ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ

ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ & ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Securing the dynamic memory manager module

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήστος Ο. Ανδρίκος

Επιβλέπον: Δημήτριος Ι. Σούντρης

Επίκουρος Καθηγητής

Αθήνα Ιούλιος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ & ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Securing the dynamic memory manager module

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήστος Ο. Ανδρίκος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Ιουλίου 2013

....	....	....
Δημήτριος Σούντρης	Κιαμάλ Ζ. Πεκμεσζή	Γιώργος Οικονομακός
Επ. Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π.	Λέκτορας

....

Χρήστος Ο. Ανδρίκος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Χρήστος Ο. Ανδρίκος 2013

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Ο διαχειριστής δυναμικής μνήμης αποτελεί μηχανή πεπερασμένων καταστάσεων. Προορίζεται για να διαχειρίζεται τη δυναμική μνήμη την οποία απαιτεί κάθε διεργασία για να εκτελεστεί ορθά και πάντοτε με κύριο στόχο τη μεγιστοποίηση της ευρωστίας του όλου συστήματος.

Σήμερα οι εφαρμογές απαιτούν όλο και περισσότερη μνήμη. Επιπλέον ο ρυθμός συναλλαγών μεταξύ εφαρμογής και δυναμικά δεσμευμένης μνήμης, αυξάνει ραγδαία, ως αποτέλεσμα της ραγδαίας αύξησης των χρηστών και των προς στις εφαρμογές απαιτήσεων τους.

Σημειωτέον είναι το γεγονός ότι οι διαχειριστές δυναμικής μνήμης είναι σχεδιασμένοι να συνδιαλλάσσονται με ασφαλείς εφαρμογές προερχόμενες από υπεύθυνο προγραμματισμό. Ωστόσο, σήμερα η πλειοψηφία των εφαρμογών δεν είναι ασφαλής λόγω τον χρηστοκεντρικών συστημάτων και των άναρχων γλωσσών προγραμματισμού.

Συνεπώς δημιουργούνται κενά ασφαλείας που οδηγούν σε επιθέσεις κατά τις καίριας οντότητας του σωρού. Αντικείμενο λοιπόν της παρούσης διπλωματικής εργασίας είναι η μελέτη σεναρίων επιθέσεων και η ανάπτυξη μηχανισμών καταπολέμησής τους.

Λέξεις κλειδιά

Διαχειριστής δυναμικής μνήμης, Κενό ασφαλείας, Σωρός, Υπερχείλιση

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον αγαπητό κ. Δ. Σούντρη ο οποίος εμπιστεύτηκε στο πρόσωπό μου την ανάθεση της παρούσης διπλωματικής εργασίας. Τις θερμές μου ευχαριστίες, στους κυρίους Ηρακλή Αναγνωστόπουλο και Αλέξανδρου Μπάρτζα που με την καθοδήγηση και την υπομονή τους συνέβαλαν τα μέγιστα στην εκπόνηση της.

NATIONAL TECHNICAL UNIVERSITY OF ATHENS

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

DIVISION OF COMPUTER SCIENCE

SECURING THE DYNAMIC MEMORY MANAGER MODULE

DIPLOMA THESIS

CHRISTOS ANDRIKOS

Table of Contents

Abstract xiii

Key Words xiv

1 INTRODUCTION 1

1.1 Process' Memory 1

1.1.1 Process' Memory Segmentation 1

1.1.2 Segments Exploration 2

1.2 Memory Transactions Leading To Dynamic Memory Managers 4

1.3 Dynamic memory mangers' overview 5

1.3.1 Free-list based allocators 6

1.3.2 BiBOP-style allocators 7

1.3.3 Allocators' functionality 9

1.4 Need for Security 10

2 Heap Attack Models 13

2.1 Basic heap attack factors 13

2.1.1 Presence of memory errors 13

2.1.2 Applications class 13

2.1.3 Window of repeated attacks 14

2.2 Basic heap attack schema 16

2.3 Dynamic Memory Manager's vulnerability 17

2.4 Attack Models Classification Based On Its Immediate Target 18

2.5 Heap Attacks Presentation 18

2.5.1 Daggling Pointer Vulnerability (DPV) Attacks 19

2.5.2 Buffer Overflows Attacks 19

2.5.3 Off by one errors 24

2.5.4 Format string attacks 24

2.5.5 Heap spraying attacks 25

2.5.6 Type overflows 27

3 RELATED WORK **29**

3.1 Canaries 30

3.1.1 Terminator Canaries 30

3.1.2 Random Canaries 30

3.1.3 Random XOR Canaries 31

3.1.4 Changing target not philosophy: From stack to heap 32

3.2 Metadata pointer encoding (Encrypted Pointers) [8] 33

3.2.1 Pointer encoding in mono-linked lists schema 33

3.2.2 Pointer encoded in dual-linked list schema 36

3.3 OpenBSD Allocator Schema 37

3.4 DieHard: Probabilistic Memory Safety [2] 37

3.4.1 Overview 37

3.4.2 Stand-alone-mode 38

3.4.3 Replicated Mode 40

3.5 Die Harder [11] 43

3.5.1 Overview 43

3.5.2 Points analysis 43

4 DEVELOPED METHODOLOGIES **47**

4.1 Dynamic memory managers' perspectives 47

4.1.1 Dynamic memory manager as server 48

4.1.2 Dynamic memory manger as finite-state monolithic resource manager .. 49

4.2 Functionalities VS Dynamic Memory Allocator's Security 51

4.3 Heap Organization Overview 51

4.4 A real attack scenario 53

4.5 Protection Security Schemas 53

4.5.1 Canaries 54

4.5.2 Encrypted Indirect Pointers 55

4.5.3 Encrypted lists 57

4.6 Basic components 58

4.7 Encryption schema 60

4.8 Keys/Random Seeds organization algorithms 62

4.9 mini-Allocator, mini-Heap implementation 64

5 METRICS – EVALUATION **67**

5.1 Overview to testing environment and scenario 67

5.2 Benchmarking DL, DieHarder and clear dmmlib 68

5.3 Benchmarking dmmlib with canaries securing mechanism enabled 70

5.4 Benchmarking dmmlib with encrypted indirect pointers securing mechanism enabled 72

5.5 Benchmarking dmmlib with encrypted lists securing mechanism enabled 74

5.6 Benchmarking dmmlib with encrypted list validation mechanism enabled 75

5.7 Benchmarking dmmlib with mixed security schemas 77

5.8 Decision schema – benchmarking explanations 79

5.9 Overall Evaluation 80

6 FUTURE WORK **81**

6.1 Server side allocator 81

6.2 Data integrity 82

Watchdogs 82

Recovery 82

6.3 Embedded systems' point of view 83

Conclusion **85**

Appendix **86**

List of Figures **86**

Chart list **88**

References **89**

Abstract

Dynamic memory manager is a finite state machine. It is designed to manage the dynamic memory that an application demands in order to work robustly, as well as efficiently.

Nowadays applications demand large memory portions. Moreover the transaction ratio between applications and dynamic allocated memory increases rapidly due to the swift growth of users' number and demands.

On the other hand dynamic memory managers is a mechanism implemented in order to interact with trusted applications, meaning applications that are self secured resulting from responsible programming. However, nowadays, the majority of applications are not self secured. The combination of scripting languages, fast development and user centralized environments, ends up with applications full of security flaws.

Many of these flaws can lead to attacks that are based on the betrayed vulnerabilities by the application dynamic memory manager, which assumes the robustness of application's security.

Consequently the **purpose** of this paper is to find out the golden section between the security issue and the dynamic memory manager's performance request. Thus it introduces several protection schemas as well as it presents in detail their internal implementation. Moreover it evaluates the specific security mechanisms through a series of benchmarking tests.

Key Words

Data: The raw information that is used by any process in producer-consumer schema

Metadata: Data used to define structures and organization of data

Chunk: Portion of memory

Block: Portion of memory (see chunk)

Raw block: Huge portion of memory

Page: The fixed size portion that kernel divides memory, in order to manage it properly

System call: Functions running in kernel space, called by user space processes, in order to take care of critical management or hardware distribution issues

Hack: Cheating a DFA defined mechanism causing a behavior that the constructor hadn't predict aiming to take the "control of a system"

Memory leaks: Using more than the needed memory or never returning portions of memory back to the system

Security flaw: A defect of a software application or component that, when combined with the necessary conditions, can lead to a software vulnerability

Vulnerability: Set of conditions that allows violation of an explicit or implicit security policy

Exploit: The way to take advantage of vulnerability

Buffer: Memory portion in physical or virtual layer

Buffer overflow: Accessing and changing a specific buffer's content without permissions by using the continuous memory property

NOP-padding: NOP instructions (No OPeration) padding

ASLR: Address Space Layout Randomization

Adaptor morpheme: Term being lent from software engineering fields meaning a mechanism that converse with different implementation and APIs offering the same functionality

OS: Operating System

Process' Memory Footprint: Total memory requests and returns that took place by a process in a specific OS

1 INTRODUCTION

1.1 Process' Memory

Address Space defines the discrete addresses which can be used by system's processes for loading/storing operations.

Virtual Address Space (VAS) is the mechanism that distinguishes the physical address space that the system affords, from the address space that the applications use. This mechanism offers to every process the illusion of being able to use the entire address space that ranges according to the **processor's word's length**[1]. This results to processes having the same address space while running in the same system. An extra benefit is the idea of the continuous process memory, fact not true because of the **ASLR**.

1.1.1 Process' Memory Segmentation[2]

Process memory is divided to five main segments:

1. text
2. data
3. bss
4. stack
5. heap

Each segment represents a special portion of memory that is set aside for a certain purpose and has specific read/write permissions.

[1] Modern systems having processor's word length 64+ bit, offers the illusion that process can use practically infinite memory space

[2] Process' address space and memory are two distinct concepts. The first one is fixed for every process being practically an infinite set (ranges according to the processor's word length thanks to VAS). The second one is agile conforming to the application's needs.

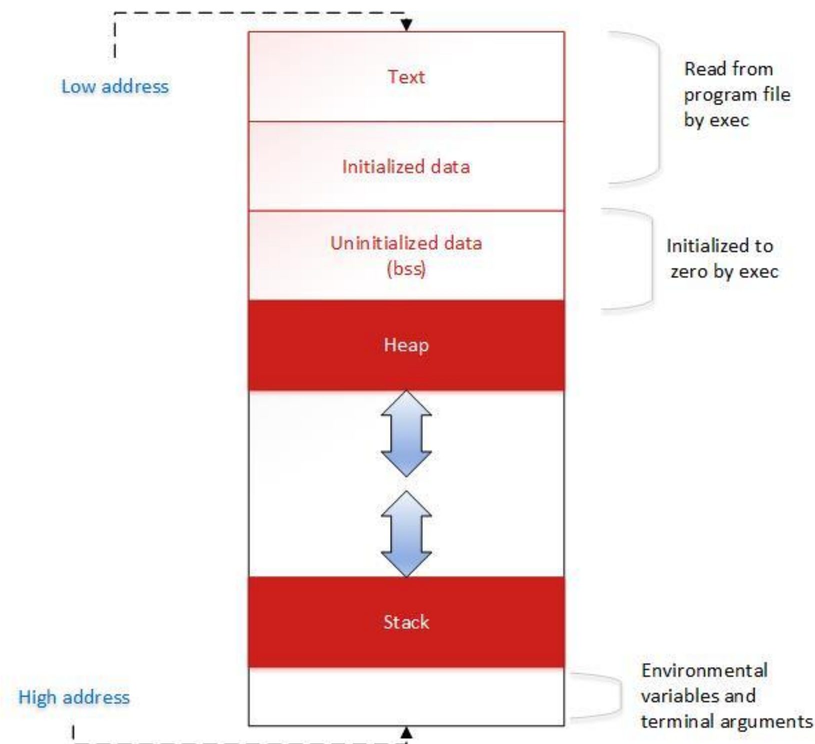

Figure 1-1: *Process' memory segmentation defined in the virtual address space*

1.1.2 Segments Exploration

Text or **code segment**: Contains the assembled machine language instructions that are to be executed. The execution of the instructions in the specific segment is non-linear, thanks to braches resulting from the well known high level control structures and function calls.

Write permission for the specific segment is disabled as it is not aimed to be used to store variables but only code segments which remain the same during any application's execution. In this way people are prevented from actually modifying the program code. Moreover any attempt to write directly to this segment of memory will cause the program to alert that something malicious happened and usually to be terminated. Finally as it is displayed in Figure 1.1, text segment consists of the lowest addresses lain on process memory organization.

Data and **bss segments** store global and static program variables and are placed consecutively after the text segment (Figure 1.1). On one hand the **data segment** is filled with the initialized global variables, strings and any other constants that are used by the process. On the other hand the **bss** stores all the uninitialized counterparts.

Both segments are writable due to the fact that global variables' values usually alter during the execution. However because of the constant crowd of all global variables, strings and constants, during execution, both segments have **fixed size**.

The **stack segment** is the first from the two segments whose size is variable, meaning it can grow larger or shrink smaller depending on the execution's needs. It is used as a temporary scratchpad to store **context info** between function calls, giving the process the ability to remember caller's context after every internal context switch.

When a process calls a function, that function contains its own set of passed variables and its code is placed at a different memory location in the text segment than the expected one pointed by the program counter register. Consequently a new record has to be generated, containing the program **c**ounter's current value (pc/eip register) and the essential context before the switch ordered by function call mechanism.

After its generation, the new stack record is placed (**push**) in the stack (FILO) segment whose **size grows**. When the called function ends the record is used to return to the hardware to the state of the caller one. At this moment we got the **pop** functionality, decreasing the size of the stack.

Of course the size of the stack segment is not exactly the same as the stack's one because this would not be efficient, however, generally the stack segment size "follows the stack's size" by reserving or returning, the necessary amounts of memory. It is notable that stack increases upwards form higher to lower memory addresses and shrinks downwards as pointed in Figure 1-1: *Process' memory segmentation defined in the virtual address space*.

Last but not least, the **heap segment** is used for the rest of the program variables, meaning all the dynamic allocated objects whose sizes are usually runtime estimated. It is the second segment that has not fixed size as the stack segment does.

All of the memory within the heap segment is managed by the allocating and deallocating algorithms – mechanisms, which respectively reserve regions of memory in the heap for use, or release reservations to allow portions of memory to be reused for later ones. Moreover, according to Figure 1.1, heap segment instead of growing upwards likes stack segments does, it grows downwards to higher memory addresses.

1.2 Memory Transactions Leading To Dynamic Memory Managers

According to aforementioned schema, every process needs an amount of memory. The first three segments are defined absolutely by the kernel when the process is born. Also the starting points of both stuck and heap segments are defined in the same time and way. A specific amount of minimal memory is also given to the process by kernel for its stack segment implementation which later can be adjusted according to the function call depth of the process.

The heap segment however is not treated in the same way. Let us consider the fact that any process and the kernel as entities of the same environment, implement a scheme **of consumer – producer** respectively. This means that kernel offers amounts of memory and process consumes them to satisfy its needs. Moreover the **scheme works vice versa** when the process returns amounts of memory, that is are not useful anymore, back to system in order to be redistributed.

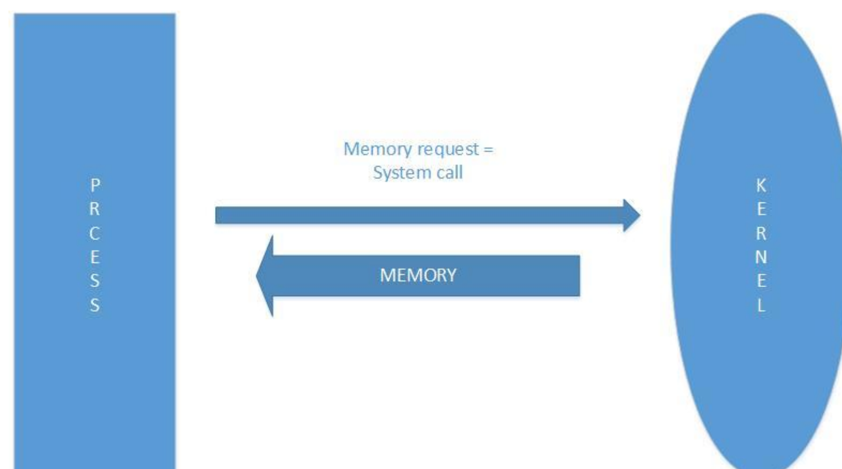

Figure 1-2: *Requesting memory by using immediate system calls*

In order to be implemented this communication – transaction mechanism, a set of special system calls exists (mmap, munmap, sbrk etc.). However the specific family of system calls manages memory in large portions, whose actual size varies through kernels and hardware platform implementations. Moreover, **large system calls ratio** leads to enormous delays, especially in modern multiprocessing systems.

The solution to these problems is the introduction of the **dynamic memory mangers,** which implement an extra mid-layer in the aforementioned scheme. Their job is to request and return memory portions, using system calls in an optimal and transparent way for the process and the system too.

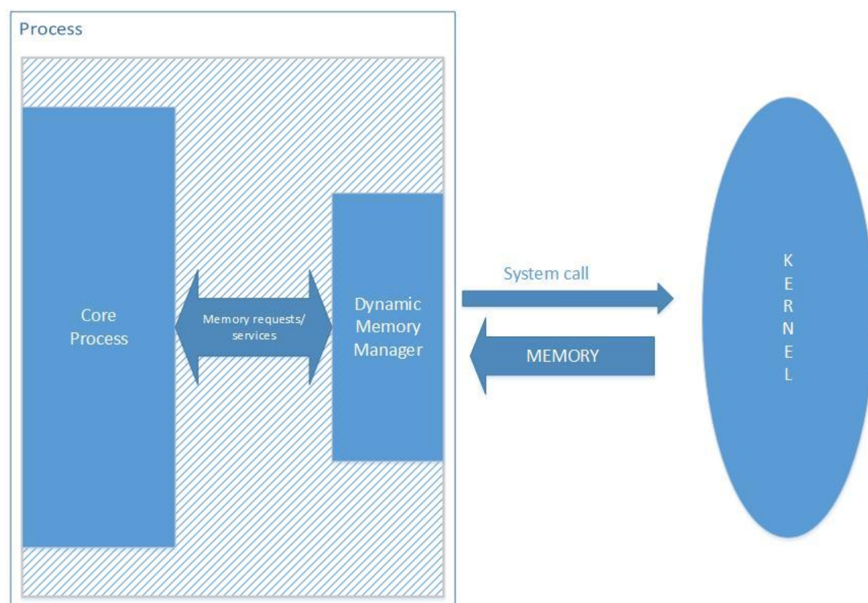

Figure 1-3: *Dynamic memory manger implementing the mid-layer memory exchange*

1.3 Dynamic memory mangers' overview

The functions that support memory management for C and C++ (malloc and free, new and delete respectively) are implemented in the C runtime library. Different operating systems and platforms implement these functions differently, with varying design designs and features. In nearly all cases, the algorithms underpinning these allocators were primarily designed to provide **rapid allocation and deallocation** while

maintaining low fragmentation aiming to **efficient memory usage** and entire host system's robustness.
Before continuing it is essential for the reader to clarify that heap has different **semantics** for the allocator and the core process itself. Process do use and modify the content of the memory blocks setting **data** as the center of its attention. On the other side, the **DMM** is oriented to the **memory blocks administration** independently from their content.

1.3.1 Free-list based allocators

The memory managers used by both Windows and Linux are freelist-based: they manage freed (and in use) space in linked lists, generally organized into bins corresponding to a range of object sizes. Figure 1-4 illustrates a fragment of freelist-based heap as it is organized by the Doug Lea allocator (DLmalloc). Version 2.7 of the Lea allocator forms the basis of the allocator in GNU libc.

