Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation

Jim Chow, Ben Pfaff, Tal Garfinkel, Mendel Rosenblum
{jchow,blp,talg,mendel}@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

Today's operating systems, word processors, web browsers, and other common software take no measures to promptly remove data from memory. Consequently, sensitive data, such as passwords, social security numbers, and confidential documents, often remains in memory indefinitely, significantly increasing the risk of exposure.

We present a strategy for reducing the lifetime of data in memory called *secure deallocation*. With secure deallocation we zero data either at deallocation or within a short, predictable period afterward in general system allocators (e.g. user heap, user stack, kernel heap). This substantially reduces data lifetime with minimal implementation effort, negligible overhead, and without modifying existing applications.

We demonstrate that secure deallocation generally clears data immediately after its last use, and that without such measures, data can remain in memory for days or weeks, even persisting across reboots. We further show that secure deallocation promptly eliminates sensitive data in a variety of important real world applications.

1 Introduction

Clearing sensitive data, such as cryptographic keys, promptly after use is a widely accepted practice for developing secure software [23, 22]. Unfortunately, this practice is largely unknown in commodity applications such as word processors, web browsers, and web servers that handle most of the world's sensitive data, e.g. passwords, confidential documents.

Consequently, sensitive data is often scattered widely through user and kernel memory and left there for indefinite periods [5]. This makes systems needlessly fragile, increasing the risk of exposing sensitive data when a system is compromised, or of data being accidentally leaked due to programmer error [1] or unexpected feature interactions (e.g. core dumps [15, 16, 14, 13], logging [5]).

We advocate a solution to this based on the observation that data's last use is usually soon before its deallocation. Thus, we can use deallocation as a heuristic for when to automatically zero data.

By zeroing data either at deallocation or within a short predictable period afterward in system allocators (heap, stack, kernel allocators, etc.), we can provide significantly shorter and more predictable data lifetime semantics, without modifying existing applications. We refer to this automatic approach to zeroing as *secure deallocation*.

We define the concept of a *data life cycle* to provide a conceptual framework for understanding secure deallocation. Using this framework, we characterize the effectiveness of secure deallocation in a variety of workloads.

We evaluated secure deallocation by modifying all major allocation systems of a Linux system, from compiler stack, to `malloc`-controlled heap, to dynamic allocation in the kernel, to support secure deallocation. We then measured the effectiveness and performance overheads of this approach through the use of whole-system simulation, application-level dynamic instrumentation, and benchmarks.

Studying data lifetime across a range of server and interactive workloads (e.g. Mozilla, Thunderbird, Apache and `sshd`), we found that with careful design and implementation, secure deallocation can be accomplished with minimal overhead (roughly 1% for most workloads).

We further show that secure deallocation typically reduces data lifetime to within 1.35 times the minimum possible data lifetime (usually less than a second). In contrast, waiting for data to be overwritten commonly produces a data lifetime 10 to 100 times the minimum and can even stretch to days or weeks. We also provide an in-depth analysis demonstrating the effectiveness of this approach for removing sensitive data across the entire software stack for Apache and Emacs.

We argue that these results provide a compelling case for secure deallocation, demonstrating that it can provide a measurable improvement in system security with negligible overhead, without requiring program source code to be modified or even recompiled.

Our discussion proceeds as follows. In the next section we present the motivation for this work. In section 3 we present our data lifetime metric and empirical results on how long data can persist. In section 4 we present the design principles behind secure deallocation while sections 5, 6, and 7 present our analysis of effectiveness and performance overheads of secure deallocation. In sections 8 and 9 we discuss future and related work. Section 10 offers our conclusions.

2 Motivation

In this section we discuss how sensitive data gets exposed, how today's systems fail to take measures to reduce the presence of long-lived sensitive data, and why secure deallocation provides an attractive approach to reducing the amount of long-lived data in memory.

