To appear in the *Intl. Symp. on Architecture Support for Programming Languages and Operating Systems* (ASPLOS), Oct 2006.

Comprehensively and Efficiently Protecting the Heap *

Mazen Kharbutli
Jordan Univ. of Science and Technology
kharbutli@just.edu.jo

Xiaowei Jiang Yan Solihin
North Carolina State University
{xjiang,solihin}@ece.ncsu.edu

Guru Venkataramani Milos Prvulovic
Georgia Institute of Technology
{guru,milos}@cc.gatech.edu

Abstract

The goal of this paper is to propose a scheme that provides comprehensive security protection for the heap. Heap vulnerabilities are increasingly being exploited for attacks on computer programs. In most implementations, the heap management library keeps the heap meta-data (heap structure information) and the application's heap data in an interleaved fashion and does not protect them against each other. Such implementations are inherently unsafe: vulnerabilities in the application can cause the heap library to perform unintended actions to achieve control-flow and non-control attacks.

Unfortunately, current heap protection techniques are limited in that they use too many assumptions on how the attacks will be performed, require new hardware support, or require too many changes to the software developers' toolchain. We propose *Heap Server*, a new solution that does not have such drawbacks. Through existing virtual memory and inter-process protection mechanisms, Heap Server prevents the heap meta-data from being illegally overwritten, and heap data from being meaningfully overwritten. We show that through aggressive optimizations and parallelism, Heap Server protects the heap with nearly-negligible performance overheads even on heap-intensive applications. We also verify the protection against several real-world exploits and attack kernels.

Categories and Subject Descriptors C [*0*]Hardware/software interfaces

Keywords Computer Security, Heap Attacks, Heap Security, Heap Server

1. Introduction

Motivation. Computer systems have evolved significantly in the last few decades, with machines that are increasingly inter-connected and run increasingly complex software. Unfortunately, such systems are also more exposed to security attacks which have not only grown in quantity, but also in variety. As a result, users are very concerned about protecting their systems from viruses, worms, trojan horses, and spyware.

In order to attack a single application, attacks often rely on overwriting a set of locations M with a set of values V with the purpose of altering the program behavior when it uses V. Attackers may use a variety of well-known techniques to overwrite M with V, such as *buffer overflows*, *integer overflows*, and *format strings*, typically by supplying unexpected external input (e.g. network packets) to the application. The desired program behavior alteration may include direct control flow modifications in which M contains control flow information such as function pointers, return addresses, and conditional branch target addresses. Alternatively, attackers may choose to indirectly change program behavior by modifying critical data that determines program behavior.

Researchers have shown that the stack is vulnerable to overwrites, so many protection schemes have been proposed to protect it. However, the heap has received less attention. In this paper, we will concentrate on protecting the heap against *heap attacks*, which are growing in number. Examples include the slapper worm on the Apache web server [21], GIF image processing library heap overflow attack on the Mozzila web browser [9], and heap corruption attack on Null HTTPD [6]. Moreover, new vulnerabilities are continuously discovered for popular programs such as Microsoft Windows [26], Microsoft Internet Explorer [30], CVS [28], and Check Point Firewall-1 [29]. Even the *Execution Protection* and *Sandboxing* heap protection schemes used by the recent Windows XP Service Pack 2 have been shown to be vulnerable [1].

The growth in heap attacks is mostly due to several main weaknesses in the way a program's heap space is managed. First, in most current implementations, memory allocations and deallocations are performed by user-level library code which keeps the heap meta-data (heap structure information) and the application's heap data stored in an interleaved fashion in contiguous locations in the main memory. Secondly, heap data and heap meta-data are not protected from each other. Such heap management implementations are inherently unsafe: they allow a vulnerability in how heap data is managed to be exploited to corrupt the heap meta-data. For example, the lack of bound checking on a heap buffer can be exploited to overflow the buffer and corrupt the heap meta-data. Heap meta-data is used by the heap management library to perform various functions such as allocating/deallocating space for an object, consolidating free space, and reallocating a recently deallocated object. With corrupted meta-data, the heap-management library performs unintended actions such as overwriting an arbitrary memory location with an arbitrary value, where the attacker can choose both the target location and its new value. By overwriting a location that contains control flow information, the attacker can hijack control flow and execute malicious code. Even if control flow hijacking is prevented, attackers can still do considerable damage, e.g. by overwriting critical program data to alter the behavior of the program.

A third weakness of current heap management implementations is the predictability of the heap layout. This allows attackers to figure out the exact location of various heap meta-data structures and of critical heap data such as function pointers.

Current Techniques are Inadequate. Despite the growing threat of heap attacks, current heap protection schemes either make

* This research was supported by NSF Early Faculty Career Awards CCF-0347425 and CCF-0447783, NSF award CCF-0429802, IBM Faculty Partnership Award, North Carolina State University and Georgia Institute of Technology. Much of this work is based on Kharbutli's PhD thesis at NCSU.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.
Copyright © 2006 ACM 1-59593-451-0/06/0010. . . $5.00.

too many assumptions on how heap attacks will be carried out, or are difficult to implement due to high performance overheads and/or demanding too many changes (to user code, library, and compiler). It is hard for users to adopt solutions that require too many changes due to the need to retool their software development toolchain.

It is useful to note that heap attacks are carried out in three basic steps [2]:

1. *Bug/vulnerability exploitation stage*: a bug or vulnerability in an application is exploited to corrupt certain structures, e.g. by corrupting heap meta-data through a buffer overflow, format string vulnerability, or other vulnerabilities.
2. *Activation stage*: the corrupted location is transformed into a dormant attack, e.g. the corrupted meta-data causes the heap library to alter program control information or critical data. This stage is not necessary for some attacks.
3. *Seized stage*: the attack leaves its dormant state and is fully executed, e.g. program behavior is altered or its control is redirected to malicious code.

Note that latter stages can be carried out with more variety than the earlier stages. Many of the current techniques make assumptions on what particular steps will be performed at the activation or seized stages and prevent the assumed steps from being successfully carried out. However, we argue that it is difficult to foresee all possible steps that attackers may perform in the latter stages, and hence it is safer to prevent the first stage (the corruption) from being carried out successfully. For example, non-executable heap [18] prevents the seized stage of an attack by not allowing injected malicious code in the heap to execute. However, the protection fails when the assumption breaks, such as when the malicious code is not in the heap area, or when the attack relies on executing existing code rather than injecting its own.

Design Criteria. The goal of this paper is to provide *comprehensive protection* for the heap. We list the following criteria that are required for a comprehensive and likely-implementable solution:

1. **Minimal assumptions** about specific steps an attack will rely on. As discussed earlier, assuming that an attack will happen in a certain way can be dangerous and overlook future attack mechanisms. As a result, our solution must prevent the first stage of an attack (the bug/vulnerability exploitation stage) as a way to prevent the corruption from happening in the first place.
2. **Low overhead**. For wide adoption, a security solution must have a low overhead even for heap-intensive applications, since applications are moving towards object-oriented design which tend to have a large and active working set in the heap.
3. **Existing hardware**. To ensure wide adoption, our design should make use of existing hardware protection mechanisms, making it implementable in existing systems.
4. **Few code modifications**. It is hard for users to adopt a solution that requires too many changes to their software development toolchain. Modifications to the code should be centralized to the heap management library and affect user code as little as possible.

Contributions. To provide a comprehensive protection against various attacks on the heap, we propose *Heap Server*. Heap Server is a separate process that performs heap management on behalf of an application. Heap Server's main protection features include:

- *Separation of heap data and meta-data*, which is achieved through two mechanisms. First, a new heap meta-data organization removes the interleaving of heap meta-data and heap data and localizes the heap meta-data in a range of contiguous locations. Secondly, for even stronger protection, Heap Server removes the heap meta-data from the application's address space and keeps it in its own address space. By keeping it in a separate address space, vulnerabilities in the application can no longer be used to corrupt heap meta-data, unless the underlying inter-process protection mechanism is broken. Using this protection, both *contiguous* and *non-contiguous* overwrite attempts can no longer corrupt the heap meta-data.
- *Layout obfuscation of heap data*. While heap meta-data is strongly protected against overwrites, Heap Server also protects the heap data against meaningful overwrites through *heap layout obfuscation*, which randomizes heap data layout. Similar to a previously proposed address obfuscation, Heap Server adds random paddings between heap objects. However, Heap Server goes beyond that and provides *random recycling*, which randomizes the selection of a heap object to recycle to satisfy an allocation request. The random seed is determined at run time based on the application's execution environment. Layout obfuscation makes it harder for an attacker to figure out the exact location to overwrite, and even when the location is figured out for one instance of a program, other instances of the same program will have different layouts and the attack can not be repeated easily.

Heap Server prevents the exploitation stage of attacks from occurring, and hence uses minimal assumptions on the specific steps used in an attack. In addition, it uses existing hardware and Operating System (OS) mechanisms for inter-process protection. Heap Server only requires modifications to the allocation/deallocation routines. In most applications, they are localized in the heap management library. Some applications manage their own heap through custom allocation/deallocation routines, usually for performance reasons. However, we found that custom routines are typically localized in very few functions, leading us to believe that they can be modified to utilize our solution without significant programming effort. Moreover, Heap Server incurs nearly-negligible execution time overheads on a real system, using both SPEC applications as well as allocation-intensive benchmarks running on a 2-way SMP 2-GHz Intel Xeon processors running RedHat Linux 8.0. Finally, we also verify the protection against several real-world exploits as well as several attack kernels.

