A Novel and Fine-grained Heap Randomization Allocation Strategy for Effectively Alleviating Heap Buffer Overflow Vulnerabilities

Zhiyong Jin, Yongfu Chen, Tian Liu, Kai Li, Zhenting Wang, Jiongzhi Zheng
School of Mechanical Science & Engineering, Huazhong University of Science and Technology, Wuhan, Hubei, 430074, China
chenyf@mail.hust.edu.cn

ABSTRACT
ASLR and other defense measures have been deployed in modern computers against memory data leakage and control flow hijacking, which are two common vulnerability exploitation methods, but attackers can always bypass these security defense mechanisms to attack the computer. For the problem of the randomization strategy in ASLR technology is partial-oriented memory space. This paper presents a heap randomization strategy for the entire heap memory space. By changing the access order of memory pool, the strategy reduces the time consumption of memory region search, a random memory block allocation algorithm is proposed to increase the unpredictability of heap memory addresses and improve the security of computer systems. The randomization algorithm increases the variation interval of memory block gap by adding randomization parameters, then through the attack experiment and time performance experiment, the appropriate randomization parameters are determined to ensure the efficiency of the algorithm.

CCS Concepts
• **Security and privacy →Virtualization and security**

Keywords
Heap buffer overflow; JIT attack; ASLR; Randomization scheme; Memory fragmentation

1. INTRODUCTION
In recent years, the numbers of buffer overflow vulnerabilities rank first among all types of vulnerabilities, which have become one of the biggest threats to the computer security [1, 2, 3]. The problem of heap buffer overflow is more critical among various buffer overflow vulnerabilities. Moreover, many well-known attacks, such as the Ghost (CVE-2015-0235) and the Oracle Virtual Box High-Risk Heap Overflow Vulnerability (CVE-2017-3332), have exploited Heap Buffer Overflow vulnerabilities. The contents of an address space in the heap can be overwritten, for instance, the pointer to a data is modified to snoop the data or code of other address spaces, or the address of the virtual function in the *vtable* is manipulated to hijack the program control flow [4, 5]. The threat to target machines, no matter what kind of malicious behavior, is enormous. If the attackers can read memory cell arbitrarily, they can search for all available short sequences in the address space at runtime, bypass the randomization protection of the code segments, and construct a JIT code reuse attack. Hijacking program control flow is the key step of all code injection attacks and code reuse attack [6, 7]. Thus, it is necessary to protect the heap space.

ICMAI 2019, April 12–15, 2019, Chengdu, China

ACM ISBN 978-1-4503-6258-0/19/04…$15.00
http://doi.org/10.1145/3325730.3325738

Initially, due to the lack of proper boundary checking of the stack memory, the memory information was rewritten (e.g., the return address of a function), and led to redirect the control flow to arbitrary code (e.g., the fabricated shellcode) in a vulnerable application, for example, a typical strategy is smashing the stack attack [8]. In order to alleviate the stack crash, a random value, which is called canary value, was introduced ahead the function return value. When the canary value is modified, the verification process will terminate program. The attacker may quickly take the corresponding measure by replacing the control flow structure with structural exception handler (SEH). And the defender responds by introducing a non-executing (NX) bit [9]. However, the attacker can directly execute malicious codes injected into the stack space. DEP invalidates code injection attacks. The tricky attacker meditates a better method turning to exploit the existing codes in memory space, which is called code reuse attack [6, 10]. The earliest code reuse attack is a *return-into-libc* attack, which reuses the existing functions in the vulnerability program to complete the attack and allows the vulnerability program to jump to an existing code sequence (e.g., a code sequence of a library function). An attacker can still use an address of a malicious code (e.g., the system function in the *libc* library) to overwrite the return address of function call during the attack, and then transfer the reconfigured parameters to run as the attacker's expectation [11]. However, the granularity of this attack is relatively coarse, and it can achieve a good defensive effect via some simple CFI verification. Later, *Hovav Shacham* proposed a more granular code reuse attack which is now widely used via Return - Oriented Programming (ROP) [12, 13]. The ROP no longer reuses the entire library functions in the *libc*, but a short sequence in the library ending with the ret instruction. The emergence of fine-grained code reuse attacks has forced defenders to seek better ways to defeat them.

Address Space Layout Randomization (ASLR) is a computer security technology against code reuse attacks to prevent the attacker from reliably jumping into a special exploit function in memory. At the beginning of the period, ASLR only randomized

the base addresses of various data segments, code segments, and heap segments, which has a coarser granularity and could easily be brute-forced. With the development of offensive and defensive technologies in the security field, the granularity of randomization has become finer and finer, whose scale is from function level to basic block level, even is explored to instruction level. Fine-grained address randomization strategies make brute-force attacks harder[14]. Although the current randomization strategy still has problems that cannot be solved. Generally, randomization is still an effective defense method against code reuse attacks.

