DieHarder: Securing the Heap

Gene Novark
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
gnovark@cs.umass.edu

Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery@cs.umass.edu

ABSTRACT

Heap-based attacks depend on a combination of memory management errors and an exploitable memory allocator. Many allocators include *ad hoc* countermeasures against particular exploits, but their effectiveness against future exploits has been uncertain.

This paper presents the first formal treatment of the impact of allocator design on security. It analyzes a range of widely-deployed memory allocators, including those used by Windows, Linux, FreeBSD, and OpenBSD, and shows that they remain vulnerable to attack. It then presents DieHarder, a new allocator whose design was guided by this analysis. DieHarder provides the highest degree of security from heap-based attacks of any practical allocator of which we are aware, while imposing modest performance overhead. In particular, the Firefox web browser runs as fast with DieHarder as with the Linux allocator.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Dynamic storage management; D.2.0 [**Software Engineering**]: Protection mechanisms

General Terms

Algorithms, Languages, Security

Keywords

buffer overflow, dangling pointer, dynamic memory allocation, memory errors

1. INTRODUCTION

Heap-based exploits are an ongoing threat. Internet-facing applications, such as servers and web browsers, remain especially vulnerable to attack. Attackers have recently developed exploits that can be triggered by viewing apparently benign objects such as PDFs and images. Even the use of memory-safe programming languages like Flash, Java, or JavaScript does not mitigate these vulnerabilities because the language implementations themselves are typically written in C or C++.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

A heap-based exploit requires both a memory management error in the targeted program and an exploitable heap implementation. Exploitable memory management errors include:

- **Heap overflows/underflows**, when heap objects are too small to hold their input, or when an index into the object can be hijacked to force an overflow.
- **Dangling pointers**, or "use-after-free" errors, when a program prematurely frees an object that is still in use.
- **Double** `free`, when an object is deleted multiple times.
- **Invalid** `free`, when a program deletes objects it never allocated (such as the middle of an array or a stack pointer).
- **Uninitialized reads**, when programs read from newly allocated objects (which generally contain data from previously-freed objects).

The key to a successful exploit is the interaction between the memory management error and the heap layout. For example, an attacker can exploit an overflow to overwrite data in an adjacent vulnerable object like a function pointer. This attack requires the ability to force the heap to place these two objects next to each other, and to overflow into that object without detection.

Fixing a particular bug prevents exploits that depend on it, but any latent memory errors leave programs vulnerable. Rather than trusting applications to be error-free, vendors have sought to harden allocators against attack.

Heap exploits have led to an arms race, where exploits are followed by countermeasures, which, in turn, are followed by new exploits that work around the countermeasures. For example, the Windows XP SP2 memory allocator added one-byte random "cookies" to the headers that precede every allocated object. The memory allocator checks the integrity of these cookies when an object is freed. However, other heap metadata was not protected, so a new attack quickly followed. This sequence of attack-countermeasure has continued from XP SP3 to Vista (see Section 5).

These *ad hoc* countermeasures have failed because it has not been possible to predict their effectiveness against new attacks. In effect, these modifications are often attempts to "fight the last war", addressing only known vulnerabilities. Their susceptibility to future attacks has remained an open question.

Contributions

This paper provides an extensive analysis of the security of existing memory allocators, including Windows, Linux, FreeBSD, and OpenBSD. It presents the first formal treatment of the impact of allocator design on security, and shows that all widely-used allocators suffer from security vulnerabilities.

It then presents the design and analysis of a new, security-focused allocator called **DieHarder**. We show that its design—comprising a combination of the best features of DieHard [8, 9] and OpenBSD's new allocator [22]—significantly reduces the exposure of programs to heap exploits. An empirical evaluation demonstrates that DieHarder provides its security gains with modest performance overhead. Across a suite of CPU-intensive benchmarks, DieHarder imposes an average 20% performance penalty versus OpenBSD, and the Firefox web browser's performance with DieHarder is effectively identical to running with the Linux allocator.

Outline

The remainder of this paper is organized as follows. Section 2 presents an overview of memory allocator algorithms and data structures, focusing on allocators in wide use. Section 3 motivates and describes our threat model. Section 4 describes heap-based attacks—abstractly and concretely—that target the weaknesses of these allocators, and Section 5 discusses the countermeasures employed to address these vulnerabilities. Section 6 describes the design of DieHarder, together with a security analysis that shows its advantages over previous allocators, and Section 7 presents empirical results on CPU-intensive benchmarks and the Firefox browser that demonstrate its modest performance overhead. Section 8 discusses related work, and Section 9 concludes.

2. OVERVIEW: MEMORY ALLOCATORS

The functions that support memory management for C and C++ (`malloc` and `free`, `new` and `delete`) are implemented in the C runtime library. Different operating systems and platforms implement these functions differently, with varying design decisions and features. In nearly all cases, the algorithms underpinning these allocators were primarily designed to provide provide rapid allocation and deallocation while maintaining low fragmentation [37], with no focus on security. We describe the allocation algorithms used by Windows, Linux, FreeBSD, and OpenBSD, focusing on implementation details with security implications. Table 1 summarizes the security characteristics of these allocators.

2.1 Freelist-based Allocators

The memory managers used by both Windows and Linux are **freelist-based**: they manage freed space on linked lists, generally organized into bins corresponding to a range of object sizes. Figure 1 illustrates an allocated object within the Lea allocator (DLmalloc). Version 2.7 of the Lea allocator forms the basis of the allocator in GNU libc [18].

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 freelists 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. Attackers have found numerous ways of exploiting this inherent weakness of freelist-based allocators, including the ability to perform arbitrary code execution (see Section 4 for attacks on freelist-based allocators, and Section 5 for countermeasures).

2.2 BiBOP-style Allocators

In contrast to Windows and Linux, FreeBSD's PHKmalloc [16] and OpenBSD's current allocator (derived from PHKmalloc) employ a heap organization known as *segregated-fits BiBOP-style*. Figure 2 provides a pictorial representation of part of such a heap. The allocator divides memory into contiguous areas that are a multiple of the system page size (typically 4K). This organization into pages gives rise to the name "Big Bag of Pages", or "BiBOP" [15]. 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.

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 [6] (see Section 4.1).

2.2.1 OpenBSD Allocator

OpenBSD originally used PHKmalloc, but recent versions of OpenBSD (since version 4.4, released in 2008) incorporate a new allocator based on PHKmalloc but heavily modified to increase security [22]. It employs the following techniques:

- **Fully-segregated metadata.** OpenBSD's allocator maintains its heap metadata in a region completely separate from the heap data itself, so overflows from application objects cannot corrupt heap metadata.

- **Sparse page layout.** The allocator allocates objects on pages provided by a randomized `mmap` which spreads pages across the address space. This sparse page layout effectively places unmapped "guard pages" between application data, limiting the exploitability of overflows.