The rest of the paper is organized as follows: Section 2 discusses related work and Section 3 illustrates heap attacks. Then, Section 4 describes our solution in details, while Sections 5 and 6 discuss our evaluation setup and present our security/performance evaluation. Finally, Section 7 summarizes our findings.

2. Related Work

Several approaches have been proposed in the past for preventing heap attacks, or security attacks in general. We examine them based on our design criteria described in Section 1. Unfortunately, none of these previously proposed solutions meets all our design criteria. Note however that some of them protect against security attacks in general, while the Heap Server focuses on protecting the heap.

Many current approaches make assumptions on the particular steps the heap attacks will perform and try to prevent the seized stage from being carried out. One proposed solution is to make the heap non-executable [18]. Non-executable pages are supported in virtual memory hardware and in recent versions of various Operating Systems (OSes), such as Microsoft Windows XP Service Pack 2 [23], Linux and OpenBSD. Dynamic hardware tracking of external inputs and their use chains have been proposed by Suh et al. [12]. If tagged data is executed or used as a branch/jump target address, an attack is detected. A related hardware scheme is

Minos [14], which tags each data item with an integrity bit that is set to '1' if the data comes from reliable sources. An attack is detected when a data item with integrity bit of '0' is used for a change in control flow. Program shepherding [16] is a technique through which policies regarding control flow transfer can be specified and enforced. Non-executable heap, dynamic tracking, Minos, and program shepherding try to prevent the last stage of an attack (the seized stage), and assume that an attack relies on control flow hijack. They fail when the attack does not rely on control flow hijack. For example, *non-control* attacks do not rely on control flow modifications, yet have recently been demonstrated to be as powerful as control-flow attacks [6]. In addition, non-executable heap also fails for control-flow attacks that do not rely on code injection, such as ones that divert the control flow to library code [22]. Our solution differs from non-executable heap, dynamic tracking, and Minos in that we prevent the first stage of an attack, rather than latter stages, hence we use minimal assumptions on how latter stages will be carried out, regardless of whether an attack relies on code injection, control-flow or non-control modifications. In addition, compared to dynamic tracking and Minos, our solution does not require new hardware mechanisms.

Storing pointers encrypted in the main memory have been proposed as a software approach in PointGuard [7] and recently as a hardware approach for stack protection [19]. The compiler or binary rewriter is required to identify type information, and apply encryption/decryption on pointer accesses. However, without sacrificing language compatibility, accurately identifying pointer accesses is difficult in C programs due to lack of type information. For example, many C-language features depend on functions with untyped buffers (e.g., `bzero` or `memcpy`) or take untyped parameters (e.g., `printf`), making it difficult to get an accurate type information. Also, existing library code does not already contain type information. Furthermore, without extra hardware for encryption, encrypting/decrypting each pointer for every write/read incurs a high performance overhead. Finally, future heap attacks may not rely on corrupting pointers but other parts of meta-data. Our solution differs from pointer encryption in that our solution only needs modifications to the allocation/deallocation routines, is compatible with existing user code, and protects all heap meta-data structures (not just pointers) against corruption.

Obfuscation techniques for heap protection and beyond have also been proposed [5, 20, 31] with a goal of making the address space less predictable and increases the effort of the attackers to make a successful attack. Transparent Runtime Randomization (TRR) [31] and Address Space Layout Randomization (ASLR) [20] are software techniques that randomize the starting address of various segments (heap, stack, BSS, etc.) and dynamic library codes when a program is loaded [31]. Although TRR and ASLR increase the difficulty of attacks that involve more than one segment, they cannot protect against attacks that are solely carried out within the heap segment. Address obfuscation [5] is another technique that, in addition to randomizing the starting address of various segments, also introduces random padding between consecutive heap chunks to make the heap layout less predictable. However, recent work by Shacham et al. [13] shows that randomization implementations on 32-bit architectures suffer from low entropy of 16-20 bits, which does not give sufficient protection against determined attackers. Additionally, in realistic implementations, the amount of padding between chunks are limited to the minimum of 25% of the requested allocation size and a threshold value (e.g. 128 bytes), to avoid excessive fragmentation of the heap space. With a maximum of 128-byte padding, there is only $\frac{128}{8} = 16$ possible padding sizes between two consecutive 8-byte aligned heap objects, which is insufficient against determined attackers. As a result, we believe that address obfuscation can only be used for heap data, where attacks require detailed knowledge of the application. Heap meta-data, however, has the same structure in all applications that use the same library, so the incentive for devising meta-data attacks is higher. As a result, meta-data needs stronger protection.

Heap Server incorporates Bhatkar et al.'s address obfuscation technique and improves it by adding *random recycling* of heap objects. We note that due to temporal locality optimizations, the heap management library often immediately recycles the most-recently freed object for a subsequent allocation. This regularity means that once attackers figure out the layout of the heap, future allocations have predictable layout. Our random recycling introduces random selection among eligible free heap objects when allocation is requested. This protection makes it harder for an attacker to figure out the target location to overwrite *throughout the program execution*.

Concurrent to our work, Berger and Zorn proposed DieHard [3]. Like Heap Server, DieHard employs similar techniques to separate heap meta-data from heap data storage, and applies randomized padding between heap chunks. DieHard also examines using multiple redundant instances of an application to detect attacks. One major difference with Heap Server is that the Heap Server further protects heap meta-data by keeping it in a separate address space, completely avoiding application vulnerabilities from overwriting it.

3. Heap Attacks and Protection

3.1 Bug/Vulnerability Exploitation Stage

The entry point of a security attack is often external input from the network. The input causes certain overwrites due to vulnerabilities in the program. In a *buffer overflow* vulnerability, a program lacks buffer bound checking and allows a long string input to overflow and overwrite adjacent bytes beyond the buffer. In an *integer overflow* vulnerability, a signed-integer variable used in indexing a structure is not checked for its valid range. A too-large value may be input and becomes a negative number in the variable, causing an access to a location beyond the structure's range. In a *format string* vulnerability, a use of `printf` family of functions that accepts formatting information in its string input may lead to both *read* and *write* attacks. For example, `%s` or `%x` format tokens can reveal the content of the stack and possibly other locations, while `%n` format token causes the number of bytes printed to be written to a location specified in format string arguments, which allows an overwrite to a random location with the value chosen by the attacker. Note that while buffer overflows allow contiguous overwrites to adjacent bytes, integer overflow and format string vulnerabilities also allow non-contiguous overwrites in a more random-access fashion.

In addition to vulnerabilities, a program may naturally have *bugs*, which manifest under certain inputs or execution environment. The bugs may be transformed into security attacks if attackers know how to exploit them. For example, in a *dangling pointer* bug, a still-in-use heap chunk is incorrectly deallocated, allowing the heap management library to write heap meta-data information to it, causing subsequent access to the chunk to use corrupted data. In an *invalid free* bug, an invalid value is incorrectly passed to the deallocation routine, resulting in the address pointed by the value to be overwritten by heap meta-data. In a *double free* bug, an already-deallocated heap chunk is deallocated again, causing inconsistencies in the free list that maintains deallocated objects. Bug exploits may be due to actual bugs in the program, or be follow-up actions after a successful vulnerability exploitation.

3.2 Activation Stage

The previous bug/vulnerability exploitation stage allows an overwrite associated with the vulnerability to occur. In many cases, the attacker still needs to convert this initial overwrite into a more powerful program behavior alteration. This is performed in the *activation stage*. The attackers may want to modify control flow infor-

mation such as function pointers, return addresses, and conditional branch target addresses. Alternatively, attackers may choose to indirectly change the program behavior by modifying critical data that determines program behavior. Such *non-control attacks* have been demonstrated recently to be as powerful as control attacks by Chen et al. [6]. For example, in WU-FTPD, overwriting the effective `UID` of an application allows the application to gain root access, while in Null HTTPD, overwriting CGI-BIN also results in root compromise. Attacks may also rely on malicious *code injection* in which code is injected by the attackers, or may simply use existing code that was not supposed to be executed, such as out-of-context library code [22].

In the activation stage, the initial overwrites due to bug or vulnerability exploitation stage is converted into an overwrite of critical control or non-control data. Sometimes, the activation stage is not necessary, such as when critical data's location is known and can be directly overwritten with the desired value. In the heap, this could happen when a heap chunk has a function pointer of which its location is known by the attacker, and vulnerabilities exist to overwrite it, e.g. the previous heap chunk has a vulnerable buffer. Because such attacks do not target the heap meta-data, but target heap data directly, we refer to this as *data overwrite attacks*.

To illustrate several types of attacks in the activation stage, we will first start with an overview of heap meta-data organization.