Our Contributions: In this paper, a novel algorithm is proposed, which is a fine-grained heap randomization strategy under the acceptable additional performance consumption and higher entropy.

- It achieves new fine-grained heap randomization.
- Through the study of the vulnerability of the heap buffer overflow and do the related investigations, we briefly state the advantages and disadvantages of the mainstream randomization strategies, figure out the existing problems about randomization at this stage, and put forward a new direction of randomization - randomizing the entire heap memory space address when it is allocated, not just the code segments.
- A new randomization algorithm is proposed, adding randomization parameters to enhance the entropy value, which increase the attack difficulty with less significant time and performance consumption, thereby reduce the predictability of the position of the objects distributed in the heap space.

2. BACKGROUND

We first state the relevant concepts, consisted of Just-In-Time code reuse attack, memory disclosure and control flow hijacking, which is easy to comprehend the later content.

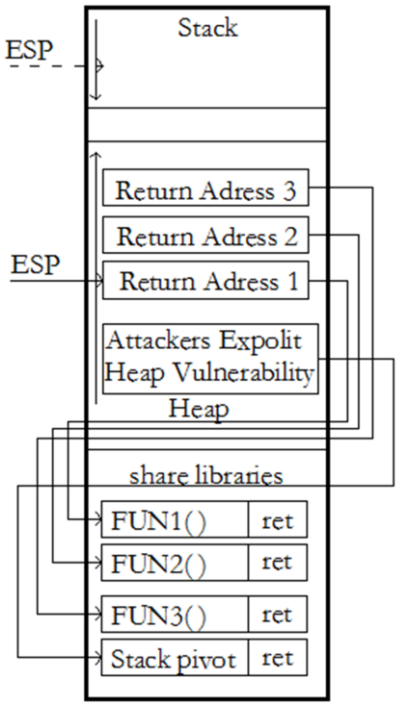

Figure 1. The basic principle of code reuse attacks.

2.1 Just-In-Time Code Reuse Attack

In 2013, Kevin *Z.Snow*[15] proposed the concept of JIT code reuse attack and successfully constructed Just-In-Time ROP attack in the JavaScript scripts of IE browser. As shown in the Figure 1, this attack uses the buffer overflow vulnerabilities to read and write any memory location, and recursively searches through the exposed function address (e.g., the address of the direct call instruction) in the code segments of the process to get lots of available codes and construct the gadget required for ROP attack. This attack invalidates fine-grained randomization of the code segment, leading to higher demand for the defender.

The original randomization strategy only randomizes the code segment, ignoring the protection of the stack, and the granularity level still stays at the base address, so memory leak occurs frequently. Is the randomization strategy obsolete? The goal of randomization is not limited to the code segment but the entire memory space. Only in this way can the attacker be prevented from making predictions on the entire memory space and be prevented substantially from using buffer overflow vulnerabilities and dangling pointer vulnerabilities.

2.2 Heap Buffer Overflow Vulnerability Exploits

On account of the continuity of the memory addresses, the emergence of the buffer overflow vulnerabilities is almost inevitable, especially in large projects or the complicated logic structure, buffer overflow vulnerabilities are often found. In the heap space, buffer overflow vulnerability exploits mainly reflect in the two following aspects: memory data leakage and control flow hijacking. The following paper will be divided into two sections.

Memory Leak. The common memory data leakage refers to the heap memory disclosure. The heap memory is an arbitrary size memory space that the program dynamically requests at run time and must be explicitly released after using. Applications generally use *malloc*, *calloc*, *realloc*, new and the other functions to apply a piece of memory from the heap to operate. Once the memory space is no longer used, the program must call the relative free or delete functions to release the memory block, if not, this memory will not be used again, it can be said that there is a leak in this memory. This memory leak is divided into the following two types: the direct and the indirect leakage. In a direct memory leak attack, the attacker can read the code pointer directly from the code page, which is usually embedded in direct branch instructions, such as *jmp* or call instruction. In an indirect leak, the attacker gets a readable code pointer in the data page, such as a return address or other pointer on the stack, and indirectly reads the code page [16, 17, 18]. As shown in the Figure 2.