2.1 The Threat Of Data Exposure

The simplest way to gain access to sensitive data is by directly compromising a system. A remote attacker may scan through memory, the file system or swap partition, etc. to recover sensitive data. An attacker with physical access may similarly exploit normal software interfaces [7], or if sufficiently determined, may resort to dedicated hardware devices that can recover data directly from device memory. In the case of magnetic storage, data may even be recoverable long after it has been deleted from the operating system's perspective [9, 11].

Software bugs that directly leak the contents of memory are common. One recent study of security bugs in Linux and OpenBSD discovered 35 bugs that can be used by unprivileged applications to read sensitive data from kernel memory [6]. Recent JavaScript bugs in Mozilla and Firefox can leak an arbitrary amount of heap data to a malicious website [21]. Many similar bugs undoubtedly exist, but they are discovered and eradicated slowly because they are viewed as less pressing than other classes of bugs (e.g. buffer overflows).

Data can be accidentally leaked through unintended feature interactions. For example, core dumps can leak sensitive data to a lower privilege level and in some cases even to a remote attacker. In Solaris, `ftpd` would dump core files to a directory accessible via anonymous FTP, leaking passwords left in memory [15]. Similar problems have been reported in other FTP and mail servers [16, 14, 13]. Systems such as "Dr. Watson" in Windows may even ship sensitive application data in core files to a remote vendor. Logs, session histories, and suspend/resume and checkpointing mechanisms exhibit similar problems [7].

Leaks can also be caused by accidental data reuse. Uncleared pages might be reused in a different protection domain, leaking data between processes or virtual machines [12]. At one time, multiple platforms leaked data from uncleared buffers into network packets [1]. The Linux kernel implementation of the ext2 file system, through versions 2.4.29 and 2.6.11.5, leaked up to approximately 4 kB of arbitrary kernel data to disk every time a new directory was created [2].

If data leaks to disk, by paging or one of the mechanisms mentioned above, it can remain there for long periods of time, greatly increasing the risk of exposure. Even data that has been overwritten can be recovered [9]. Leaks to network attached storage run the risk of inadvertently transmitting sensitive data over an unencrypted channel.

As our discussion illustrates, data can be exposed through many avenues. Clearly, reducing these avenues e.g. by fixing leaks and hardening systems, is an important goal. However, we must assume in practice that systems will have leaks, and will be compromised. Thus, it behooves us to reduce or eliminate the amount of sensitive data exposure that occurs when this happens by minimizing the amount of sensitive data in a system at any given time.

2.2 What's Wrong with Current Systems

Unfortunately, most applications take no steps to minimize the amount of sensitive data in memory.

Common applications that handle most sensitive data were never designed with sensitive data in mind. Examples abound, from personal data in web clients and servers, to medical and financial data in word processors and databases. Often even programs handling data known to be sensitive take no measures to limit the lifetime of this data, e.g. password handling in the Windows login program [5].

Applications are not the only culprits here. Operating systems, libraries and language runtimes are equally culpable. For example, in recent work we traced a password typed into a web form on its journey through a system. We discovered copies in a variety of kernel, window manager, and application buffers, and literally dozens of copies in the user heap. Many of these copies were long lived and erased only as memory was incidentally reused much later [5].

Consequently, even when programmers make a best-effort attempt to minimize data lifetime, their efforts are often flawed or incomplete as the fate of memory is often out of their control. A process has no control over

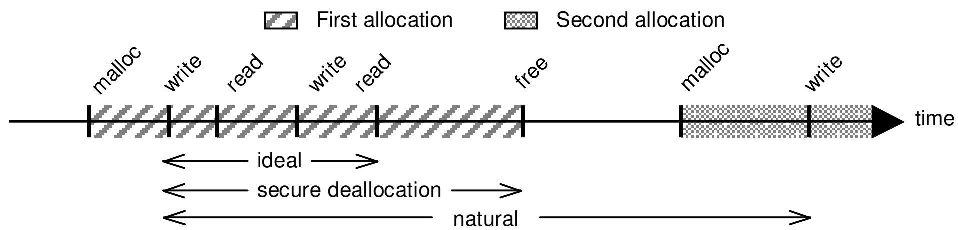

Figure 1: A time line showing the relationship of different memory events for a particular memory location. The span from first write to last read is the *ideal lifetime*. The data must exist in the system at least this long. The span from first write to deallocation is the *secure deallocation lifetime*. The span from first write to the first write of the next allocation is the *natural lifetime*. Because programs often rely on reallocation and overwrite to eliminate sensitive data, the natural lifetime is the expected data lifetime in systems without secure deallocation.