- **Destroy-on-free.** Optionally, OpenBSD's allocator can scramble the contents of freed objects to decrease the exploitability of dangling pointer errors.

	Windows	DLMalloc 2.7	PHKmalloc	OpenBSD	DieHarder
No freelists (§ 2.1)			✓	✓	✓
No headers (§ 2.1)			✓	✓	✓
BiBOP (§ 2.2)			✓	✓	✓
Fully-segregated metadata (§ 2.2.1)				✓	✓
Destroy-on-free (§ 2.2.1)				✓?	✓
Sparse page layout (§ 2.2.1)				✓	✓
Placement entropy (bits) (§ 2.2.1)	0	0	0	4	$O(\log N)$ (§ 6)
Reuse entropy (bits) (§ 2.2.1)	0	0	0	5.4	$O(\log N)$ (§ 6)

Table 1: Allocator security properties (see the appropriate section for explanations). A check indicates the presence of a security-improving feature; a question mark indicates it is optional. While both OpenBSD's allocator and DieHarder (Section 6) employ the full range of security features, DieHarder provides higher entropy for certain key features (where N is the size of the heap) and is thus more secure.

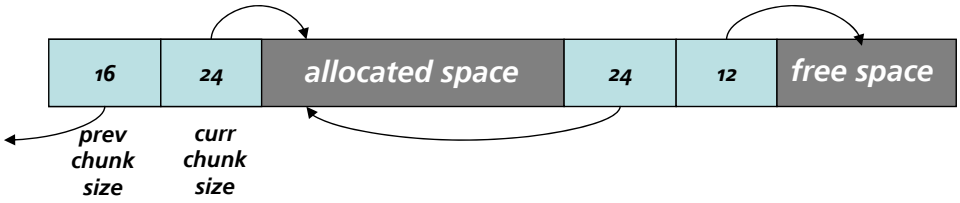

Figure 1: A fragment of a freelist-based heap, as used by Linux and Windows. Object headers precede each object, which make it easy to free and coalesce objects but allow overflows to corrupt the heap.

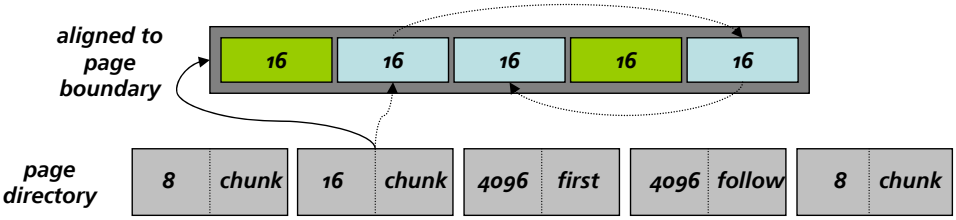

Figure 2: A fragment of a segregated-fits BiBOP-style heap, as used by the BSD allocators (PHKmalloc and OpenBSD). Memory is allocated from page-aligned chunks, and metadata (size, type of chunk) is maintained in a page directory. The dotted lines indicate the list of free objects inside the chunk.

- **Randomized placement.** Object placement within a page is randomized by a limited amount: each object is placed randomly in one of the first 16 free chunks on the page.
- **Randomized reuse.** The allocator delays reuse of freed objects using a randomly-probed delay buffer. The buffer consists of 16 entries, and on each `free`, a pointer is stored into a random index in this buffer. Any pointer already occupying that index is then actually freed.

Together, these modifications dramatically increase security, although the randomized placement and reuse algorithms are of limited value. We discuss these limitations further in Sections 4.1.3 and 4.3.1.

3. THREAT MODEL

This section characterizes the landscape for heap-based attacks and presents our threat model.

3.1 Landscape

The power of potential heap attacks is affected by several factors, including the presence of memory errors, the kind of application being attacked, and whether the attacker has the ability to launch repeated attacks.

Presence of memory errors.

The first and most important factor is the existence of a memory error, and the attacker's ability to trigger the code path leading to the error. A program with no memory errors is not vulnerable to heap-based attacks.

Application class.

The kind of application under attack affects the attacker's ability to control heap operations. Many attacks assume an unfragmented heap, where the effects of heap operations are predictable. For example, when there are no holes between existing objects, new objects will be allocated contiguously on a fresh page. Many attack strategies assume the ability to allocate enough objects to force the heap into a predictable state before launching the actual attack.

When attacking a web browser, the attacker can run scripts written in JavaScript or Flash. In most current browsers, JavaScript objects are allocated in the same heap as the internal browser data, allowing the attacker to control the state of the application heap. Sotirov describes a sophisticated technique called *Heap Feng Shui* that allows attacks on browsers running JavaScript to ensure predictable heap behavior [34].

Server applications are generally less cooperative. The number and types of allocated objects can be fixed by the application. How-

ever, an attacker may be able to place the heap into a predictable state by issuing concurrent requests, forcing the application to allocate a large number of contemporaneously live objects.

Other applications may provide attackers with no ability to cause multiple object allocations. For example, many local exploits target setuid root binaries which may run for a short time and then terminate. In many cases, the attacker is limited to controlling the command-line arguments and the resulting heap layout.

Ability to launch repeated attacks.

An application's context defines the attacker's ability to repeatedly launch attacks. In a web browser, if the first attempt fails and causes the browser to crash, the user may not attempt to reload the page. In this case, the attack has only one chance to succeed per target. On the other hand, server applications generally restart after crashes to ensure availability, providing the attacker with more opportunities. If the server assumes an attack is in progress and does not restart, then the vulnerability becomes a denial of service.

Given enough time, an attacker with any probability of success will eventually succeed. However, if the allocator can decrease this probability, the system maintainer may be able to analyze the attack and fix the application error before the attacker succeeds.

Randomization techniques such as address-space layout randomization (ASLR) are designed to provide such unpredictability. 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 [32]. 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.

While one can imagine a hypothetical supervisor program that detects incoming attacks, such a system would be hard to make practical. While it could detect a series of crashes coming from a single source, sophisticated attackers control large, distributed networks which allow them to coordinate large numbers of attack requests from different sources. Shacham et al. discuss the limitations of such systems in more detail [32].

However, more sophisticated techniques can limit the vulnerability of systems to repeated attacks. Systems such as Rx [29], Exterminator [24, 25], and ClearView [28] can detect heap errors and adapt the application to cope with them. For example, Exterminator can infer the size of an overflow and pad subsequent allocations to ensure that an overflow of the same size does not overwrite data.

The threat model.

We assume the attacker has the power to launch repeated attacks and allocate and free objects at will. Repeated attacks are most useful against Internet servers, while the unlimited ability to allocate and free objects is most useful against web browsers (especially when executing JavaScript). This model thus assumes the worst-case for prime attack targets in the real world.

We analyze vulnerabilities based on a single exploit attempt. The lower the likelihood of success of a single attack, the longer the expected time before the application is compromised. Given enough time, the error can be corrected manually, or a system like Exterminator can adapt the application to correct it.