Inline metadata
Like most freelist-based allocators, the Lea allocator prepends a header to each allocated object that contains its size and the size of the previous object. This metadata allows it to efficiently place freed objects on the appropriate free list (since these are organized by size), and to coalesce adjacent freed objects into a larger chunk In addition, freelist-based allocators typically thread the freelist through the freed chunks in the heap. Freed chunks thus, contain the size information (in the headers), as well as pointers to the next and previous free chunks on the appropriate freelist (inside the freed space itself). This implementation has the significant advantage over external free-lists of requiring no additional memory to manage the linked list of free chunks. Unfortunately, inline metadata also provides an excellent attack surface. Even small overflows from application objects are likely to overwrite and corrupt allocator metadata. This metadata is present in all applications, allowing application-agnostic attacks techniques.
Numerous ways have been implemented exploiting the inherent weakness of freelist based allocators occurring from process corruption until arbitrary code execution (see Section 2.5)

Figure 1-4: *A fragment of a freelist-based heap, as used by Linux and Windows. Object headers precede each object, which make it easy to free and coalesce objects*

1.3.2 BiBOP-style allocators

In contrast to Windows and Linux, FreeBSD's PHKmalloc [10] and OpenBSD's current allocator (derived from PHKmalloc) employ a heap organization known as segregated-fits BiBOP-style. Figure 1-5 provides a pictorial representation of part of such a heap. The allocator divides memory into contiguous areas that are multiple of the system page size (typically 4K). This organization into pages gives rise to the name "Big Bag of Pages", or "BiBOP". BiBOP allocators were originally used to provide cheap access to type data for high-level languages, but they are also suitable for general purpose allocation. In addition to dividing the heap into pages, both PHKmalloc and OpenBSD's allocator ensure that all objects in the same page have the same size, in other words, objects of different sizes are segregated from each other. The allocator stores object size and other information in metadata structures either placed at the start of each page (for small size classes), or allocated from the heap itself. A pointer to this structure is stored in the page directory, an array of pointers to each managed page. The allocator can locate the metadata for individual pages in constant time by masking off the low-order bits and computing an index into the page directory. On allocation, PHKmalloc first finds a page containing an appropriately sized free chunk. It maintains a list of non-full pages within each size class. These freelists are threaded through the corresponding page metadata structures. Upon finding a page with an empty chunk, it scans the page's bitmap to find the first available free chunk, marks it as allocated, and returns its address. [3]

Page-resident metadata:

As opposed to freelist-based heaps, BiBOP-style allocators generally have no inline metadata: they maintain no internal state between allocated objects or within freed objects. However, they often store heap metadata at the start of pages, or within metadata structures allocated adjacent to application objects. This property can be exploited to allow arbitrary code execution when a vulnerable application object adjacent to heap metadata can be overflowed.

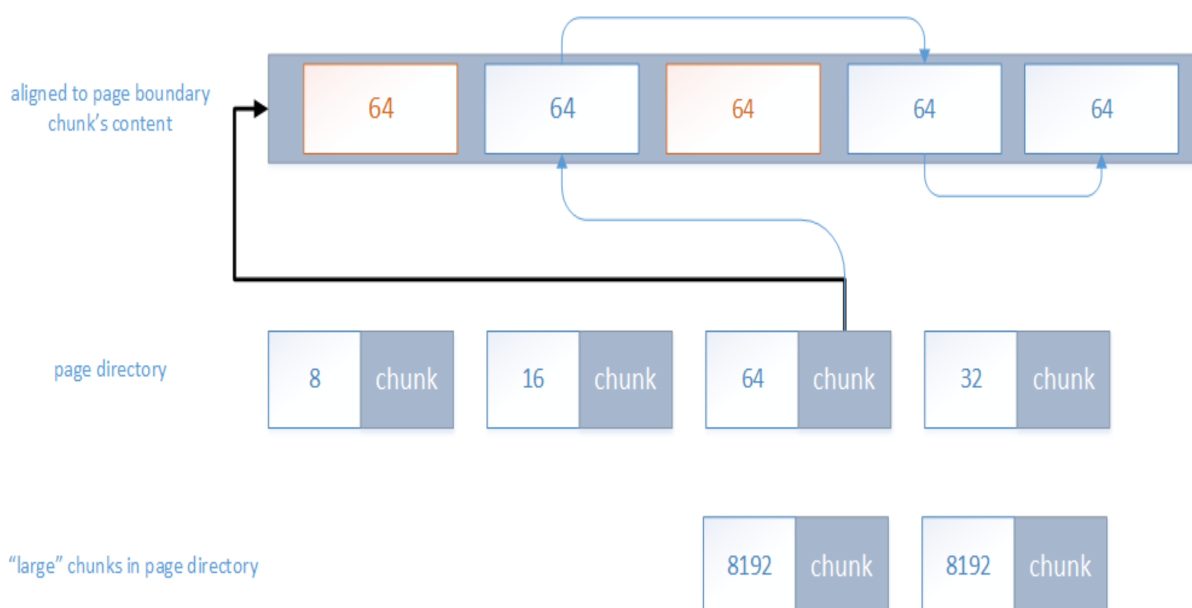

Figure 1-5: *A fragment of a segregated-fits BiBOP-style heap, as used by the BSD allocators (PHKmalloc and OpenBSD). Memory is allocated from page-aligned chunks, and metadata (size, type of chunk) is maintained in a page directory. The light lines indicate the list of free objects inside the chunk and the thick one indicates the owner of the chunk's interior.*

1.3.3 Allocators' functionality

As it has been aforementioned, allocators as mechanisms have been implemented aiming to increase application's performance and to **assist the production of platform independent code**.

The last "functionality" is obvious. Having an allocator to play the **role of the adaptor morpheme** among so many kernel implementations, the compatibility problems, at least for memory management routines, are absolutely vanished.

The issue of performance is the next in priority request that allocator as mechanism fulfills. Investigating the way that the heap increases and decreases, reserving and returning to the system portions of memory respectively, two things are noteworthy: **the number of system calls** that take place and the **utilization of the reserved memory**.

Starting with the **system calls**, every memory exchange, without any **dmm manager** implementation, between process and kernel would lead to one system call at least. This should end up saturating the whole system due to **locks** and **hardware conflicts**.

The way that an allocator solves the specific problem is that it collects all the memory requests, interacts with the OS through a large size system call and then it serves the small initial requests[3].

Moreover the **memory footprint** of any application in case of dynamic memory allocator absence should be enormous compared with the one resulting by the same application using the allocator. This results from the fact that the quantum for any memory exchange between process and OS is the one **system page (page)**, portion of memory larger than almost any single memory request coming from a process[4].

Finally the extra functionalities that allocators offer vary from one to another. Some of them do split or merge free memory blocks in order to reuse them in a more

---

[3] Actually instead of gathering requests it triggers a system call and because system call returns more space than it needs, every following request is served immediately from this extra space.

[4] This is not always true. There may be process that request memory to allocate objects having multiple of the page size but this is not rule.

efficient way, others prefer to return free blocks back to system as soon as possible instead of others which try to apply spatial locality techniques in order to maximize the efficiency of memory reuse etc.

1.4 Need for Security

"Memory management is the act of managing computer memory. The essential requirement of memory management is to provide ways to **dynamically** allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed. This is **critical** to any computer system."

"Heap is actually a set of data contained in a specific memory space, described, defined and organized by specific metadata placed in the same address space."

The combination of the two paragraphs above, the definition of memory management and the description of the heap "vulnerable" organization, results to how essential is the **heap protection** against any evil wills. Moreover the variety of the already defined heap attack model is an extremely indicative clue justifying why security is essential.

As it mentioned before heap consist of data and metadata. Each one of this content class is the module of memory of two different finite state machines which actually corporate with, application and allocator respectively. However the huge problem is that **application's and allocator's behaviors have different risk thresholds for the entire system**. Allocator's behavior and metadata maintenance are more critical, with its corruption to create large system vulnerabilities. Moreover data's nature make them more vulnerable compared with metadata because of the probable security flaws due to application's interaction with external entities (users). Considering the fact that data and metadata are located in the same memory segment it is obvious the need for a mechanism that primarily secures the maintenance of metadata and secondly protect the application's data, in a memory field full of transaction and modifications.

2 Heap Attack Models

The specific section describes the meaning of a vulnerable and exploitable landscape for heap attacks, and presents some of the most common heap-attack techniques.

2.1 Basic heap attack factors

It is obvious that any kind of attack request a specific landscape to take place into. Consequently the power of potential heap attacks is affected by numerous factors such as the presence of memory errors, the type of application being attacked and mainly of the ability of the attacker to launch repeated attacks.

2.1.1 Presence of memory errors

First of all it must be mentioned that **dynamic memory mangers are not vulnerable** themselves. However because of the fact that they are finite state machines dedicated to operate efficiently and unmistakably with **invulnerable** applications, they provide important security flaws leaving the whole responsibility for security issues to applications' layer programmers. Consequently the most important factor for an application to be heap vulnerable is a **memory error existence**. Moreover this is not enough. It has to be combined with the attacker's ability to trigger the code path leading to the error.

2.1.2 Applications class

The attacker's ability to control heap operations is determined by the kind of the application. Many attacks require plenty of object allocations in order to force the heap into a predictable and vulnerable state. Other attack strategies assume unfragmented heap where the effects of heap operations are predictable. For example

the lack of holes between existing objects indicates new objects allocations on a fresh page.

Nowadays attacking in web browsers is something just less than trivial. The web browser attackers choose to “feed” browser with **Javascript** or **Flash** written scripts because in most current browsers, **Javascript** objects are allocated in the same heap as the internal browser data, which are essential for browsers natural functionality. **Heap Feng Shui** is a sophisticated technique that allows attacks to browsers running Javascript to ensure predictable behavior.

On the other hand server applications are generally less cooperative. This is because of the fact that application limits the number of allocation and deallocations. Moreover because of the fact that servers are implemented very carefully there is not much vulnerability to abuse. However the heap may be placed in a predictable state under a large or, better defined, a specific number of concurrent requests forcing the application to create a corresponding amount of contemporaneously live objects.

Finally the last class of application forbidding multiple object allocations limiting the attacker to exploits based on command line arguments in order to lead heap to an exploitable transformation.

2.1.3 Window of repeated attacks

A great factor in every effort to illegally gain control of a particular system is the amount of the repeated attacks the attacker can launch, until the attainment of its target. For example the most common server attack (not heap based) is the **flooding** aiming to the **denial of service**. Its implementation assumes plenty of clients to send at least **ping** requests to the server again and again. In other words the vulnerability that **flooding server attack** is based on, is that it permits large trial and client windows. Of course modern server systems recognize these attacks limiting both trial and client windows.

In order to create the heap proportional scheme, let us consider again the web browser example. In a web browser, if the first attempt fails causing the browser corruption (assumed that attacker wants to implement a malicious code injection in order to gain

a specific system's component control), the user may not attempt to reload the page. In this case the attack has only one chance to succeed per target. Moreover consider the attacks whose success depends on mid-target success. In this case if repeated attack window is not large enough there will not be any success.

On contradiction to web browsers where the attacks are user dependent, web servers restart after any corruption in order to ensure availability of service. This provides the attacker with more than one opportunity. Even if a server assumes that an attack is being occurred refusing to restart the attacker succeeded a **denial of service**.

Well, in case of large enough repeated attacks window, an attacker with probability of success will eventually succeed. This is obvious due to the fact that even if the attacker predicts heap organization wrongly, after many attack trials, his predictions will be closer to the reality making the application more and more vulnerable until its exploitation. However if the allocator decreases this probability the system maintainer got the time to analyze the attack and fix the security flaw before the attacker will succeed.

Such unpredictability is provided by randomization techniques such as ASLR (**A**ddress **S**pace **R**andomization). The exact way that ASLR supports security philosophy is by involving heap consistence of randomly arranging pages. Thus the system pages that the heap consists of, are not literally continuous forbidding overflows or any predictability in system page layer. For example, Shacham et al. showed that ASLR on 32-bit systems provides 16 bits of entropy for library address and can thus be circumvented after about 216 seconds [3]. On 64-bit systems providing 32 bits of entropy, however, the attack would require an expected 163 days. During this time, it would be feasible to fix the underlying error and redeploy the system.

Despite this is not enough, as system pages grow enormous while the platforms are evolving, being multiple of applications' usual memory requests. The vulnerabilities that ASLR vanishes in system page layer are moving down in the interior of every reserved by the application page. Consider the fact that every system page can be segmented to many heap entities described by metadata entities. Consequently the vulnerabilities move to the internal of system pages.

While one imagines a hypothetical supervisor program detecting incoming attacks, such a system would be hard even unpractical to be made. This is why despite the fact that it would be able to detect a series of crashes launched by a single source, attackers control large sophisticated distributed networks allowing them to coordinate large number of attack requests from random for the process sources. Shacham et al. discuss the limitations of such systems in more detail [3].

2.2 Basic heap attack schema

Before introducing the attack models, is essential to be described the stage sequence and the structural ingredients schema of any heap attack.

Starting with the structural ingredients schema there are three main attack entities to consider, the security flaw, the vulnerability and the exploit. These entities are fundamental for every attack in a way that their absence forbids any kind of heap attack setting the running process purely "untouchable".

- As **security flaw** is defined any defect of a software application or component that, when combined with the necessary conditions, can lead to a software vulnerability.
- **Vulnerability** is a set of conditions that allows violation of an explicit or implicit security policy
- **Exploit** is a piece of software or a technique that abuse vulnerability to violate an explicit or implicit security policy

Continuing to the attack "schedule" the following three stages describe the philosophy and the steps that are vital for an attack to be launched correctly having the planned side-effects:

1) **the exploitation stage**: vulnerabilities are located and exploitation mechanism is set up ready to be triggered
2) **the activation stage**: exploitation mechanism is triggered occurring illegal modifications to the process, its environment or causing malicious code execution on behalf of running process or even having administrator's credentials

3) **the influence stage**: the final stage of the attack, it is not always the one have been predicted by the attacker but it is sure that the system has been influenced in some illegal way

As well as any heap-attack is a coordination of attacker's "movements" and allocators behavior first and third of the tree following stages depends on the "**traps**" (exploitable states) that the attacker creates while the second stage involves the **ability of allocator to avoid them**.

2.3 Dynamic Memory Manager's vulnerability

As mentioned before dynamic memory managers offer **efficiency in the manner of memory utilization and system resources consumption**. This is succeeded through an internal organization extremely "fragile" and attack sensitive. This is because the coexistence of data and metadata in the same **address space**[5] made up from literally continuous portions of memory[6]. Moreover, because of the fact that DMM is actually the mid-layer between application and kernel layer, in the manner of memory transactions, it actually accesses variables and memory portions critical for the running process and the system itself.

[5] Address space must not be confused with sequential address space. Address space in the specific sentence means a large portion of memory where the applications have the higher possible credentials such as accessing and modifying all the contents of this area even metadata.

[6] Memory is not literally continuous. Address space in continuous. However some sequential addressees are assigned to continuous memory space, e.g. the internal of a reserved system page.

2.4 Attack Models Classification Based On Its Immediate Target

Investigating section 1.3 it is expected to classify the attacks to as many categories as the semantics of the heap.

Attacking heap metadata is the first class, containing every attack aiming to change the allocator's behavior through heap's metadata illegal modification.

The second class**, attacking the application's data**, consists of attack models aiming to offer the attacker an easily exploitable environment though a special behavior of a running process that the programmer has not forecasted.

Both attack model classes use memory manager and application vulnerabilities and success similar influences to the applications and to the systems that they were tried to. Their difference is only in semantics[7].

2.5 Heap Attacks Presentation

Formally the target of almost every attack in the heap is the malicious **code injection,** meaning to obligate CPU to execute a sequence of instructions not intended to do. There are barely some times that attack aims to **process corruption** causing denial of service. Every other attack types, like those aiming to just commit illegal modifications, are vestibules to process corruption or code injection attacks.

Despite the fact that attacking to vulnerabilities is a formally defined procedure aiming to specific mid targets until the ultimate one, it contains a pinch of art also. Moreover a specific type of vulnerability can be used in different ways in different attack scenarios. Consequently **the vulnerability and attack sets are not connected**

[7] For example a buffer overflow can occur either on heap metadata aiming to exploit allocator routines or on application's data aiming to exploit application's routines. Both mechanisms have the same target, to take control of a subsystem of the main system, both implement the same attack but they have different semantics!

by a one to one function. That is why it is essential attack models, vulnerabilities and exploits to be presented side by side in a scheme indicating how the vulnerability is exploited leading to a successful attack.

2.5.1 Daggling Pointer Vulnerability (DPV) Attacks

To justify the claim lain out in section **Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.**, consider the daggling pointer vulnerability. **Dangling pointers are pointers to memory locations that are no longer allocated**. In most cases dereferencing a dangling pointer will lead to a program crash.

However in heap memory, it could also lead to a **double free vulnerability**, where a memory location is freed twice. Such a double free vulnerability could be misused by an attacker to modify the management information associated with a memory chunk (chunk's metadata) and as a result could lead to a **code injection attack**.

This kind of vulnerability is not present in all memory managers, as some will check if a chunk is free or not before freeing it a second time. It may also be possible to write to memory which has already been reused, while the program think it is still writing to the original object.

This can also lead to further vulnerabilities which usually are much harder to be exploited in general programs than a double free. The possibility of exploiting them will most likely rely on the way the program uses the memory rather than by using the memory manager to the attacker's advantage.

2.5.2 Buffer Overflows Attacks

Buffer overflows exploits are the most common vulnerability abuse resulting in **code injection attack**. It requires knowledge or extremely accurate prediction of the heap organization and the allocator functionalities' side-effects.

The main goal of the specific technique is to overwrite essential heap fields by taking advantage of continuous memory in chunks, or by **abusing indirectly allocator's**

functionalities such as **memcopy**, defragmentation techniques etc. The **content** and the **type of the modified memory cells**[8] depend on **the goal and the class (data/metadata attack) of the tried attack**.

The first choice to launch a **data attack** aiming to overflow some critical for the application data. By this way the application will change its behavior and may create a larger vulnerability in the whole system organization. This is **an application centered attack** meaning it can be launched against only a specific application or application class and maybe for a restricted number of times.

The second strategy, more common and global than the first one, is to focus on metadata aiming **to pointer subterfuge** or **other critical memory cells modifications**. Pointer subterfuge, in other words pointer overwriting, is a very common exploit targeting to illegally change pointers values.

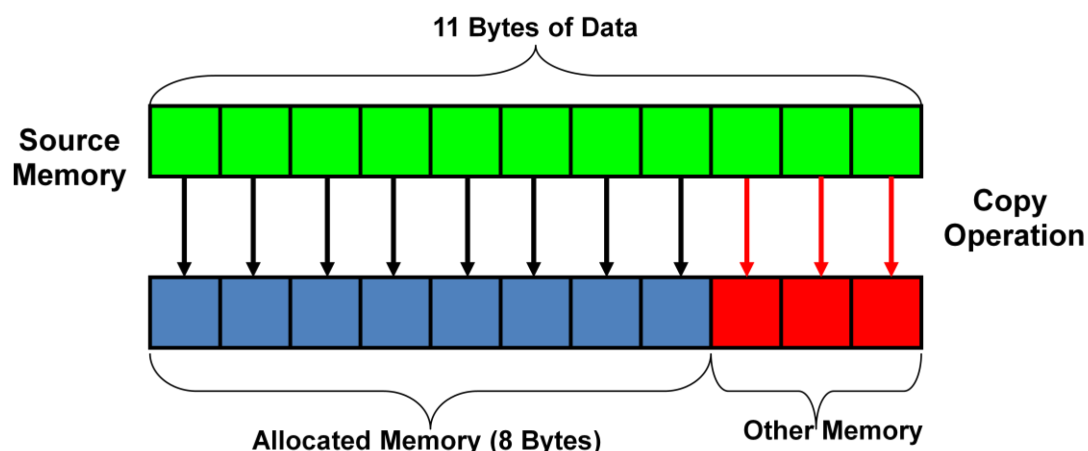

Figure 2-1: *Buffer overflow example using the embedded to the allocator copy functionality. The allocator mistakenly copies a larger buffer's content to a smaller one and as a result it overwrite the following fields. In case that the tree overwritten bytes are heap's metadata the corruption is close.*

Buffer overflows family contains two worthy to be presented attack schemes, forward consolidation and backward consolidation attacks.

---

[8] Memory portions of having specific size and specific significance such as an integer variable a string buffer etc.

2.3.1.1 Forward consolidation

The specific attack's target is to inject indirectly malicious code into the text or to the stack segment of the process's memory. By this way the attacker obliges the system to let its malicious code to be executed on behalf of the process, offering it the process's credentials. The idea is to illegally modify the values of **fd** and **bk** pointers (see Figure 2-2) so the allocator will do the job changing the text or stack segment's chosen field of the allocated by the application memory.

Investigating a real example in detail, let us consider the UNIX's free-list based allocator, the DL-allocator (aka Doug Lea allocator). Moreover we consider a snapshot of its state involving four memory chunks A, B, C and D. Assuming that chunk B is a free memory chunk, it has to be connected with C and D belonging to the maintained by the allocator free-list. Moreover chunk B is continuous with chunk A having its start just after the end of the data segment of A (buffer), while A is in use. Moreover a user's input is pending to be stored in buffer of chunk A.

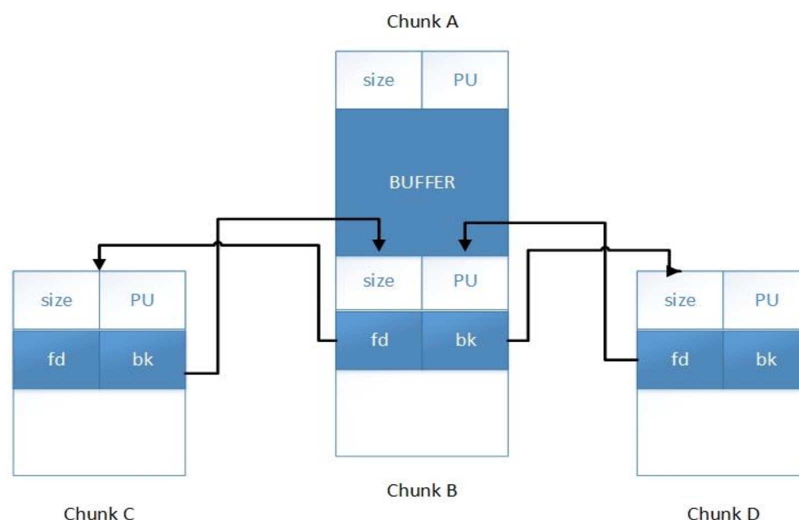

Figure 2-2: *Memory Chunks A, B, C, D organized in a vulnerable way just before the attack to DL-allocator is launched. Size: chunk's size, PU: previous chunk in use, fd: points the next free chunk, bk points the previous free chunk*

When the application requests an allocation, the chunk B is chosen to be removed from the free-list changing its state to "in use". Consequently the following code lines stored in the text segment will be executed:

b->fd->bk=B->bk;

b->bk->fd=b->fd;

as it is expected.