kernel buffers, window manager buffers, and even over application memory in the event that a program crashes.

3 Characterizing Data Lifetime

We begin this section with a conceptual framework for understanding secure deallocation and its role in minimizing data lifetime. We then present an experimental results quantifying how long data persists in real systems.

3.1 Data Life Cycle

The *data life cycle* (Figure 1) is a time line of interesting events pertaining to a single location in memory:

Ideal Lifetime is the period of time that data is in use, from the first write after allocation to the last read before deallocation. Prior to the first write, the data's content is indeterminate, and after the last read the data is "dead," in the sense that subsequent writes cannot affect program execution (at least for normal process memory). Thus, we cannot reduce data lifetime below the ideal lifetime without restructuring the code that uses it.

Natural Lifetime is the window of time where attackers can retrieve useful information from an allocation, even after it has been freed (assuming no secure deallocation). The natural lifetime spans from the first write after allocation to the first write of a later allocation, i.e. the first overwrite. This is the baseline data lifetime seen in today's systems.

Secure Deallocation Lifetime attempts to improve on the natural lifetime by zeroing at time of deallocation. The secure deallocation lifetime spans from the first write after allocation until its deallocation (and zeroing). The secure deallocation lifetime falls between the natural and ideal lifetimes.

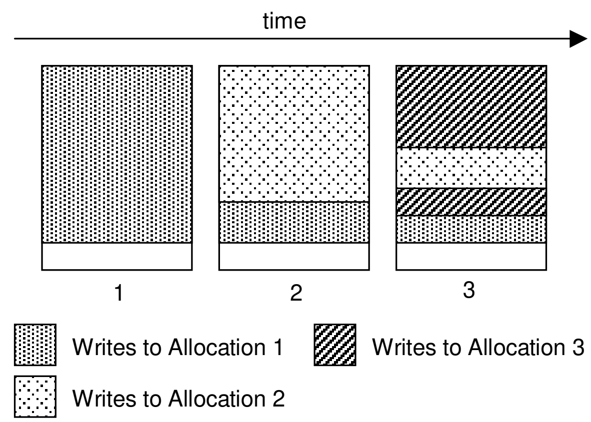

Figure 2: Incomplete overwrites, or *holes*, lead to the accumulation of data from previous allocations in current ones. This time line shows how a given block of memory gradually accumulates data from three different allocations.

Defining data lifetime in this manner provides a framework for reasoning about the effectiveness of zeroing policies. The degree to which the secure deallocation lifetime matches the ideal lifetime gives us a metric for understanding how well secure deallocation approximates an optimal policy.

Reallocation and Holes When memory is reallocated and used for a different purpose, it is not uncommon for the previous contents of the memory to be incompletely overwritten, allowing some data from the previous allocation to survive. We refer to the sections of surviving data as *holes*. Holes may arise from unused variables or fields, compiler-added padding for stack frame alignment or in `structs`, or unused portions of large buffers.

For example, it is common for user-level file name handling code to allocate `PATH_MAX` (at least 256) byte buffers even though they aren't completely used in most situations, and Linux kernel code often allocates an entire 4,096-byte page for a file name. The unused portion of the buffer is a hole. This is important for data lifetime because any data from a previous allocation that is in the hole is not overwritten. Figure 2 illustrates the accumulation of data that can result from these holes.

