacent chunk [5], by calling the mechanism in the heap (such as unlink, etc.) to achieve arbitrary Address writing or controlling the contents of the chunk to control the execution flow of the program. In the source code implemented by the current glibc malloc, there are certain protection measures against attacks in the heap mechanism such as unlink, but the hacker can use the heap overflow to perform the forgery of the chunk and bypass the unlink protection mechanism to attack. The prerequisite for a successful attack is that it can successfully overflow to the next heap from the current heap, as long as the number of bytes that can be written to the heap is sufficient to cover the next physically adjacent heap block, there is no corresponding protection for this.

In 2003, William Robertson and other scholars at the University of California added a magic field and a pad field as padding based on the existing chunk header structure [6]. The value of the Magic field contains the block header checksum generated by the random seed. The process is initialized with a random value during startup. When free the chunk, the checksum is calculated again. If the stored value does not match the calculated value, the chunk structure is destroyed. At this point, the program will throw an exception and send an alert. However, the drawback of this method is that the magic field is written before the prev_size field, which reduces the chunk's storage capability. The prev_size field cannot be used to store the data of the previous chunk. If the random seed is too simple, there is also a possibility of being brutally cracked. In 2006, Qiang Zeng *et al.* proposed to add two canary values and encrypt the size field before the buffer available to the chunk user and the user buffer [7] to confuse the size of the attacker's user buffer and the address.

The Unlink attack utilizing the allocation, free, and merge operations of the chunk, tampers with the linked list pointer of the free chunk, causing the program pointer variable to be overwritten, to hijack the program control flow. For the Unlink attack, the Linux system sets the heap free protection mechanisms such as detection, double-linked list conflict detection, and chunk size detection to prevent the program control flow from being hijacked. In recent years, the exploit practice has proved that the Unlink attack is still an effective heap overflow exploiting method under certain conditions. For the latest detection me-

thod of unlink attacks, in 2018, Huang Ning *et al.* proposed a detection method based on symbolic execution. This method extracts the variation characteristics of the attack process by inputting the error of the heap overflow as the pollution source and designs an unlink attack model to filter out heap overflow vulnerabilities that may lead to unlink attacks. The downside of this approach is that a series of fuzzing tests are needed in advance to find malicious attack vectors that can trigger the unlink detection system, if the detection system is no attack vector that causes the program to crash, it may be missed [8] [9].

In [6], adding the magic field before the prev_size field of the chunk structure, this method makes the physical adjacent chunk unable to use the prev_size field of the next chunk to store information, and the encryption method is simple and easy to be brute force cracked. In [7], the double canary field is added to the chunk structure, which makes the chunk structure too complicated. The modification method of the chunk structure proposed in this paper takes into account the security and utilization efficiency. Only one canary field is added to the glibc 2 chunk header, and the md5 algorithm is used as the verification algorithm, which is not easy to be cracked and can effectively defend against unlink attacks.

The first part of this paper introduces the existing chunk structure and related knowledge. The second part introduces the improvement scheme proposed in this paper. Finally, the experiment proves that our method is effective.

2. Related Knowledge

2.1. Chunk Structure in Ptmalloc 2

In the process of execution in the program, the memory requested by malloc is a chunk. This memory is represented by the malloc_chunk structure inside ptmalloc [10]. When the chunk requested by the program is free, it will be added to the corresponding idle management list. Regardless of the size of a chunk, whether it is in the allocation state or t free state, they all use red the same structure. Their representations will be different depending on whether they are free. The structure of the Chunk is shown in **Figure 1**.

1) Prve_size. If the chunk of the current physical neighboring address (physical memory low address) of the current chunk is idle, the field records the size of the previous chunk (including the chunk header). If the previous chunk is already used, the fields can be used to store data from a physical chunk that is adjacent to the previous chunk.