There are two main dangers of memory leak: First, the attacker obtains all the codes, and breaks the fine-grained randomization. Second, the attacker obtains information from the data segments to prepare for an attack. In order to prevent memory leak, it must be sure that memory resources are free and the memory pointer is empty after using the resources

Control Flow Hijacking. Control flow hijacking is one of the most common attacks against computer software. It achieves the aim of attack by hijacking the original control flow and executing malicious codes which violates own code logic. Firstly, a specific

attack vector is constructed. Then aforementioned memory leak vulnerabilities are exploited to modify the control data. After bypassing the deployed security defense mechanism, the attacker hijacks the control flow and executes the malicious codes to ultimately implement the invasion. For example, during the run time of the program, the attacker first obtains the pointer of object A in the *vtable*, and determines the starting address of the entire *vtable* according to the address of the pointer pointing object A. The attacker continues to use the vulnerability to modify some useful virtual function. At the moment, the attacker hijacks the control flow by calling the corresponding virtual function of the object A [5].

Obviously, the attacker can easily construct available codes due to the high prediction of heap allocation, and then snoop the memory data or hijack control flow. Therefore, the direction of defense in this paper is to make the distributions of different objects unpredictable, which will increase the difficulty for attackers to exploit the heap buffer overflow vulnerability.

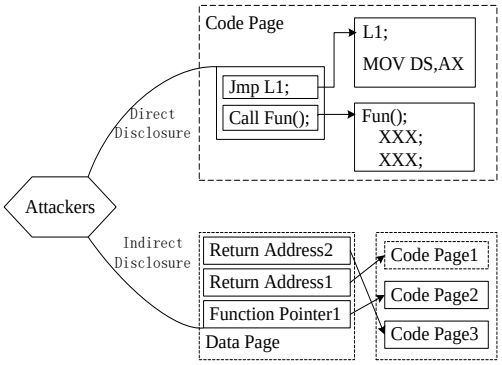

Figure 2. The direct and indirect memory disclosure.

3. RELATED WORK

3.1 Ptmalloc2 Algorithm

As we all know, the switching between kernel mode and user mode in the operating system requires a lot of overhead, so the operating system does not provide memory management interface functions such as *malloc* to the application. In general, the management of heap space depends on base library of the operating system. This paper chooses ptmalloc2 algorithm in GNU *libc* library as the research object which is widely used and open source. The memory allocation algorithm *malloc* and recovery algorithm *free* in ptmalloc2 algorithm are briefly described below.

Core data structure of *malloc* function is *malloc_chunk*. It is the basic unit of memory allocation, which contains four important fields which are *pre_size*, *size*, *fd*, and *bk*. *Pre_size* denotes the size of the previous free chunk. *Size* denotes the size of the current chunk. *fd* denotes the pointer to the last chunk block in the Bin, and *bk* denotes the pointer to the next chunk block in the bin. To manage chunks of different size, there are four kinds of bin named in *malloc*: fast bin, unsorted bin, small bin, and large bin. The size of the chunks in each fast bin and small bin is the same, and between the different fast bin are different. The size of chunks in the fast bin can reach to 80 bytes, and it can reach to 512 bytes

in the small bin. The size of chunks in unsorted bin and large bin are not determined.

In the main thread heap, the last piece of memory space is named *topchunk*, which is a large chunk of cache. When the appropriate chunk is not found in the four bins, the *topchunk* is split in order. When the *topchunk* memory is insufficient, the heap memory space is requested from the operating system to expand the *topchunk*. When the free chunks are merged, the *topchunk* exceeds a certain threshold, and then returns part memory space of the *topchunk* to the operating system.

As shown in the Figure 3 and Figure 4 which are the workflow of the *malloc* and *free* algorithm in the ptmalloc2.

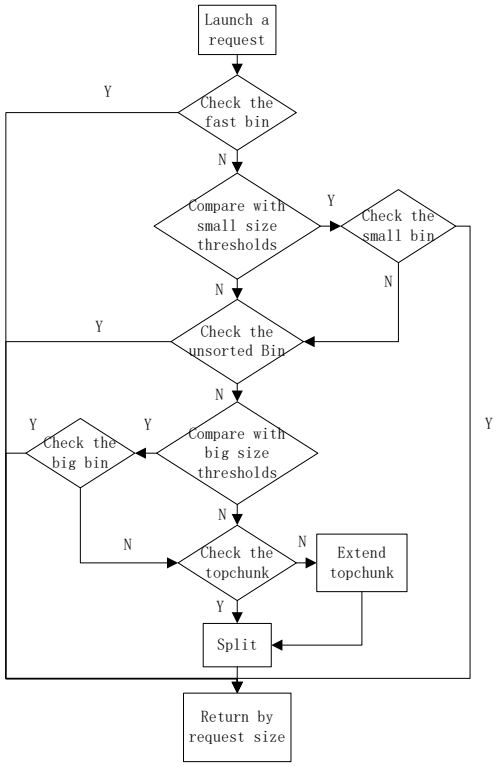

Figure 3. The workflow of the *malloc* algorithm.