Heap Meta-Data Organization. Different memory allocation libraries differ in how they keep heap meta-data. We illustrate some of the attacks using the heap organization used in the GNU standard C library [11], but note that other libraries are vulnerable to similar attacks, such as the System V implementation in IRIX and Solaris operating systems [2].

The GNU standard C library keeps track of heap memory in terms of *chunks*. Figure 1 shows how each chunk stores its meta-data and data. Memory locations of a chunk are shown top to bottom, starting with lower-address locations at the top of the figure. In an allocated chunk, the 4-byte `size` field indicates the number of bytes occupied by the chunk, whereas the 1-bit `PIU` (*Previous-In-Use*) field indicates whether the chunk that is located right before the current chunk is also allocated [1]. If that previous chunk is not in use, its last memory word is used by the current chunk as a 4-byte `prev_size` field that contains the previous chunk's size. Finally, a freed chunk is placed as a node in a doubly-linked *free list* of similar-size deallocated chunks. The successor (`fd`) and predecessor (`bk`) pointers for this list are also maintained in the chunk itself. Heap meta-data attacks rely on corrupting the meta-data information (`size`, `prev_size`, `PIU`, `fd`, and/or `bk`), while heap data attacks rely on corrupting the heap data information such as function pointers.

Heap chunks can be either allocated or freed. Chunks are allocated by calls to *malloc()* or similar functions. Allocated chunks are still in use by the application, whereas, freed chunks are chunks that were allocated by the application, used, and then freed (by a call to *free()* or similar functions).

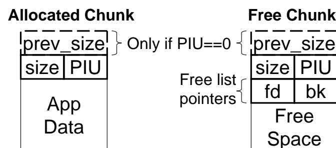

Figure 1. Heap chunk structure used in GNU C [11].

[1] Actually, since heap chunks are 8-byte aligned, the last three bits in the `size` numeric field are used to store the `PIU` field and some additional information.

Freed chunks are organized into bins of equal or similar sizes using a doubly-linked list structure. Upon an allocation request (e.g. *malloc()*), those bins are searched for a freed chunk of suitable size. If a chunk of suitable size is found, the address of the application data section of that chunk is returned to the user. If no chunk of suitable size is found, a larger chunk is divided into two chunks, one to be used to serve the allocation request and the other is considered a new freed chunk. If the allocation request cannot be served using freed chunks, it is served from the *top* of the heap, the chunk bordering the end of available heap memory.

On a call to *free()*, or similar functions, the chunk is first consolidated with neighboring chunks if they are also free and of suitable sizes. [2] Then the consolidated chunk is placed into a suitable bin based on its size. Finally, when a freed chunk is removed from a bin (either to be allocated or consolidated), its predecessor and successor in the doubly-linked list bin are linked using their *fd* and *bk* pointers, respectively. This can be done, for example, using code similar to the one below:

```
/* Assuming the chunk to be removed is P */
   P->bk->fd = P->fd
   P->fd->bk = P->bk
```

We will now describe various ways of attacking the heap meta-data and heap data, assuming that the attack relies on heap buffer overflow exploit, although other exploits are possible. For illustration purposes we use a *control-flow attack*, but note that *non-control attacks* can be performed similarly.

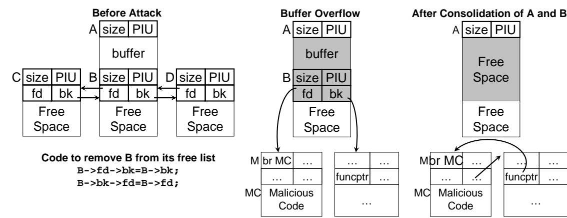

Figure 2. Steps of a forward consolidation attack.

Forward consolidation attack on heap meta-data. Figure 2 shows the steps of a forward consolidation attack. Again, the assumption is that chunk A is allocated and contains a buffer. Chunk B, on the other hand, is free and part of a free list in which the next and previous chunks are C and D. The attack starts by overflowing chunk A's buffer to overwrite B's `fd` pointer with a *data value*, such as the address of a *malicious code* M. Also, B's `bk` pointer is overwritten to point to a *target data location* minus a constant displacement equal to the difference between a chunk's address and the address of its `fd` pointer. The attack is now dormant until chunk A is deallocated or until B is allocated again. The memory management library tries to avoid fragmentation of free heap space by merging consecutive free chunks into a larger one. As a result, deallocation of chunk A results in merging A with B. To do this, B is removed from its free list, merged with A, and the resulting chunk is inserted into another free list. Similarly, chunk B may be removed from its free list if it is allocated again. Removal of B from its free list is accomplished by copying B's `fd` pointer into the `fd` of the chunk pointed to by B's `bk` pointer as described earlier. This results in copying the malicious code address M into the target memory location which can be, for example, a return address on the stack or a function pointer. When this return address is used to return from a function or the function pointer is used to call a function, the attacker's malicious code M is executed. The malicious code M could

[2] Chunks smaller than a certain threshold are considered *fast chunks* and are not consolidated normally.

have been injected in chunk A's buffer or anywhere else. Note also that the attacker may not insert malicious code at all, but use the attack to redirect the execution flow as desired.

Backward consolidation attack on heap meta-data. Figure 3 shows a backward consolidation attack where the meta-data of an allocated chunk B is overwritten by overflowing a buffer in chunk A. The attack creates a fake heap chunk record in the buffer and overwrites B's `PIU` field to falsely indicate that the previous chunk is free. The fake `size*` of the previous chunk enables the attacker to control at which address the fake chunk starts. The fake pointers `fd*` and `bk*` are in the buffer controlled by the attacker, and point to malicious code and to a target location minus displacement. When chunk B is deallocated, the memory management library tries to consolidate B and the fake chunk in A into a single larger chunk. The first step in this consolidation is to remove the fake chunk, which result in overwriting the target location with attacker's value. Compared to forward consolidation, backward consolidation is relatively easier to perform because only four bytes in the next chunk need to be overwritten, and it relies only on allocated chunks which are often more numerous than free chunks.

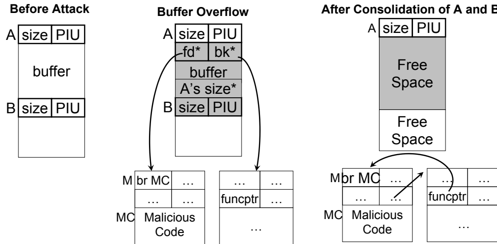

Figure 3. Steps of a backward consolidation attack. `fd*`, `bk*`, and A's `size*` are fake fields created by the attacker.

Other attacks on heap meta-data A hybrid consolidation attack combines forward and backward consolidations, and has been used by the Slapper worm [21]. Attacks to other components of heap meta-data are also feasible and have been demonstrated to be effective [1].

3.3 Economics of Heap Attacks

It is easy to see why attacks on the heap meta-data are attractive from the attackers' point of view. To the attackers, the incentive to devise an attack depends on the *cost* of attack design and deployment effort, and the *coverage* of the attack (how many cases the attack can be applied to in order to amortize the cost). Hence, the incentive is, roughly speaking, proportional to $\frac{coverage}{cost}$. Let us consider the case of heap attacks. Suppose a program consists of two components: user code and the heap library code. The heap library code is small, does not directly interact with external inputs (from the networks or I/O), and has been debugged more rigorously over many years. Thus, on its own, it has few, if any, exploitable vulnerabilities. Thus, the cost of directly attacking the heap library code is high, but since many applications use it, its coverage is also high. On the other hand, user code is typically less debugged, large, and directly interact with external inputs, and thus it has lower cost, but at the same time lower coverage. An attacker can choose to attack the user code with $incentive = \frac{low}{low}$ or the library code with $incentive = \frac{high}{high}$. However, if an attacker uses vulnerabilities in the user code to compromise and use the library code to attack any applications that rely on it, his/her incentive has increased to $\frac{high}{low}$. From this simple analysis, we note that heap meta-data definitely need stronger protection than heap data due to its higher incentive for attackers, although for comprehensive protection, both heap meta-data and heap data need to be protected.

3.4 Scope of Protection

The goal of a security attack on a single application is to alter the application's behavior such that it results in a security breach, such as elevating the privilege level of the application, execution of malicious code with or without code injection, or simply application crash. Of those, an application crash is considered much less harmful than the others since the crash does not allow the attackers to follow up with larger damages. In addition, it is very difficult to protect against crashes because any random overwrite can potentially cause a crash. Hence, we do not seek to protect against an application crash.

We note that vulnerability exploitation techniques can be directed to corrupt the stack, heap meta-data, heap data, or others. Heap Server seeks to protect the heap, so we only discuss attacks on the heap. However, we note that the security of a system is only as strong as its weakest component, so if only one of the stack or the heap is protected, the system is overall still vulnerable. For example, let us assume that the stack is protected by StackGuard [8]. StackGuard places a canary value between a return address and local variables in the stack. A stack buffer overflow that overwrites the return address also overwrites the canary value, which is checked before returning from the function to detect the attack. However, if the heap is unprotected, some heap attacks can directly overwrite a single memory location with a desired value and hence bypass StackGuard's protection by overwriting the return address directly (without overwriting the canary value), or overwriting the default canary value itself [18]. Alternatively, if the heap is protected but the stack is not, attackers would just choose to attack the stack, without bothering to break the Heap Server.