2) Size. The size of the current chunk, the size must be an integer multiple of 2*SIZE_SZ. If the requested memory size is not an integer multiple of 2*SIZE_SZ, it will be converted into a multiple of the smallest 2*SIZE_SZ that satisfies the requested size. In a 32-bit system, SIZE_SZ is 4, and the SIZE_SZ in the 64-bit system is 8. And the low three bits in size field does not affect the chunk size. From high to low, it is represented as: NON_MAIN_ARENA, and records whether the current chunk belongs to the chunk of the main thread. 1

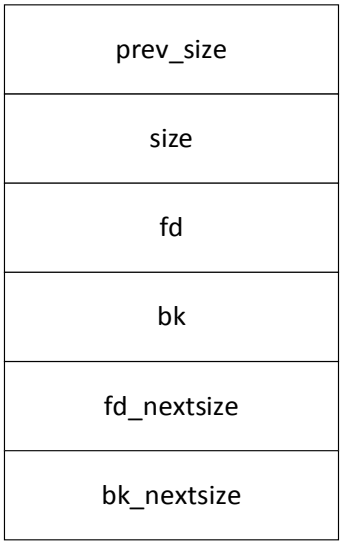

Figure 1. Chunk structure.

means not belonging to, and 0 means were belonging to. IS_MAPPED, record whether the current chunk is allocated by mmap. PREV_INUSE record whether the previous chunk is allocated.

3) When fd, bk.chunk is in the allocation state, it is the user's data from the fd field. When the chunk is free, it will be added to the corresponding idle management list. The meaning of the fields is as follows: fd points to the next (non-physical neighbor) free chunk, bk points to the previous (non-physical neighbor) free chunk. Through fd and bk, the free chunk can be added to the free chunk block list for unified management.

4) Fd_nextsize, bk_nextsize, is only used when the chunk is free, but it is used for larger chunks (large chunk). fd_nextsize points to the previous free chunk that is different from the current chunk size, and does not contain the header of the bin. The pointer bk_nextsize points to the next free block that is different from the current chunk size, and does not contain the header pointer of the bin.

2.2. Canary

Canary is protection against stack overflow. This technology has existed as the first line of defense for system security until now [11]. Canary is very simple and efficient in terms of implementation and design and is prone to overflow in the stack. The tail of the high-risk area is inserted with a value. When the function returns, it is detected whether the value of canary has been changed to determine whether the overflow occurs. The structure of Canary in the stack is shown in Figure 2.

When writing data to a local variable, if you want to overflow the overlay ebp or even return the address, the value of canary will be changed. If the canary has been illegally modified, the program will execute into __stack_chk_fail, which will fail the overflow.

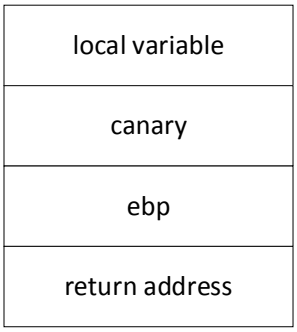

Figure 2. Chunk stack layout.

2.3. Unlink Mechanism and Attack Principle

Unlink takes a free block from a doubly linked list and merges it with the physically adjacent free chunk when chunk free, and unlink performs a series of checks before the free chunk is merged. The first check:

if (__builtin_expect (chunksize(P)! = prev_size (next_chunk(P)), 0))

malloc_printerr ("corrupted size vs. prev_size");

Unlink first checks whether the size of the chunk in the free list is consistent. There are two places in the chunk structure that record the size of the current chunk, one is the current size field, and the prev_size of the chunk of the high address of the current chunk physical neighbor. If the values of the two fields are equal, the unlink mechanism considers that the heap block does not have an exception and will successfully pass the first check.

if (__builtin_expect (FD->bk != P||BK->fd! = P, 0))

malloc_printerr (check_action, "corrupted double-linked list", P, AV)

The second check to see if the current chunk's previous free chunk in the free list points to the current chunk [10], and whether the next chunk's previous one points to the current chunk. As shown in **Figure 3**.

In the free list, the fd field of the current chunk points to the next free chunk (non-physical neighbor) in the linked list, and the bk field points to the previous free chunk. When the program overflows, free the overflowed chunk and modified the flag can reach the pointer data and hijack the program flow smoothly. Unlink's existing inspection mechanism can be bypassed smoothly.

3. New Chunk Structure

The primary cause of the Unlink attack can be successfully used because the header content of the next chunk can be modified by overflow, to deceive the unlink check mechanism, and then successfully execute the unlink. To fundamentally defend the unlink, it is necessary to directly prevent the overflow to the next one. From the perspective, a new type of chunk structure is proposed, as shown in **Figure 4**.