It might seem that secure deallocation is a superfluous overhead since the job of overwriting sensitive data can simply be handled when the memory is reused. However, in some programs, holes account for the vast majority of all allocated data. Thus, simply waiting for reallocation and overwrite is an unreliable and generally poor way to ensure limited data lifetime. The next section shows an example of this.

3.2 Long-Term Data Lifetime

On today's systems, we cannot predict how long data will persist. Most data is erased quickly, but our experiments described here show that a significant amount of data may remain in a system for weeks with common workloads. Thus, we cannot depend on normal system activities to place any upper bound on the lifetime of sensitive data. Furthermore, we found that rebooting a computer, even by powering it off and back on, does not necessarily clear its memory.

We wrote Windows and Linux versions of software designed to measure long-term data lifetime and installed it on several systems we and our colleagues use for everyday work. At installation time, the Linux version allocates 64 MB of memory and fills it with 20-byte "stamps," each of which contains a magic number, a serial number, and a checksum. Then, it returns the memory to the system and terminates. A similar program under Windows was ineffective because Windows zeroes freed process pages during idle time. Instead, the Windows version opens a TCP socket on the localhost interface and sends a single 4 MB buffer filled with stamps from one process to another. Windows then copies the buffer into dynamically allocated kernel memory that is not zeroed at a predictable time. Both versions scan all of physical memory once a day and count the remaining valid stamps.

Figure 3 displays results for three machines actively used by us and our colleagues. The machines were Linux and Windows desktops with 1 GB RAM each and a Linux server with 256 MB RAM. Immediately after the fill program terminated, 2 to 4 MB of stamps could be found in physical memory. After 14 days, between 23 KB and 3 MB of the stamps could still be found. If these stamps were instead sensitive data, this could pose a serious information leak.

In the best case, the Linux server, only 23 KB of stamps remained after 14 days. We expected that these remaining stamps would disappear quickly, but in fact, after an additional 14 days, about 7 KB of stamps were still left. A closer look found most data retained over the long term to lie in holes in pages owned by the Linux slab allocator, which divides pages into smaller blocks of equal size for piecemeal use. Most block sizes do not fit evenly into the page size, so leftover space (up to hundreds of bytes worth) follows the final block in a slab, and some blocks also contain data members that are rarely used. This unused space retains data as long as the slab page itself persists—at least as long as any block in the page is in use and ordinarily longer—and slab pages tend to be deallocated in large numbers only under memory pressure. Thus, we expect data that falls into a hole in a slab to persist for a long time on an ordinarily loaded system, explaining our observations.

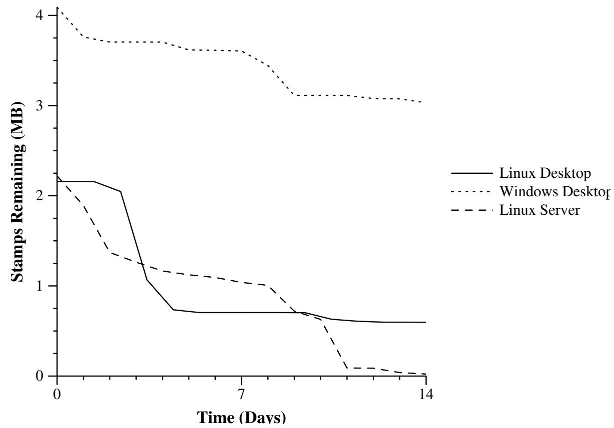

Figure 3: Lifetime of 20-byte "stamps" written to memory by a test program run on several machines used daily. This shows that data can often persist in memory for days or weeks under common workloads. Despite appearances, the Linux server did not drop quickly to 0 KB: at 14 days, it retained about 23 KB; at 28 days, about 7 KB.