However imagine that, before the allocation of chunk B the attacker fed the application with an input that was to be stored in the buffer contained in chunk A, having size larger than the appropriate. In this way he could easily overwrite the two pointers fd, bk of free chunk B with an existing legal function's address that would be called later by the process and with a pointer to a heap area (maybe in the same buffer) containing a malicious code segment, respectively. Consequently when allocators try to remove from free-list chunk B, it will end up linking a pointer located in the text segment, with a malicious routine.

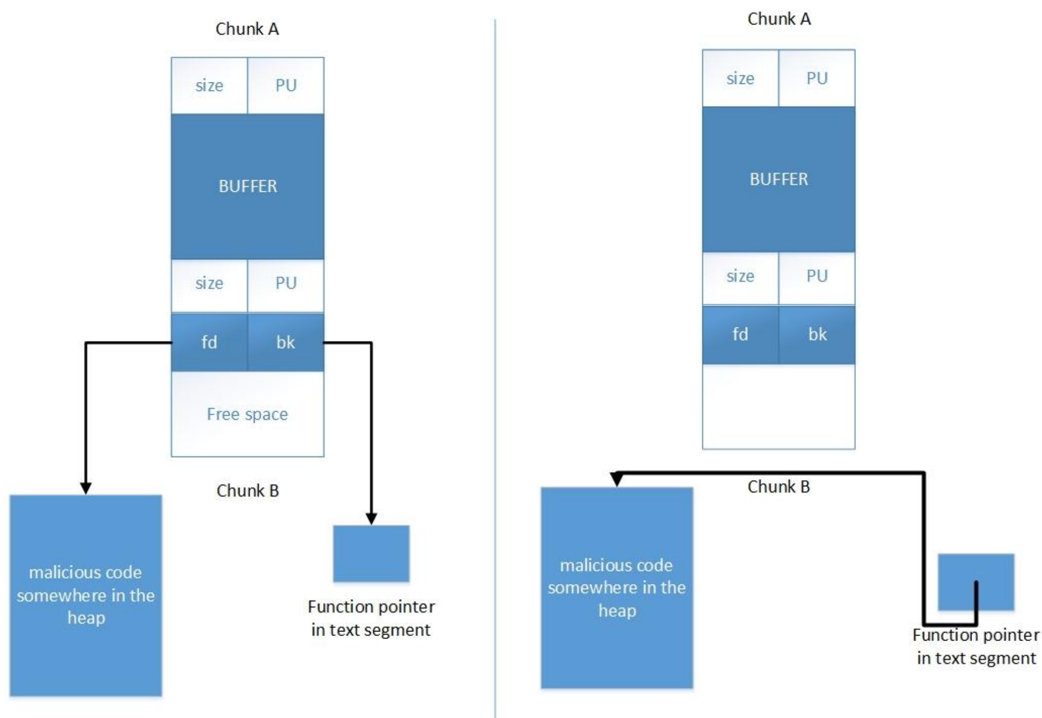

Figure 2-3: *Malicious code injection through indirect pointer overwriting. Allocator overwrite text area. In the left half it is presented the buffer overflow exploitation and its result is obvious in the right half.*

Finally, the attack succeeds when the application calls the pointed by the overwritten pointer function. Consequently there should be a malicious code execution instead of the legal function routine expected one.

2.3.1.2 Backward Consolidation

The second notable attack scheme, the backward consolidation, is similar to the aforementioned forward consolidation scheme. The target of the attack is once again the malicious code injection to the text segment through the modifications of fd and bk pointers of some free chuck before its allocation by the process. The distinguishing difference although of the two models, is that in backward consolidation attacks there is not actual overflow in any free chunks metadata. Having a large size buffer in his hands, the attacker creates **a fake free memory chunk**. Its metadata is customized by the attacker, who assigns fd and bk pointers the values that will connect a function pointer located in the text segment, with a malicious code routine, resulting to malicious code execution on behalf of the attacked process.

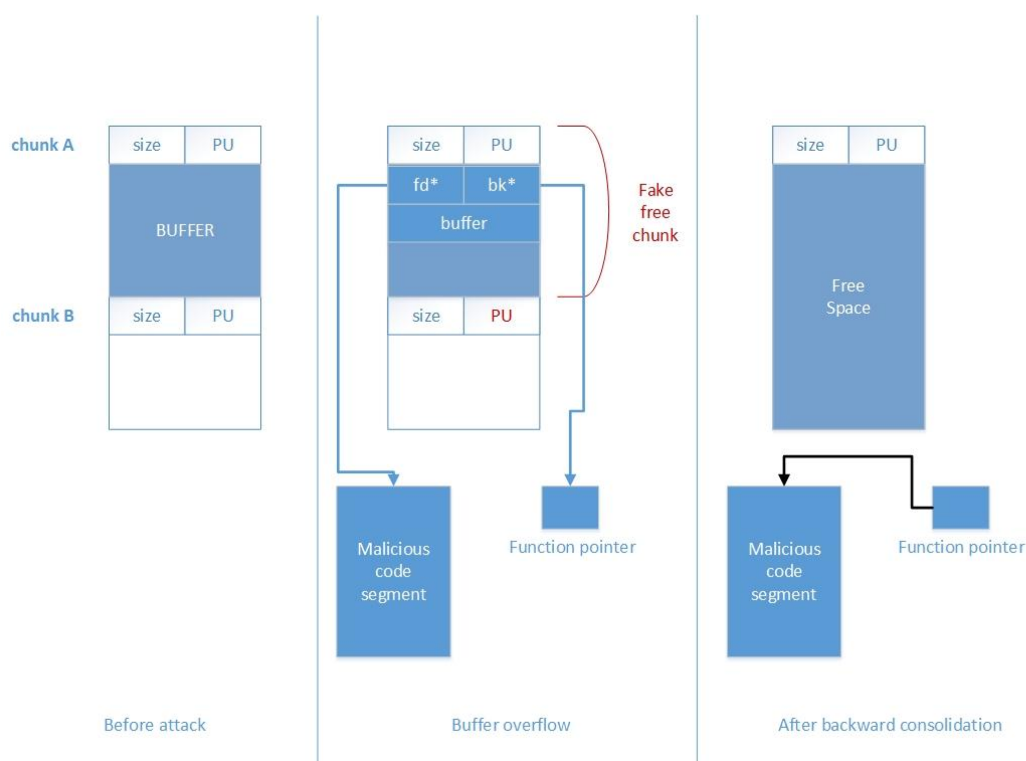

Figure 2-4: *Buffer overflow: Backward Consolidation scheme. Note that* ***fake chunk must be denoted as free****. This can be done in two ways. The first is overflowing the following chunk's metadata and* ***unset "previous in use" attribute*** *while the second one, in case of large enough buffer, is to create another continuous fake chunk pretending to be in use, and denoting to its PU attribute that the early one is free.*

2.5.3 Off by one errors

An off by one error is a special case of the buffer overflow. When an off by one occurs, **the adjacent memory location is overwritten by exactly one byte**. This often happens **when a programmer loops through an array but typically ends at the array's size rather than stopping at the preceding element** (because arrays start at 0). In some cases these errors can also be exploitable by an attacker. A more generally exploitable version of the off by one for memory allocators is an off by five, while these do not occur as often in the wild, they demonstrate that it is possible to cause a **code injection attack** when little memory is available. These errors are usually only exploitable on little endian machines because the least significant byte of an integer is stored before the most significant byte in memory.

2.5.4 Format string attacks

The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application. In this way, the attacker could execute code, read the stack, or cause a segmentation fault in the running application, provoking side-effects able to compromise the security or the stability of the system.

To understand the attack, it's necessary to understand the components that constitute it.

- the **Format Function** is an ANSI C conversion function, like printf, fprintf, which converts a primitive variable of the programming language into a human-readable string representation
- the **Format String** is the argument of the Format Function and is an ASCII Z string which contains text and format parameters, like:

  printf ("The magic number is: %d\n", 1912);

- the **Format String Parameter**, like %x %s defines the type of conversion of the format function

The attack could be executed when the application doesn’t properly validate the submitted input. In this case, if a Format String parameter, like %x, is inserted into the posted data, the string is parsed by the Format Function, and the conversion specified in the parameters is executed. However, the Format Function is expecting more arguments as input, and if these arguments are not supplied, the function could read or write the stack.

In this way, it is possible to define a well-crafted input that could change the behavior of the format function, permitting the attacker to cause denial of service or to execute arbitrary commands.

If the application uses Format Functions in the source-code, which is able to interpret formatting characters, the attacker could explore the vulnerability by using many already developed tools. [7]

2.5.5 Heap spraying attacks

A heap spray does not actually exploit any security issues but it can be used to make a security issue easier to exploit. A heap spray by itself cannot be used to break any security boundaries: a separate security issue is needed.

Exploiting security issues is often hard because various factors can influence this process. Chance alignments of memory and timing introduce a lot of randomness (from the attacker's point of view). A heap spray can be used to introduce a large amount of order to compensate for this and increase the chances of successful exploitation. Heap sprays take advantage of the fact that on most architectures and operating systems, the start location of large heap allocations is predictable and consecutive allocations are roughly sequential. This means that the sprayed heap will roughly be in the same location each and every time the heap spray is run.

Exploits often use specific bytes to spray the heap, as the data stored on the heap serves multiple roles. During exploitation of a security issue, the application code can often be made to read an address from an arbitrary location in memory. This address is then used by the code as the address of a function to execute. If the exploit can force the application to read this address from the sprayed heap, it can control the flow of execution when the code uses that address as a function pointer and redirect it

to the sprayed heap. If the exploit succeeds in redirecting control flow to the sprayed heap, the bytes there will be executed, allowing the exploit to perform whatever actions the attacker wants. Therefore, the bytes on the heap are restricted to represent valid addresses within the heap spray itself, holding valid instructions for the target architecture, so the application will not crash. It is therefore common to spray with a single byte that translates to both a valid address and a NOP instruction on the target architecture. This allows the heap spray to function as a very large NOP sled.

A common example of heap sprays for web browsers which are implemented in JavaScript and spray the heap by creating large strings. The most common technique used is to start with a string of one character and concatenating it with itself over and over. This way, the length of the string can grow exponentially up to the maximum length allowed by the scripting engine. Depending on how the browser implements strings, either ASCII or Unicode characters can be used in the string. The heap spraying code makes copies of the long string with shellcode and stores these in an array, up to the point where enough memory has been sprayed to ensure the exploit works.

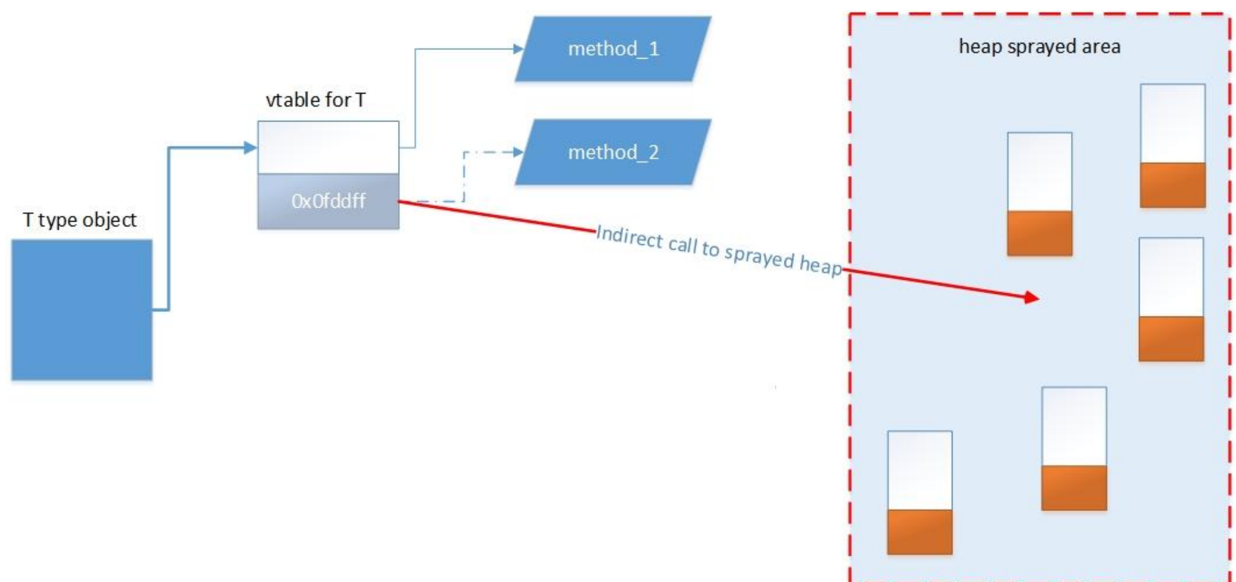

Figure 2-5: *Heap spraying attacks scheme. When the indirect call will happen it is extremely possible that the malicious code will be executed instead of the "called" method.*

2.5.6 Type overflows

Type overflows or **illegal type** casting attacks refer to attacks that aim to change the value of a counter that manages a loop or any other essential for the execution variable.

This offers the attacker from the opportunity to gain extra time until the next attack is ready to be fired, up to succeed the **denial of service**.

The specific attack can easily take place by feeding any variable which expect "input" of a certain type, with input belonging to larger type group (e.g. instead of unsigned int, -1 is fed).

3 RELATED WORK

In this section there are described in detail, the most successful heap security mechanisms according to bibliography. However, before moving on, it is essential to distinguish the protection policies to **active** and **passive** like vehicles' manufactures do.

- The class of passive security mechanisms consists of all the security mechanisms that aim to interact with the attack in way that they lead application or allocators to corrupt without being informed the reason why. Almost every passive heap security mechanism results in indirect segmentation fault corruption in case it is triggered by an attack. There are although some cases that the application does not corrupt because of segmentation fault but actually it starts behaving unpredictably.
- On the other hand, active security mechanisms react differently. They **detect** the launched attack as soon as possible and they either alert the allocator or the application about it, or **they handle the danger themselves** using special routines that they afford. Many times active security mechanisms cause program termination using segmentation fault. However this must not be misunderstood as program corruption[9] because it does not signaled by the OS but from the security mechanism that is a module of the application's allocator.

[9] For the needs of the specific paper program corruption is consider equivalent to program's irregular termination by the OS because of being dangerous for the system's behavior.

3.1 Canaries

The first described mechanism is the canaries or canary words. Canaries where introduced as stack's security mechanism. The terminology is a reference to the historic practice of using canaries in coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a biological warning system. They are known words, from the application's internal mechanisms (stack implementation libraries), that are placed between buffers and control data in stack in order to monitor buffer overflows. In this way, any buffer overflow is barely possible to take place without corrupting the guarding canary leading to a canary verification failure which respectively will raise an overflow alarm.

Bibliography indicates three types of canaries in use: **Terminator, Random** and **Random XOR.** Current stack protection mechanisms use all three kinds.

3.1.1 Terminator Canaries

The majority of buffer overflow attacks are based on certain string operations. This observation resulted to **terminator canaries.** Actually they are built by NULL terminators, CR, LF and -1. The undesirable result is that the canary in known. Consequently terminator canaries do protect stack from application or system failures, however any attacker can easily violate the specific security mechanism by potentially overwriting its known value and control information with mismatched values passing the canary verification.

3.1.2 Random Canaries

Random canaries are randomly generated, usually by an **entropy-gathering daemon**, in order to prevent an attacker to predict its value. Usually it is not plausible to read the canary for exploiting as well as only the buffer overflow protection mechanism knows its value.

Normally, a random canary is generated at program initialization, and stored in a global variable. This variable is usually padded by unmapped pages, so that

attempting to read it using any kinds of tricks that exploit bugs to read off RAM results to segmentation fault, terminating the program. It may still be possible to read the canary, if the attacker knows where it is, or can get the program to read from the stack.

However if a canary remains constant during many system cycles then it offers the attacker a large window of thought and attacks, Consequently it is possible for the attacker to be informed for its actual value by exploiting other application's vulnerabilities in order to gain read access in the stack locating and reading its value. Moreover a possible scenario should be that somehow the attacker locates the variable where the original canary value is stored to. Again it is extremely possible that the launched attack will eventually succeed.

3.1.3 Random XOR Canaries

Random XOR Canaries are random canaries produced in the aforementioned way (subchapter 3.1.2) that are XOR scrambled using all or a part of the protected buffer data. In this way once buffer's data is clobbered, the canary value mismatches the expected one, resulting to verification failure.

Random XOR Canaries have actually the same vulnerabilities. The distinguishing difference is that Random XOR Canaries makes the "read from stack" method much more complicated. This is because the fact that the attacker has to get the canary, the algorithm as well as the buffer contents in order to generate the original canary for re-encoding into the canary he needs to spoof the protection.

In addition Random XOR Canaries offer **security against a larger set of exploits** than Random Canaries. Certain types of attacks, not avoided by Random Canaries, such as overflowing a buffer in a structure into a pointer to change the pointer to point illegally a piece of control data, can be prevented. This is because of the XOR encoding which involves the actual content of the buffer to the canary assertion. Consequently buffer's content corruption will lead to canary corruption even if the corruption is occurred without immediate overflowing over canary (dangling pointer vulnerability can lead to this type of attacks in heap – see also 2.5.1 & 2.5.2).

3.1.4 Changing target not philosophy: From stack to heap

Despite the fact that canary's mechanisms were introduced for stack security they actually behave being applied to the heap organization too. However its reaction is a bit more complicated. In order to clarify this fact let us consider the way that actually canaries work applied to heap.

As it mentioned in section 1.4 heap contains data and metadata. The second entity's security is more critical to the system maintenance. Moreover the majority of the attacks, targeting metadata, are based in transactions, which modify data, between application and heap. Thus the need to restrict primarily the overflows form data to metadata fields as well as secondly generally any kind of overflow.

Actually this restriction is implemented in a large scale by canary use by many modern allocators. The key difference in the way that they implement the canary mechanism is the algorithm[10] that they use and the position of the guards they set. For example, list-based allocators use canaries to secure metadata fields by setting canary guards in the boundaries of memory chunks. On the other hand BiBOP style may use canaries for securing data sections setting guards between allocated objects located in the same system page as figure 3-1 presents.

As well as the attacker is not aware of the canary's value it is hard enough to predict it. Most of modern allocators, implementing the specific mechanism, produce canary guards runtime, using pseudo random mechanisms, making actually impossible any of their values to be predicted. However, the efficiency of the canary security model depends on the algorithm it is based on and the attack windows that it allows.

---

[10] The algorithm that a canary mechanism implements "describes" the way that the secondary structures (random seeds etc) are stored in the address space, the randomization techniques and the algorithms that the canary-guards are being produces.

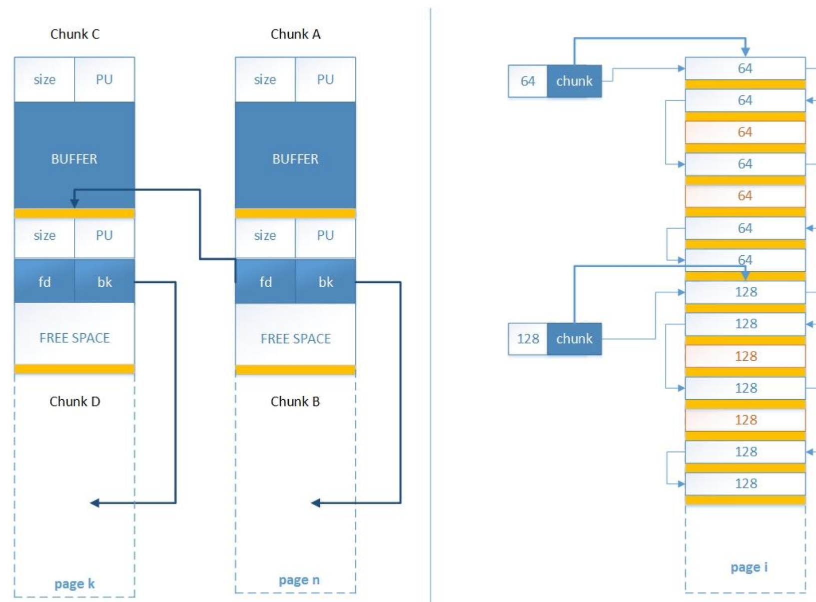

Figure 3-1: *In left canary mechanism application in free-list based allocator. In right canary mechanism application in BiBOP-style allocator.*

3.2 Metadata pointer encoding (Encrypted Pointers) [8]

3.2.1 Pointer encoding in mono-linked lists schema

The specific technique was introduced, mainly to protect any list-based allocator. The springboard of its creation is the large vulnerabilities set existing in free-list based allocators and especially in Doug Lea's dynamic memory manager which is mainly adopted by almost any DebIan based Linux distribution.