Finally, we note that read attacks (through format string vulnerabilities) need special attention because attackers may read memory contents to break address/layout obfuscation in a particular instance of an application.

Table 1. Attacks on the heap and scope of Heap Server's protection: full vs. probabilistic protection. N/A = Not Applicable.

Type	Attacks	Meta-Data	Data
Vulnerability Exploit	Buffer overflow	Full	Prob
	Integer overflow	Full	Prob
	Format string	Full	Prob
Bug Exploit	Dangling pointers	Prob	Prob
	Invalid free	Full	Full
	Double free	Full	N/A
Activation Stage	Fwd Consol.	Full	N/A
	Bkwd Consol.	Full	N/A
	Other meta-data	Full	N/A

Table 1 shows attacks on the heap and scope of Heap Server's protection, whether it is full or probabilistic protection. We note that any exploits that try to overwrite parts of the heap meta-data are completely avoided because the heap meta-data is separately protected in the Heap Server's address space. These include all the vulnerability exploits and all activation stage attacks. Protection from data overwrites is probabilistic, through layout obfuscation.

Because of Heap Server's protection, bug exploits can no longer be used to directly overwrite heap meta-data. In addition, unlike traditional heap management libraries, the heap meta-data organization in Heap Server provides an easy mechanism to double check the validity of a deallocation request. For example, invalid free (deallocation to an invalid location) and double free (deallocation to an already-deallocated location) can be easily found by the Heap Server because, in order to service a request, it first reads and verifies the validity of heap meta-data corresponding to the location in question. An invalid free will be detected as a deallocation

not to the start of a heap chunk or to a location outside the heap range. A double free will be detected as a deallocation to a chunk whose heap meta-data indicates that the chunk is already deallocated. Attempts to deallocate non-heap data (in order to overwrite it) will also be caught because the Heap Server will not find a valid meta-data record corresponding to the requested location.

Only dangling pointer exploit (deallocation of a still-live object) may not be fully protected against. A deallocation request to a valid and live heap object is always accepted by the Heap Server. However, unlike in traditional libraries which overwrite parts of the object with heap meta-data causing data corruption, Heap Server never corrupts data when it updates the meta-data as a result of the deallocation request. In the future, if the object if finally deallocated, Heap Server will detect it as a double free attempt. However, if the deallocated chunk is recycled before the actual deallocation occurs, two heap objects may incorrectly share a single object space. This may likely lead to crash, but meaningful exploitation by attackers is unlikely, because layout obfuscation prevents attackers from knowing which two objects are sharing that space.

4. Heap Server Design and Implementation

Heap Server is a separate process that performs heap management on behalf of an application. Heap Server's protection comes from three mechanisms. First, it uses a new bitmapped heap meta-data organization (Section 4.1) that stores heap meta-data separately from heap data, but still allows fast meta-data lookups with low storage overhead. Secondly, the heap meta-data is moved from the application's address space to the Heap Server's address space (Section 4.2). The application communicates its allocation and deallocation requests through inter-process messaging to the Heap Server, which manages the heap meta-data for the application. Finally, the Heap Server obfuscates the heap layout to make it difficult for the attackers to attack heap data which resides in the application's address space (Section 4.3).

4.1 Heap Meta-Data Organization

The efficiency of heap management implementation depends on how the heap meta-data is stored and looked up. Specifically, it should (1) use little extra storage, (2) allow efficient lookup of the meta-data of a chunk for deallocation purpose, (3) allow efficient lookup of the meta-data of a chunk's neighbors for consolidation, and (4) allow efficient lookup of a free chunk to reuse on an allocation request.

If we expect the average heap object size to be large, we can keep a heap object's meta-data as a fixed-size node, and nodes can be organized in a binary search tree or a hash table. To lookup the meta-data of a chunk, we can use the chunk's address to search the tree or index the hash table. However, at any given time the number of allocated chunks N can be large (sometimes in the millions), there may also be many allocations/deallocations per unit time, and the average chunk size may be very small (tens of bytes). In such cases, a tree search and rebalance would incur $\Theta(log_2 N)$ time on every allocation or deallocation request, which is clearly not acceptable in terms of performance. A search on an m-entry hash table would take $\Theta(1 + \frac{N}{m})$ which is expensive unless m is large (i.e. the hash table is large).

As a result, we choose a *bit-mapped* implementation [3]. In our implementation, every eight bytes of heap data are associated with 2 bits of meta-data, for a 3.125% space overhead relative to the heap data space. This bit-mapped organization is especially storage-efficient when chunks are small, simple indexing ($\Theta(1)$ time) finds the meta-data for a chunk, and meta-data of a chunk's neighbors can be found in contiguous locations.

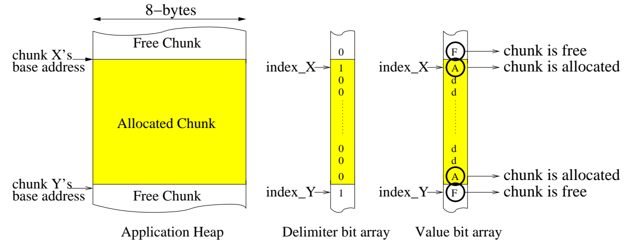

Figure 4. Bit-mapped meta-data information that is kept by the Heap Server.

Because correct alignment of allocated blocks requires every chunk to begin at a multiple of eight bytes (double-word), we associate each double-word of the application with one bit in a *delimiter bit array* and another bit in a *value bit array*, as illustrated in Figure 4. The delimiter bit is set to '1' if it corresponds to the first double-word of a chunk, otherwise it is set to '0'. Such encoding implicitly stores the size of the chunk, which can be computed by multiplying eight (bytes/bit) with the distance (in number of bits) between two consecutive delimiter bits with values of '1'. The bits in the value bit array have two uses. The first and the last bits for a chunk indicate whether the chunk is free ('F') or allocated ('A'). Other value bits for the chunk are ignored ('d' stands for "don't care") unless there are enough of them to store the chunk's size directly. Figure 4 shows an example meta-data information for chunk X. The first delimiter bit is '1', indicating the start of chunk X, while the value bit is 'A' indicates that X is an allocated chunk. All other delimiter bits for X are zero. The start of the next chunk (Chunk Y) is indicated by a delimiter bit value of '1'. Chunks that are before and after X are free as indicated by their 'F' value bits.

Given a chunk's address, its meta-data is located by simple indexing to the delimiter and value bit arrays, and the bits corresponding to the chunk are extracted through bit masking. To compute the sizes of a chunk and its neighbors, delimiter bits are fetched as 32-bit words, and the locations of '1's are found using bit scan forward (`bsfl`) and bit scan reverse (`bsrl`) x86 instructions. For chunks whose value bits span over more than two 32-bit words, the chunk's size is stored and read directly from the value bit array.

To facilitate fast re-allocation of deallocated chunks, we also maintain free-list structures similar to traditional C heap library implementations [11]. However, instead of keeping the `prev_size` and `size` fields, we only keep the addresses of free chunks and use them to access the bit-arrays where the rest of the meta-data can be found. We additionally reduce storage overhead of the free lists by using singly-linked lists, in which each node only stores the chunk's address and the list's forward pointer.

On an allocation or deallocation request, the heap meta data bitmap from the requested location is read and checked for validity, e.g. that a deallocation is to a valid chunk that is currently allocated. An invalid free is detected as a deallocation to a non-valid chunk: the address is out of range, no bits exist for that address, or the delimiter and value bits for the supposed start of the chunk are not '1A'. A double free is detected as a deallocation to a free chunk: the delimiter and value bits are already '1F', except if the chunk is already consolidated in which case the Heap Server detects it as a deallocation to a non-valid chunk.

Compared to traditional interleaved meta-data implementation, Heap Server incurs an extra 3.125% overhead to keep the delimiter and value bit arrays. Our experiments found that the performance overhead due to our bitmapped organization is negligible.

[3] Bit-mapped meta-data organization is standard in garbage collection techniques [15] for tracking an object's generation and accesses.

In some cases, the organization improves performance because the fragmentation due to heap meta-data and data storage interleaving is removed, resulting in improved spatial locality for heap data.

4.2 Heap Server Communication

The heap meta-data organization alone can already separate the storage of heap meta-data and heap data and improve security. However, as long as the heap meta-data and data are located in the same protection domain (i.e., same address space), it is still vulnerable to targeted attacks. Consequently, we place the heap meta-data in the Heap Server's address space, and provide a communication protocol between the application and Heap Server.