Based on the original chunk header structure, a canary protection mechanism similar to that in the stack is added between the prev_size field and the size field.

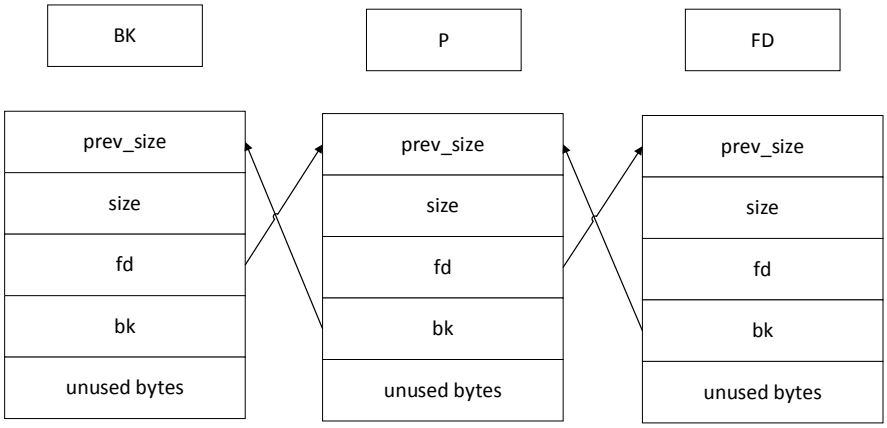

Figure 3. Free double link list.

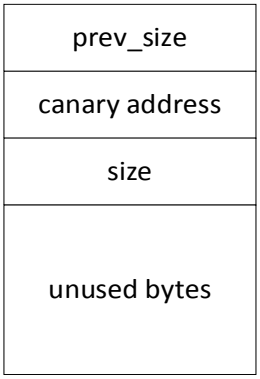

Figure 4. New chunk structure.

When the program uses malloc to apply for memory [12], the system firstly uses the size of the requested byte in the heap to open a memory space. The size of this memory space contains at least the size of the chunk header. At this time, the system will generate a random seed between 1 and 100,000 and generate a good random seed as an offset. The value is added to the first address of the bss segment of the program to get a new address. The bss segment is the memory area used by the program to store uninitialized global variables and static variables. It features readable and writable. Since the bss segment address space is very large, it can be used to store additional data. The new address generated by adding the random seed to the first address of the bss segment is used as the storage address for storing the canary value. This address is the real address for storing the canary value. It is added to the chunk header. The new field is just the address of the bss segment where the canary is stored.

The canary protection mechanism in the stack is to prevent stack overflow by obtaining the value of a certain position in the fs register and inserting it before the return address. The canary generated by this mechanism can be guessed by brute force cracking. Especially in many server programs, such as the famous Apache program, the fork function is used [13]. The fork function will copy the canary set in the parent process to the child process, so the canary values in the parent and child processes are the same so that will face a byte-for-byte attack

vulnerability. It can detect whether it is the correct value by one byte and one byte guess. To prevent the canary value in the chunk header from being violently guessed, the canary value will be encrypted. Here, MD5 is chosen as the encryption algorithm for canary.

MD5 is a widely used cryptographic hash function that produces a 128-bit hash value [14], which is 16 bytes. Due to the nature of MD5 itself, when the encrypted plaintext has minor changes, it is The MD5 value generated by encryption will change very much, so it is very suitable for verifying whether the information has been tampered with. When generating a new bss address, the heap manager determines whether the previous chunk is being used, that is, reading lowest bit in the size field. If the previous chunk is being used, the size field and the first address of the chunk header are used as plaintext. Otherwise, the value of the prev_size field, the value of the size field, and the first address of the chunk header are used as plaintext for the MD5 algorithm. Because the previous chunk of the physical neighbor is being used, then the current chunk's prev_szie can be used to store the contents of the previous chunk, which will interfere with the value generated by the MD5 algorithm. To enhance the security of the MD5 algorithm, last 8 bytes of generated. MD5 value will be are stored in the bss segment address, and then the bss segment address is inserted into the chunk header. The entire application process is shown in **Figure 5**.