4. ATTACKS

We now explain in detail how heap-based exploits work, and how these interact with the underlying heap implementations. Exploits often directly exploit heap-based overflows or dangling pointer errors (including double frees), but can also start with **heap spraying** attacks [14] and then later exploit a vulnerability.

We abstract out each of these attacks into an **attack model**. We illustrate these models with examples from the security literature, and show how particular memory management design decisions facilitate or complicate these attacks.

4.1 Heap Overflow Attacks

Perhaps the most common heap attack strategy exploits an overflow of an object adjacent to heap metadata or application data.

4.1.1 Overflow attack model

Abstractly, an overflow attack involves two regions of memory, one **source chunk** and one or more **target chunks**. Target chunks can include application data or heap metadata, including allocator freelist pointers. The attacker's goal is to overwrite some part of target chunk with attacker-controlled data.

A real attack's success or failure depends on application behavior. For example, an attack overwriting virtual function table pointers only succeeds if the application performs a virtual call on a corrupted object. However, details of such application behavior is outside the scope of our attack model, which focuses only on the interaction between the heap allocator and overflows. For purposes of analysis, we are pessimistic from the defender's viewpoint: **we assume that an attack succeeds whenever a target chunk is overwritten**.

Note that the attacker's ability to exploit a heap overflow depends on the specific application error, which may allow more or less restricted overflows. For example, off-by-one errors caused by failure to consider a null string termination byte allow only the overflow of 1 byte, with a specific value. In general, `strcpy`-based attacks do not allow the attacker to write null bytes. On the other hand, some errors allow overwrites of arbitrary size and content.

4.1.2 Specific attacks

An overflow attack may target either heap metadata or application data. In some cases, a single, specific heap object may be the target, such as a string containing a filename. In others, there may be many targeted chunks. For example, a potential target for application data attacks is the virtual function table pointer in the first word of C++ objects with virtual functions. In some applications, many objects on the heap have these pointers and thus are viable targets for attack. Other attacks target inline heap metadata, present in the first words of every free chunk.

Early attacks.

The earliest heap overflow attacks targeted application data such as filename buffers and function pointers [12]. A susceptible program allocates two objects, the source (overflowed) chunk and an object containing a function pointer (the target chunk). A successful attack forces the allocator to allocate the source chunk and victim chunk contiguously. It then overflows the buffer, overwriting the function pointer with an attacker-controlled address. If the chunks are not adjacent, a more general attack may overwrite multiple objects in between the buffer and the vulnerable object.

Freelist metadata attacks.

Solar Designer first described an attack relying on specifics of the heap implementation [33]. The attack applies to any allocator that embeds freelist pointers directly in freed chunks, such as DLmalloc and Windows. The specific attack described allowed a hostile web server to send a corrupt JPEG image allowing arbitrary code execution within the Netscape browser.

This attack overwrites words in the free chunk header, overwriting the freelist pointers with a specific pointer (generally to shellcode) and the address of a target location. Candidate target locations include function pointers in heap metadata structures, such as `_free_hook` in DLmalloc, which is called during each `free` operation. When the corrupted free chunk is reallocated, the allocator writes the pointer to the target location.

In the worst case for this attack, every free chunk is a target. Once a free chunk is corrupted, the attacker can simply force allocations until the chunk is reused. However, existing attacks in the literature target a single, attacker-controlled free chunk.

Other metadata attacks.

BBP describes overflow attacks targeting PHKmalloc metadata, which resides at the beginning of some pages and also allocated within the heap itself [6]. In this case, the attacker does not directly control the target chunks. However, he may be able to indirectly force the allocation of a metadata object by allocating a page worth of objects of certain size classes. Target chunks include these page info structures.

4.1.3 Allocator Analysis

A number of allocator features have a direct impact on their vulnerability to overflow attacks.

Inline metadata.

Allocators such as DLmalloc and Windows that use inline metadata inherently provide many target chunks. For some attacks, effectively any chunk in the heap could be overwritten to cause a remote exploit. For example, a patient attacker relying on freelist operations (an "`unlink` attack") could overwrite the freelist pointers in an arbitrary free chunk, then simply wait for that chunk to be reused. These allocators are similarly vulnerable to other such attacks, such as those targeting an object's size field.

Page-resident metadata.

Allocators with no inline metadata, such as PHKmalloc, may still have allocator metadata adjacent to heap objects. PHKmalloc places page info structures at the beginning of some pages (those containing small objects), and allocates others from the heap itself, in between application objects. Those allocated from the heap itself are obviously vulnerable to overwrites, especially if the attacker can control where they are allocated due to determinism in object placement. PHKmalloc also lacks guard pages, meaning that the page info structures placed at the beginning of pages may also be adjacent to overflowable application chunks.

Guard pages.

Guard pages can protect against overflows in multiple ways. First, for allocators like PHKmalloc which place metadata at the beginning of some pages, guard pages could be used to protect that metadata against overflows (though they are not). Deterministically placing a guard page before each page with metadata provides protection against contiguous overruns (the most common case), but not against underruns or non-contiguous overflows (such as an off-by-one on a multidimensional array).

Second, guard pages provide gaps in memory that cannot be spanned by contiguous overflow attacks, limiting the number of heap chunks that can be overwritten by a single attack. In this sense, guard pages protect application data itself. However, if the allocator is sufficiently deterministic, an attacker may be able to ensure the placement of the source chunk well before any guard page, allowing an attack to overwrite many chunks.

Canaries.

The use of canaries to protect heap metadata an application data may protect against overflows in some cases. However, their effectiveness is limited by how often the canaries are checked. Metadata canaries may be checked during every heap operation and can substantially protect metadata against overflows. However, allocators that place canaries between heap objects must trade off runtime efficiency for protection. For example, an overflow targeting a function pointer in application data requires no heap operations: only the overwrite and a jump through the pointer. Since allocators that check canaries only do so on `malloc` and `free`, they cannot protect against all such attacks.

Randomized placement.

All existing allocators that do not explicitly randomize object placement can be forced to allocate contiguous objects, assuming enough control of allocations and frees by the attacker. Techniques such as Heap Feng Shui are designed to force the allocator into such a deterministic state in order to enable reliable exploitation of vulnerabilities.

OpenBSD randomizes placement of heap objects to a limited extent. This approach reduces the reliability of overflow exploits by randomizing which heap chunks are overwritten by any single overflow. Attacks that depend on contiguous objects are also complicated, since it is unlikely that any given objects will be contiguous in memory.

However, overflow attacks able to span multiple heap chunks need not rely on contiguously-allocated objects. As long as the target object is placed after the source object on the same page, the attacker can overwrite the target. The extent of placement randomization affects the probability of such an attack's success.

OpenBSD's limited randomization allows certain such attacks to succeed with high probability. In an unfragmented heap, successive allocations of the same size objects will be clustered on the same page, even though their placement is randomized within that page. An attacker that can control heap operations so that the source and target are allocated on the same page has a 50% probability of allocating the source at a lower address than the target, enabling the attack to succeed.