4.2.1 Modes of Operation and Optimizations

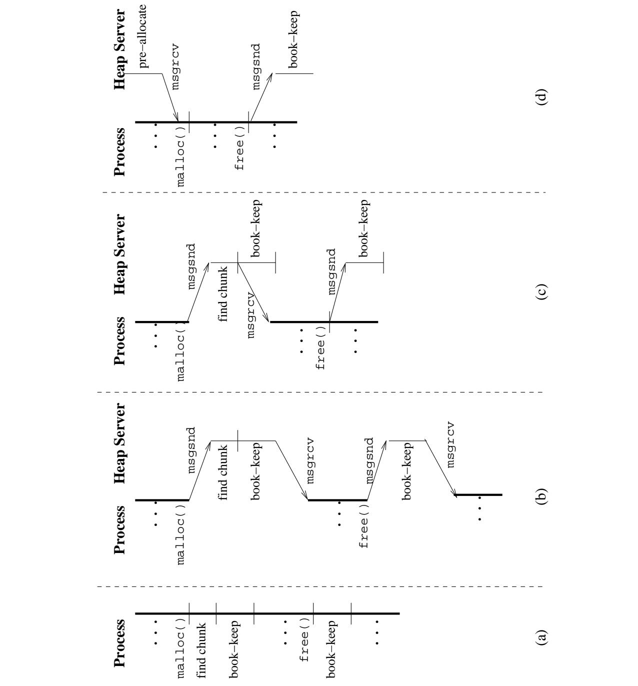

Figure 5. Traditional heap management (a) and Heap Server using blocking communication (b), non-blocking communication (c), non-blocking communication pre-allocation (d).

Figure 5a shows the timing of memory allocation and deallocation with a traditional implementation of heap management, where allocations and deallocations are executed as part of the running application and are blocking. On an allocation request, a free chunk of a suitable size is found, book-keeping is performed, and the application's regular execution resumes. On a deallocation request, book-keeping is performed and the library then returns to the application's code.

Heap Server is a separate process that is forked by the application when it starts. Although it is possible to run Heap Server as a daemon process which runs all the time and serves multiple different application processes, it has quite different implementation issues and is beyond the scope of this work. Figure 5b shows the "base" unoptimized Heap Server, while the remaining parts (c and d) show different Heap Server modes of operation, which represent different levels of optimization. The base Heap Server implementation (Figure 5b) operates similarly to a traditional heap management implementation, but uses standard System V `msgsnd()` and `msgrcv()` primitives [17] to pass heap management operations to the Heap Server process. This fully blocking implementation of Heap Server introduces a significant overhead due to the high inter-process communication latency.

Non-Blocking Communication Optimization. Some of the high inter-process communication latency can be hidden by making Heap Server requests *non-blocking* (Figure 5c). In this mode of operation, deallocation is completely non-blocking and the application continues execution as soon as a deallocation request is sent to Heap Server. Allocation requests still block the application, but Heap Server sends its response as soon as a suitable chunk is found and performs book-keeping in the background. In this way, part of the communication latency for deallocations and book-keeping latency for both allocations and deallocations are hidden from the application and are done in parallel with the application's execution. However, frequent deallocation requests can occupy the Heap Server and delay processing of allocation requests, for which the application is waiting.

Bulk Deallocation Optimization. To avoid delaying allocation requests due to high Heap Server occupancy, our *bulk deallocation* optimization groups multiple deallocation requests into a single request. The application temporarily stores each deallocation request locally, and when a limit is reached ($BULK_DEALLOC_PTRS$ = 64 requests in our implementation), a new bulk deallocation request is created and sent to the Heap Server. Upon receiving the bulk request, the Heap Server handles each deallocation sequentially. Although handling each deallocation in a bulk deallocation request takes just as much time as handling each deallocation request in the unoptimized case, the Heap Server overhead is reduced because it spends a lot less time fetching messages from the communication queue (a single message as opposed to $BULK_DEALLOC_PTRS$ messages). Note that postponing the handling of a deallocation request does not affect correctness, although it may lead to a bounded increase of the memory footprint. Finally, since bulk deallocation targets applications with a high deallocation frequency, its use is triggered only after a certain number of deallocations are performed ($BULK_DEALLOC_THRESH$ = 1024 in our implementation).

Pre-allocation Optimization. Non-blocking and bulk deallocation optimizations do not tackle the high inter-process communication latency suffered by allocation requests, which is especially a problem in applications with frequent allocations. Fortunately, we observe that, in such applications, such frequent requests are typically caused by repeated allocations for only a few different types of small data structures. We exploit this observation by pre-allocating several chunks of those sizes in anticipation of future allocations (Figure 5d). Pre-allocation is triggered when the total number of allocation requests exceeds a certain threshold PRE_ALLOC_THRESH = 1024, indicating that the application probably has a high allocation frequency. Once pre-allocation is triggered, the application sends a pre-allocation request to Heap Server, and Heap Server responds by returning pointers to PRE_ALLOC_PTRS = 512 chunks of the specified size to the application. The heap library takes the addresses of the chunks and places them in an array of pre-allocated chunks. On an allocation request, the library first checks whether the requested size is already pre-allocated. If so, it retrieves a chunk from the pre-allocation array without communicating with the Heap Server. When all the pre-allocated chunks for a certain size are consumed and there is a new request for that size, a new pre-allocation request is sent to the Heap Server. Moreover, Heap Server attempts to hide the allocation time, in addition to the communication time, by pre-allocating new PRE_ALLOC_PTRS chunks as soon as it replies to a pre-allocation request in anticipation of the next pre-allocation request. This way, at any given time, there are $2 \times PRE_ALLOC_PTRS$ pre-allocated chunks of every common

size in the system, half of them at the application's side, and the other half at the Heap Server's side.

Pre-allocation mispredictions are largely inconsequential. Unused pre-allocated chunks may result in memory fragmentation, but large fragmentation is avoided by only using pre-allocation for small chunk sizes (less than 512 Bytes). Overall, pre-allocation optimization hides the communication and allocation overhead for most frequent allocations and, together with non-blocking and bulk deallocation optimizations, allows the Heap Server and application execution to proceed almost fully in parallel.

4.2.2 Communication Protocol

Communication between the application and its Heap Server process uses standard System V message-passing. The application sets up two message queues: a *Request Queue* for sending heap requests to the Heap Server, and a *Reply Queue* for receiving Heap Server's replies. A message `M_small` has three fields: `type` identifies the type of the message, `mem_ptr` is a pointer to a memory location, and integer `value` contains additional information for some requests.

Table 2 lists the request and reply message types and their associated `mem_ptr` and `value` contents. When the application process performs its first heap allocation, it uses an `sbrk` system call with a zero argument to request the starting address of its heap memory space from the operating system, which returns a pointer to the base address of the heap memory. Then the application creates the message queues and forks a Heap Server process. The Heap Server then initializes its meta-data structures using the application's heap base address, and connects to the request and reply message queues.

Due to space limitation, we will only describe communication protocol for `malloc` request. For a `malloc` library call, a `MALLOC` request is sent to Heap Server. The Server replies with a `MPTR` message that contains the address of the newly allocated chunk. If Heap Server cannot satisfy an allocation request because the application's current heap region is too small, it requests additional heap memory by sending a positive value in the `sbrk_size` field of the `MPTR` reply. The application then extends its heap region by the requested size through an `sbrk` call. We note that Heap Server cannot directly use `sbrk` on behalf of the application because it runs in a separate address space. The `sbrk_size` value may also be a negative value indicating that the application must trim its heap region through an `sbrk` call. This ensures that the application's memory footprint is kept to a minimum.

When the application process completes execution, it can send a `DONE` message to the Heap Server, which deallocates the process' heap meta-data and terminates execution.

4.2.3 Security Considerations

One potential vulnerability of the standard messaging system in System V implementation is that the message queues are global and any process that belongs to the same user ID (UID) can listen to them. Although it is unlikely that remote attackers have an access to such a process, a combination of attackers' effort and the local users' carelessness in handling the message queue IDs can break the heap server protection. To avoid that, we could add process id authentication to the messaging system. A message queue contains the list of process ids that can read from or write to the queue, and only the owner of the queue (the application) can modify the list. Furthermore, the queue can only be closed by the owner, or by the OS if it detects that the queue has been orphaned, i.e. the owner process has died but the queue and Heap Server are still around.

A second concern specific to Heap Server is related to the bulk deallocation and pre-allocation optimizations, where the application maintains bulk deallocation requests and pre-allocated pointer list in its address space. These structures can be considered as a new type of meta-data that may be targeted by attacks. Consequently, we protect them rigorously. First, because the amount of new meta-data is small and bounded in size, we can duplicate it by writing N identical copies at different random locations. Before meta-data is used, all copies are fetched, compared, and an attack is detected if not all copies are identical. Note that keeping separate identical copies of meta-data cannot be easily or cheaply done if the application maintains *all* of its heap meta-data in its address space because of the sheer amount of meta-data involved. Our current Heap Server implementation includes *two separate copies* of bulk deallocation requests and pre-allocated pointers at the application's side. Secondly, we delimit both meta-data copies with write-protected pages so that a contiguous meta-data overwrite attempt will be instantly detected. Finally, the meta-data in the pre-allocation array is not the only copy in the system, since Heap Server maintains a redundant meta-data information. Thus, Heap Server has some ability to detect anomalies. For example, when an address of a pre-allocated chunk is corrupted, the subsequent deallocation request of the chunk will fail to correspond to the valid chunk information and be detected by the Heap Server.