When using the free function to free the chunk being used back to the heap manager, read whether the state of the previous chunk is being used, and if it is being used, perform the MD5 operation on the first address of the chunk header and the size field, and take the first 8 bytes of the result. Read the canary address in the chunk header, and take the value of the last 8 bytes of the MD5 result in the process of requesting memory from the stored bss segment address, and connect it with the first 8 bytes of the current calculation result. And compare it with the currently calculated MD5 value to see if it is equal. If not equal, the chunk must be overflowed, and the heap manager will terminate the current free process and throw an exception.

After the free function is successfully executed, the lowest bit of the size field of the physically adjacent high address is modified to identify that the previous chunk state has been changed. When the size field flag of the next chunk is modified, the above is performed again. MD5 judges the verification process, and recalculates the MD5 value of the new chunk header after the flag bit is successfully modified. The free verification process of the free function is shown in **Figure 6**.

Due to the existence of the bin mechanism in the ptmalloc 2 heap manager, the free chunk will be added to the bin list. When the next chunk of the same size is applied, the chunk will be taken out from the bin list for reuse. The chunk process also needs to recalculate the MD5 checksum to ensure security.

4. Experimental Results and Analysis

The new chunk structure designed in this paper is based on Ubuntu 16.04 64-bit system, simulating the definition of chunk structure in glibc library function,

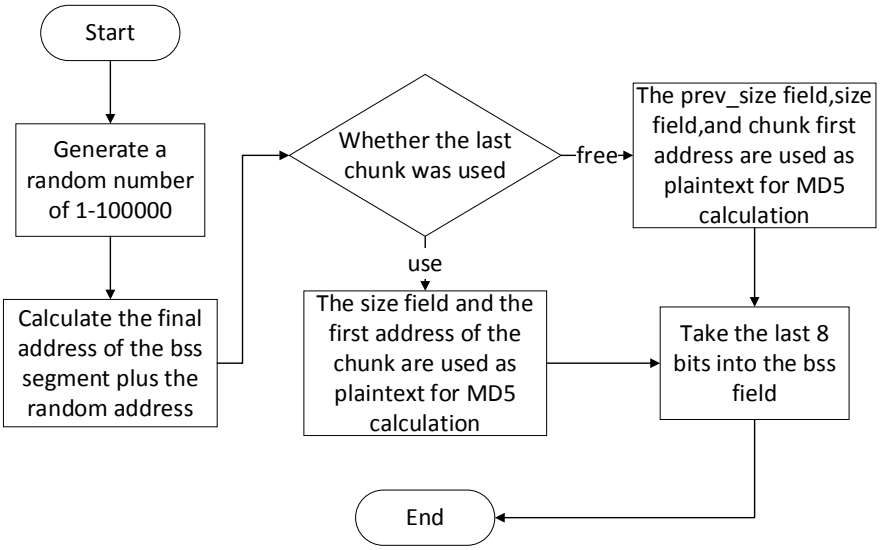

Figure 5. Canary generation process.