For small objects, object placement is not fully randomized within a page because the allocator uses only 4 bits of entropy for a single allocation. For example, two successive allocations on a fresh page will always lie within 16 chunks of each other. An attack can exploit this property to increase attack reliability by limiting the length of the overflow, reducing the risk of writing past the end of a page and causing the application to crash.

4.2 Heap Spraying Attacks

Heap spraying attacks are used to make exploitation of other vulnerabilities simpler. In modern systems, guessing the location of heap-allocated shellcode or the address of a specific function for a return-to-libc attack can be difficult due to ASLR. However, on many systems, the heap lies within a restricted address space. For example, on 32-bit systems the heap generally lies within the first 2 GB of virtual address space. If the attacker allocates hundreds of megabytes of shellcode, jumping to a random address within this 2 GB region has a high probability of success.

4.2.1 Heap spraying attack model

To successfully exploit a heap spray, the attacker must guess the address contained within some (large) set of attacker-allocated objects. However, the attacker need not guess a pointer out of thin air. The simplest attack exploits an overflow to overwrite an ap-

plication pointer with the guessed value. However, if this pointer already references the heap, overwriting only the low-order bytes of the pointer on a little-endian machine results in a different pointer, but to an address close to the original address. An attacker often knows the address of a valid heap object and can use this knowledge to guess the address of a sprayed object. This knowledge may be acquired either implicitly due to a partial overwrite, or explicitly based on information leakage.

To account for these effects, we consider two heap spraying attack models. Both require the attacker to guess the address of one of a specific set of sprayed objects, V. The models differ in the information known to the attacker:

- **No *a priori* knowledge.** In the first model, the attacker must guess an address with no *a priori* knowledge of valid heap addresses.

- **Known address attacks.** In the second attack model, the attacker knows the address of a valid heap object. In some cases, the attacker may control when the known object is allocated. For example, allocating it temporally between two shellcode buffers makes it easy to guess a shellcode address if the heap allocates objects contiguously. This model is more general and significantly stronger than the ability to partially overwrite a pointer value. In the latter case, the attacker does not know an exact address, and can only guess addresses within 256 or 64K bytes (when overwriting 1 or 2 bytes, respectively).

4.2.2 Allocator Analysis

We quantitatively analyze allocator design choices with respect to heap spraying attacks under both attack models.

No a priori *knowledge.*

First, we analyze the probability of an attacker guessing the address of one of a set V of target objects without *a priori* information. V models the set of objects sprayed into the heap. Note that unlike the overflow case, the attacker can cause $|V|$ to be close to $|H|$, the size of the heap. Thus, the probability of guessing the address of heap-allocated shellcode or other target heap data is equivalent to guessing the address of any heap object in the limit.

More formally, we consider the probability $P(a \in V)$, that is, the probability of address a pointing to a valid heap object. Under this model, the attacker knows only the approximate value of $|H|$, the amount of allocated heap memory.

$P(a \in V)$ is almost entirely dependent on the target system's ASLR implementation. For example, on systems without ASLR, an attacker knowing the the size of the heap can always guess a valid address. Even with ASLR, an attacker spraying hundreds of megabytes of data into the heap on a 32-bit system has a high probability of guessing the address of a sprayed object. Note that $P(a \in V)$ need not be uniform with respect to a: if the system allocates memory contiguously and $|H| > 2$ GB, then address $a =$ `0x80000000` must contain valid data. However, if the allocator allocates pages randomly and non-contiguously, then the probability need not depend on a itself.

On 64-bit systems, however, the situation is vastly improved. Even on modern x86-64 systems which limit the effective virtual address range to 48 or 52 bits, physical memory limitations restrict the attacker's ability to fill a significant portion of this space. If ASLR randomizes the addresses of the `mmap` region across the entire space, the probability of guessing a valid address is low. Further evaluation of existing ASLR systems, which have been discussed by Shacham et al. [32] and Whitehouse [35], is outside the scope of this paper.

Known address attacks.

From the allocator's perspective, the problem of guessing the address of an object in V given the address of a heap object $o \in V$ depends upon the correlation of valid addresses with that of o. In most allocators, this correlation is due to contiguous object allocation. The addresses of contiguous objects are dependent upon each other. For example, if the entire heap is contiguous, then the addresses of all heap objects are mutually dependent, and thus amenable to guessing.

Quantitatively, we can evaluate the predictability of object addresses by considering the conditional probability distribution $P(a \in V \mid o \in H)$. An allocator that minimizes this probability *for all* $a \neq o$ is the least predictable.

In a contiguous heap, the address distribution is highly correlated. An address δ bytes after the known object o is valid if and only if o lies within the first $H - \delta$ bytes of the heap. In the worst case, we have no knowledge of the position of o within the heap. The probability of a being a valid address is thus dependent on its distance from o. The validity of the addresses surrounding o are highly correlated.

By contrast, consider the Archipelago allocator [19]. Archipelago allocates each object in a random position on a separate page, compressing cold pages to limit its consumption of physical memory. Since it allocates objects randomly throughout a large address space, Archipelago delivers minimal correlation because all object addresses are independent. The probability $P(a \in V \mid o \in H) \approx P(a \in V)$[1].

Practical allocators must trade off performance with predictability. While Archipelago works well for programs with small heap footprints and allocation rates, it is by no means a general purpose allocator. Practical allocators must allocate multiple objects on the same page in order to provide spatial locality to the virtual memory system. The page granularity thus limits the entropy an allocator can provide, and thus the protection it can supply against heap spray attacks.

4.3 Dangling Pointer Attacks

Temporal attacks rely on an application's use of a free chunk of memory. If the use is a write, the error is generally called a dangling pointer error. If the subsequent use is another free, it is called a double-free error. There are two general attack strategies targeting these errors, one based on reuse of the prematurely freed object, and another based on a freelist-based allocator's use of free chunks to store heap metadata.

4.3.1 Reuse Vulnerabilities

The first strategy exploits the reuse of chunks still referred to by a dangled pointer. The attacker's goal is to change the data contained in the object so that the later (incorrect) use of the first pointer causes an unintended, malicious effect. For example, if the dangled object contains a function pointer, and the attacker can force the allocator to reuse the chunk for an attacker-controlled object, he can overwrite the function pointer. Later use of the original pointer results in a call through the overwritten function pointer, resulting in a jump to an attacker-controlled location. This attack strategy has been described elsewhere [36], but we know of no specific attacks described in the literature.

[1]There is still minimal dependence, as Archipelago ensures at most one object is allocated per page.

This strategy exploits the predictability of object reuse by the allocator. A reliable attack can only be created if the attacker knows when the dangled chunk will be recycled. We formalize this by designating the dangled chunk as the **target chunk**. The attacker succeeds by forcing the allocator to recycle the target chunk.