In case of free-list based allocators, bins for free chunks are either singly or doubly linked lists. If a bin is constructed as a singly linked list, only the forward link fd is used. For doubly linked lists, both fd and bk pointers are used. Thus buffer overflow attacks to the specific pointers abuse allocator's functionality to overwrite other more significant pointers (usually correlated with text segment's contents). In any case, link pointers can protect themselves by encryption.

In this scheme, the DMM encrypts a pointer linking free chunks immediately after its definition. The pointer is decrypted only when it is necessary the assigned address to be revealed in order to access its contents before the dereferencing. When the link pointer is simply copied to other structures, it is neither encrypted nor decrypted.

Consequently in case of the attacker wants to modify link pointers value, overwriting is not enough. Attacker must possess the key value and the algorithm, also, to encrypt it correctly so as latter decryption will not give unpredictable values.

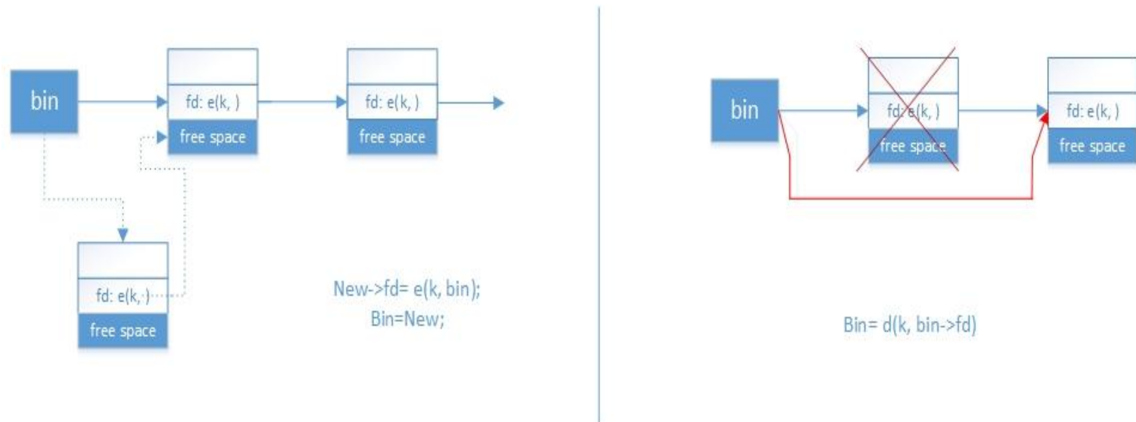

Figure 3-2: *Operation of singly linked lists. The encoding function is symbolized as e, while the decrypting one as d and key as k. Left half: the operation of insertion. Right half: the operation of deletion.*

For insertion the encoded address of the bin is copied into the fd cell of chunk New in order to be inserted, and the bin is set to the address of chunk New. Deletion also occurs at the head of the bin. The fd field of the first chunk is decrypted and assigned to the bin.

Other bins have the structure of a doubly linked list with a header element, where both the fd and the bk link fields are encrypted. Fig. 3-3 shows the insertion and deletion operations. For insertion, chunk New is inserted between two chunks, in which one or both of FD and BK are given. The address of chunk New is encoded by an encoding function **e** and a key **k**. Given the decrypted address of a chunk both previous and next chunks in the linked list can be accessed by decrypting the fd and bk pointers of the initial one. Deletion of chunk New simply copies encrypted pointers' values to the destinations as shown in Fig. 3-3. It needs to decode only two of the pointers' values to get the real target addresses.

For encryption and decryption, it is used the exclusive-OR (XOR) operation. The decision is made by considering performance as well as security, since too much

overhead renders data pointer encoding impractical. Guessing two (a and b) algorithm components from the third (e) is difficult given that e = a XOR b. With 32-bit keys, the probability of a successful break is $\frac{1}{2}^{32}$. XOR operations can be done by a single instruction in microprocessors counting no overhead.

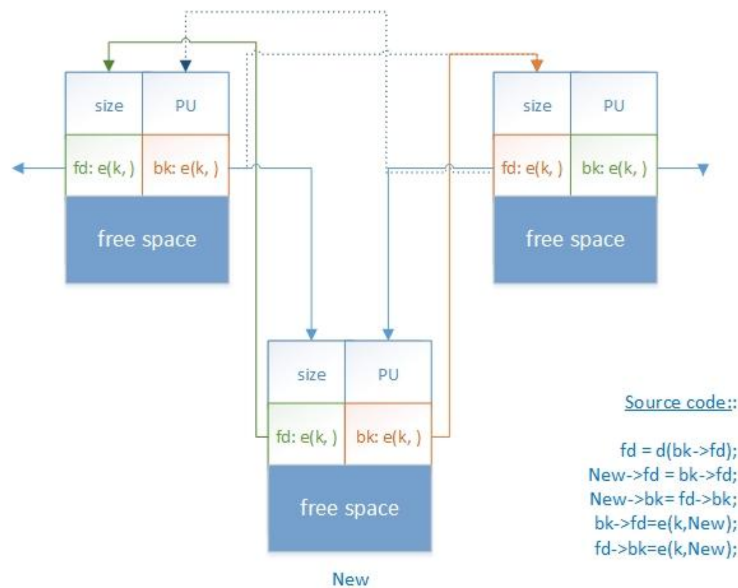

Figure 3-3: *Operation of double linked lists. The encoding function is symbolized as **e**, the decrypting one as **d** and key as **k**. Insertion functionality is displayed. Deletion happens in similar way.*

Obviously Pointer encoding in mono-linked lists schema is a passive security technique. This is because in case of any attack to heap's freelist linking pointers, application is modified in way that is extremely possible to corrupt due to segmentation fault, without "being alerted" about the reason why the corruption occurred.

3.2.2 Pointer encoded in dual-linked list schema

Traditional linked lists have mono-link fields for each node in order to access next or previous one. With a data pointer encoding mono-linked lists being attacked will end up probably with a segmentation fault or in worst case scenario with a truly unpredictable behavior. Thus Kyungtae Kim and Changwoo Pyo upgraded the pointer encoded to **mono**-linked list scheme that they have introduced, to pointer-encoded to **dual**-linked scheme, using instead of the traditional list element structure the updated one figured out in figure 3-4.

Free chunks of dual-linked lists use two link fields that process independently encrypted values coming from the same address. In this way it is implemented active security instead of the passive scheme that mono-linked lists offer. Any illegal modification of the encrypted fields can be perceived, by decrypting and comparing both fields. The verification of the two encrypted pointers takes place whenever a chunk changes state being inserted and removed from the freelist.

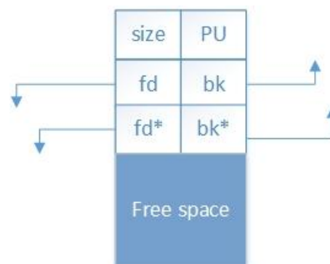

Figure 3-4: *Left: list element structure for dual linked lists.*

Pointer encoded in dual-linked list schema is an active security scheme by the means that the attack is detected barely after its launch and special code routines undertake the responsibility of coping with it.

3.3 OpenBSD Allocator Schema

OpenBSD originally used PHKmalloc. However 2008 it introduced a new allocator based in PHKmalloc but heavily modified aiming to security increment. The techniques that it employs in order to fulfill the security demand is the following three:

- **Data – metadata segregation**: it maintains all of its metadata structures in a memory region completely isolated from the one that stores data
- **Sparse page layout**: the amounts of memory that the allocator reserves consists of pages provided by the mmap, a system call offering randomization because it is based on the ASLR. Consequently the probability of reserving two continuous pages limits to zero as the sophisticated computer systems afford lots of memory meaning plenty of system pages
- **Destroy-on-free**: every time that a memory portion changes its state from "**in use**" to "**freed**" allocate scramble its contents. In this way dangling pointer exploitability is decreased dramatically (see section 2.5.1).

3.4 DieHard: Probabilistic Memory Safety [2]

3.4.1 Overview

Die hard is a runtime system that provides probabilistic memory safety. Actually probabilistic memory safety is a way to create a dynamic memory allocator or to modify an existed one providing a mathematical proof using probabilities that certain error types are not possible to be occurred. So Emery D. Berger and Benjamin G. Zorn in their homonymous paper introduce Die Hard and prove that it offers extremely high security levels for specific errors.

These errors fall into the following categories:

Dangling pointers: If the program mistakenly frees a live object, the allocator may overwrite its contents with a new object or heap metadata.

Buffer overflows: Out-of-bound writes can corrupt the contents of live objects on the heap. Heap metadata overwrites: If heap metadata is stored near heap objects, an out-of-bound write can corrupt it.

Uninitialized reads: Reading values from newly-allocated or unallocated memory leads to undefined behavior.

Invalid frees: Passing illegal addresses to free can corrupt the heap or lead to undefined behavior.

Double frees: Repeated calls to free of objects that have already been freed cause freelist-based allocators to fail.

Of course there are tools that allow programmer to pinpoint the exact location of these memory errors (Valgrind is an excellent example [13]). However they actually reveal only those bugs detected during testing. Consequently deployed programs remain vulnerable to crashes or attacks.

Moreover there are other security policies that can protect application from vulnerabilities caused by the aforementioned error classes but they provide fail-stop services which of course is inadequate in front of the self-treat that DieHard affords.

DieHard provides two modes of operation: a **stand-alone mode** that replaces the default memory manager, and a **replicated mode that runs several replicas simultaneously**. Both rely on novel randomization memory manger that allows computation of the exact probabilities of detecting or avoiding memory errors.

3.4.2 Stand-alone-mode

DieHard allocates objects in different segments according to their size. Each segment's size is multiple of the object's size that it stores. Objects of the same size are placed randomly in the same segment. The resulting spacing between objects makes it likely that **buffer overflow will end up overwriting only empty space**. Randomized allocation also makes it unlikely that a newly-freed object will soon be overwritten by a subsequent allocation, thus **avoiding dangling pointer errors**. It also improves application robustness by segregating all heap metadata from the heap

(avoiding most heap metadata overwrites) and ignoring attempts to free already-freed or invalid objects.

3.4.2.1 Setting Memory Management Concerns

However there are two main **problems** on standalone DieHard allocator. Allocator degrades the **spatial locality** and it use memory in an extremely inefficient way. In order to justify why these problems exist let us consider first of all the strength of the DieHard, the **infinite heap approach**.

3.4.2.2 Probabilistic Memory Safety

There actually is defined an idealized, but unrealizable, runtime system called **infinite heap manager** which provides **infinite-heap semantics**. In such a system, the heap area **is infinitely large**, so there is no risk of heap exhaustion. Objects are allocated infinitely far apart from each other. Consequently buffer overflows, in the way that are known, are benign –never overwrite live data. Moreover the problems of heap corruption and daggling pointers also vanish because frees are ignored and allocated objects are never overwritten. However uninitialized reads to heap remain undefined[11].

3.4.2.3 Infinite heap approximation

While an infinite-heap memory manager is unimplementable, DieHard probabilistically approximates its behavior. By replacing the infinite heap with one that is **M times larger** than the maximum required, it obtains an M-approximation to infinite-heap semantics. By placing objects uniformly at random across the heap, DieHard get a minimum expected separation of E[minimum separation] = $M-1$ objects, making overflows smaller than $M-1$ objects benign. Finally, by randomizing

[11] Unlike Java, C and C++ objects are not necessarily initialized to specific contents.

the choice of freed objects to reclaim, recently freed objects are highly unlikely to be overwritten.

3.4.2.4 Justifying memory management concerns

M times larger heap than the maximum required, is the justification for both concerns that described above. This is a fact indicting the inefficient memory usage. The side effects in restricted resources systems are obvious, as well as the saturation of spatial locality, offered by memory layer, because of the random space between data that should be managed simultaneously.

3.4.3 Replicated Mode

This memory manager approximates most aspects of infinite-heap semantics as M approaches infinity. However, it does not quite capture infinite-heap semantics, because it does not detect uninitialized reads. In order to detect these, another mode is implemented, the replicated mode.

This mode requires the free portions of the infinite heap to be filled with random values. Thus it can then detect uninitialized reads by simultaneously executing at least two replicas with different randomized allocators and comparing their outputs. An uninitialized read will return different results across the replicas, and if this read affects the computation, the outputs of the replicas will differ.

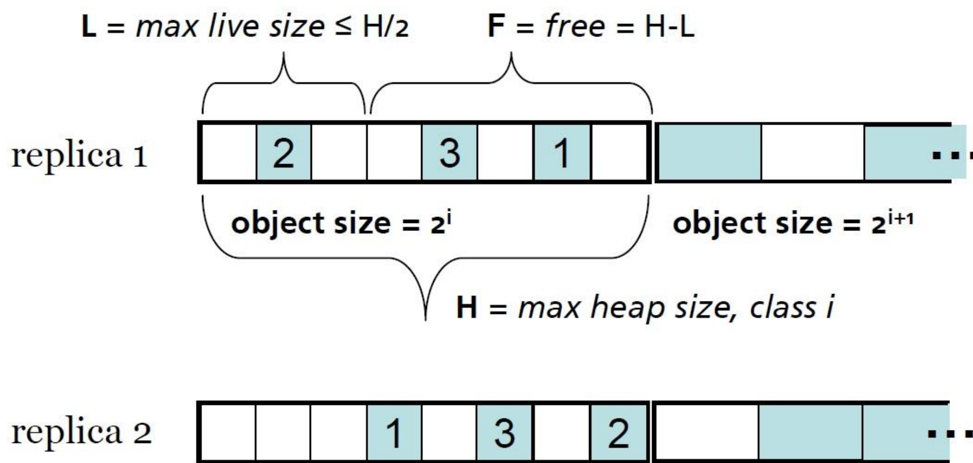

Figure 3-5: *DieHard 's heap layout. The heap is divided into separate regions allocating in random positions in each region, objects of the same size. Notice the different layouts across replicas.*

3.4.3.1 Replicas and Input

DieHard spawns each replica in a separate process using LD_PRELOAD environment variable to redirect all calls to malloc and free in the application to DieHard's memory manager. By this way more than one replicas of the same application runs simultaneously. However each replicas heap is actually initialized under different conditions and as memory manager pick a different random number seed in every invocation, all replicas execute with different sequences of random numbers.
DieHard communicates with replicas using both of standard mechanisms, pipes and shared memory. Pipes are used in order to receive each replica its standard input from DieHard and then shared memory is used vise versa containing one communication buffer per replica.

3.4.3.2 Voting scheme

DieHard manages output from replicas by periodically synchronizing at barriers. Whenever all replicas finish or all shared memory gets full of their result the **voter** asserts the execution progress of the whole application by comparing replicas' outputs stored in the communication buffers. If all buffers agree, no mistakes were occurred.

Consequently the contents of one of the buffers are sent to standard output and execution proceeds normaly.

However, if there is no homophony in buffers, it means that at least one of replicas has at least one error. In this scenario, the **voter** chooses output buffer agreeing by at least two replicas, and send it to standard out. Two replicas suffice because odds are slim that two randomized replicas with memory errors would return the same error.

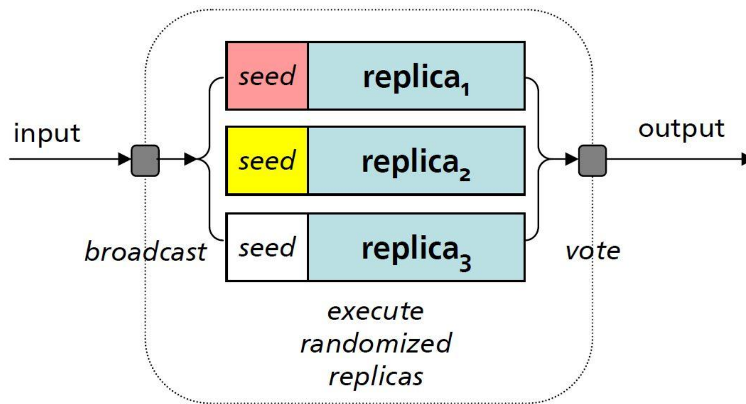

Figure 3-6: *Replicas voting mechanism as it is documented in [11]***[11]**

3.5 Die Harder [11]

3.5.1 Overview

DieHarder is a **bitmap-based** fully randomized lighter allocator based on DieHard, implementing however only **passive protection mode**. Actually it can be classified as BiBOP-style allocator. It allocates memory from increasingly large chunks called **miniheaps**. Each miniheap contains objects of exactly one size having exactly the same sense of page entity, in BiBOP-style allocators. DieHarder allocates new miniheaps to ensure that, for each size, the allocated objects (positions in use) per total objects (positions that can be used for storage) is not more than 1/M (see infinite heap approximation, chapter 3.4.2.3). Moreover each new miniheap is twice as large as the previous one and thus it holds as many objects as the previous does.

Of course the heart of the heap security that DieHarder offers to the application, is the **random allocations**. Every allocation randomly probes a miniheap's bitmap for the given size class for an unset bit (0). In case of **bit hit** (meaning the bit is found) allocator changes its value to set (1) and allocates the respective position for the object. Otherwise it reserves extra space (system call) in the name of the application and assigns it to the respective size class. Afterwards it repeats the search for the newly allocated space where it is obvious that an unset bit hit will immediately take place.

3.5.2 Points analysis

It is fact that by rendering the heap's unpredictable organization and the way that it is used by the application, in order to satisfy its allocation and deallocation needs, the security levels, which are offered by, are increased radically.

For this reason, DieHarder's implementation targets to heap's randomization. This is succeeded by randomizing the placement of allocated objects and the length of the time before freed objects are recycled. This technique is not prototype. It is applied in OpenBSD's allocator. The key difference However is that DieHard randomizes placement and reuse to the largest practical extent instead of limited randomization, that OpenBSD offers.

3.5.2.1 Randomized Placement

When choosing where to allocate a new object, DieHard choice may end up with a position belonging uniformly in anybody of free chunks having the proper size. Furthermore, DieHard's overprovisioning ensures the existence of O(N) free chunks, with N denoting the number of allocated objects. Thus DieHard provides O(logN) bits of entropy for the position of allocated objects, significantly improving on OpenBSD's 4 bits. This entropy decreases the probability that overflow attacks will succeed. The probability depends upon the limitations of the specific application error. For example, small overflows (at most the size of a single chunk) require that the source object be allocated contiguously with the target chunk.

Berger and Novak have justified the security level by introducing and proving the following theorem:

"The probability of a small overflow overwriting a specific vulnerable target under DieHard is O(1/N), where N is the number of allocated heap objects when the later of the source or target chunk was allocated." [11]

To generalize the theorem's point let us calculate the probability of X-chunk overflow attack to be successive, overwriting even one of the V objects contained in a larger set of N allocated ones. Also, consider that there are k objects slots following the source object (the one that the attack starts from) that can actually be overflowed. [11]

Because of the allocator's policy, for N allocated objects, there are at least MN available positions. Consequently the probability for the **first** of the allocated objects, to be stored outside this slots area, is **(MN-k)/MN**, for the **second** one is (**MN-k-1)/MN**, the **third** one is **(MN-k-2)/MN,** resulting to the fact that **i**th object's possibility is **(MN-i-1)/MN.** Starting counting from 0 instead of 1, we got the generalized probability for the **i**th element, **not** belonging to the range attack area, to be **(MN-i)/MN**.

Of course these probabilities are reserved and so their multiplication results to the probability of the case that "**none of the allocated objects are placed in even one of the k slots after source object":**

$$\frac{(\boldsymbol{MN}-\boldsymbol{k})!}{\boldsymbol{MN}(\boldsymbol{MN}-\boldsymbol{k}-|\boldsymbol{V}|-\mathbf{1})!}$$

And of course the opposite case, ”**even one of the allocated objects is located in the attacked area”,** is described by the possibility:

$$\mathbf{1}-\frac{(\boldsymbol{MN}-\boldsymbol{k})!}{\boldsymbol{MN}(\boldsymbol{MN}-\boldsymbol{k}-|\boldsymbol{V}|-\mathbf{1})!}$$

And this is the probability the launched attack to success in a k-vulnerable[12] application. As soon as |V|<< N the probability of successful attack limits to

$$1-\left(\frac{MN-k}{MN}\right)^{|V|}$$

3.5.2.2 Randomized Reuse

Die harder chooses the newly-allocated chunks randomly across all free chucks of the proper size. The probability of returning the most-recently-freed chunk is at most 1/MN. This bound holds even if the after s a single allocation, a large quantity of continuous allocations takes place because of the M overprovisioning factor. So attacks that exploits recently freed space has 1/MN probability to success. [11]

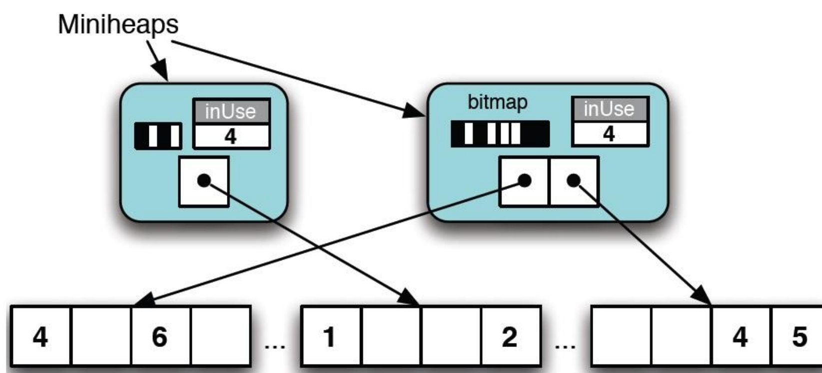

Figure 3-7: *Overview of DieHarder's heap layout as it is documented in [11]*

---

[12] An application is k-vulnerable that allows overflows having size k times or less than the source object

3.5.2.3 Concerns' Justification

The problem of the specific mechanism is the same having the one based on, memory usage efficiency and spatial locality degradation. The overprovisioning factor M points that memory efficiency does not belongs to DieHarder's priorities. And considering that maximization of security probabilities demands maximization of M for constant N it is obvious the memory scalability problems that can be occurred.