4.3 Heap Layout Obfuscation

While heap meta-data is separately protected and only accessible by Heap Server, heap data needs to be protected as well. To protect heap data, Heap Server uses address obfuscation [5], which inserts random padding between heap chunks to prevent attackers from knowing exact locations of critical heap data. The padding size is between zero bytes (no padding) and the minimum of N bytes and $X\%$ of the requested chunk size (we use $N = 64$ and $X = 12.5\%$). The choice of N and X is a tradeoff between protection and fragmentation. We skip padding for very small chunks to avoid fragmentation.

In order for different program instances to have different randomization, the random seed to use can be determined at run-time by hashing together the high-resolution real-time clock, application characteristics, and system state such as the number of currently-waiting processes, total available memory, etc. This randomization does not necessarily restrict debuggability of the program since the heap meta-data state maintained by the Heap Server can itself be used as valuable debugging information.

However, address obfuscation alone still leaves the system too vulnerable. First, although padding between chunks is random, allocation and deallocation sequences are not random. This means that different instances of a program, given the same input, will have the same sequential layout of chunks in memory, albeit with different amount of paddings between chunks. For example, if in one instance of a program chunk A and B are consecutive, then in another instance they will also be consecutive. We refer to this as *layout predictability*. Layout predictability allows a brute-force guessing of padding amount, in which an attack that works on one instance of a program can also work on another as long as the padding amount is correctly guessed. We already note in Section 2 that padding-based obfuscation techniques have low entropy, so a small number of guesses are needed to successfully break it. Another factor contributing to layout predictability is the way current chunk recycling algorithms work. To optimize for temporal locality, in many libraries a deallocated chunk is often immediately recycled (re-allocated) when there is an allocation request to the same size. Such temporal locality optimization creates a predictable pattern that allows attackers to guess which chunks are likely to be consecutive in the heap, and then use the previous chunk to overflow into the function pointer in the next chunk.

In order to remove layout predictability and predictability in the recycling pattern, our proposed *layout obfuscation* randomizes chunk recycling patterns by selecting a random chunk to reallocate from the free list. To implement the random recycling, we wait until a free list has at least four chunks before allowing a chunk from the list to be recycled for allocation (this requirement is relaxed for

Table 2. Types of request and reply messages. * indicates that it is only generated for blocking communication.

Request	Request Message			Reply Message		
	type	mem_ptr	value	type	mem_ptr	value
malloc()	MALLOC	N/A	request size	MPTR	new chunk ptr	sbrk_size
calloc()	CALLOC	N/A	request size	MPTR	new chunk ptr	sbrk_size
realloc()	REALLOC	old chunk	new size	MPTR	new chunk ptr	sbrk_size
free()	FREE	memory region to be freed	N/A	ACK*	N/A	N/A
	BULK_DEALLOCATE	N/A	N/A	N/A	N/A	N/A
	PRE_ALLOCATE	N/A	request size	new message with the pre-allocated chunks		
Terminate	DONE	N/A	N/A	ACK	N/A	N/A

free lists that manage huge chunks since there are very few of such chunks). On an allocation request, we generate a random number and traverse the list by that number of nodes, and select the node at the end of the traversal for recycling. To reduce the overhead of traversal, the maximum nodes traversed is set to 16. Through random recycling, consecutiveness of chunks in one instance of a program do not imply consecutiveness of the same chunks in another instance.

5. Evaluation Methodology

Machine configuration. We evaluate our protection scheme on a bus-based symmetric multiprocessor (SMP) with two 2GHz Intel Xeon processors. Each processor has two thread contexts, a small L1 data cache, a small L1 instruction trace cache, and a unified 512KB L2 cache. The memory controller is part of the Intel 860 Chipset and the main memory is 512MB of Rambus RDRAM. The operating system on the machine is Red Hat Linux 8.0, kernel version 2.4.20. The machine is run under a relatively light load, where normal Linux OS processes and daemons run, but we do not run major applications except the application and the Heap Server.

Benchmarks. To evaluate the Heap Server, we use all 16 C/C++ benchmarks from the SPEC CPU 2000 benchmark suite [27] with reference input sets: ammp, art, bzip2, crafty, eon, equake, gap, gcc, gzip, mcf, mesa, parser, perlbmk, twolf, vortex, and vpr. In order to stress the performance of our scheme, we use eight additional C/C++ allocation-intensive benchmarks: boxed, cfrac, deltaBlue, espresso, lindsay, LRUsim, richards, and roboop. These benchmarks are widely used for testing heap management implementations due to their high allocation/deallocation rates [4, 10].

Each experiment is run ten times [4] and the wall-clock times are averaged when reporting them to reduce measurement noise. The benchmarks and the Heap Server are compiled with `gcc` version 3.2 with a `-O3` optimization level. All benchmarks are run from start to completion without skipping or sampling. Table 3 lists all the benchmarks, their sources, programming languages, inputs, and run times in seconds.

Heap Management Library. For ease of statistics collection and timing, we develop a base heap management library that faithfully implements all major features of Doug Lea's heap management library v.2.7.2 [11], which is a very popular heap management library used in GNU C and in other systems. We compared the performance of our library versus GNU C library and verified that both always perform within 1% of another for all the applications we use in this paper.

Attacks. To evaluate our scheme's security protection ability, we obtained two real-world exploits that perform heap attacks: (1) Wu-Ftpd File Globbing Heap Corruption Vulnerability [24] against Washington University's FTP daemon (ftpd), and (2) Sudo Password Prompt Heap Overflow Vulnerability [25] against the Linux/Unix sudo utility. Because these attacks were not specifically designed to break Heap Server, we also inject our own heap attacks based on buffer overflow vulnerability exploit [2]: backward consolidation attack (*backon*), forward consolidation attack (*forcon*), and data overwrite that targets a function pointer in heap chunks (*funptr*). In all the injected attacks, we vary the number of words that are overflowed from 1 word to the maximum padding amount.

[4] except for roboop and deltaBlue which are run for 20 times because of their very short run times.

Table 3. The 24 benchmarks, their sources, languages, inputs, and run times in seconds.

Benchmark	Source	Lang.	Input	Time(s)
ammp	SpecFP2000	C	ref	644
art	SpecFP2000	C	ref	465
bzip2	SpecINT2000	C	ref (input.source)	90
crafty	SpecINT2000	C	ref	158
eon	SpecINT2000	C++	ref (cook)	101
equake	SpecFP2000	C	ref	180
gap	SpecINT2000	C	ref	160
gcc	SpecINT2000	C	ref (200.s)	59
gzip	SpecINT2000	C	ref (input.source)	45
mcf	SpecINT2000	C	ref	319
mesa	SpecFP2000	C	ref	274
parser	SpecINT2000	C	ref	318
perlbmk	SpecINT2000	C	ref (splitmail)	62
twolf	SpecINT2000	C	ref	597
vortex	SpecINT2000	C	lendian2	87
vpr	SpecINT2000	C	ref (place)	172
boxed	Heap Layers	C	-n 50 -s 1	85
cfrac	Heap Layers	C	a 40 digit number	34
deltaBlue	Other	C++	100000	4
espresso	Heap Layers	C	largest.espresso	251
lindsay	Heap Layers	C++	script.mine	92
LRUsim	Heap Layers	C++	20,000,000 accesses	48
richards	Other	C++	100000	447
roboop	Heap Layers	C++	bench	4

6. Heap Server Evaluation

6.1 Attack Avoidance

We tested two real-world attacks WU-ftpd and Sudo on GNU C library and confirm that control flow is hijacked as a result. The two attacks attempt to overwrite heap meta-data through buffer overflows. We then test them on Heap Server, and they complete execution without suffering any effect from the attacks. This is because heap meta-data is no longer in the application's address space, so the attacks do not corrupt it. Furthermore, the buffer overflow amount is small and falls in the padding area inserted through layout obfuscation.

To test the Heap Server further, we inject attacks (forcon, backcon, and funptr) to both GNU C library and to our Heap Server. The application is a simple linked list program in which each node has a function pointer and a large vulnerable buffer. To attack the application, attackers only need to overflow one buffer into the following chunk. All the attacks successfully hijack the control flow with GNU C library. For Heap Server, we increase the buffer overflow size up to 128 bytes (the maximum padding size) to specifically break our layout obfuscation scheme. All attempts of forward and backward consolidation fail to hijack the control flow. However,

the impact on the application varies with the overflow amount. If the overflow is entirely within the chunk's padding, the attack has no effect on the application. If the overflow overwrites heap data in the next chunk, the application either crashes or produces wrong computation results.

Function pointer overwrite attempts, however, sometimes succeed in hijacking the control flow when the actual padding amount is correctly guessed. When the overflow is larger than the padding amount, the overflow also corrupts data in the next chunk. Since our attack kernels are applied against a very regular application which only has heap objects of the same size and each object has both a function pointer and a vulnerable buffer, any overflow into the next chunk results in a successful control flow hijack. Real applications have fewer vulnerable buffers and function pointers in the heap and are much less prone to data overwrite attacks.