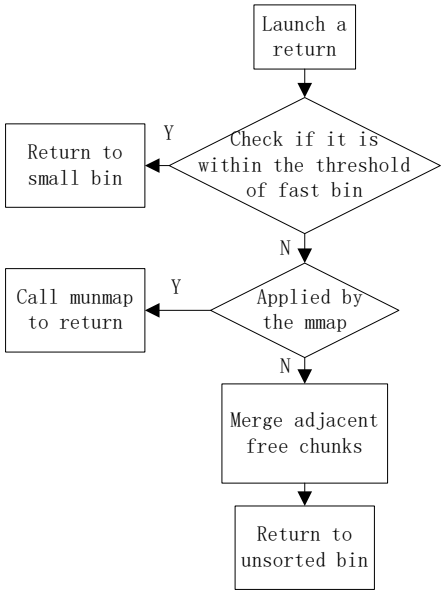

Figure 4. The workflow of the *free* algorithm.

3.2 Other Memory Randomization Strategies

In order to mitigate the threat of code reuse attacks, the computer system is proposed to apply various randomization techniques. We define a proper, effective and efficient randomization algorithm for security and use it to motivate the design and implementation of our strategy. Before showing the idea, we firstly explain more details of traditional fine-grained memory randomization.

Compiler-Based Randomization. The main idea of compiler-based approach is to randomize the layout of a program and install the randomized copies on different computers, which make the program layout in memory unpredictable for an adversary. But the primary disadvantage of compiler-based randomization is that the diversified program layout will be disrupted once the program is updated, which increases the risk of being attacked [19, 20, 21].

Base Address Randomization. The early ASLR technique was to pre-populate random bytes into the process stack (the stack top subtracting the random size bytes) while the process was loading, so that the address space loaded by the process base address was randomized. Because the random memory address area (stack memory) is single, it is unable to deal with other forms of memory attacks [1, 22, 23].

Instruction level randomization: Firstly, *Kil et al*. [23] proposed an efficient support re-randomization strategy, which statically rewrites ELF executable to permute all executable functions and data objects. Then *J. D. Hiser et al*. [24] proposed instruction location randomization (ILR), which translated each address to a randomized version while executing in a process virtual machine. But both of them still have unsolved issues. *R.Wartel et al*. [25] introduced a new technique, called STIR, which imbues x86 native code with the ability to self-randomize its instruction addresses each time it is launched. However, the difficulty of compatibility and overhead has been still unsolved. Finally, *Guiffrida et al*. [26] introduced a fine-grained randomization for system kernels. it allows re-randomization at the runtime, but it still has limited application.

Basic block level randomization. The randomization design is based on control flow graph (CFG) which is consisted of a series of basic blocks. The goal of randomization is to rearrange basic blocks of CFG. In order to retain the connection of the basic blocks, the author have fixed *jmp*, branch and call instructions; In order to obtain the position of the basic blocks in the special mode, the author tracks and analyzes every usage of the address in following codes and rewrites the offsets based on binary rewriter [27, 28].

Compared with various randomization techniques at present, the following issues should be noted:

- **System overhead**. In general, the system overhead of deployment randomization and other defense strategies are relatively small, frequent adjustment of the space layout will increase system overhead and make it difficult for users to accept it.
- **Achieve difficulty**. Note that it is difficult to modify the source code or binary code to achieve a wide range of applications.
- **Randomized scale.** To prevent the attacker from brute force, the randomized entropy value should be higher.
- **Granularity level.** A more granular defense mechanism can effectively resist memory information exposure, which can often achieve better performance.
- **Strategic cooperation.** A platform does not only deploy a single defense strategy. Coordinated with other defense mechanisms can effectively increase the difficulty to be conquered, such as, the combination of randomization strategies and intrusion detection system.

4. FINE-GRAINED HEAP RANDOMIZATION

4.1 Design

In this paper, we redesign the memory allocation strategy, which omit the access judgment of fast bin of the traditional strategy and directly judge the size of the application block. If the size is not larger than 512 bytes, the workflow enters directly into the small size heap application branch; if larger than 512 bytes then it enters the large size heap application branch.