Moreover the 1/MN reuse probabilities, can exclude freed chunks from being reused, while many of them should be essential for spatial locality maintenance, as well as system call low ratio maintenance.

4 DEVELOPED METHODOLOGIES

4.1 Dynamic memory managers' perspectives

Before starting with the security policies that were followed to offer security principals to dynamic memory manager, it is essential to clarify the way that dynamic memory mangers were generally treated in this paper.

Dynamic memory mangers are nothing more than finite state machines that were introduced in order to service applications memory requests through managing specific memory portions. The memory resources that feed them with these portions are not a concern for the specific paper.

Consequently there are two ways to treat dynamic memory managers, as **servers**, because they do serve application requests, and as **finite-state monolithic resource managers,** as they do mange internal memory resources and consisting of one module able to be depicted by a sequence of specific states. The first point of view is closer to **web** functionalities, while the second one closer to **kernel** layer. And these actually are the entities that we treat dynamic memory mangers like, defining our security issue's approximation.

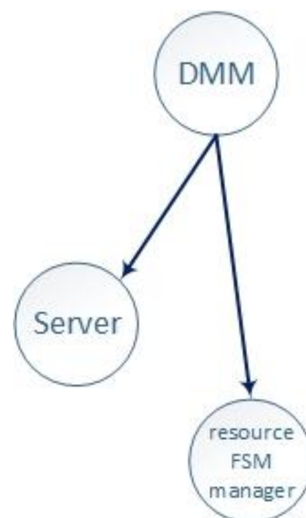

Figure 4-1: *Dynamic memory managers classification according to application's and system point of views*

4.1.1 Dynamic memory manager as server

Every application requests object allocations and deallocations. These requests are "passed" to dynamic memory manger to be served according to its policy schemes. In this manner, despite the fact that dynamic memory manger is a module of the main process (mother process), it can be considered as a server providing its services to the application itself. Memory organization schemes and efficiency are allocator's concerns, leaving application to worry only about the internal management of the allocated objects.

The introduction of the described scheme, offers an extremely different security approximation. Allocators interact with mother application in a request-response scheme with application requesting allocation/deallocation of memory portions of **specific size** and allocator finding magically[13] a way to serve them.

Each application's request to the allocator is considered to be a **datagram** consisting of **two** components, the request itself and a number denoting the size of the requested memory portion in bytes. Thus the **allocator is not informed about the way that application aims to use the requested space but only for its size**. This lack of information restricts security mechanisms that can be implemented around heap protection.

However, by treating allocator as server it is obvious that security can be supported by a **large quantity of special memory requests**, treating memory according to its content and not only as a pure bit container.

Thus, **semantics** are been provided to the heap, based on programming language's type-systems and application's demands, in order to **identify block contents** and to set **clear restrictions** that are based on and proved in formal mathematical way. In this way security moves on, **from heap integrity to application's data integrity**.

---

[13] It actually maintains structures of free memory allocation or it interacts with kernel to request enough memory for serving the application's demands.

Although this scheme provides the maximum of security it does not respect the **transparency's law**. In case that someone will try to secure an allocator or to create a new one from the scratch as described in section 4.1.1, he would end up with a dynamic memory manager's API mismatching the one used by the majority of the existing applications.

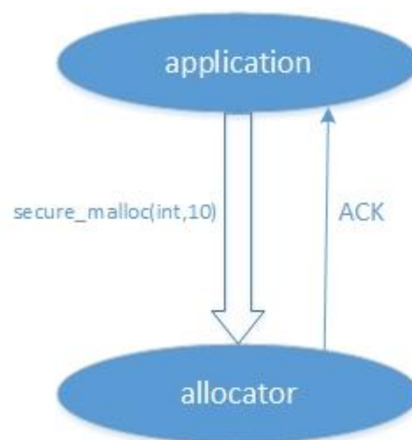

Figure 4-2: *Application allocation request providing semantics to dynamic memory manager.*

4.1.2 Dynamic memory manger as finite-state monolithic resource manager

This is the way that dynamic memory mangers have been treated until now, from anyone occupied with their security issue, as expected because of the need for transparency (see previous subchapter and chapter).

Every dynamic memory manager consisting of only one module, the manager itself, and can described by a set of finite states complementary to the process's states. Each time the allocator's routines are executed the main process's routines, related with

allocation and deallocation scheme[14], passes indirectly to “wait mode”. This consideration reminds the way that two process do corporate with, managed by the kernel’s scheduler. However this is not actually a corporation scheme as the executed allocating/deallocating routines belong to the process itself.

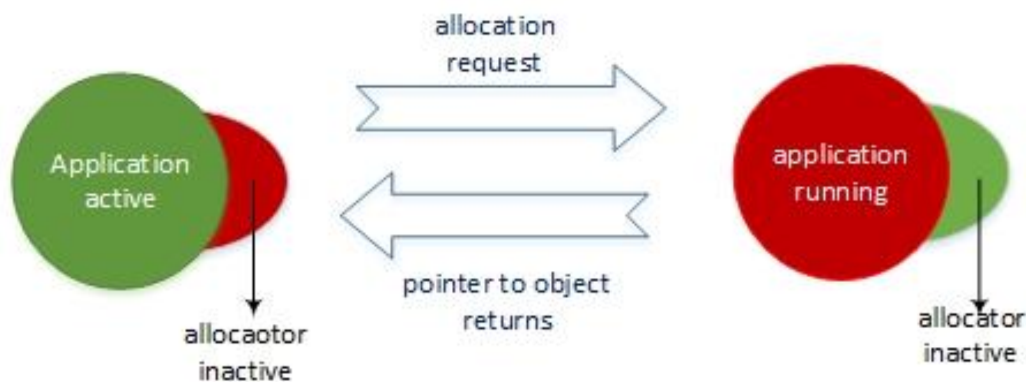

Figure 4-3: *Application request allocation from the dynamic memory manager, control passes to DMM and again in application.*

The above assumption helps us to understand the manners and the critical location that the security mechanisms can protect. Heap’s content investigation results to two classes, **data** and **metadata**. Source code investigation leads to two classes too, **application instructions** and **dynamic memory manger instructions**. As it has been clarified in subchapter 4.1.1, allocator is actually informed only about the size of requested objects. Thus we understand that any try to secure allocator should point to **interaction between its instructions execution and heap’s metadata**.

[14] There are multithread process so the assumption refers only to the thread requesting allocation or deallocation

4.2 Functionalities VS Dynamic Memory Allocator's Security

The subject used for our research is an open source library, **dmmlib** [12]. It is free-list based allocator offering to any process the same functionality with DL, UNIX's default allocator.

Moreover it maintains some extra internal functionality features aiming to memory usage efficiency and low system calls ratio. For example it supports evolutionary **splitting** and **coalescing** mechanisms that resize portion of memory being freed by application in order to serve each allocation call in a more memory efficient way. Moreover the specific allocator supports a numerous policies defining the way that object allocations are served, policies like **first fit**, **best-fit** etc.

Finally through heap's internal organization permitting alignment policies and raw blocks with various sizes, the target of hardware platform transparency is fulfilled (as long as it is possible) as well as distributed memory transactions supported based on NOC[15] hardware and software implementations.

4.3 Heap Organization Overview

The allocator that we based on for our research is described by a **layered architectural** scheme figured out in Figure 4-5.The allocator classifies memory portions, according to their significance to **blocks** and **raw blocks**.

Starting with **blocks**, they are the known, as well as, memory chunks, meaning memory portions which contain a header to specify their location in the whole heap's organization and being oriented for maintaining application dynamic allocated objects. Moreover blocks are the structural components of raw blocks.

---

[15] NOC: Network On Chip

Raw blocks are large portion of memory that allocator requests from kernel through some system call. The size of these large portions is fixed and defined during allocator's source code compilation meaning that depending on the specified system there should be analogous allocator's binary. Raw blocks consist of continuous blocks.

So the dynamic memory manager implements the above structures to wrap application's data. Moreover it uses single linked lists to manage and organize these structures. Actually we have two basic lists, one ordering the raw blocks and one maintaining the free blocks. The linked lists mechanism is implemented in macros in order to achieve best able performance inheriting linux queue.h source code file.

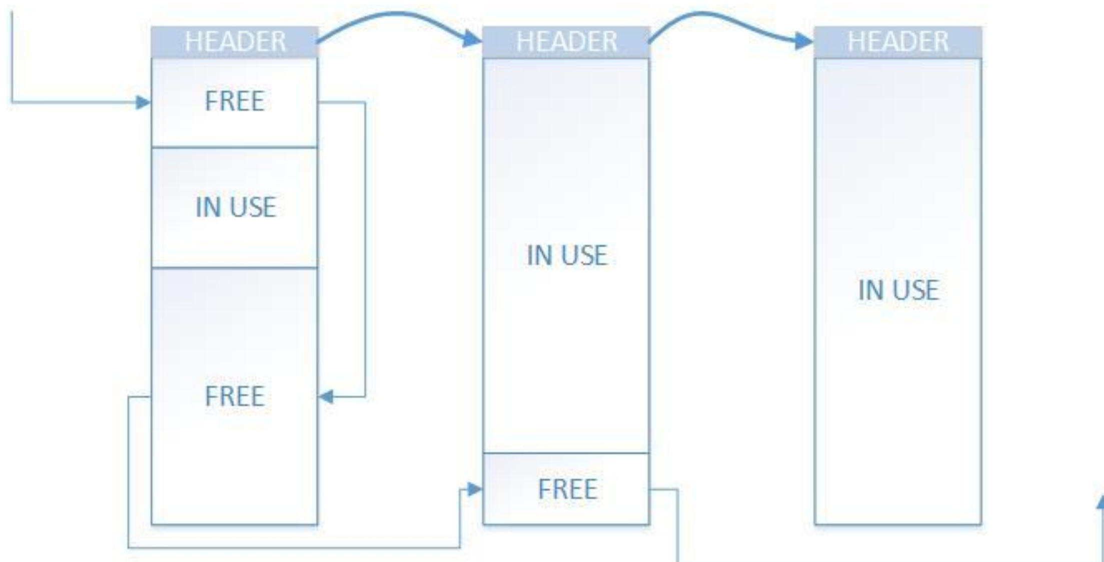

Figure 4-4: *Snapshots of tree raw blocks organized in a linked list while the free blocks that they contain are organized in another linked list too.*

4.4 A real attack scenario

Consider that there are two raw blocks having size less than one page[16], in the address space of an application's heap. The first one consists of five blocks and the second one of four blocks. In case of unprotected string pointer, pointing to the internal of one of these nine blocks, there are only two blocks that forbid block overlap attack to be successful. These are the last one block of each raw block. Any trial for forward consolidation (see 2.5.2) buffer overflow attack to the specific blocks will probably end to segmentation fault. So the probability the launched attack not to be succeeded is two out of nine cases. And because of the probabilities regularity, **7/9 is the probabilities of a successful attack** indicating to the need of security implementations mechanisms.

4.5 Protection Security Schemas

In chapter 4.3 we have discussed the two layered memory organization of heap, **blocks** and **raw blocks** organization schemas. Consequently all the protection mechanisms that have been implemented within this paper's framework, aim to protect this specific layer's organization integrity in order not to permit indirect overflows through heap, introducing canaries techniques and encrypted pointers for the lower level (blocks) as well as encrypted lists and encrypted lists validations mechanisms for the higher one (raw blocks).

All the security mechanisms are active (see chapter 3) meaning that they do find out any illegal modification, but they act like passive ones, as in case of illegal modification only a segmentation fault is triggered. However because of the fact that our implementation uses the adaptor morpheme at this point, it is obvious that any other more complicated activity can replace the passive action of segmentation fault triggering.

---

[16] Usually raw blocks' size is equal to page size. This because, in this case, allocator scales, as it collaborates with OS efficiently. There are although cases in which raw block's size is multiple of the page size. In this case the probabilities of our example are different

4.5.1 Canaries

This is the first of the security mechanism introduced to the subject allocator. It aims to protect blocks metadata from any illegal modifications mainly happening through buffer overflows and daggling pointers vulnerabilities. However, despite the fact that it succeeds to protect heap's metadata significantly as it was described in previous chapter (see 3.1), the specific mechanism is not able to track any **double free** attack or any allocator's effort to **read uninitialized data**. This is the reason why, in our paper, we describe an evolution of the specific security mechanism. "**Semantics**" are being introduced in canary guard's value for first time, meaning that **chunk's state defines canary guard's value**. This actually takes place by implementing **different encryption schemas for "in use" and for free blocks**.

The block header structure, that our allocator uses to organize blocks, contains a range of fields having different significance depending on blocks state (in use/free). This leads us to include in our encryption schema, different metadata fields, in order to produce each block canary guard, according to its state. Figure 4-5 presents the block header's fields used in encryption schema, depending on the block's status.

The produced mechanism is able, thus, to detect all the attacks that its stateless ancestor detects. Moreover it is moving on in detection of double frees and uninitialized data reads. For example a double free trial will lead to canary validation failure. This is not because of wrong decryption random-seed/key (see 4.7) but because of wrong header field invocation.

Figure 4-5: *Block header fields used by canary mechanism in order to distinguish blocks' states. Red denotes the one used for free blocks while green the one used for in use blocks.*

4.5.2 Encrypted Indirect Pointers

The basic schema

The second mechanism that was implemented and tested is encrypted pointers. Actually, the dynamic memory manger that we focus on, supports two types of linked lists. The first one is the one explained before which connects directly, via pointers, entities of heap organization such as blocks and raw blocks. The second one, on the other hand, connects indirectly all the blocks belonging on the same raw block regardless their status. This is supported by an extremely low level arithmetic software component that use the pervious size field of block header and the data segment's start of the specific block in order to locate previous one. The equation bellow describes the mechanism connecting chunk A and B included in the same raw block:

$$startingAddress_A = startingAddres_B - previousSize_B - sizeof(B)$$

The mechanism that we introduce for this reason is based on encrypting some of the essential, for the described above equation, operators resulting to e**ncrypted indirect pointing mechanism**. Because of the fact that the arithmetic function used for

indirect pointing is the addition, which is linear and distributive, both addends (a, b) are of the same size as well as the chosen encrypting function is XOR (see 4.7) we got:

$$encrypt(a) + \ b \ \Leftrightarrow encrypt(a + b)$$

The above equation denotes that formally our encryption schema produces a fully encrypted indirect heap metadata pointer, by encrypting one of the two addends that the pointer consists of[17].

The chosen policy of our implementation is to elect the **previous size** block's header field because it is exposed to buffer overflow attacks and its value is manipulated directly by dynamic memory manager, fact false for the starting address of any block as it depends on system. However our choice has **a small flaw**. The allocated chunk's sizes create a **canonical distribution**. Consequently in case of using a static key set, having size smaller than the encryptions amount, the resulted environment will not be secure enough. A statistical observation would be enough, to squeal the described security mechanism. The last side effect of static keys choice can be bypassed by using fully random keys.

Consequently the indirect encryption mechanism creates a randomization cloud around the raw blocks internal organization, in a way that the probabilities do not help the attacker to succeed its will. However this does not necessitate the definition of the applications behavior, leaving to the luck the results of the launched attack.

The replicated mode

Despite the fact that the described security mechanism is enough to offer protection against buffer overflow attacks, both forward and backward consolidation, through unpredictability, it is not enough to compete the validation functionality that canaries implementation offers and the pros of any other active security mechanism. That is why we have created the replicated mode of the described mechanism.

---

[17] Addition and subtraction mathematically is the same operation. Moreover do not forget that b can considered the sum: $startingAddres_B - \ sizeof(B)$

In this scheme it is kept an extra replica of each one already encrypted field. And as it is expected, the value stored in each extra field, comes from the original one, encrypted in the same way, but using a different random seed (key). Of course every key is again produced in the same way that the respective one used in the first replica does.

Every time that an internal operation takes place allocator decrypts both original and replicated field and compares their values. Their mismatching denotes an attack launch thus the dynamic memory manger behaves analogous (in our case it raises a segmentation fault). By this way, we have passed the encrypted indirect pointer mechanism from the class of **passive** to the one of **active** security schemes.

4.5.3 Encrypted lists

As it is mentioned before dmmlib organizes heap's entities (blocks/raw blocks) in large single linked lists. Consequently there is an extensive actual pointer use in order to connect heap's metadata. In chapter 2 it is clarified that maintaining pointer to metadata and letting the allocator managing them freely can be proven pernicious for the system's integrity.

Passive schema

The first mechanism implemented aiming to offer passive protection transparently to any list that participates in heap's organization. This is supported by the encryption of any existing list pointer according to the encryption mechanism that chapter 4.7 presents. In this way the attacker without being aware of the key that is used for the encryption-decryption mechanism cannot track the next element of the list. Consequently the whole heap's organization cannot be revealed. Moreover even if the attacker finds out the essential for its attack clues, an attack based on list linking pointer overflow would lead to application to behave in a way that even the attacker cannot predict let alone take advantage of it.

This hostile environment combined with short attack windows can be proved valuable against a large set of security flaws.

Active schema

However this security mechanism is not able to detect any security violation in order to take action and defend at once the application as well as the whole system. That is the reason why we have implemented its descendant, the encrypted list validation scheme.

In encrypted list validation mechanism every list link[18] has been replicated. Both the original and the replicated links are encrypted. Every time that a list operation, such as insertion, deletion, reordering etc., is going to take place the validation mechanism is triggered. Thus both values are decrypted and compared and in case of mismatch the allocator alerts for illegal modification and terminates the process through segmentation fault. Of course, as it is mentioned in previous mechanisms, the segmentation fault part can be replaced by a more innovative behavior.

Efficiency Assertion

It is obvious that generally encrypting all the list links that connect heap's entities is not something efficient. The main problem is the list exploration and not any other list operation. This is because, despite the fact that a deletion/insertion requests only one decryption (three in case of active schema), every list element access involves a pointer decryption. Thus probably the security schemas are not appropriate for applications that maintain high heap reorganization intensity. However it is not actually prohibitive as there are plenty of applications maintaining low heap reorganization intensity.

4.6 Basic components

The described security mechanisms are actually divided into four main distinguished modules, **the core, the encryption, the key production and the key maintenance**

[18]Pointer that connects two list elements

module. The last three modules are named side-modules, because they support the main module and they are not involved with each described security scheme (they behave the same way regardless the implemented security module they corporate with). These modules, to corporate with, recommend a layered architecture schema of peer to peer interaction points that we have implemented in a modular way offering the opportunity to replace any one of them with some other implementation.

Starting with the core application it consists of all the modifications and code routines that have been created in order to secure the dynamic memory manager. It has the most complicated connection with the dynamic memory manager in unrepeatable way.

The encryption module is the one responsible for any encrypting or decrypting operation that happens during application's memory requests. The basic algorithm the specific sub-mechanism is based on is described in detail in section 4.7.

The random seed production unit is responsible for key production. In the specific paper we use the already existed random functionality, that c libraries offer, which produces pseudorandom numbers. However it can be replaced easily from any other more sophisticated randomization mechanism. Moreover because of one time encryption politics the randomization c utility is wrapped with an extra feature, the key uniqueness. Every time a new random seed is generated **a uniqueness request is sent to the key holder module**. Then if the new random seed is unique, it is added to the structures that key holder maintains. On the other hand, if it already exists, we got a regeneration of it. This is ordered by one time padding philosophy (see 4.7).

Last but not least the biggest concern, talking about side-modules, is the key storing and maintenance module. This is because keys have to remain secreted and so they have to be protected by any third. However allocator needs to find them in low cost in order to be effective and competitive. Thus the specific side-module is extremely important for our research so we have dedicated to it both chapters 4.8, 4.9.