Effect of Rebooting In the course of setting up experiments, we rebooted some machines multiple times and found, to some surprise, that some stamps put into memory before reboot remained. We investigated further and found that a "soft" reboot, that is, a reboot that does not turn off the machine's power, does not clear most of RAM on the machines we tested. The effect of a "hard" reboot, that is, one that powers off the machine, varied. On some machines, hard reboots cleared all stamps; on others, such as IBM ThinkPad T30 laptops, many were retained even after 30 seconds without power. We conclude that it is a bad idea to assume a reboot will clear memory without knowledge about the specific hardware in use.

4 Designing Secure Deallocation

In this section we describe the design principles behind secure deallocation.

4.1 A Conservative Heuristic

Secure deallocation clears data at deallocation or within a short, predictable time afterward. This provides a *conservative heuristic* for minimizing data lifetime.

Secure deallocation is a *heuristic* in that we have no idea when a program last uses data. We just leverage the fact that last-time-of-use and time-of-deallocation are often close together (see section 5). This is *conservative* in that it should not introduce any new bugs into existing programs, and in that we treat all data as sensitive, having no *a priori* knowledge about how it is used in an application.

This approach is applicable to systems at many levels from OS kernels to language runtimes, and is agnostic to memory management policy, e.g. manual freeing vs. garbage collection. However, the effectiveness of secure deallocation is clearly influenced by the structure and policy of a system in which it is included.

4.2 Layered Clearing

We advocate clearing at every layer of a system where data is deallocated including user applications, the compiler, user libraries, and the OS kernel. Each layer offers its own costs and benefits that must be taken account.

- *Applications* generally have the best knowledge of what data are sensitive and when the best time to clear them is. For example, an application that pops elements off a circular queue knows immediately that the space used to store those elements can and should be cleared. Because such operations are usually implemented in terms of simple pointer increments and decrements, the heap storage layer simply has no way of knowing this data could have been cleared.

  Unfortunately, it can be complex and labor intensive to identify all the places where sensitive data resides and clear it appropriately. We explore an example of modifying a piece of complex, data-handling software (the Linux kernel) to reduce the time that data is held in section 6.

- *Compilers* handle all the implicit allocations performed by programs (e.g. local variables allocated on the stack), therefore they can handle clearing data that programs do not explicitly control. Clearing data at this level can be expensive, and we explore the trade-offs in performance in section 4.4.

- *Libraries* handle most of the dynamic memory requests made by programs (e.g. `malloc`/`free`) and are the best place to do clearing of these requests. Clearing at this level has the caveat that we must depend on programs to deallocate data explicitly, and to do so as promptly as possible. We explore the efficacy of this approach in section 5.

- *Operating system kernels* are responsible for managing all of an application's resources. This includes process pages used in satisfying memory requests, as well as pages used to buffer data going to or coming from I/O devices.

  The OS is the final safety net for clearing all of the data possibly missed by, or inaccessible to, user programs. The OS kernel's responsibilities include clearing program pages after a process has died, and clearing buffers used in I/O requests.

Why Layered Clearing? Before choosing a layered design, we should demonstrate that it is better than a single-layer design, such as a design that clears only within the user stack and heap management layer.

Clearing only in a lower layer (e.g. in the kernel instead of the user stack/heap) is suboptimal. For example, if we do zeroing only when a process dies, data can live for long periods before being cleared in long running processes. This relates back to the intuition behind the heuristic aspect of secure deallocation.

Clearing only in a higher layer (e.g. user stack/heap instead of kernel) is a more common practice. This is incomplete because it does not deal with state that resides in kernel buffers (see section 6 for detailed examples). Further, it does not provide defense in depth, e.g. if a program crashes at any point while sensitive data is alive, or if the programmer overlooks certain data, responsibility for that data's lifetime passes to the operating system.

This basic rationale applies to other layered software architectures including language runtimes and virtual machine monitors.

The chief reason against a layered design is performance. But as we show in section 7, the cost of zeroing actually turns out to be trivial, contrary to popular belief.