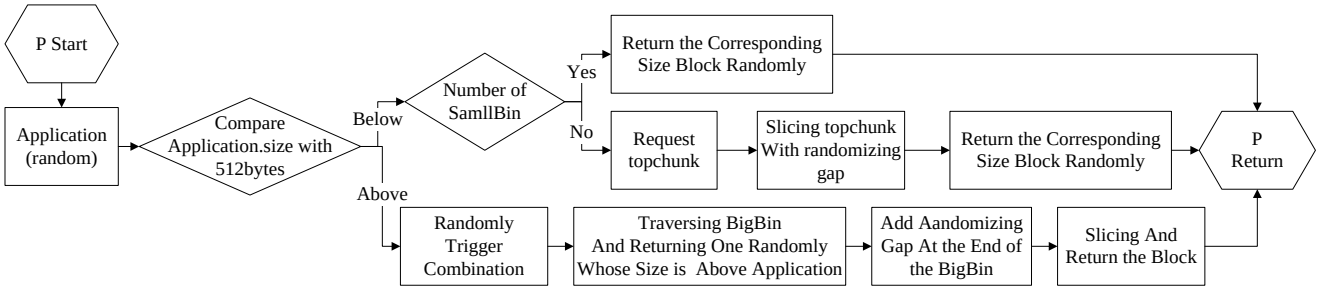

Figure 5. The direct and indirect memory disclosure.

As shown in the Figure 5, if the workflow enters the chunk blocks in small bin. If it is more than the threshold K_R, the OS iterates through all the corresponding size chunk blocks and randomly returns a chunk. If it is less than the threshold K_R, the top chunk is applied to be split. Every time it is split, a randomized gap is necessary to ensure the address of the next chunk is unpredictable. When the application is larger than 512 bytes, the workflow enters into Big bin branch (The merging operation will be triggered at a certain probability before the application is returned in big bin. More details will be described later). The OS traverses all chunk blocks in big bin, and randomly returns one of them, which is larger than the application. And at the end of the returned chunk, a small randomized memory chunk is added. Based on the end address of the small randomized memory block, the chunk block

after adding is split along the direction of heap growth and the split address is returned.

The core idea of the randomization strategy has two aspects: First, random size gaps are added behind the memory block every time a certain size of memory is requested. Second, every time the number of the caches in small bin is insufficient, the top chunk memory block is split in advance and placed into the buffer pool, and one of them is randomly returned. As shown in the Figure 6. However, frequently splitting will increase fragmentations of the heap memory. In the following sections, we will briefly introduce how to avoid excessive fragmentations of the heap memory.

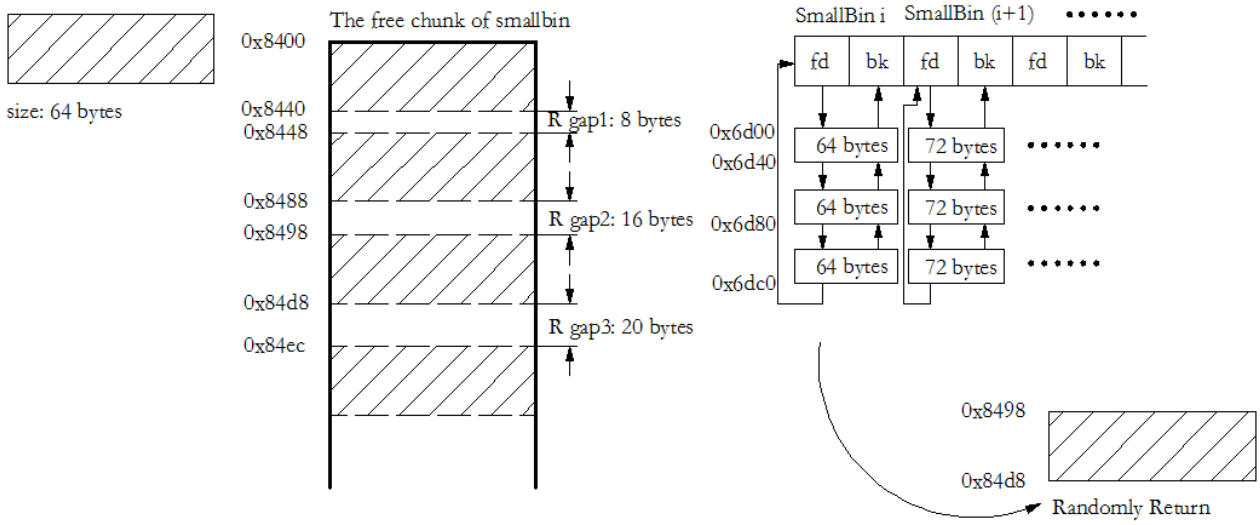

Figure 6. The principle of randomly splitting.

4.2 Maintaining Caches of Small Bin

It helpful to prevent attackers from brute force attacks to maintain a certain number of buffers in small bin. In order to ensure a defensive effect, we should have enough free blocks in the buffer. Therefore, if the number of free chunks in the small bin is less than K_R , it is necessary to supplement the free chunks by randomly splitting. In the process of the operation, in order to quickly get the number of free chunk blocks in the buffer-pool, the OS should have a counter to record the quantities. Although it brings additional performance overhead, it is acceptable if the number of chunks is not large.