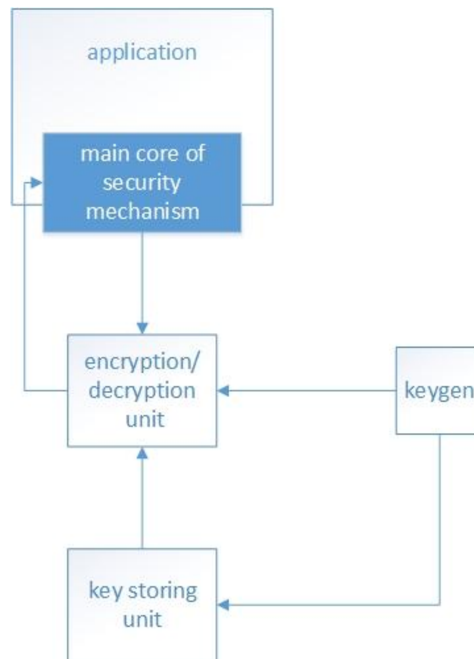

Figure 4-6: *The four modules of the security mechanism, corporation schema. Arrows indicate the keys or the encrypted/decrypted values exchange directions.*

4.7 Encryption schema

As it is declared every one of the above security mechanisms, is based on a specific cryptographic scheme. Because of the fact that, in cryptography, the **one-time-pad** has been proven to be impossible to crack **in case of correct use**, one-time-pad is the one chosen algorithm to support the aforementioned security mechanisms.

In one-time-pad each bit or character from the plaintext is encrypted by a modular addition with a bit or character from a secret random key (or pad) of the same length as the plaintext, resulting in a ciphertext. If the key is **truly random**, **as large** as or greater than the plaintext, **never reused** in whole or part, and **kept secret**, the ciphertext will be impossible to be decrypted or break without knowing the key.

It is obvious that despite the fact that one-time-pad is impossible to crack, there are some serious difficulties to its application. Starting with the truly random number demand, there is not commodity hardware able to be found on any pc system which can produce truly random numbers. Thus the random numbers (seeds) in our

implementation are produced by using c-embedded random generator. It is actually a generator of pseudo-random numbers based on system clock.

It is chosen not to spend a lot of time in finding a truly random number generator because it should be something out of this paper's scope. Moreover consider that the heap is a large dynamic structure, changing shape continuously and rapidly, resulting to windows of time not large enough for the attacker to crack the ciphertext. Although, academically, a ciphertext break is possible. For this reason our implementation uses the adaptor morpheme[19] connecting modularly the randomization mechanism to the main project. In this way its replacement from **one producing truly random numbers,** is a piece of cake.

The second problem that any heap security mechanism faces is to keep keys (random seeds) secret and protected. This is actually a great challenge due to the fact that random seeds and encrypted values are located in the same address space (heap segment). Most described security mechanisms use hashing mechanism to assign keys and encrypted fields of heap. However hashing is generally a predictable procedure so any encryption protection is cancelled because of possible key access. Even if memory space is out of range of heap segment which application can access a dangling pointer could easily lead to the keys area.

Because of the aforementioned concern, we created another storing schema. The random produced keys are stored as nodes of an **avl-tree** placed in a memory space located and maintained by a special **mini-Allocator** module. The search key of the avl-tree is even a block's address or the encrypted value itself (see 4.5). By this way two targets are being fulfilled, storing keys in an inaccessible by the process area and ordering them randomly.

[19] Morpheme used to link a specific routine with main project in one or more locations giving the opportunity of changing the routines implementation even definition without modifying any code line from the main project.

A small modification

Our implementation changes a little the algorithm depending on the allocator's level that we secure. For the lowest one (canaries, encrypted pointers) the encrypting function that is used is:

$$encrypt(a, key) = a\ XOR\ key$$ [20]

While in the higher one (encrypted lists, encrypted lists validation) it is used as encrypting function the simple value key permutation.

$$encrypt(a, key) = a \Leftrightarrow key$$

This results from the dependencies that exist in the avl-tree storing mechanism which needs a unique search key to connect encrypted value and stored key. While in lowest level the unique search key is the block address that encrypted field is contained to, in higher level there is no clear similar address because of the complicated organization. For example there should be **a raw block** belonging in more than one linked lists leading to conflicts.

4.8 Keys/Random Seeds organization algorithms

As it is clarified in previous subchapters (4.5.1, 4.5.2, 4.5.3, 4.7) all the security mechanisms that we have implemented and described in this paper are based on keys entities. Any encryption decryption scheme is based on **key existence and availability**. Also, speed and low cost metadata modifications are evry dynamic memory manager's demands.

However it is obvious that encryption decryption schemas, cross checks, and replicas do cost in resources as well as in time too. The resources are not a problem nowadays. However the time cost of any protection mechanism is the main concern. To understand the causes of any delay because of security mechanism addition to the

[20] Of course in case of canaries **a** is produced reversely by executing XOR between an accumulator and each chosen field and finally with the random seed.

dmmlib we have to consider the bottleneck of each operation. And this is not the encryption decryption operation themselves but necessary key retrieval.

To investigate the above statement we have implemented tree different key production and maintenance schemas, the static keys, the single list linked keys and the AVL-stored keys.

Static keys are values produced randomly run-time during initialization stage of the dynamic memory management library. Static keys are a set of random values each one specified to be used by some of the described securing mechanisms. For example literally, every canary guard should use only one of these pre-produced values as seed, or any field and its replica of any block should be encrypted with the same pair of pre-produced values that was used by the same fields of any other heap's block.

It is obvious that the above implementation is not effective enough to protect large scale heaps. However it is not useless. In case of application having small heaps which does not last enough to offer large attack windows and demand extremely fast memory transactions, this implementation should be suitable due to its negligible memory transaction dependency.

The second organization scheme that was implemented is the **stored in link lists keys**. According to this mechanism every hot field[21] is assigned to its private key through its owner block starting address or through its own value depending on the core module's algorithm. Canaries scheme assigns the random seed used for canary creation by implementing an object-element containing block's header starting address, as primary search key, as well as the seed itself. A similar keys maintaining mechanism is used for indirect encrypted pointers. Again an object-element is created containing as primary search key block's header address and both random seeds, the one used for the original field and the one for the replica, respectively.

On the other hand in order to implement a universal security mechanism encrypting transparently all the linked lists the dmmlib creates and maintains during programs execution, we had to solve the problem of lack of distinguishing reference points to

[21] Meaning fields whose values result from the encryption component.

use them as primary search keys. For example a raw block can belong to more than one linked lists. That is why in case of encrypted lists and encrypted lists validating security mechanisms create objects contacting the encrypted value as primary key and the original one too. Furthermore remember that each encrypted value is nothing more than the respective random seed (see 4.7). Of course the restriction of key uniqueness will prevents any primary search key conflict in the whole structure.

Single linked lists do not scale properly, as it is expected, because of their linearity; consequently we introduced another organization for stored key schema, AVL-trees. It is considered a better solution because of its O(logn) search, insert and delete complexity instead of Θ(n) that single-linked lists have for all of these operations[22].

Concluding, bibliography indicated that the most efficient way to store the keys is by using hash functions. We agree with the specific claim. However we consider that hash functions are a predictable technique. Consequently in case of large attack windows, the attacker after identifying the algorithm should bypass the security mechanism by accessing the original keys location. On the other had our string organization implementation inherit the randomization feature from independent "authorities". In case of lower level security mechanisms the legacy comes from the ASLR and internal entropy while in case of list security mechanisms the randomization is inherited by the randomization independent mechanism itself. In any case there is no obvious way that the keys layout.

4.9 mini-Allocator, mini-Heap implementation

Of course all the described algorithms cannot be implemented in the same heap's segment that the application uses for its needs and the dynamic memory mangers manages. In this case we would have a logical recursion leading us to an infinite loop. Moreover it is impossible to implement efficiently the described in section 4.8 structures in huge memory portions without a special mechanism's support.

[22] Of course the specific complexities are corresponding to the operation for a random element. In this way security schemas treat the key storing organization, by requesting random elements

The gap between the keys storing organization is completed by an extra layer module, mini-Allocator which manages a mini-Heap. In reality mini-Allocator is a small and efficient memory manager. Mini-Heap is the memory organization that it orders to a protected memory portion that it request directly from the OS. Consequently every dynamic memory request related with one of the algorithms described in section 4.8 is serviced by the mini-Allocator's routines which allocate the requested space in the mini-Heap segment which actually is a protected segment of the known heap, accessible only by specific routines.

Mini-Allocator's organization indicates that it belongs to BiBOP dynamic memory managers family. It consists of big portions of memory dynamically increasing and decreasing, bags of pages. Each bag can store only objects of a specific size, which is defined during the bags initialization. Objects of various size will be stored in various pages. This property makes it ideal for storing elements of the same type. Considering linked lists and AVL-trees, they actually consist of objects having the same size, list elements and AVL-nodes respectively.

Each bags exploration is based on bitmap philosophy. The bag consists of pages. Each is divided in equal portions, cells, where the newly created objects are plugged in. Moreover every bag contains a header in which it store essential for bags organization metadata such as pointer to the next page which belongs to the same bag too, a counter displaying the number of the allocated cells, an LRU reference, and the essential for the platform padding.

Continuously after its header, every bag maintains a bitmap indicating exactly which cells are in use (respective bit is set) and which are not (respective bit is unset).

Finally on the top of the whole construction, a single linked list exists. Every node belonging to this list is actually each bag's header maintaining critical for the bag metadata such as number of pages of specific bag, a pointer to the first page of the bag as well as a pointer to the next larger bag. By ordering the bags we succeed to index the list and to refer to each bag in constant time.

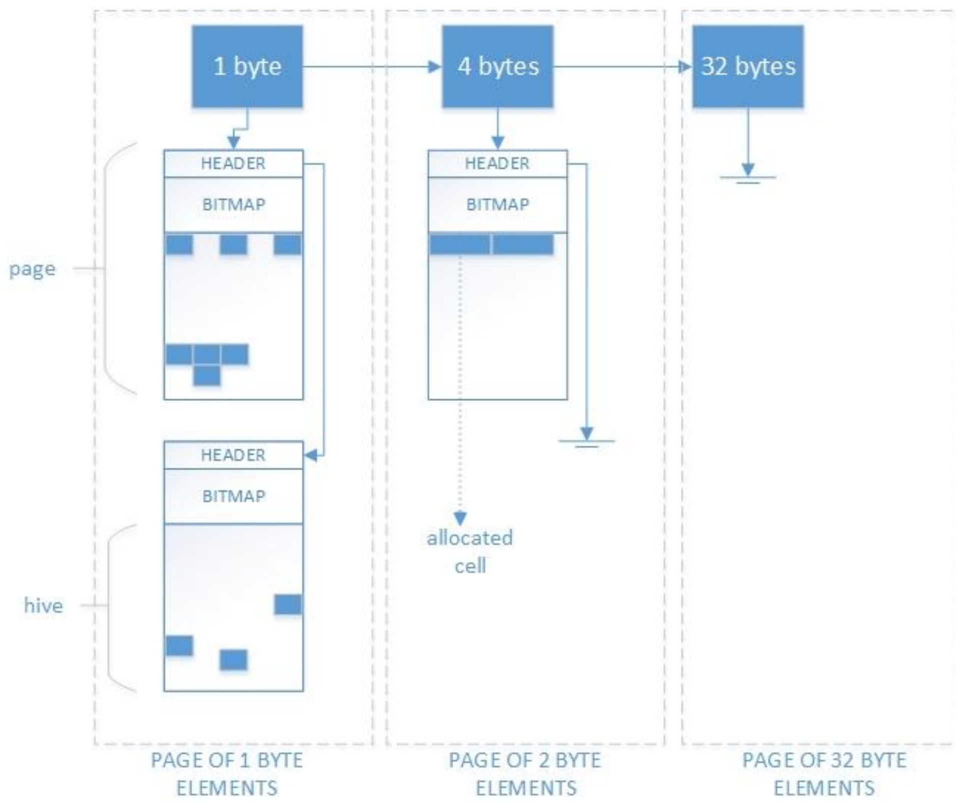

Figure 4-7: *mini-Heap's internal organization*

5 METRICS – EVALUATION

5.1 Overview to testing environment and scenario

The object of the specific paper is not only the implementation of security mechanisms, but their **evaluation through hard testing too**. For testing we have chosen to use spec2006 and especially cpu2006 benchmark suite [9]. This is a widely recognized hardware and component architecture evaluation suite oriented mainly to CPU assertions. Consequently it is a sufficient indicator for our implementation's performance.

cpu2006 suite's benchmarks are classified to two class, floating point and integer benchmarks respectively, according to their intention of floating point operations. Both sub-suites have equal significance to our assertion.

Moreover it must be mentioned that they have been used three programming languages for benchmarks' implementation, **c, c++, fortran**. Of course despite previous benchmark classification was not significant for our research this one must be taken under serious consideration, before evaluating benchmarking results. For this reason, until the end of this paper, every time that a benchmark's name is displayed, it will be followed by **c, cpp or f indicating the programming language that has been used for its development**. This is because only c applications mange memory by sending explicit requests to dynamic memory manager. On the other hand c++ uses a garbage collector which increases its memory managing complexity.

Concluding, the benchmarking process's result evaluation is based on comparison between competitive implementations. Two competitive implementations have been benchmarked, the UNIX default DL allocator, the Die Harder [11] implementation. Moreover it was judged essential to measure the performance of dmmlib without any security mechanism enabled.

5.2 Benchmarking DL, DieHarder and clear dmmlib

This chapter focuses on the presentation of the benchmarking results of the three competitive dynamic memory manager implementations. In the following tables it is presented the **time** in second that was **consumed** by each benchmark when using each of the three **allocators**.

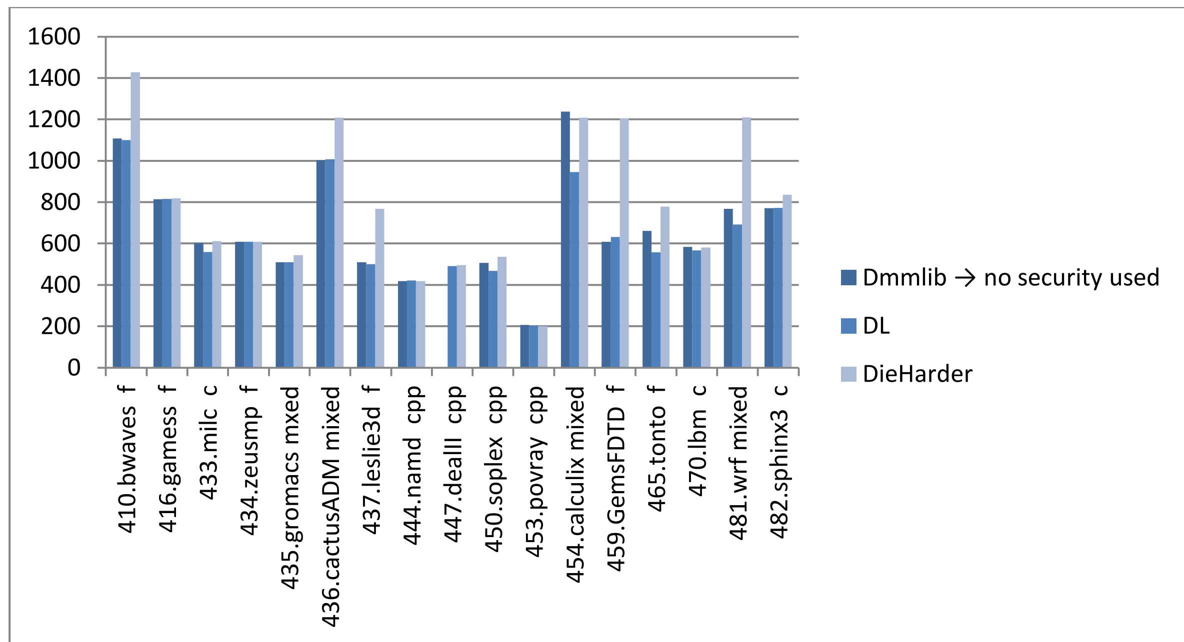

Chart 5-1: ***Floating point*** *benchmarking results for* ***DieHarder, DLallocator and dmmlib*** *without any security mechanism enabled.*

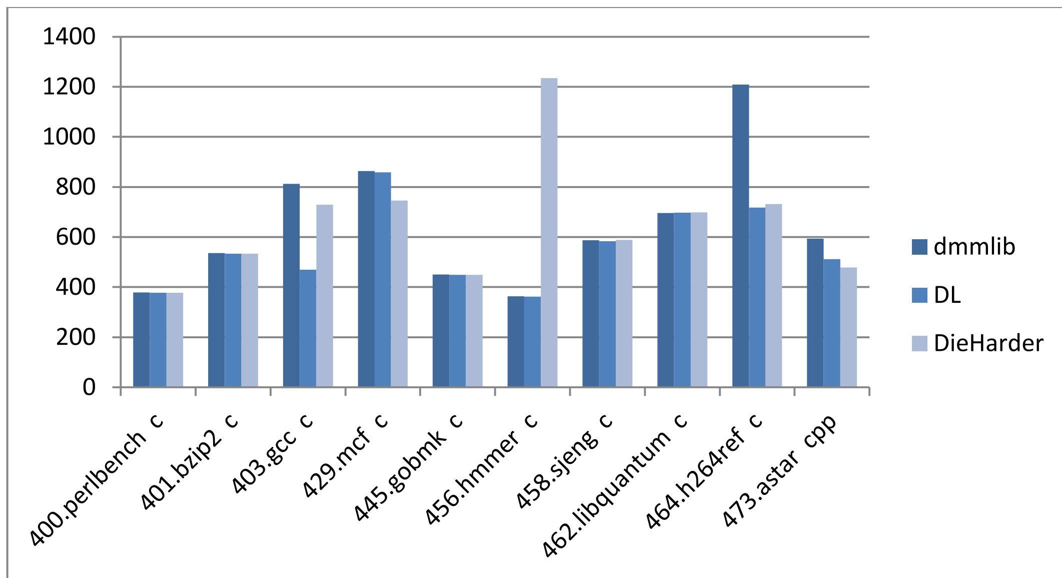

Chart 5-2: ***Integer*** *benchmarking results for* ***DieHarder, DLallocator and dmmlib*** *without any security mechanism enabled.*

The conclusions that the above charts leads us to, is the fact that dmmlib is an extremely competitive library, considering the fact that despite it offers a large number of extra functionalities it actually is as efficient as the other two benchmarked dynamic memory managers, in the majority of benchmarks.

As it is mentioned before, back in chapter 4 and especially in section 4.8, there were implemented three **keys production-organization schemas**, static keys, AVL-stored keys, single linked stored keys. Thus it is important to measure the performance that every mechanism presents when combined with canaries, encrypted indirect pointers, encrypted lists, and encrypted lists validation securing policy respectively.

The benchmarking results have no "commercial" value if they are not compared with the existing competition. Every chart contains the results of dmmlib, without any security mechanism enabled (**clean dmmlib**), that have been compared with the competitive DieHarder and DL allocators (charts 5-1, 5-2). This is the link among all the charts that are presented in this paper.

Moreover charts 5-1 and 5-2 offer us a clear perspective of benchmarks behavior. Meaning that benchmarks that resulted to the similar measurements for all three dynamic memory managers have low memory intensity and those that resulted in

measurements indicating large divergence it is obvious that have high memory intensity and dynamic memory manager performance dependency.

5.3 Benchmarking dmmlib with canaries securing mechanism enabled

Considering the evaluation results taken place in section 5.1 in the specific section we focused on evaluating the results of canaries security mechanism benchmarking.

In chart 5-3 they are presented the floating points benchmarking results. They indicate that in the majority of the benchmarks our security mechanism implementation did not add any performance overhead. All three versions had performance close to clear dmmlib. Especially static keys implementation has same behavior with clear dmmlib, AVL-tree implementation gives close to dmmlib benchmarking results, while single linked lists benchmarking results indicates some performance issues to some of the benchmarks. However this is an expected status as single linked list version is by default an extremely inefficient implementation which can be saturated easily by a large number of accesses to keys requests.