Unlike buffer overflows, where each attempt by the attacker may cause the program to crash (e.g., by attempting to overflow into unmapped memory), repeated attempts to reallocate the dangled chunk need not perform any illegal activity. The attacker just allocates objects and fills them with valid data. This strategy limits the ability the runtime system to cope with such an attack, unless it somehow prevents the original dangling pointer error (e.g., via conservative garbage collection).

To combat reuse-based attacks, an allocator can implement a variety of strategies to delay reuse. First, it can delay reuse for as long as possible, e.g., by using a FIFO freelist. Unfortunately, in a defragmented heap, this policy has little effect.

The allocator could also impose a minimum threshold before objects are recycled. While a fixed threshold would be predictable and thus exploitable, randomized reuse would generally make attacks less reliable. For example, if an attacker has only one chance to force an application to call the overwritten function pointer, randomized object reuse reduces the probability of success.

OpenBSD implements limited reuse randomization by storing freed pointers in a random index of a 16-element array. The object is only actually freed when a subsequent free maps to the same array index. Each subsequent free is thus a Bernoulli trial with a 1/16 probability of success, making the distribution of t, the time before the object is reused, follow a geometric distribution with approximately 5.4 bits of entropy.

4.3.2 Allocator Analysis

In this section, we analyze the effect of allocator design on the predictability of object reuse. We evaluate each allocator feature by analyzing the entropy of t, the random variable representing the number of allocations before a just-freed object is recycled.

Freelists.

Freelist-based allocators commonly use LIFO freelists. Independent of other allocator features such as coalescing, such freelists always return the most-recently allocated object, providing zero entropy and thus perfect predictability.

BiBOP-style allocators.

BiBOP-style allocators may implement different reuse policies. PHKmalloc tracks a freelist of pages, and allocates in address-ordered first-fit within the first page on the freelist. Thus, t depends on the number of free chunks on a page. If the freed object creates the only free chunk on the page, the page was not previously on the freelist, and so the allocator will place it at its head. The subsequent allocation will choose this page, and return the only free chunk, which is the just-freed chunk. An attacker can force this behavior by allocating objects from the same size class as the target in order to eliminate fragmentation before the call to `free`.

Coalescing.

Most freelist-based allocators perform coalescing, the merging of adjacent free chunks. When a free chunk is coalesced with an existing free chunk, its size class will change, and thus be placed on an unpredictable freelist. While coalescing is deterministic, it relies on several aspects of the heap layout, making it difficult to create attacks when it occurs. However, in a defragmented heap, the probability of coalescing occurring is low, making it straightforward to work around in existing allocators.

4.3.3 Specific Attack: Inline Metadata

The second strategy relies on the behavior of the allocator itself. Freelist-based allocators write metadata into the contents of free chunks. If a dangling pointer points to a free chunk, then it points to overwritten, invalid data. If the attacker can control or predict the data the allocator writes into the freed chunk, he can maliciously corrupt the contents of the object.

Example.

Afek describes an exploit that relies on the object layout of C++ objects, combined with the freelist behavior of the Windows heap [1]. On most implementations, the first word of a C++ object with virtual functions contains the pointer to the virtual function table. This same word is also used by freelist-based allocators to store the pointer to the next object on the freelist. Afek's technique allocates a fake vtable object containing pointers to shellcode, then frees the object. Then, the dangling pointer error is triggered, placing the dangled chunk at the head of the freelist and storing a reference to the fake vtable in the first word. When the application erroneously uses the dangled pointer and performs a virtual function call, the runtime looks up the address of the target function from the forged vtable installed by the allocator, resulting in a jump to shellcode.

4.3.4 Allocator Analysis

This vulnerability is specific to freelist-based allocators, and does not affect allocators with no inline metadata. BiBOP-style allocators do not write metadata to free chunks, so they cannot be forced to write attacker-controlled data into dangled objects. This vulnerability also exploits deterministic reuse order, discussed in detail in Section 4.3.2.

5. COUNTERMEASURES

Allocator implementors have introduced a variety of techniques to protect inline metadata against attacks. The first countermeasures were freelist integrity checks, included in modern freelist-based allocators to prevent unlink attacks. Instead of naïvely trusting the free chunk header, the allocator ensures that memory pointed to by the heap chunk header is a valid chunk that refers back to the supplied chunk, and thus forms a valid doubly-linked list.

In addition to freelist integrity checks, Windows XP SP2 added an additional countermeasure. Each object header contains a 1-byte cookie computed from a per-heap pseudorandom value and the chunk address. The allocator checks the integrity of this cookie on each `free` operation, (possibly) aborting the program if it fails. An attack that contiguously overflows the previous object must correctly forge this value in order to overwrite freelist pointers. However, some heap metadata, notably the size field, lies before the cookie, allowing small overwrites to modify the inline metadata without corrupting the cookie. McDonald et al. describe a technique that can achieve a single null byte overflow, such as a string terminator [20] (used in the "heap desynchronization" attack described in that work). Furthermore, there are only 256 possible 1-byte values, so if an attack can repeatedly guess random cookies, it will succeed after a relatively low number of trials.

Despite the introduction of these countermeasures, attackers have found new methods of corrupting heap metadata to allow arbitrary code execution. McDonald and Valasek present a comprehensive summary of attacks against the Windows XP Service Pack 2/3 heap [20], and Ferguson provides an accessible overview of techniques targeting DLmalloc [13].

While freelist integrity checks were added to the Windows XP heap in service pack 2, a similar structure called the lookaside list (LAL) was left unprotected, allowing similar attacks. Similarly, the allocator did not consistently check the header cookie (in particular, during LAL operations), making it possible exploit certain chunk header overwrites without guessing the correct value [4].

More comprehensive protection for chunk headers was added in Windows Vista. In Vista, the entire chunk header is "encrypted" by XORing with a random 32-bit value. All uses of header fields must be decrypted before use, meaning that the allocator *must* consistently check the header integrity in order to function correctly. In order to supply a specific value to a header field, an attacker must determine the 32-bit value, which is harder to brute force than the single-byte cookie.

While header encryption has effectively eliminated the ability of simple buffer overflows to successfully attack heap metadata, the technique is just the most recent reaction to inline metadata attacks. All of these techniques simply cope with an underlying design flaw: allocators with no inline data are not susceptible to this kind of attack.

6. DIEHARDER: DESIGN AND ANALYSIS

In this section, we present the design of DieHarder, a memory allocator designed with security as a primary design goal. As its name implies, DieHarder is based on DieHard, a fully randomized heap allocator designed to improve the resilience of programs to memory errors [8]. Our implementation is based on the adaptive version of DieHard [9].

While DieHard was designed to increase reliability, it does so by fully randomizing the placement and reuse of heap objects. Due to this randomization, allocator behavior is highly unpredictable, a primary goal for our secure allocator. We first describe the DieHard allocator and analyze its strengths and weaknesses with respect to our attack models. We then present DieHarder, our secure allocator which modifies DieHard to correct those weaknesses. Figure 3 presents an overview of DieHarder's architecture.