4.3 New Logic of Combination

The new algorithm is redesigned with the memory allocation logic. The conventional algorithm traverses all the chunks in fast bin and merges the free chunks whose neighbors are also free, and these operations conflict with new randomization. So Fast bin is replaced with Small bin in the new algorithm. Meanwhile, in order to avoid memory fragmentation, the new algorithm triggers new merge operation at a certain probability before large block applications returned. The new merge task mainly involves two aspects:

The merge operation for Small Bin. When the size of memory block is less than the threshold (e.g., 64 bytes or 128 bytes) and the adjacent block is free, the merge mechanism is triggered. The OS compares the combined chunk size with the threshold and puts it into the corresponding Bin.

The merge operation for top chunk. When the memory block in front of the top chunk is free, the merge mechanism triggered. Once the combined chunk size is larger than the threshold, the OS collects the part of the memory. The new design is based on the following three reasons:

- In order to prevent memory from being repeatedly split and over fragmented, only tiny chunks are combined and the slightly large chunks are directly reserved in the buffer.
- Considering that the number and size of buffers always tend to meet the requirements, what we should consider for small block applications is the time efficiency rather than the space efficiency. Therefore, the merge mechanism triggered before large block applications return (if the merge operation is also performed for small size heap application, it will have a negative impact on the time efficiency).
- Due to the memory space is limited, a process cannot take up too many memory resources, or the performance of the machine will be reduced. Therefore, the OS merges the free and adjacent blocks to release the memory resources.

5. ALGORITHM IMPLEMENTATION

5.1 Randomization Algorithm

The randomization algorithm must have high enough entropy to ensure the allocated address unpredictable. When the number of buffer is insufficient, the OS executes randomly splitting. When the application is a large size block, the OS executes the random complement. The random entropy is determined by the fine-grained heap randomization parameter K_R: The larger the value of K_R, the more chunks in the buffer; the larger the value of K_R, the more unpredictable the offset between adjacent chunks. Algorithm 1 is the pseudo-code of the random process.

Algorithm 1:

Input: P（Pointer that pointing to the top chunk）
Output: NewP (Return the remainder pointer of the top chunk)
K_R ← Randomization parameter to ensure sufficient entropy
Application ← a block in memory by randomizing
Small bin ← Linkedlist composed of chunk blocks
BYTE_ALIGNMENT ← 8 bits for X86 and 16 bits for 64
TempVariable ← A return chunk satisfied the requirements of size from Big bin by Randomly
Rest ← The remain part after Slicing
rGap ← random(1, K_R) * BYTE_ALIGNMENT

if Application.size < 512 bytes **then**
  if small bin.number > K_R **then**
    Retrun NewP ← a member of a chain in small bin Randomly
  else
    for all Rest > Application do
      Slicing(P, rGap)
    end for
  end if
else
  NewP ← TempVariable + rGap
end if

5.2 Heap Memory Fragmentation

When it comes to randomly split for a small size heap application every time, it is easy to cause a serious problem: Severe heap memory fragmentation. The fragmentation, on the one hand, can cause a great waste of memory; On the other hands may cause the system to crash. Since the fragmentation problem is not to be avoided, we simply settle as following:

First, all remaining gap blocks after being split are managed in the Small bin buffer. There are two advantages: The one is increasing the number of buffers to avoid memory being split many times. Another is satisfying the requests for subsequent redistribution. This operation can greatly relieve the pressure from the memory fragmentations. Generally, it is apparently rare to apply a mass of blocks which have the same size. If we have a large number of heap requirements with the same size blocks, we generally use dynamic arrays. The size of the required heap space, in most programs, is random.

Second, adjacent free blocks are merged at some random moment with the new merge operation, which is necessary because it will greatly reduce space consumption. It can prevent the memory from being excessively fragmented, and confuse the distribution of objects in the heap. The random time mentioned here indicates that the merge behavior must be unpredictable so that no backdoor is left for the attacker.

5.3 Fine-grained Randomization Parameter

Almost all defense solutions are inevitably facing a problem - performance over-consumption. Under the acceptable performance consumption, what we should pay more attention to is how to increase the entropy. Thus, a fine-grained randomization parameter K_R (K_R is a positive integer) is introduced in this paper. The larger the K_R is, the higher the randomized entropy is, namely, the larger performance loss is in theory. It extremely enhanced the flexibility and practicality of the algorithm.

6. EVALUATION

6.1 Effectiveness

The defense effect of buffer overflow vulnerability attack is mainly affected by two aspects. One is the randomization parameter, which directly affects the size of the relative offset between two objects, and this offset will affect the probability of the average number of successful attacks by the attacker. The other is how many objects needed to launch an attack. Theoretically, when n objects are relied on, the relationship between success rate and objects is:$P = \frac{1}{K_R^n}$. Under normal code reuse attack, it generally depends on the relative distance between two objects, so the attack success rate is: $P = \frac{1}{K_R^2}$.