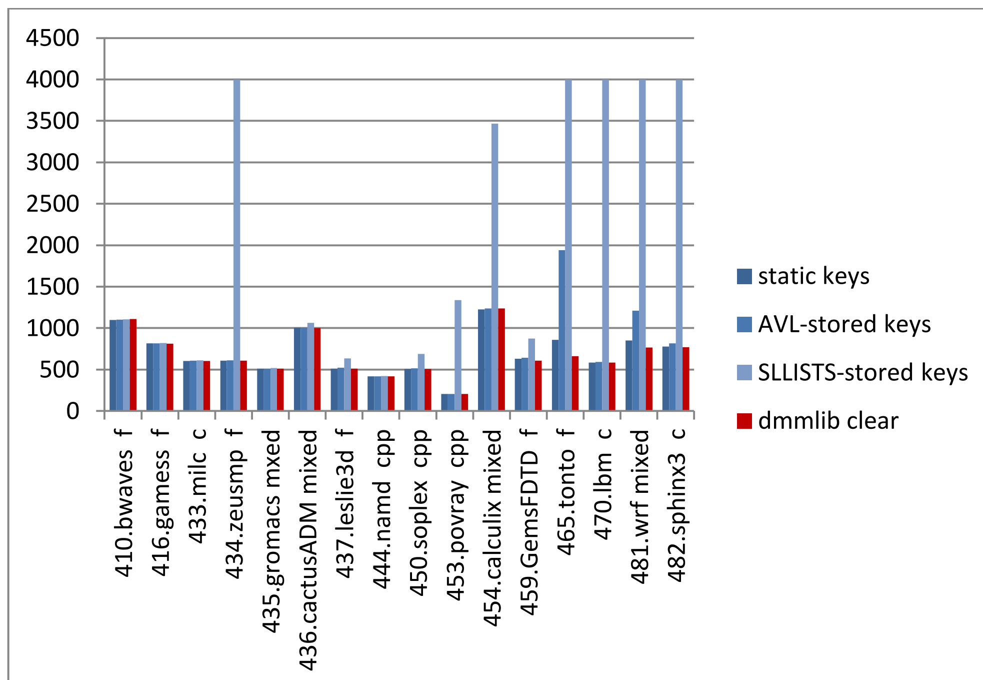

Chart 5-3: ***Floating point*** *benchmarking results of dmmlib with* ***canaries*** *securing mechanism enabled for all of three key producing organizing schemes.*

The same "clinical" view is indicated in the following chart, which actually presents integer benchmarking results. In this case single linked lists implementation indicates a better measurement set closer to dmmlib. On the other hand both single lists implementation and AVL-tree implementation add serious performance overhead according gcc benchmark's results. Of course this is an expected behavior as gcc demands a large amount of allocations and deallocations, so the continuous canary guards validations lead to keys invocations lead naturally to application saturation.

This indicates that gcc is not an appropriate benchmark as the security mechanisms are dedicated to protect applications that have low application intensity and highly user interactivity.

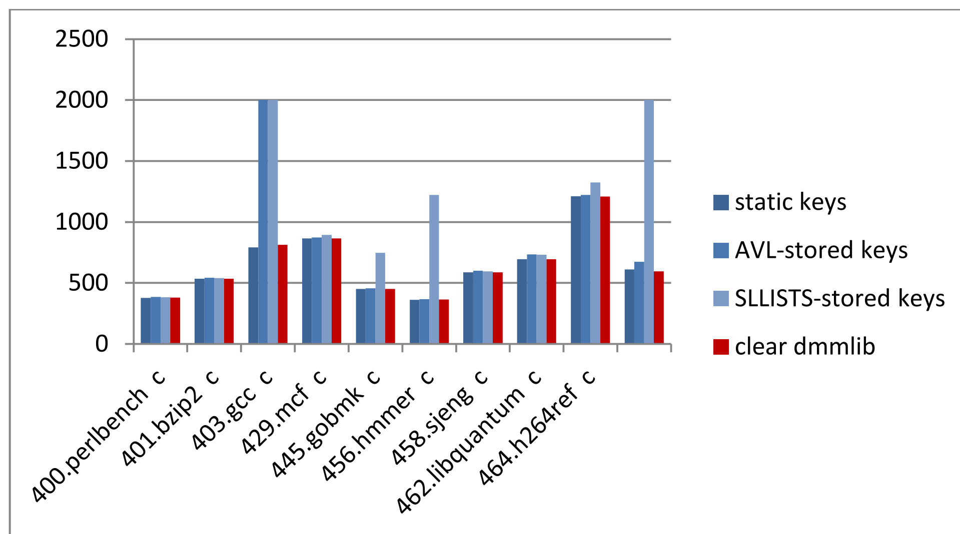

Chart 5-4: ***Integer benchmarking*** *results of dmmlib with* ***canaries*** *securing mechanism enabled for all of three key producing organizing schemes.*

5.4 Benchmarking dmmlib with encrypted indirect pointers securing mechanism enabled

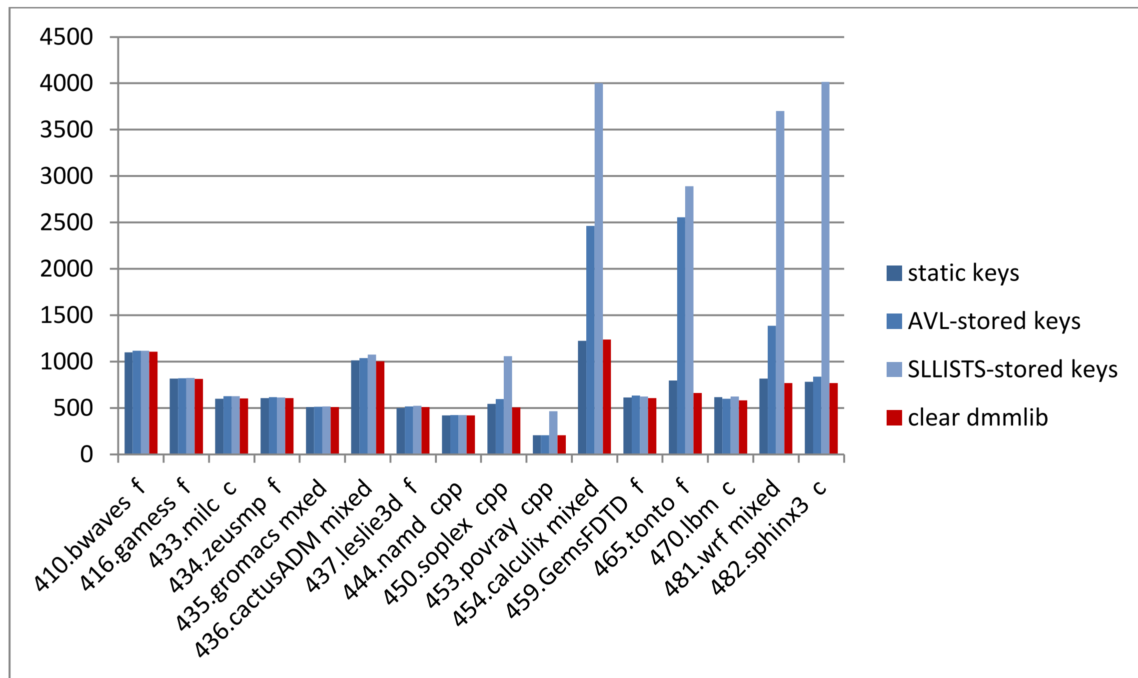

Chart 5-5: ***Floating point*** *benchmarking results of dmmlib with* ***encrypted indirect pointers*** *securing mechanism enabled for all of three key producing organizing schemes.*

Mining the conclusions of the above chart it is noticeable again that flat pointing benchmarking results indicate that there is no prohibitive overhead in our implementation. Of course serious overhead is noted in **465.tonto** f and **454.calculix** mixed benchmarks for AVL-tree implementation. This results us to the conclusion that maybe there is some collaborating problems between the way that fortran and AVL-trees implementation as both tonto and calculix contains fortarn code segments also.

As it was expected, static keys implementation adds zero overhead, while single linked lists implementation suffer from some greater overhead as two of the benchmarks using this implementation passed the predefined time limits. (see 5.8).

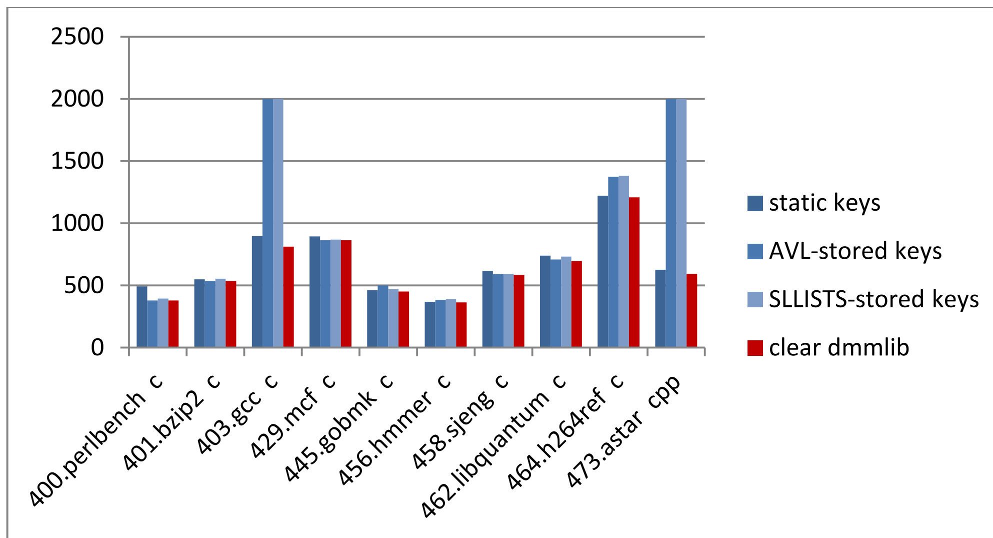

Chart 5-6: ***Integer benchmarking*** *results of dmmlib with* ***encrypted indirect pointers*** *securing mechanism enabled for all of three key producing organizing schemes.*

On the other hand, integer benchmarking indicates that the overhead added because of encrypted indirect pointers is the close to the one added by the canaries mechanism. This is because mini-Heap transactions ratio is the same as both mechanisms validate memory block fields at the same rate. Allocator validates canary guard or encrypted indirect pointer and its replica, before every block's status change, fact that justifies the similarity of charts 5-6 and 5-3. The differences between the two charts are because of double encryption-decryption operations taking place in encrypted indirect pointer mechanisms, as well as due to decryptions that raw block interior exploration demands.

5.5 Benchmarking dmmlib with encrypted lists securing mechanism enabled

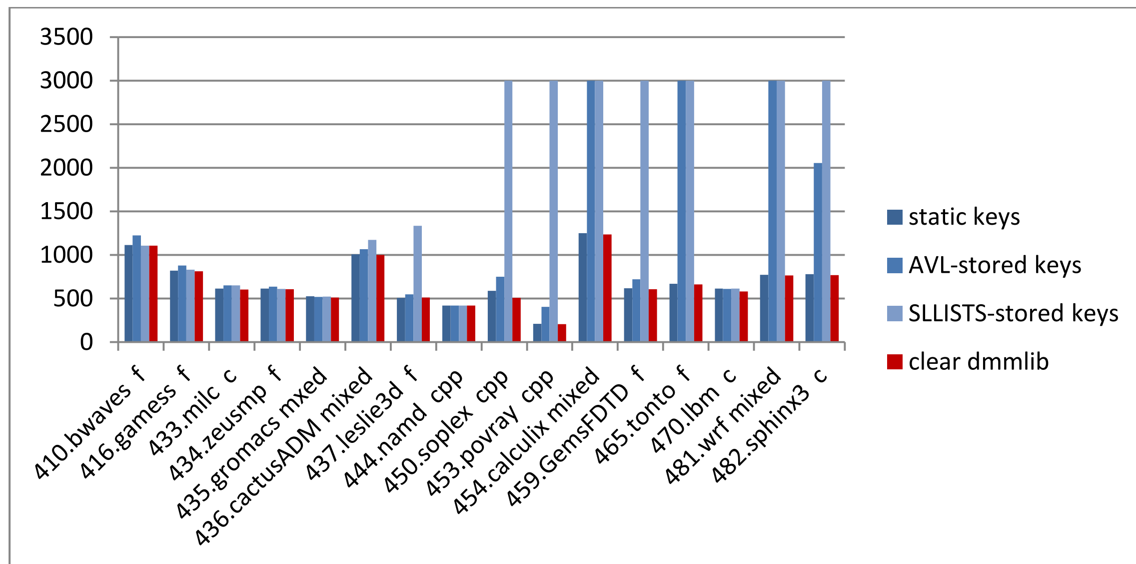

Chart 5-7: ***Floating point*** *benchmarking results of dmmlib with* ***encrypted lists*** *securing mechanism enabled for all of three key producing organizing schemes.*

It is clear that encrypted list security policy adds serious overhead for the two dynamic implementations (AVL-trees and SLLISTS). Heap is organized in linked lists, free-lists as well as raw-block lists. Because of it's tend to be reorganized in order to offer memory efficient use, it has to encrypt and decrypt all the list links of the list that it need update. Consequently it presents a high transaction ratio with keys pool and of course as the AVL-implementation can serve its requests faster, it adds less overhead than SLLISTS.

This perspective results from both Charts 5-7 and 5-8. Again the overhead that the specific securing implementation adds it is not prohibitive as it comes with a serious security principles guaranteeing linked lists robustness.

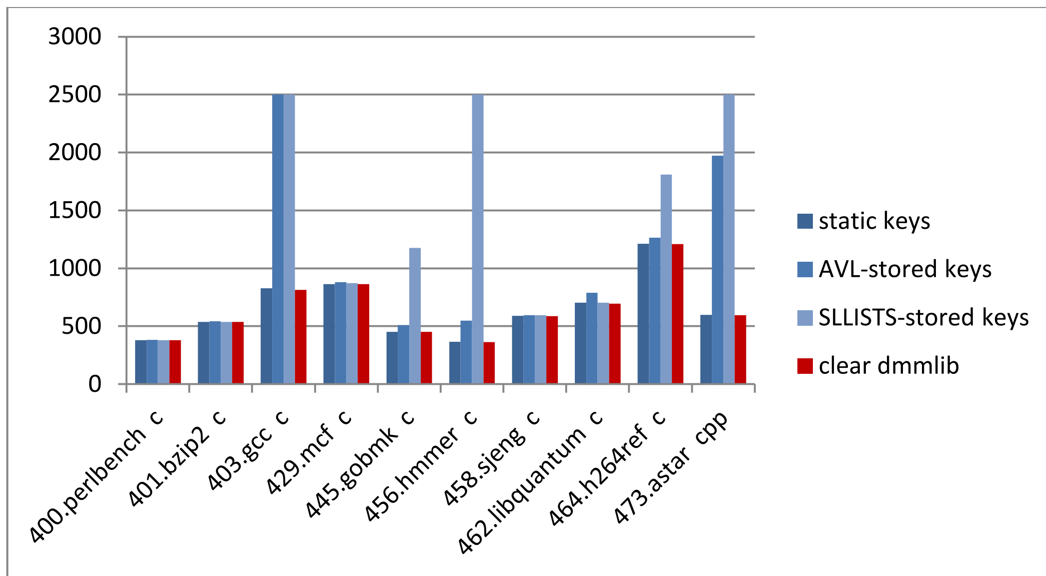

Chart 5-8: ***Integer benchmarking*** *results of dmmlib with* ***encrypted lists*** *securing mechanism enabled for all of three key producing organizing schemes.*

5.6 Benchmarking dmmlib with encrypted list validation mechanism enabled

In this section we present the results of benchmarking encrypted list validation mechanism. Encrypted list validation mechanism is a bit more pretentious implementation then its ancestor whose benchmarking results is presented in chapter 5.7. This is because the validation scheme that it implements. Of course as is figured out in charts 5-9 and 5-10 below, the behavior of encrypted list and encrypted list validation mechanisms are close. However the small extra overhead that encrypted list validation mechanism adds is negligible considering that it offers the principles of maintaining all every heap list protected by any **list link oriented attack** in an active way as it can detect it.

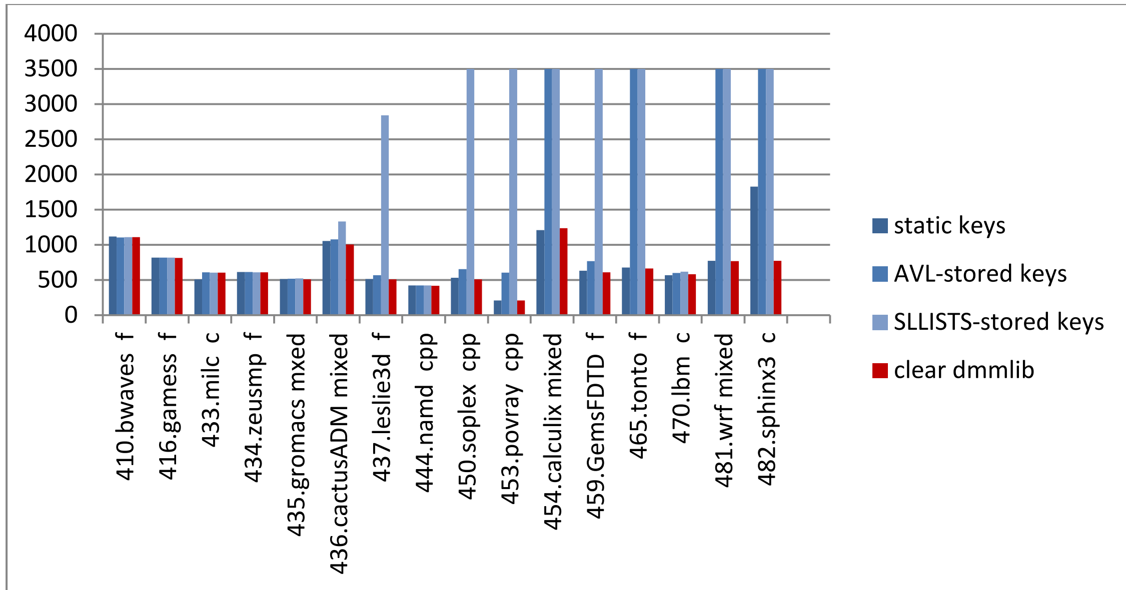

Chart 5-9: ***Floating point*** *benchmarking results of dmmlib with* ***encrypted lists validation*** *securing mechanism enabled for all of three key producing organizing schemas.*

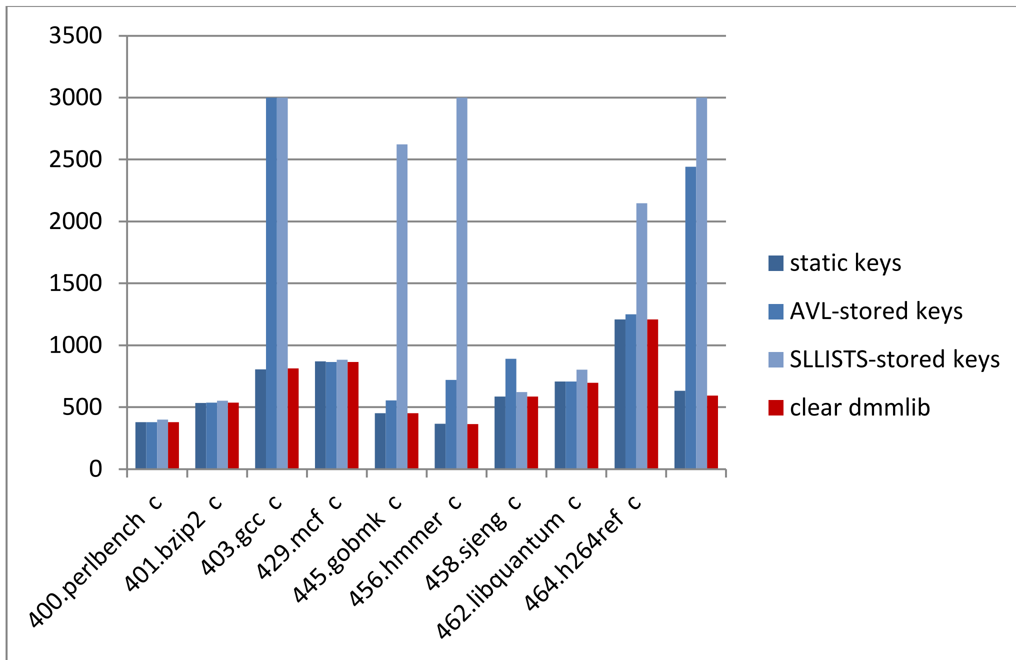

Chart 5-10: ***Integer benchmarking*** *results of dmmlib with* ***encrypted lists validation*** *securing mechanism enabled for all of three key producing organizing schemes.*

5.7 Benchmarking dmmlib with mixed security schemas

Finally we tested two extra scenarios where there is enabled mixed security schema. The first consists of canaries and encrypted list validation security mechanisms (protecting both heap organization layers) while the second consists of encrypted indirect pointers and again encrypted list validation. The results are displayed in the following four charts.