4.3 Caveats to Secure Deallocation

Secure deallocation is subject to a variety of caveats:

- *No Deallocation.* Some applications deallocate little of their memory. In experiments we perform in section 5, for example, we see workloads where less than 10% of memory allocated was freed. In short-lived applications, this can be handled by the OS kernel (see section A.1). In longer-lived applications little can be done without modifying the program itself. Static data has the same issue because it also survives until the process terminates.

- *Memory Leaks.* Failing to free memory poses a data lifetime problem, although we'll see in section 5 that programs usually free data that they allocate. Fortunately, leaks are recognized as bugs by application programmers, so they are actively sought out and fixed.

  Long-lived servers like `sshd` and Apache are generally written to conscientiously manage their memory, commonly allowing them to run for months on end. When memory leaks do occur in these programs, installations generally have facilities for handling them, such as a `cron` job that restarts the process periodically.

- *Custom Allocators.* Custom allocators are commonly used to improve application performance or to help manage memory, e.g. by preventing memory leaks. Doing so, however, hides the application's use of memory from the C library, reducing the effectiveness of secure deallocation in the C library.

  Region-based allocators [8], for example, serve allocation requests from a large system-allocated pool. Objects from this pool are freed en masse when the whole pool is returned to the system. This extends secure deallocation lifetimes, because the object's use is decoupled from its deallocation.

  Circular queues are another common example. A process that buffers input events often does not clear them after processing them from the queue. Queue entries are "naturally" overwritten only when enough additional events have arrived to make the queue head travel a full cycle through the queue. If the queue is large relative to the rate at which events arrive, this can take a long time.

These caveats apply only to long-lived processes like Apache or `sshd`, since short-lived processes will have their pages quickly cleaned by the OS. Furthermore, long-lived processes tend to free memory meticulously, for reasons described above, so the impact of these caveats is generally small in practice.

These challenges also provide unique opportunities. For example, custom allocators designed with secure deallocation can potentially better hide the latency of zeroing, since zeroing can be deferred and batched when large pools are deallocated. Of course, a healthy balance must be met—the longer zeroing is deferred, the less useful it is to do the zeroing at all.

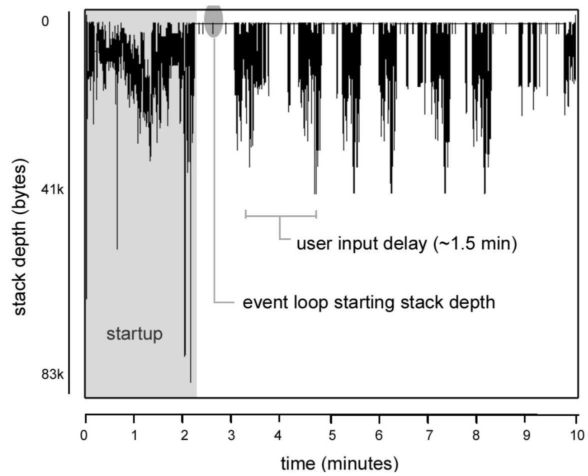

Figure 4: Crests and troughs of stack usage over time for a web browsing session under Firefox 1.0 (stack grows downward). Firefox typifies stack usage for a GUI application: the main window event loop sits high atop the stack and occasionally makes excursions downwards to do processing, in this case web page rendering, only to return back to the event loop. Intervals between excursions are on a human scale (seconds or minutes).

4.4 Implementing Clearing

In this section we provide some practical examples of design trade-offs we made in our secure deallocation implementation.

Compilers and Libraries Secure deallocation in compilers and libraries is relatively simple, and consists of clearing the heap and the stack.

All heap allocated data is zeroed immediately during the call to `free`. Data is cleared immediately because the latency imposed appears to be negligible in most cases, given the speed of zeroing.

For the stack, we explored two strategies: zeroing activation frames immediately as their function returns and periodically zeroing all data below the stack pointer (all old, currently unused space). The latter strategy amortizes the performance overhead of stack over many calls and returns, although it has the disadvantage of missing "holes" in the stack (see section 3.1).