In this paper, the effectiveness and time performance consumption of the algorithm are selected as evaluation targets and observed to obtain the optimal randomization parameters. The GHOST attack module will be loaded 100 times with the *metaspolit* tool under the *KaliLinux* system, and the number of successful acquisition of the target host shell will be recorded.

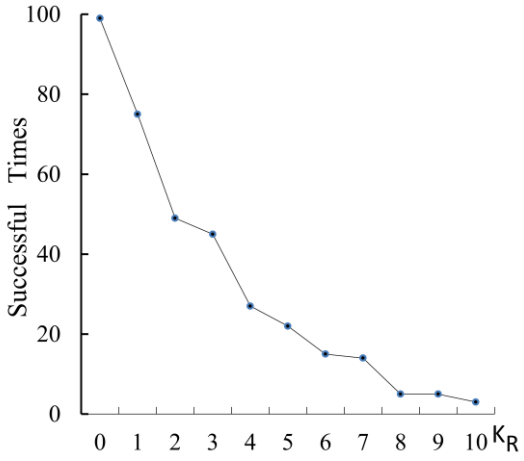

Figure 7. The times of successful attack.

As shown in the Figure 7, when the randomization strategy is not adopted, the attack success rate is up to 100%. As the randomization parameter K_R increases, launching an attack becomes more and more difficult. When K_R equals 8,9,10, the number of successful acquisition of the target host shell is close to 0. When K_R is between 2 and 6, the actual successful rate is higher than the theoretical values. It is speculated that the reason is that the merge operation in the algorithm is performed too frequently, which reducing on a degree of randomization of the relative distance between objects in heap.

6.2 Time Performance Consumption

In this paper, the algorithm uses random split to build cache and creates new merge logic, which will inevitably bring additional time consumption. However, the algorithm in this paper abandons the fast bin and unsorted bin in the ptmalloc2 algorithm, which reduces the time consumption of search. At the same time, the increase of the number of cache areas can make the request for small size block quickly satisfied, so it makes up for the time

consumption to a certain extent. Since random split will bring more cache, the space cost will increase. However, when the scale of program is large, its own heap space increases too, so the above additional space consumption can be no longer considered within the scope.

This paper uses benchmark which is written by ourselves to test the performance of the algorithm. The process of this benchmark is as follows: First, 1000 objects of random size are applied, and the size of the objects varies from 1 byte to twice times the size of the maximum chunk in small bin. All heap space is then returned and the time taken for the entire process is recorded. Test environment: Intel(R) Core(TM) i3-2348 cpu@2.30ghz 8GB memory, Ubuntu 14.04 64-bit operating system. The test procedure is to run benchmark 100 times under different K_R values, and take its average value as the result under the corresponding K_R value. The final statistical results are shown in the figure 8.

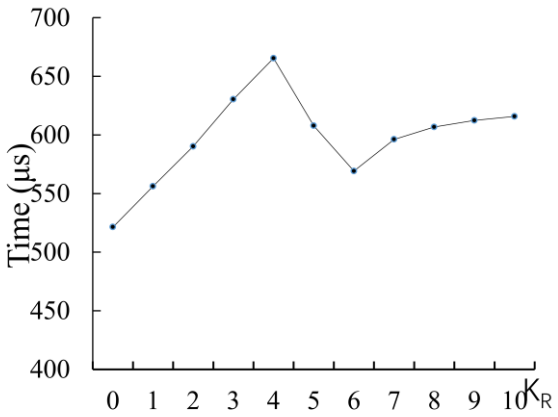

Figure 8. Time consumption.

As shown in the Figure 8, when the randomization parameter is between 2 to 6, the time consumption is not stable. When K_R takes 8,9 and 10, the time from the application object to the memory address of the returned object tends to be stable. It follows that the larger the K_R, the higher the randomization, and the larger the memory address offset applied between the two objects, which theoretically the space occupied by the cache will also increase. Combine the above two pictures, K_R=8 is optimal which can achieve the desired defend effect with lower space consumption.

7. CONCLUSION

JIT code reuse attack is still a popular attack method which uses existing code in memory. In this paper, we discuss the principle of these attacks and design an efficient mitigation algorithm that is inspired by the principle of randomization. Our algorithm takes higher entropy based the randomization parameters into account and give consideration to both effectiveness and practicality of the algorithm. The heart of the algorithm lies in randomized splitting, new merge logic, and random layout for the whole heap memory space. According to the experiments, we obtain a more appropriate randomization parameter, which meets the requirements for security and performance.