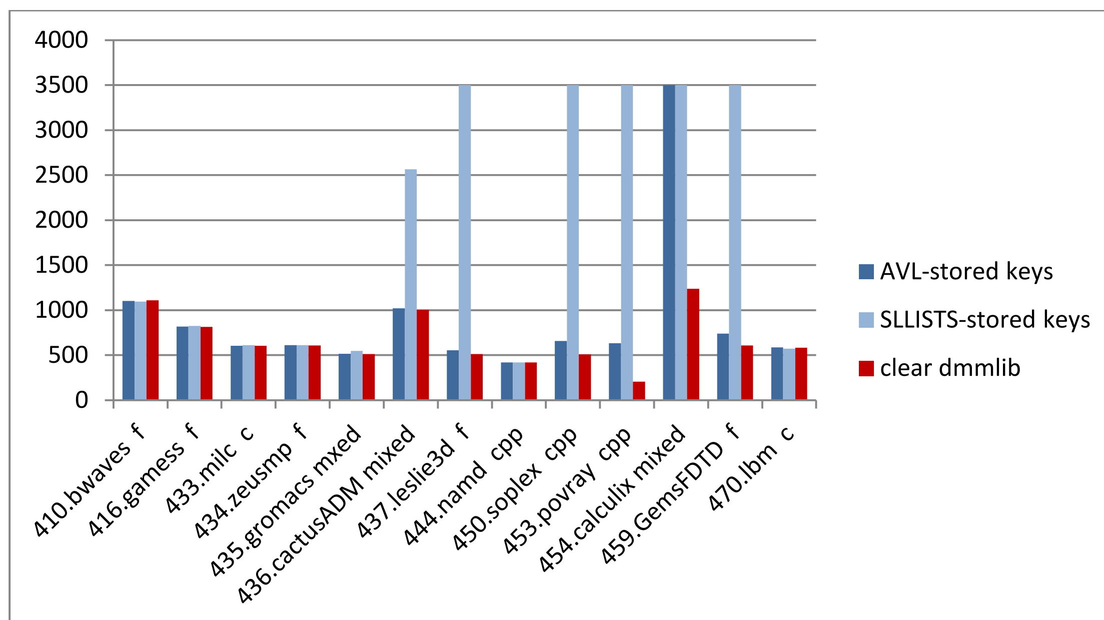

Chart 5-11: ***Canaries*** *combined with* ***encrypted lists validation*** *security mechanisms,* ***floating point benchmarking****.*

Benchmarking of the first mixed schema (chart 5-11) indicates that the overhead of the coexisting security schemas is cumulative rather than cascading. This results from the fact that dynamic memory manger is actually a monolithic single thread module that runs on behalf of mother process without any multitasking touch. If a canary is going to be validated and a list link to be decrypted, both request will be serialized and serviced the one behind the other. However mini-Heaps implementation guarantees that the one mechanism will not add any overhead to the other. This is because of the fact that every mechanism maintains its separate key pool. In this way it is ensured that there is no interaction between the collaborating security schemas.

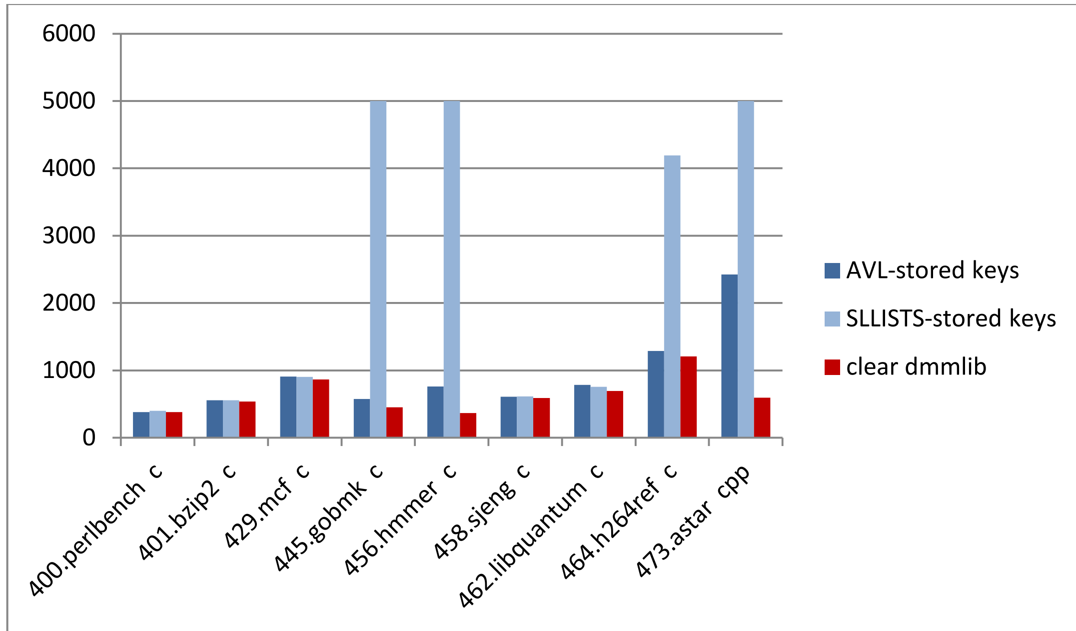

Chart 5-12: ***Canaries*** *combined with* ***encrypted lists validation*** *security mechanisms,* ***integer benchmarking.***

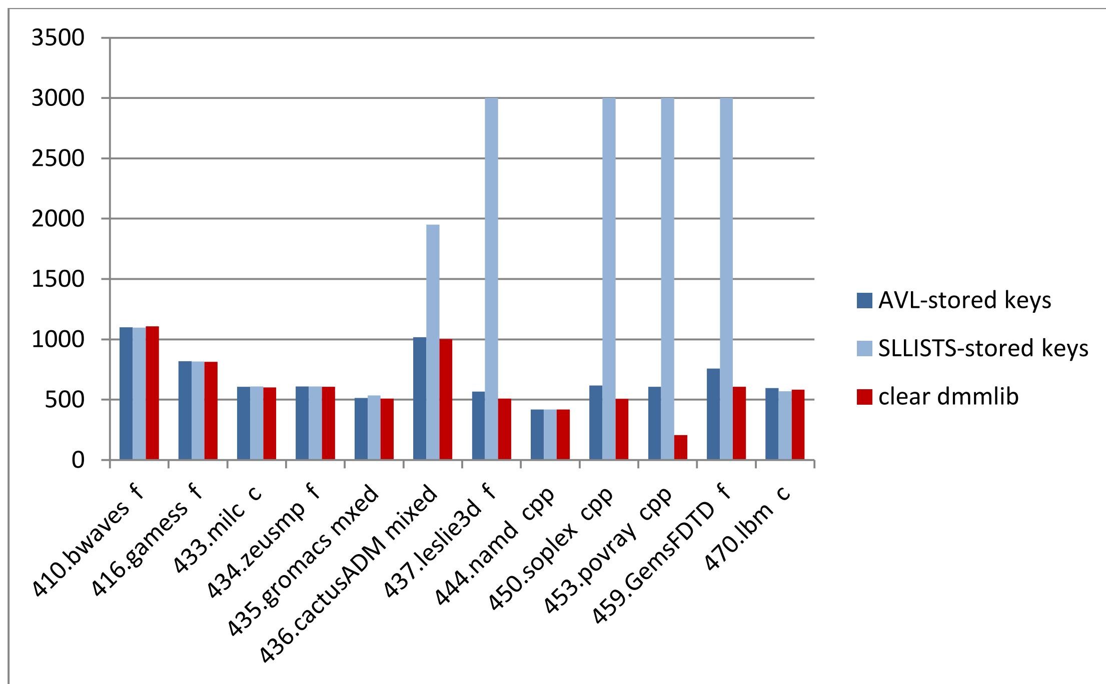

Chart 5-13: ***Encrypted pointers*** *combined with* ***encrypted lists validation*** *security mechanisms,* ***floating point benchmarking***.

Thus coexistence of two different security schemas does not affect the implementation's performance dramatically. Of course the number of benchmarks that do not finish in the predefined time bounds increases without although to be a dissuasive factor for combining more than one mitigation technique to create a security mechanism as we done here.

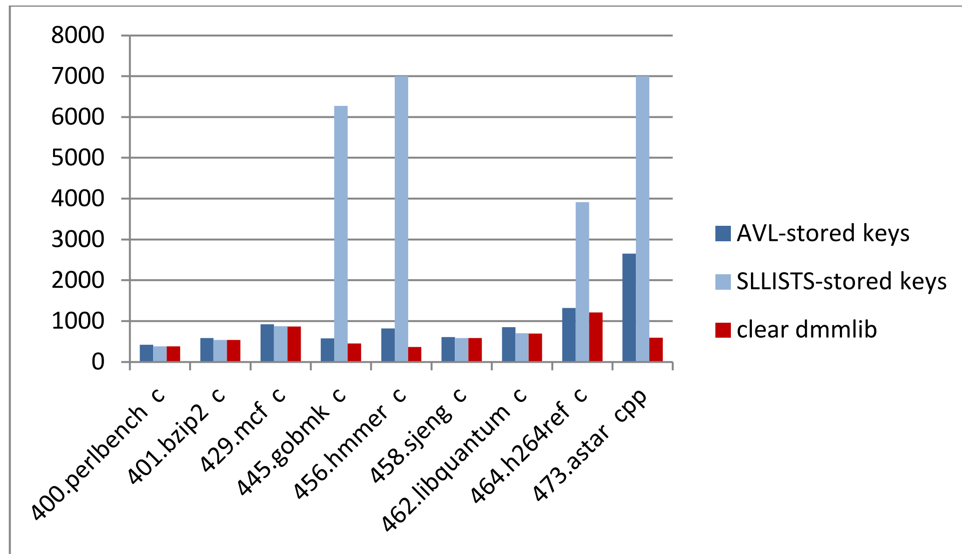

Chart 5-14: ***Encrypted pointers*** *combined with* ***encrypted lists validation*** *security mechanisms,* ***integer benchmarking.***

5.8 Decision schema – benchmarking explanations

First of all it must be clarified that in every chart, the highest displayed time value measurements (highest bars) **are not actually true**. They actually **denote the fact that for the specific benchmark, the requested time was more than the charts scalability could afford**. If measurements, higher than the displayed ones, were compromised in charts, charts would not be able to scale correctly in an A4 paper.

Moreover benchmarking script has been fed with time limits in order to deteriorate unreasonable resources consumption. A benchmarking threshold produced dynamically from previous benchmark results, have been considered enough to offer us clear performance view.

As it was mentioned before, the evaluation of the benchmark results is based on a simple comparison scheme. First of all dmmlib, with no security mechanism enabled, is benchmarked and compared with the state of art mechanisms. In case of satisfying comparison results, the allocator that we based our research is competitive enough, so we can continue to our module indirect benchmarking.

Consequently benchmarking was used not only as a hard testing and evaluating procedure but also as a **feedback tool helping us to understand if we follow an efficiently competitive route** during implementation stage.

5.9 Overall Evaluation

As it was expected, our security mechanism adds some time overhead to the running process. This overhead depends on the type of the security mechanism that is enabled and the key organization that it uses **in direct combination with the application itself**.

Consequently the above measurements cannot be explained just by referring to the dynamic memory manager that benchmarks have used and the security mechanism that were enabled during benchmarking stage. Of course **static keys** schema is the less while **single linked lists** is the most pretentious of all the three schemes. However this behavior can be hidden behind a bottleneck which is caused by a completely different reason.

Concluding, charts denote that the secured dmmlib is not as efficient as the original. However its efficiency indicator is close to the original’s one and considering the security principals that it affords, it is a really competitive dynamic memory manager.

6 FUTURE WORK

After this first approach to the dynamic memory managers' security issue, plenty of ideas have been born, based on evolutionary implementations and on the already described principles. These principles and ideas are being invigorated by modern hardware implementations. Multicore systems, NOC (Network On Chip), lightweight processes (threads), embedded systems, as well as universal software platforms (JVM) demand special target oriented instead from the described global transparent solutions.

6.1 Server side allocator

As it is aforementioned in chapter 4.1.1, dynamic memory manager can be treated as a server. Consequently a newly designed allocator can be designed as a server to serve one or more running applications. By embedding all the security principles used by the modern client-server big scale architectures combining them with a robust and self secure implementations it is possible to end up with a scalable competitive dynamic memory manager which maintains more than one heaps oriented from equal number of running applications vanishing many security issues.

Considering the performance issue, it is noteworthy the plethora of possibilities that such an implementation offers to any **interoperable** or **collaborating** environment which are rising minute after minute. Issues such as shared memory and message passing aiming to synchronization could be solved by a universal server allocator instead of kernel based solutions. Furthermore most of the security issues confrontation and application data maintenance should be allocator's responsibility, thus programming such collaborating environments should be easier.

6.2 Data integrity

Every proposed solution for heap security until know, focuses on heap's metadata integrity. Of course this, as it is described in previous chapters, is the biggest concern of the security oriented developers. Nevertheless, it is essential nowadays for an environment to be self secure to guarantee not only its internal organization but also its **effectiveness**. Consequently, the effectiveness demand necessitates not only the metadata but also the **data integrity**.

Watchdogs

Any schema offering application's **data** protection should be real time[23]. Thus the security policy ordering the allocator to find out if any attack has been launched just before any allocation or deallocation is not enough to offer real time data protection. Thus watchdogs are being proposed. Watchdogs are lightweight processes, threads, whose purpose is to validate continuously the data of their range of interest. In this way we can have real time investigation involving real time protection using **parallelism**.

Nowadays chips come with many cores, cores fast and powerful. NOC[24] is also a powerful architecture for parallelism embedding large scale distribution philosophy that modern cloud systems are based on, on just any multicore chip. Thus, one from the most useful ways to use this good, is to use it for security completion.

Recovery

Finally it is essential data recovery mechanisms to be implemented in order to recover data after their corruption's detection in way that the application will not be affected at all. Of course the data recovery mechanism can support metadata recovery offering a really robust heap organization.

---

[23] Real time protection refers to corruption detection just in the next clock cycle, after the one that it took place

[24] NOC aka Network On Chip

6.3 Embedded systems' point of view

Nowadays embedded systems are something more than trivial. Thus it is essential to focus on these systems security too. Because of the fact that embedded systems has a number of hard restrictions defined by their purpose there is not obvious solution that can offer to all of them the same security and efficiency qualifications. Moreover it is not obvious which components of each implementation can be hardware and which can be software implemented. Thus embedded systems cannot be treated just like consumers as software applications are, but also as assistants, meaning that the dynamic memory manager and its security policies have to be essential parts of their design and implementation.

Conclusion

The specific paper classifies, defines and describes some of the well known attacks that can be launched against any dynamic memory manger. It also describes the best of the existing efforts to secure any dynamic memory manger as well as the main principles of mitigation implementations.

Moreover there have been developed, tested and benchmarked some new securing methodologies based on the preexisting ones and on the defined securing principles, as well as it has been proposed the principles of a futuristic dynamic memory manager focused primarily on security without underestimating performance requests.

The implemented allocator [14], based on open source dmmlib is **efficient enough** in way that it can be compared with the existing competitors, while it offers protection against numerous attacks. However the specific implementation cannot guarantee that it offers a totally trusted dynamic memory manger, because of the numerous existing and newly born attack morphemes.

Finally, as it is explained before, the security of an application depends firstly on the application itself and secondly to the dynamic memory manger. So in order to have a trusted environment there should be responsible programming!

Appendix

List of Figures

Figure 1-1: *Process' memory segmentation defined in the virtual address space* 2

Figure 1-2: *Requesting memory by using immediate system calls* 5

Figure 1-3: *Dynamic memory manger implementing the mid-layer memory exchange* 5

Figure 1-4: *A fragment of a freelist-based heap, as used by Linux and Windows. Object headers precede each object, which make it easy to free and coalesce objects* 7

Figure 1-5: *A fragment of a segregated-fits BiBOP-style heap, as used by the BSD allocators (PHKmalloc and OpenBSD). Memory is allocated from page-aligned chunks, and metadata (size, type of chunk) is maintained in a page directory. The light lines indicate the list of free objects inside the chunk and the thick one indicates the owner of the chunk's interior.* 8

Figure 2-1: *Buffer overflow example using the embedded to the allocator copy functionality. The allocator mistakenly copies a larger buffer's content to a smaller one and as a result it overwrite the following fields. In case that the tree overwritten bytes are heap's metadata the corruption is close.* 20

Figure 2-2: *Memory Chunks A, B, C, D organized in a vulnerable way just before the attack to DL-allocator is launched. Size: chunk's size, PU: previous chunk in use, fd: points the next free chunk, bk points the previous free chunk* 21

Figure 2-3: *Malicious code injection through indirect pointer overwriting. Allocator overwrite text area. In the left half it is presented the buffer overflow exploitation and its result is obvious in the right half.* 22

Figure 2-4: *Buffer overflow: Backward Consolidation scheme. Note that fake chunk must be denoted as free. This can be done in two ways. The first is overflowing the following chunk's metadata and unset "previous in use" attribute while the second one, in case of large enough buffer, is to create another continuous fake chunk pretending to be in use, and denoting to its PU attribute that the early one is free.* 23

Figure 2-5: *Heap spraying attacks scheme. When the indirect call will happen it is extremely possible that the malicious code will be executed instead of the "called" method.* 26

Figure 3-1: *In left canary mechanism application in free-list based allocator. In right canary mechanism application in BiBOP-style allocator.* 33

Figure 3-2: *Operation of singly linked lists. The encoding function is symbolized as e, while the decrypting one as d and key as k. Left half: the operation of insertion. Right half: the operation of deletion.* 34
Figure 3-3: *Operation of double linked lists. The encoding function is symbolized as e, the decrypting one as d and key as k. Insertion functionality is displayed. Deletion happens in similar way.* 35
Figure 3-4: *Left: list element structure for dual linked lists.* 36
Figure 3-5: *DieHard 's heap layout. The heap is divided into separate regions allocating in random positions in each region, objects of the same size. Notice the different layouts across replicas.* 41
Figure 3-6: *Replicas voting mechanism as it is documented in [11]* 42
Figure 3-7: *Overview of DieHarder's heap layout as it is documented in [11]* 45
Figure 4-1: *Dynamic memory managers classification according to application's and system point of views* 47
Figure 4-2: *Application allocation request providing semantics to dynamic memory manager.* 49
Figure 4-3: *Application request allocation from the dynamic memory manager, control passes to DMM and again in application.* 50
Figure 4-4: *Snapshots of tree raw blocks organized in a linked list while the free blocks that they contain are organized in another linked list too.* 52
Figure 4-5: *Block header fields used by canary mechanism in order to distinguish blocks' states. Red denotes the one used for free blocks while green the one used for in use blocks.* . 55
Figure 4-6: *The four modules of the security mechanism, corporation schema. Arrows indicate the keys or the encrypted/decrypted values exchange directions.* 60
Figure 4-7: *mini-Heap's internal organization* 66

Chart list

Chart 5-1: *Floating point benchmarking results for DieHarder, DLallocator and dmmlib without any security mechanism enabled.* 68
Chart 5-2: *Integer benchmarking results for DieHarder, DLallocator and dmmlib without any security mechanism enabled.* 69
Chart 5-3: *Floating point benchmarking results of dmmlib with canaries securing mechanism enabled for all of three key producing organizing schemes.* 71
Chart 5-4: *Integer benchmarking results of dmmlib with canaries securing mechanism enabled for all of three key producing organizing schemes.* 71
Chart 5-5: *Floating point benchmarking results of dmmlib with encrypted indirect pointers securing mechanism enabled for all of three key producing organizing schemes.* 72
Chart 5-6: *Integer benchmarking results of dmmlib with encrypted indirect pointers securing mechanism enabled for all of three key producing organizing schemes.* 73
Chart 5-7: *Floating point benchmarking results of dmmlib with encrypted lists securing mechanism enabled for all of three key producing organizing schemes.* 74
Chart 5-8: *Integer benchmarking results of dmmlib with encrypted lists securing mechanism enabled for all of three key producing organizing schemes.* 75
Chart 5-9: *Floating point benchmarking results of dmmlib with encrypted lists validation securing mechanism enabled for all of three key producing organizing schemas.* 76
Chart 5-10: *Integer benchmarking results of dmmlib with encrypted lists validation securing mechanism enabled for all of three key producing organizing schemes.* 76
Chart 5-11: *Canaries combined with encrypted lists validation security mechanisms, floating point benchmarking.* 77
Chart 5-12: *Canaries combined with encrypted lists validation security mechanisms, integer benchmarking.* 78
Chart 5-13: *Encrypted pointers combined with encrypted lists validation security mechanisms, floating point benchmarking.* 78
Chart 5-14: *Encrypted pointers combined with encrypted lists validation security mechanisms, integer benchmarking.* 79

References

[1] http://en.wikipedia.org/wiki/Buffer_overflow_protection

[2] Emery D. Burger, Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languges.

[3] H. Shacham, M. Page, B. Pfaff, E. Jin Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In CCS ’04: Proceedings of the 11th ACM conference on Computer and communications security, 2004.

[4] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 1–11, New York, NY, USA, 2007. ACM Press.

[5] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. Communications of the ACM, 51(12):87–95, 2008.

[6] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In J. N. Matthews and T. E. Anderson, editors, SOSP, pages 87–102. ACM, 2009.

[7] https://www.owasp.org/index.php/Format_string_attack

[8] Kyungtae Kim, Changwoo Pyo. Securing heap memory by data pointer encoding.

[9] http://www.spec.org

[10] P.-H. Kamp. Malloc(3) revisited. http://phk.freebsd.dk/pubs/malloc.pdf

[11] Gene Novark, Emery D.Burger. Dieharder: Securing the heap.

[12] https://bitbucket.org/joko/dmmlib

[13] http://valgrind.org/docs/manual/mc-manual.html

[14] https://bitbucket.org/mcchran/securing-the-dynamic-memory-allocator