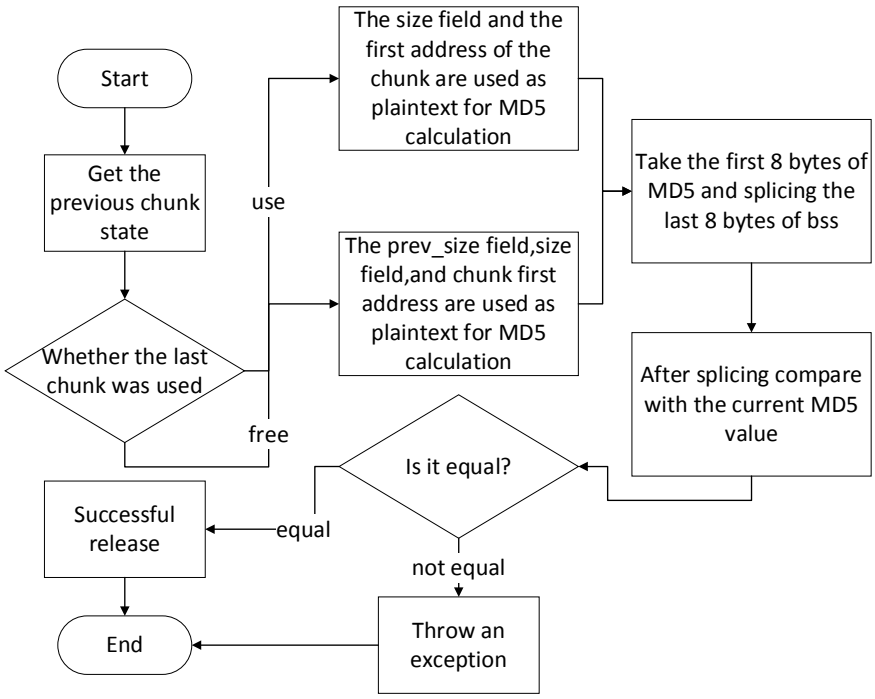

Figure 6. Free function release process.

and a series of operations, such as allocate, free, etc. Other mechanisms in heap memory stay the same. The experiment will be carried out in two cases. One is the defense effect against the ordinary heap overflow, and the other is the defense effect against the unlink attack.

Firstly, the structure defined by the malloc function in the existing glibc source code is analyzed, and the chunk structure is improved on the basis of this, and the address field of the canary is added. The core code of the improved chunk structure is shown in **Figure 7**.

```
struct malloc_chunk{

INTERNAL_SIZE_T     prev_size;
INTERNAL_SIZE_T     canary_address;
INTERNAL_SIZE_T     size;

struct malloc_chunk* fd;
struct malloc_chunk* bk;

struct malloc_chunk* fd_nextsize;
struct malloc_chunk* bk_nextsize;
}
```

Figure 7. New chunk structure code.

Insert the canary_address between the prev_size and size fields. The size of the chunk header changes from 16 bytes to 24 bytes. The minimum chunk size is also changed at the time of application.

Use the new chunk structure to apply for two chunks in a heap, one larger and one smaller. The second one must be slightly larger when applying. If it is too small, the chunk will be in the range of fastbin size, even adjacent idle chunks will not trigger unlink. Experimental data and results are shown in **Table 1**, **Table 2**.

The experimental overflow data is divided into two categories: garbage data and carefully constructed data. The first row in the table uses the garbage data to fill the first chunk and then continue to overflow to the second chunk, and uses the garbage data to cover header of the chunk. When using the free function for release, it will be addressed from the chunk header canary_address. Since the canary_address has been overwritten by the garbage data, the address is an illegal address, and the program throws an exception and terminates the release. The second row of data is only displayed to overwrite the prev_size field of the second chunk. The canary_address address is normal when released, and 8 bytes after the MD5 check value of the bss segment is successfully acquired at the time of application. However, since the prev_size field is overwritten, the checksum is calculated. The first 8 bytes and the bss segment value are compared with the currently calculated MD5 value, which are not equal. The verification failure throws an exception and terminates the release. The third row of data shows the prev_size and Canary_address of the second chunk field are overwritten. Since the canary_address field is overwritten, the result is consistent with the experimental data of the first line.

The fourth line of data is the classic unlink attack mode. First, the fake chunk is constructed in the first chunk and the prev_size field of the second chunk is modified to the size of the first chunk. The canary_address field remains unchanged, and the lowest position of the size field is 0 to forge the idle chunk flag for merging. After all the construction is completed, use the free function to start the attack. Since the unlink will manually apply for two new chunks, then the second chunk identifies the first chunk is being used, so the calculated checksum

Table 1. Experimental result description.

Overflow Data	Override Field	Experimental Results
junk data (AAAA)	override chunk header	Canary address is overwritten and addressed abnormally
junk data (AAAA)	override prev_size	Canary addressing is normal MD5 checksum verification failed
junk data (AAAA)	override prev_size & canary address	Canary address is overwritten and addressed abnormally
create fake chunk and modify chunk size	The chunk header is completely covered	Canary addressing is normal MD5 checksum verification failed
normal write	no coverage	Canary addressing normal MD5 checksum is normal

Table 2. Experimental result data.