6.1 DieHard Overview

DieHard consists of two features: a bitmap-based, fully-randomized memory allocator and a replicated execution framework; we discuss only the former [8]. The adaptive version of DieHard, upon which DieHarder is based, adaptively sizes its heap to be some multiple M larger than the maximum needed by the application (for example, M could be 2) [9]. This version of DieHard allocates memory from increasingly large chunks called *miniheaps*. Each miniheap contains objects of exactly one size. DieHard allocates new miniheaps to ensure that, for each size, the ratio of allocated objects to total objects is never more than $1/M$. Each new miniheap is twice as large, and thus holds twice as many objects, as the previous largest miniheap.

Allocation randomly probes a miniheap's bitmap for the given size class for a 0 bit, indicating a free object available for reclamation, and sets it to 1. Freeing a valid object resets the appropriate bit. Both `malloc` and `free` take $O(1)$ expected time. DieHard's use of randomization across an over-provisioned heap makes it likely that buffer overflows will land on free space, and unlikely that a recently-freed object will be reused soon.

6.2 DieHard Analysis

Like OpenBSD, DieHard randomizes the placement of allocated objects and the length of time before freed objects are recycled. However, unlike OpenBSD's limited randomization, DieHard randomizes placement and reuse to the largest practical extent. We show how these two randomization techniques greatly improve protection against attacks by decreasing predictability.

Randomized Placement.

When choosing where to allocate a new object, DieHard chooses uniformly from every free chunk of the proper size. Furthermore, DieHard's overprovisioning ensures $O(N)$ free chunks, where N is the number of allocated objects. DieHard thus provides $O(\log N)$ 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.

THEOREM 1. *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.

PROOF. Due to overprovisioning (by a factor of M) there are at least MN free heap chunks to choose for each allocation. Each of these slots is equally likely to be chosen. The probability of the chunks being allocated contiguously is thus at most $2/MN$, assuming free chunks on both sides of the first-allocated chunk (otherwise, the probability is lower). □

The probability of a k-chunk overflow overwriting one of V vulnerable objects generalizes this result. To derive the result, we consider the k object slots following the source object. The first object in V, v_0 has a $(MN-k)/MN$ chance of being outside these k slots, since there are MN possible positions. Each successive v_i has a $(MN-k-i)/MN$ chance, since each $v_0 ... v_{i-1}$ consumes one possible position. Multiplying these probabilities gives

$$\frac{(MN-k)!}{MN \cdot (MN-k-|V|-1)!},$$

the probability of all vulnerable objects residing outside the overwritten region. Thus the overwrite succeeds with probability

$$1 - \frac{(MN-k)!}{MN \cdot (MN-k-|V|-1)!}.$$

If $|V| << N$, each factor is approximately $(MN-k)/MN$, making the probability of a successful attack

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

Randomized Reuse.

DieHard chooses the location of newly-allocated chunks randomly across all free chunks of the proper size. Because of its overprovisioning (M-factor), the number of free chunks is always proportional to N, the number of allocated objects. Thus the probability of returning the most-recently-freed chunk is at most $1/MN$. This bound holds even if we continuously allocate without freeing, since the allocator maintains its M overprovisioning factor. In other words, the allocator is sampling with replacement. Thus, like OpenBSD, t follows a geometric distribution with $p = 1/MN$. Unlike OpenBSD, which has low fixed reuse entropy, DieHard provides $O(\log N)$ bits, making reuse much less predictable.

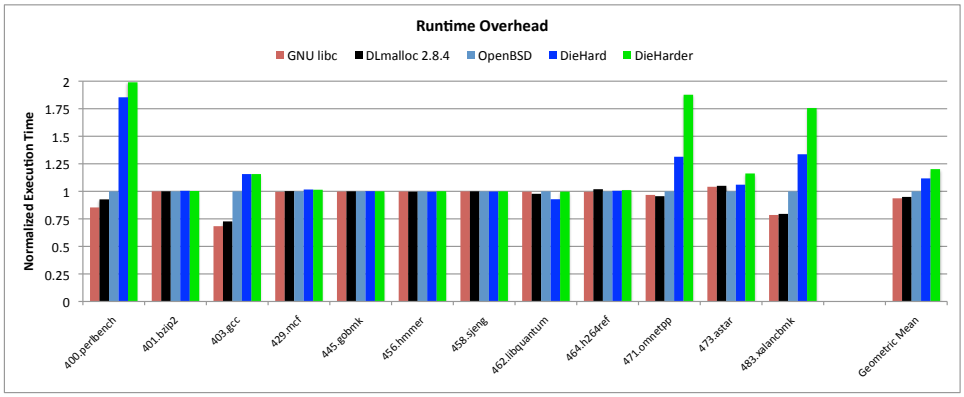

Figure 4: Runtime overhead of the different allocators, normalized to their runtime using OpenBSD's allocator. In exchange for a substantial increase in entropy, DieHarder imposes on average a 20% performance penalty vs. OpenBSD for CPU-intensive benchmarks, though it has no performance impact on Firefox (see Section 7).

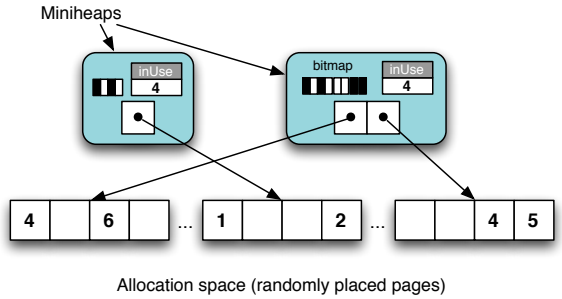

Figure 3: An overview of DieHarder's heap layout.

6.3 DieHarder Design and Implementation

As shown in the previous section, DieHard provides greater security guarantees than other general-purpose allocators. However, DieHard was designed to increase reliability against memory errors rather than for increased security. Several features of DieHard enable the program to continue running after experiencing memory errors, rather than thwarting potential attackers. In this section, we describe DieHarder's changes to the original DieHard allocator that substantially enhance its protection against heap-based attacks.

Sparse Page Layout

DieHard's first weakness is its use of large, contiguous regions of memory. Allocating such regions is more efficient than sparse pages, requiring fewer system calls and smaller page tables. This heap layout results in large regions without guard pages, allowing single overflows to overwrite large numbers of heap chunks.

In contrast, OpenBSD's allocator uses a *sparse page layout*, where small objects are allocated within pages spread sparsely across the address space. This approach relies on OpenBSD's ASLR to allocate randomly-placed pages via `mmap`. On 64-bit systems, ASLR makes it highly unlikely that two pages will be adjacent in memory. As a result, a single overflow cannot span a page boundary without hitting unmapped memory and crashing the program.