6.2 Benchmark Characteristics

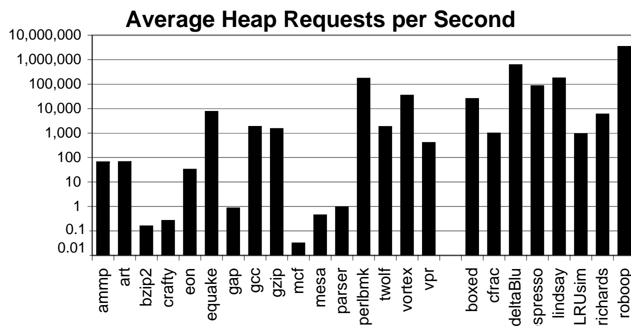

Figure 6. Average heap requests per second (logarithmic scale)

Heap request frequency. Figure 6 shows the heap request frequency, measured as the total number of allocation and deallocation requests per second, for each application. SPEC2000 benchmarks are shown on the left, while the allocation-intensive benchmarks are shown on the right of the figure. The figure shows that the benchmarks have a wide range of heap requests frequency: high – more than 2,000 up to 3,367,281 (deltaBlue, equake, espresso, lindsay, perlbmk, richards, roboop, and vortex), medium/low – the rest of the benchmarks. We pay particular attention to benchmarks with high heap request frequencies, because they stress the performance of our heap protection mechanisms the most. Most of the allocation-intensive benchmarks except cfrac and LRUsim are in this category. Note that roboop, a C++ object-oriented robotic manipulator simulation, has at least one order of magnitude higher heap request frequency compared to all other benchmarks. On average, it makes one heap request every 594 processor cycles, and does very little computation beyond making heap requests. Hence, roboop represents a good worst-case scenario to test our schemes.

Heap request types and sizes. Due to the space limitations, we only provide a summary of our data on heap request type and size. We found that in all benchmarks with high heap request frequencies, deallocation requests account for roughly half of all heap requests. This makes sense because allocations are frequent, and heap memory would grow quickly without frequent deallocations. As a result, these benchmarks are expected to benefit from our deallocation optimizations (bulk deallocations and non-blocking communication) as well as from allocation optimizations (pre-allocations and non-blocking communication). We also found that all benchmarks with high heap request frequencies have a small average heap request size, ranging from 2 bytes for lindsey to 235 bytes for deltaBlue. This makes sense because large objects would take more time to initialize and use, preventing the application from making heap requests as frequently. The inverse, however, is not true: some applications with low heap request frequencies also have small request sizes. Both heap request types and sizes for benchmarks with high heap request rates support our choice of bit-mapped heap meta-data organization because they are efficient in storage when the average heap chunk size is small, and allow fast lookups for these demanding benchmarks.

6.3 Heap Server's Performance

Execution time overheads. Figure 7 shows the execution time overheads of Heap Server when some or all of its components are implemented, compared to the execution time of the base (no protections) library. The *Bitmap* bars show the overheads when the bitmapped heap meta-data organization is used, but is stored in the application's address space and no obfuscation is used. The *Bitmap+Obfus* bars show when layout obfuscation is added. Finally, the *FullHS* bars show a full Heap Server implementation that includes the bitmapped heap meta-data organization, layout obfuscations, keeping meta-data in the Heap Server process, and using all Heap Server communication optimizations. The figure shows that our bit-mapped meta-data organization (Section 4.1) has nearly negligible performance overheads of -0.5% and 2.5% on average for SPEC2000 and allocation-intensive benchmarks, respectively. In some cases, it even speeds up execution (9% in equake and 4% in twolf), because the more compact heap data layout produces better spatial locality. In other cases, it slows down the execution: 6% in ammp, deltaBlue, and espresso, 9% in roboop, and 2% in lindsay and eon. This is due to the extra meta-data lookup time and due to the storage overhead to store the meta-data. The *Bitmap+Obfus* bars indicate that our layout obfuscation leaves average performance largely unchanged. However, the fragmentation in the heap data space due to random padding and longer traversal for heap object recycling penalize roboop by 29%. The *FullHS* bars show that a fully-optimized full implementation of Heap Server, on average, produces -0.4% overhead for SPEC2000 benchmarks and 2% overheads for the allocation-intensive benchmarks. The full implementation even speeds up seven benchmarks (bzip2, crafty, equake, gap, twolf, deltaBlue, and LRUsim), and in three benchmarks the speedups are significant: 14% in deltaBlue, 10% in equake, and 7% in twolf. *FullHS* implements everything in *Bitmap+Obfus* and also suffers from the high inter-process communication latencies. If it were unoptimized, its execution time overheads would be strictly higher. The fact that *FullHS* performs better than *Bitmap+Obfus* in most cases signifies that the parallelism between the application and the optimized Heap Server more than offsets the inter-process communication latencies.

Execution time overheads of an alternative to Heap Server. One may imagine an alternative to the Heap Server in which we use bitmapped heap meta-data to separate the storage of heap meta-data from heap data, layout obfuscation to protect heap data, but add page-level write-protection to the heap meta-data. The heap management library unprotects the pages that hold the heap meta-data when it needs to modify them, but immediately write-protects the pages before returning to the user code. To implement this, we simply wrap the heap management library functions with `mprotect` system calls to unprotect and protect the heap meta-data pages. Figure 8 shows the resulting performance overheads, which are huge for benchmarks with high heap request frequency: the slowdown is more than 20X in deltaBlue, espresso, perlbmk, and roboop. We investigated this further by counting the number of `mprotect` calls, and found out that on average, one such system call introduces thousands of cycles of overhead due to an exception, pipeline flush, context switch latency, TLB flush, and some cache flushes. Even if the system calls are highly optimized (e.g. through fast system calls), it is unlikely that these overheads can be reduced to the level of our Heap Server implementation.

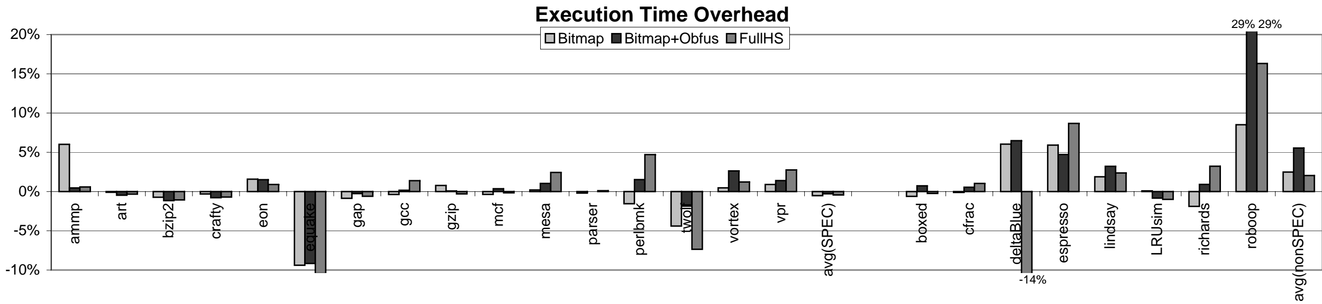

Figure 7. Execution time overheads of Heap Server with some or all its components implemented.

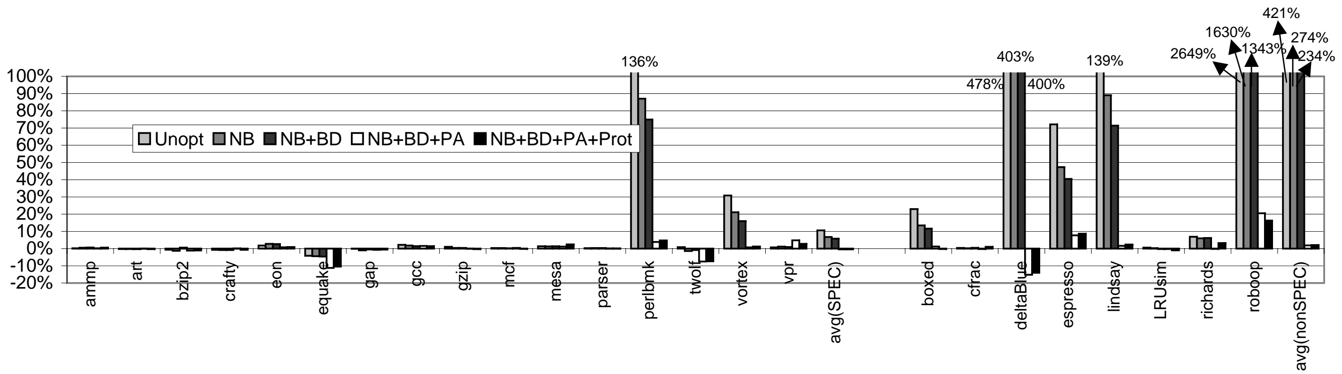

Figure 9. Execution time overhead of Heap Server with different optimizations.