8. REFERENCES

[1] S. Bhatkar, R. Sekar, and D. C. DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In USENIX Security Symposium. Baltimore, Maryland, USA.

[2] Piromsopa. K, Enbody. R. J. 2011. Survey of Protections from Buffer-Overflow Attacks . Engineering Journal. DOI: 10.4186/ej.2011.15.2.31.

[3] Rinard. M, Cadar. C, and Dumitran. D. 2004. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). Computer Security Applications Conference, IEEE Computer Society. Tucson, Arizona, USA.

[4] Jang Y, Lee S, and Kim T. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. ACM Sigsac Conference on Computer and Communications Security. Hofburg Palace, Vienna, Austri.

[5] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D Song. 2015. VTint: Defending Virtual Function Tables' Integrity. In 22nd Annual Network and Distributed System Security Symposium. San Diego, CA, USA.

[6] V. Pappas, M. Polychronakis, and A. D. Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In IEEE Symposium on Security and Privacy. San Francisco, CA, USA.

[7] C. Zhang, T. Wei, and Z. Chen. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. Security and Privacy, IEEE. Berkeley, CA, USA.Ding, W. and Marchionini, G. 1997. A Study on Video Browsing Strategies. Technical Report. University of Maryland at College Park.

[8] D. Litchfield. 2003. Defeating the stack based buffer overflow exploitation prevention mechanism of microsoft windows 2003 server. In Black Hat Asia. Singapore.

[9] Kugle. C, Müller. T. 2015. Separated Control and Data Stacks to Mitigate Buffer Overflow Exploits. In 36th IEEE Symposium on Security and Privacy, S&P . San Jose, CA, USA.

[10] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A-R. Sadeghi, and T. Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In 36th IEEE Symposium on Security and Privacy, S&P. San Jose, CA, USA.

[11] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. 2011. On the expressiveness of return-into-libc attacks. In Proceedings of the 14th international conference on Recent Advances in Intrusion Detection. Menlo Park, CA, USA.

[12] Gupta. A, Kerr. S, and Kirkpatrick. M. S. 2013. Marlin: A Fine Grained Randomization Approach to Defend against ROP Attacks. Network and System Security. Heidelberg, Germany.

[13] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. 2014. SoK: Automated software diversity. In 35th IEEE Symposium on Security and Privacy, S&P. Berkeley, CA, USA.

[14] Backes. M. 2014. Oxymoron: making fine-grained memory randomization practical by allowing code sharing. Usenix Conference on Security Symposium. San Diego, CA, USA.

[15] Snow. K. Z, Monrose. F, and Davi. L. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address

Space Layout Randomization. Security and Privacy, IEEE. Berkeley, CA, USA.

[16] Akritidis. P, Cadar. C, and Raiciu. C. 2008. Preventing Memory Error Exploits with WIT. IEEE Symposium on Security and Privacy. Berkeley, CA, USA.

[17] Gu. Y, Lin. Z. 2016. Derandomizing Kernel Address Space Layout for Memory Introspection and Forensics. ACM Conference on Data and Application Security and Privacy. New Orleans, Louisiana, USA.

[18] Yang. Y, Guan. Z, and Liu. Z. 2014. Protecting Elliptic Curve Cryptography Against Memory Disclosure Attacks. Information and Communications Security. Springer International Publishing. Hong Kong, China.

[19] Cohen. F. B. 1993. Operating system protection through program evolution. Elsevier Advanced Technology Publications.

[20] Franz. M. 2010. E unibus pluram: massive-scale software diversity as a defense mechanism. Proceedings of the New Security Paradigms Workshop. Cumberland Lodge, Windsor, UK.

[21] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. 2011. Moving Target Defense. Chapter 4.

[22] Evtyushkin. D, Ponomarev. D, and Abughazaleh. N. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. International Symposium on Microarchitecture, IEEE. Taipei, Taiwan, China.

[23] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In Annual Computer Security Applications Conference. Miami Beach, Florida, USA.

[24] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. 2012. ILR: Where'd My Gadgets Go? In IEEE Symposium on Security and Privacy. San Francisco, CA, USA.

[25] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In ACM Conference on Computer and Communications Security (CCS). Raleigh, NC, USA.

[26] Giuffrida. C, Kuijsten. A, and Tanenbaum. A. S. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In the 21st USENIX Security Symposium. Bellevue, WA, USA.

[27] Barua. R. K, Smithson. M. 2013. Binary rewriting without relocation information. University of Maryland, Tech. Rep.

[28] Zhan. X, Zheng. T, and Gao. S. 2014. Defending ROP Attacks Using Basic Block Level Randomization. IEEE Eighth International Conference on Software Security and Reliability-Companion. San Francisco, CA, USA.