Our first enhancement to DieHard is to use sparse page allocation. Similarly to OpenBSD, DieHarder randomly allocates individual pages from a large section of address space. DieHarder treats these pages like DieHard, carving them up into size-segregated chunks tracked by an allocation bitmap. Allocation is also performed as in DieHard, with an extra level of indirection to cope with sparse page mapping.

Object deallocation is more complicated, since finding the correct bitmap given an object address is not straightforward. DieHard finds the correct metadata using a straightforward search, exploiting its heap layout to require expected constant time. With sparse pages, however, using DieHard's approach would require $O(N)$ time. DieHarder instead uses a hash table to store references to page metadata, ensuring constant-time `free` operations.

Address Space Sizing

To achieve full randomization under operating systems with limited ASLR entropy, DieHarder explicitly randomizes the addresses of small object pages. It does so by mapping a large, fixed-size region of virtual address space and then sparsely using individual pages. This implementation wastes a large amount of virtual memory, but uses physical memory efficiently as most virtual pages are not backed by physical page frames.

While the size of the virtual region does not affect the amount of physical memory used by application data, it does affect the size of the process's page tables. The x86-64 uses a 4-level page table. Contiguous allocations of up to 1 GB (2^{18} pages) require only 1 or 2 entries in the top three levels of the table, consuming approximately 512 pages or 2 MB of memory for the page table itself. In contrast, sparsely allocating 1 GB of pages within the full 48-bit address space requires mostly-complete middle levels of the table. Each 512-entry second-level page-middle directory (PMD) spans 1 GB, and the expected number of pages contained within each 1 GB region is 1. The resulting page table would thus require on the order of $2 \cdot 2^{18}$ page table entries (PTEs) and PMDs, for a staggering 2 GB page table.

Even if physical memory is not an issue, these sparse page tables can drastically decrease cache efficiency when the application's working set exceeds the TLB reach. When each PMD and PTE is sparse, the cache lines containing the actual entries have only 1/8 utilization (8 of 64 bytes). Combined with needing a line for each PMD and PTE, the effective cache footprint for page tables grows by 16X under a sparse layout.

To combat this effect, we restrict DieHarder's randomization to a much smaller virtual address range.

Destroy-on-free

DieHarder adopts OpenBSD's policy of filling freed objects with random data. While this policy empirically helps find memory errors, within the context of DieHarder, it is required to limit the effectiveness of certain attack strategies.

Unlike allocators with deterministic reuse, repeated `malloc` and `free` operations in DieHarder return different chunks of memory. If freed objects were left intact, even an attacker with limited control of heap operations (e.g., only able to hold only one object live at a time), could fill an arbitrary fraction of the heap with attacker-controlled data by exploiting random placement. In the same scenario, overwriting the contents of freed objects ensures only one chunk at a time contains attacker-controlled data.

6.3.1 DieHarder Analysis

Using a sparse page heap layout provides greater protection against heap overflow attacks and heap spraying. Unlike DieHard, DieHarder does not allocate small objects on contiguous pages.

Overflows

The sparse layout provides two major protections against overflow attacks. First, because pages are randomly distributed across a large address space, the probability of allocating two contiguous pages is low. Thus pages are protected by guard pages on both sides with high probability. Overflows past the end of a page will hit the guard page, causing the attack to fail.

The chance of hitting a guard page depends on H, the number of allocated pages and S, the size in pages of DieHarder's allocated virtual address space. The chance of having a guard page after any allocated page is $(S - H)/S$. This probability increases with S; however, large values of S can degrade performance, as discussed in Section 6.3.

Combined with randomized object placement, the memory immediately after *every* allocated object has a significant probability of being unmapped. The worst case for DieHarder is 16-byte objects, since there are 256 16-byte chunks per page. The probability of a 1-byte overflow crashing immediately is at least

$$\frac{(S-H)}{S} \cdot \frac{1}{256}.$$

The first term represents the probability of the following page being unmapped, and the second term the probability of the overflowed object residing in the last slot on the page.

Heap Spraying

DieHarder's sparse layout protects against heap spraying attacks by providing more entropy in object addresses. DieHarder's fully-randomized allocation eliminates dependence between the addresses of objects on different pages. The number of objects easily guessable given a valid object address is limited to the number that reside on a single page, which is further reduced by DieHarder's overprovisioning factor, inherited from DieHard.

7. DIEHARDER EVALUATION

We measure the runtime overhead of DieHarder compared to four existing allocators, GNU libc (based on DLmalloc 2.7), DLmalloc 2.8.4, DieHard, and OpenBSD. We enabled DLmalloc 2.8's object footers that improve its resilience against invalid frees. We use the adaptive version of DieHard [9]. To isolate allocator effects, we ported OpenBSD's allocator to Linux. We run DieHarder using a 4 GB virtual address space for randomizing small object pages. We discuss the impact of this design parameter in Section 6.3.

Our experimental machine is a single-socket, quad-core Intel Xeon E5520 (Nehalem) running at 2.27GHz with 4 GB of physical memory. We first evaluate the CPU overhead of various allocators using the SPECint2006 benchmark suite. Unlike its predecessor (SPECint2000), this suite places more stress on the allocator, containing a number of benchmarks with high allocation rates and large heap memory footprints.

Figure 4 shows the runtime of the benchmark suite using each allocator, normalized to its runtime under OpenBSD's allocator. DieHarder's overhead varies from -7% to 117%, with a geometric mean performance impact of 20%. Most benchmarks exhibit very little overhead (less than 2%). The benchmarks that suffer the most, `perlbench`, `omnetpp`, and `xalancbmk`, significantly stress the allocator due to their unusually high allocation rates.

Firefox.

In addition to the SPECint2006 suite, we evaluated the performance of the Firefox browser using both DieHarder (4 GB virtual address space) and GNU libc. In order to precisely measure Firefox's performance, we used the Selenium browser automation tool to automatically load a sequence of 20 different web pages. We used an offline proxy, wwwoffle, to minimize the effect of network latency and ensure identical behavior across all experiments. We repeated this experiment 15 times for each allocator.

The results show no statistically significant difference in performance between allocators at the 5% level. The mean runtimes for GNU libc and DieHarder, respectively, were 44.2 and 41.6 seconds, with standard deviations of 7.13 and 6.12. This result qualitatively confirms that DieHarder is practical for use.

8. RELATED WORK

8.1 Memory allocator security

Most previous work to increase the security of memory allocators has focused on securing heap metadata and the use of randomization to increase non-determinism.

One approach is to secure the metadata via encryption: Robertson describes the use of XOR-encoded heap metadata [31], a countermeasure that was incorporated (in slightly modified form) by Lea into DLmalloc version 2.8 (a later version than the basis of GNU libc's allocator). Younan et al. instead present a modified version of the Lea allocator that fully segregates metadata, but which implements no other security enhancements [38]. Kharbutli et al. describe an approach to securing heap metadata that places it in a separate process [17]. Isolation of heap metadata helps prevent certain attacks but, for example, does not mitigate attacks against the heap data itself. Like DieHard, DieHarder completely segregates heap metadata, and its randomized placement of heap metadata in a sparse address space effectively protects the metadata.