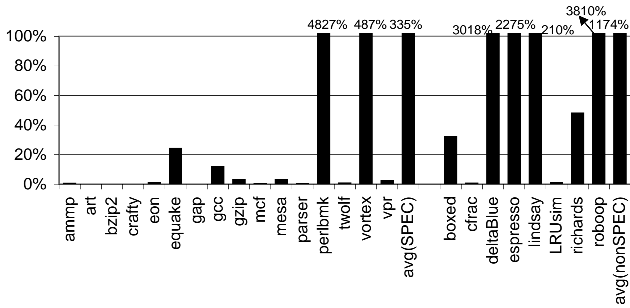

Figure 8. Execution time overhead using kernel-level page protection.

Impact of various optimizations. Figure 9 shows the execution time overheads for Heap Server with various optimizations, relative to the base (no-protection) heap library. *Unopt* implements the Heap Server with layout obfuscation in the fully blocking mode, *NB* adds non-blocking communication, *NB+BD* adds bulk deallocation optimization, and *NB+BD+PA* adds pre-allocation optimization. Finally, *NB+BD+PA+Prot* adds extra protection to the bulk-deallocation and pre-allocated pointers at the application's side, by keeping two identical copies of this new meta-data, and delimiting both copies with write-protected pages (Section 4.2.3). Note that *NB+BD+PA+Prot* is equivalent to *FullHS* in Figure 7. Figure 9 demonstrates that essentially *Unopt*, *NB*, and *NB+BD* have low overheads for many benchmarks, but quickly jump to unacceptably high overheads for benchmarks with high heap request rates. When pre-allocation optimization is added (*NB+BD+PA* bars), overheads drop to almost-negligible levels, and we even observe speedups in some benchmarks. The overhead in roboop goes down from 1,343% to only 17%, while deltaBlue's 400% overhead becomes a 14% speedup. The reason for this speedup is that an allocation request is often quickly followed by initialization code that uses the block's address, and this data dependence stalls the processor significantly if allocation is not serviced quickly. This is in contrast to a deallocation request, which typically has no data dependences with the subsequent code.

Effectiveness of Bulk Deallocation and Pre-Allocation optimizations. Due to space limitations, we only summarize our data. On average, pre-allocation eliminates 49% and 97% of all allocation requests for Spec2000 and allocation-intensive benchmarks, respectively. We investigated further and found that high heap request frequencies are usually caused by a small number of data structures with a large number of same-sized nodes being allocated and deallocated frequently. Thus, there are very few allocation sizes that are repeated very frequently, leading to a pattern that is easily predictable by our pre-allocation optimization. On average, bulk deallocation eliminates 42% and 85% of all deallocation requests in Spec2000 and allocation-intensive benchmarks, respectively. The low coverage of pre-allocation and bulk deallocation in SPEC2000 benchmarks is due to the small number of allocation/deallocation requests in some benchmarks, or large allocation sizes in others. None of the low-coverage benchmarks can benefit from pre-allocation and bulk deallocation optimizations since they have low heap request frequencies.

Heap Server's processor occupancy. Heap Server incurs almost-negligible storage and execution time overheads for all the benchmarks tested. However, it occupies a thread context or a processor. We have argued in Section 4 that the trend towards Chip Multi-Processor architectures plays in favor of trade-offs between execution time and the number of processors used. However, we

are still interested in finding just how much Heap Server utilizes a processor. Figure 10 shows a breakdown of time the Heap Server process spends on servicing heap requests from the application. *waiting* indicates that the Heap Server is idle, not having a heap request to service. The figure shows that the Heap Server is busy less than 6% of the time for all but one benchmark. Even for roboop, the Heap Server is busy only 24% of the time. Hence, the extra processor utilization of Heap Server is actually very small. We leave how this observation can be exploited for future study.

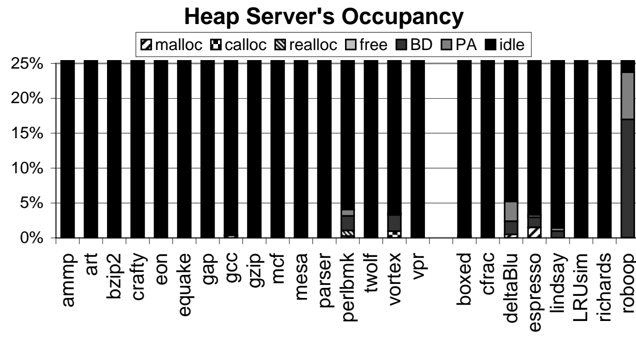

Figure 10. Heap Server's occupancy.

7. Conclusions

We have proposed and evaluated a new scheme for comprehensively protecting the heap meta-data as well as heap data from security attacks. In its full implementation, our scheme protects against contiguous and non-contiguous overwrites on heap meta-data, and makes overwrites of heap data more difficult. We show that our approach uses minimal assumptions on the mechanisms of the latter stages of an attack, utilizes existing hardware protection mechanisms, and requires modifications only to the allocation/deallocation routines.

We demonstrate that an alternative heap meta-data protection through existing kernel-level page protection produces unacceptable performance overheads. In contrast, our schemes achieves nearly-negligible performance overheads for applications with a wide-range of heap behavior. We achieve such low performance overheads using aggressive optimizations and by exploiting parallelism between the application and its Heap Server. Since many techniques have been proposed for stack protection but few for heap protection, we believe that the heap protection offered by our scheme is a significant contribution towards the overall security of an application.

References

[1] Alexander Anisimov, Positive Technologies. Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass. *http://www.maxpatrol.com/defeating-xpsp2-heap-protection.htm*, 2005.

[2] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), 2001.

[3] E. Berger and B. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2006.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *in Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, 2000.

[5] S. Bhatkar, D. C.DuVarney, and R.Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. *in Proc. of the 12th USENIX Security Symp.*, pages 105–120, 2003.

[6] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. *in Proc. of the 14th USENIX Security Symp.*, pages 177–192, 2005.

[7] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. *in Proc. of the 12th USENIX Security Symp.*, pages 91–104, 2003.

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *in Proc. of the 7th USENIX Security Symp.*, pages 63–78, 1998.

[9] Darkeagle. Mozzila GIF Image Processing Library Remote Heap Overflow Vulnerability. *http://www.securityfocus.com/bid/12881/exploit*, 2005.

[10] D. L. Detlefs, A. Dosser, and B. Zorn. Memory Allocation Costs in Large C and C++ Programs. *Software Practice and Experience*, pages 527–542, 1994.

[11] Doug Lea. A Memory Allocator. *http://gee.cs.oswego.edu/dl/html/malloc.html*, 2000.

[12] G. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. *In Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems. Boston, MA*, 2004.

[13] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address space randomization. *In Proc. of the ACM Conf. on Computer and Communications Security*, 2004.

[14] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. *To appear in Proc. of the 37th Intl. Symp. on Microarchitecture. Portland, OR*, 2004.

[15] Jones, Richard, and Rafael Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. *John Wiley & Sons, New York*, 1996.

[16] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *11th USENIX Security Symp.*, 2002.

[17] Linux Programmer's Manual. Man Pages MSGOP(2). 2002.

[18] Matt Conover and w00w00 Security Team. w00w00 on Heap Overflows. *http://www.w00w00.org/files/articles/heaptut.txt*, 1999.

[19] Nathan Tuck, Brad Calder and George Varghese. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow. *Proc. of the 37th annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 209–220, 2004.

[20] PaX Team. PaX Address Space Layout Randomization (ASLR). *http://pax.grsecurity.net/docs/aslr.txt*, 2003.

[21] F. Perriot and P. Szor. An Analysis of the Slapper Worm Exploit. *http://securityresponse.symantec.com/avcenter/reference/ analysis.slapper.worm.pdf*, 2003.

[22] R. Wojtczuk. Defeating Solar Designer Non-executable Stack Patch. *http://seclists.org/lists/bugtraq, experimental study of security vulnerabilities caused by errors. In Proc. of the IEEE Intl. Conf*, 1998.

[23] S. Andersen and V. Abella. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. *http://www.microsoft.com/technet/prodtechnol/winxppro/ maintain/sp2mempr.mspx*, 2004.

[24] Security Focus. Wu-Ftpd File Globbing Heap Corruption Vulnerability. *http://www.securityfocus.com/bid/3581*, 2002.

[25] Security Focus. Sudo Password Prompt Heap Overflow Vulnerability. *http://www.securityfocus.com/bid/4593*, 2003.

[26] Security Focus. Microsoft Windows winhlp32.exe Heap Overflow Vulnerability. *http://www.securityfocus.com/archive/1/385332/2004-12-20/2004-12-26/2*, 2004.

[27] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. *http://www.spec.org/osg/cpu2000/*, 2000.

[28] US-CERT. CVS Heap Overflow Vulnerability. *www.uscert.gov/cas/techalerts/index.html*, pages TA04–147A, 2004.

[29] US-CERT. HTTP Parsing Vulnerabilities in Check Point Firewall-1. *www.uscert.gov/cas/techalerts/index.html*, pages TA04–036A, 2004.

[30] US-CERT. Microsoft Internet Explorer vulnerable to buffer overflow via FRAME and IFRAME elements. *http://www.kb.cert.org/vuls/id/842160*, page VU 842160, 2004.

[31] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent Runtime Randomization for Security. *in Proc. of the 22nd Intl. Symp. on Reliable Distributed Systems*, pages 260–269, 2003.