Chunk Start Address	Random Number	Bss Start Address	Canary Address	Prev_size	Size	MD5 Value
0x663028	0x5c04	0x601060	AAAA AAAA	AAAA AAAA	AAAA AAAA	72d4a6274faff6d3453fc4 1de00b999b
0x663028	0x14106	0x601060	0x615166	AAAA AAAA	0x48	27fb59194aafe9554e0b0b c4099ff4db
0x663028	0x14fcb	0x601060	AAAA AAAA	AAAA AAAA	0x48	27fb59194aafe9554e0b0b c4099ff4db
0x663028	0x8cca	0x601060	0x609d2a	0x30	0x90	0eb7c0e623557c5b371a7 5f6056abf05
0x663028	0x6de3	0x601060	0x607e43	used	0x49	5d15b747f6d4cd992e23f 4343c1c26ee

is not using the prev_size field. After the size field is modified by overflow, the lowest bit of the size field is set to 0. The free process calculates the prev_size field together when calculating the MD5 checksum, and the calculated checksum is not equal to the first calculation. It must not be equal to the first calculation, which makes the verification failed. Although the layout required for the unlink attack is successfully performed in the heap, the unlink attack is successfully defended because the free function cannot be triggered.

It can be seen from the experimental results that the new chunk structure highly enhances the defense against heap overflow, and defends the unlink from the trigger condition of the unlink attack, which is an enhancement to the existing security mechanism. The verification mechanism implemented in this paper not only needs to know the used verification algorithm, but also needs to modify the value stored in the bss segment address of each random change and check which fields are used as plaintext. This attack has a limited possibility of succeeding.

5. Conclusion

Based on the chunk structure in ptmalloc 2 used in existing Linux, this paper absorbs the research experience of the predecessors and proposes a new type of

chunk structure. It defends the unlink attack from the structure and has a good detection effect on common heap overflow. It is also a good complement to heap security. However, the content proposed in this program still has the possibility of being cracked. How to improve and enhance is the research emphasis and development direction in the future

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] von Hagen, W. (2006) Building and Installing Glibc. In: *The Definitive Guide to GCC*, Apress, Berkeley, CA, 247-279. https://doi.org/10.1007/978-1-4302-0219-6_12

[2] Conover, M. (1999) w00w00 on Heap Overflows.

[3] Sumi, A. (2003) Bounds Checking for C and C^{++}. *Social Science Japan Journal*, **1**, 165-168.

[4] Dinakar, D. and Vikram, A. (2006) Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In: *Proceedings of the* 28*th International Conference on Software Engineering*, ACM, New York, 162-171. https://doi.org/10.1145/1134285.1134309

[5] Lhee, K. and Chapin, S.J. (2002) Type-Assisted Dynamic Buffer Overflow Detection. In: *USENIX Security Symposium*, San Francisco, CA, 81-88.

[6] Robertson, W.K., Kruegel, C., Mutz, D. and Valeur, F. (2003) Run-Time Detection of Heap Based Overflows. In: 2003 *Large Installation Systems Administration Conference*, San Diego, CA, 51-60.

[7] Zeng, Q., Wu, D. and Liu, P. (2011) Cruiser: Concurrent Heap Buffer Overflow Monitoring Using Lock-Free Data Structures. In: *ACM SIGPLAN Notices*, ACM, New York, 367-377. https://doi.org/10.1145/1993316.1993541

[8] Kharbutli, M., Jiang, X., Solihin, Y., Venkataramani, G. and Prvulovic, M. (2006) Comprehensively and Efficiently Protecting the Heap. *ACM SIGOPS Operating Systems Review*, **40**, 207-218. https://doi.org/10.1145/1168917.1168884

[9] Miller, C., Caballero, J., *et al.* (2010) Crash Analysis with BitBlaze. *Revista Mexicana De Sociología*, **44**, 81-117.

[10] Heelan, S. (2009) Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. University of Oxford, Oxford.

[11] Zhu, J. (2017) Research on Enhanced Stack Protection Technology Based on Canary. Nanjing University, Nanjing.

[12] Berger, E.D. (2006) HeapShield: Library-Based Heap Overflow Protection for Free. UMass CS TR, 6-28.

[13] Wang, Y., Cao, Y. and Wang, M. (2018) Study on Stack Protection Technology Based on GCC Plug-in. *Microelectronics & Computer*, **35**, 133-136.

[14] Wang, X. and Yu, H. (2005) How to Break MD_5 and Other Hash Functions. In: Cramer, R., Ed., *Advances in Cryptology-EUROCRYPT* 2005. *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, 19-35. https://doi.org/10.1007/11426639_2