Several uses of randomization have been proposed to increase the non-determinism of object placement and reuse, including locating the heap at a random base address [10, 26], adding random padding to allocated objects [11], and shuffling recently-freed objects [17]. None of these approaches generate as much entropy as DieHarder.

8.2 Object-per-page allocators

Several memory allocators have been proposed that use one page for each object (PageHeap [21], Electric Fence [27], and Archipelago). The first two were designed specifically for debugging and are not suitable for deployment. Archipelago provides higher performance and significantly reduces space overhead, but its overhead still makes it prohibitive for use in many situations.

8.3 Other countermeasures

We briefly describe other countermeasures not mentioned in Section 5 that are orthogonal and complementary to DieHarder.

One noteworthy countermeasure by Ratanaworabhan et al. called Nozzle addresses heap spraying attacks aimed at preventing code injection attacks [30]. Nozzle operates by scanning the heap looking for valid x86 code sequences—a large number of such sequences indicates that a spray attack is in progress, and can be used to trigger program termination.

Libraries like LibSafe and HeapShield can prevent overflows that stem from misuse of C APIs like `strcpy` [5, 7]. HeapShield itself was integrated into DieHard [9] and is also integrated into DieHarder, although we do not evaluate its impact here.

Finally, compilers can prevent buffer overflows (though not dangling pointer errors) by implementing bounds checks [23, 3, 2]. Limitations of these techniques include their restriction to C, the need to recompile all code (including libraries), and in most cases, a substantial performance penalty. WIT [2] works for both C and C++ and protects against overwrites (but not out-of-bound reads) with low overhead. Baggy Bounds Checking [3] relies on the same insight exploited by HeapShield, namely that bounds checks can be implemented efficiently for BiBOP-style allocators, and thus could easily be modified to use DieHarder as its allocation substrate.

9. CONCLUSION

We present an extensive analysis of the impact of memory allocator design decisions on their vulnerability to attack. Guided by this analysis, we design a new allocator, DieHarder, that provides the highest level of security from heap-based memory attacks. It reduces the risk of heap buffer overflow attacks by fully isolating heap metadata from application data, by interspersing protected guard pages throughout the heap, and by fully randomizing object placement. It limits dangling-pointer based exploits by destroying freed data and destroys the contents of freed objects, and by fully randomizing object reuse. We show analytically that, compared to past allocators, DieHarder's design decisions greatly complicate the task of the attacker both by limiting exposure to some attacks and by dramatically increasing entropy over past memory allocators. Our empirical evaluation shows that DieHarder imposes modest runtime overhead—on average, running 20% slower than OpenBSD across a suite of CPU-intensive benchmarks, and performing equivalently to the Linux allocator on the Firefox web browser.

Acknowledgements.

The authors would like to thank Ben Zorn for conversations that helped inspire this work, Doug Lea for his feedback on a previous draft of this paper, Justin Aquadro for his assistance with benchmarking Firefox, and the anonymous reviewers for their comments that helped shape the final version of this paper.

The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. This material is based upon work supported by Intel, Microsoft Research, and the National Science Foundation under CAREER Award CNS-0347339 and CNS-0615211. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] J. Afek and A. Sharabani. Dangling pointer: Smashing the pointer for fun and profit. In *Black Hat USA*, 2007.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[3] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, pages 51–66. USENIX, Aug. 2009.

[4] A. Anisimov. Defeating Microsoft Windows XP SP2 heap protection and DEP bypass, 2005.

[5] K. Avijit, P. Gupta, and D. Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *Proceedings of the 13th USENIX Security Symposium*. USENIX, Aug. 2004.

[6] BBP. BSD heap smashing. http://www.ouah.org/BSD-heap-smashing.txt.

[7] E. D. Berger. HeapShield: Library-based heap overflow protection for free. Technical Report UMCS TR-2006-28, Department of Computer Science, University of Massachusetts Amherst, May 2006.

[8] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 158–168, New York, NY, USA, 2006. ACM Press.

[9] E. D. Berger and B. G. Zorn. Efficient probabilistic memory safety. Technical Report UMCS TR-2007-17, Department of Computer Science, University of Massachusetts Amherst, Mar. 2007.

[10] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120. USENIX, Aug. 2003.

[11] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286. USENIX, Aug. 2005.

[12] M. Conover and the w00w00 Security Team. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt, January 1999.

[13] J. N. Ferguson. Understanding the heap by breaking it. In *Black Hat USA*, 2007.

[14] S. Gonchigar. Ani vulnerability: History repeats. http://www.sans.org/reading_room/whitepapers/threats/ani-vulnerability-history-repeats_1926, 2007.

[15] D. R. Hanson. A portable storage management system for the Icon programming language. *Software Practice and Experience*, 10(6):489–500, 1980.

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

[17] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and efficiently protecting the heap. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 207–218, New York, NY, USA, 2006. ACM Press.

[18] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[19] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 115–124, New York, NY, USA, Mar. 2008. ACM.

[20] J. McDonald and C. Valasek. Practical Windows XP/2003 heap exploitation. In *Black Hat USA*, 2009.

[21] Microsoft Corporation. Pageheap. http://support.microsoft.com/kb/286470.

[22] O. Moerbeek. A new malloc(3) for OpenBSD. In *EuroBSDCon*, 2009.

[23] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139. ACM Press, Jan. 2002.

[24] 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.

[25] 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.

[26] PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[27] B. Perens. Electric Fence v2.1. http://perens.com/FreeSoftware/ElectricFence/.

[28] 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.

[29] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies: A safe method to survive software failures. In *Proceedings of the Twentieth Symposium on Operating Systems Principles*, volume XX of *Operating Systems Review*, Brighton, UK, Oct. 2005. ACM.

[30] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th USENIX Security Symposium*, pages 169–186. USENIX, Aug. 2009.

[31] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *LISA '03: Proceedings of the 17th Large Installation Systems Administration Conference*, pages 51–60. USENIX, 2003.

[32] 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.

[33] Solar Designer. JPEG COM marker processing vulnerability in Netscape browsers. http://www.openwall.com/advisories/OW-002-netscape-jpeg/, 2000.

[34] A. Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe*, 2007.

[35] O. Whitehouse. An analysis of address space layout randomization on Windows Vista. http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf, 2007.

[36] Wikipedia. Dangling pointer — Wikipedia, the free encyclopedia, 2010. [Online; accessed 16-April-2010].

[37] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116, Kinross, Scotland, Sept. 1995. Springer-Verlag.

[38] Y. Younan, W. Joosen, F. Piessens, and H. V. den Eynden. Security of memory allocators for C and C++. Technical Report CW 419, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, July 2005.