The intuition for periodically zeroing the stack is illustrated by Figure 4, obtained by instrumenting Firefox 1.0. Although applications do make excursions downwards to do initialization or complex processing, many, particularly long-lived ones like network server daemons or GUI programs, spend most of their time high atop the stack, waiting in an event loop for a network/user request.

Clearing in the Kernel In the kernel we leveraged our greater knowledge of the semantics of different data structures to selectively clear only memory that may contain sensitive data. We chose this approach because the kernel is performance sensitive, despite the greater effort and implementation complexity required.

Ideally, this approach would provide the same reduction of sensitive data lifetime as we would obtain by clearing everything in the main kernel allocators, perhaps better, as specific data structures such as circular queues are cleared as well. However, as this is not conservative, there is a greater risk that we may have overlooked some potentially sensitive data.

The kernel has two primary responsibilities for zeroing. First, it must clear user space memory which has been deallocated, e.g. by process death or unmapping a private file mapping. Next, it must clear potentially sensitive state residing in I/O buffers e.g. network buffers (e.g. `sk_buffs` in Linux), tty buffers, IPC buffers.

Due to the range and complexity of zeroing done in the kernel we have deferred most of our discussion to section 6 and further in appendix A.

Zeroing Large Pools of Memory An unusual aspect of kernel zeroing is the need to clear large areas such as the pages in a terminated process. This requires significant care in order to balance the demand for short and predictable data lifetime against the need for acceptable latency.

To provide predictable data lifetime, we would like to have some sort of deadline scheduling in place, e.g. a guarantee that sensitive pages are zeroed within n seconds of deallocation. We would like n to be as small as possible without imposing unacceptable immediate latency penalties on processes. On the other hand, if n is too large many dirty pages could accumulate, especially under heavy load. This could lead to long and unpredictable pauses while the system stops to zero pages. Intuitively, this is very similar to garbage collection pausing a program to free up memory.

Sometimes proactively zeroing memory can actually improve system responsiveness. Even an unmodified kernel must zero memory before allocating it to a user process, to prevent sensitive data from one protection domain from leaking into another. Often this is done on demand, immediately before pages are needed. Doing this before pages are needed can improve performance for process startup. Zeroing memory can also increase page sharing under some virtual machine monitors [24].

Another important consideration is ensuring that zeroing large pools of memory does not blow out caches. We discuss this issue in section 7.1.

A more complete treatment of zeroing performed by the kernel is provided later in section 6 and appendix A.

Side Effects of Secure Deallocation Secure deallocation only modifies data with indeterminate content, e.g. freed data on the heap. This should not introduce bugs in correct programs. Some buggy software, however, depends on the value of indeterminate data.

Use of indeterminate data takes two forms. Software may use data before it has been initialized, expecting it to have a constant value. Alternately, software may use data after it has been freed, expecting it to have the same value it had prior to deallocation.

By making the value of indeterminate data consistent, some buggy code will now always break. However, this also changes some non-deterministic "Heisenbugs" into deterministic "Bohr bugs," e.g. returning a pointer to a local, stack-allocated variable will always break, instead of just when a signal intervenes between function return and pointer dereference. This can be beneficial as it may bring otherwise hard to find bugs to the surface. Conversely, secure deallocation may eliminate some bugs permanently (e.g. data is always initialized to zero as a programmer assumed).

Implementers of secure deallocation should consider this issue when deciding what value they wish to use for clearing memory. For example, matching the value the existing OS uses for clearing process pages (e.g. zero on Linux x86, `0xDEADBEEF` on AIX RS/6000), is a good heuristic for avoiding the introduction of new use-before-initialization bugs.

5 Data Lifetime Reduction

Evaluating the effectiveness of secure deallocation requires us to answer a variety of questions, including: How often do applications deallocate their own memory? Can we rely on incidental reuse and overwriting to destroy sensitive data—do we even need secure deallocation? What kind of delay can we expect between last use of data and its deallocation?

Using the conceptual framework introduced in section 3, we try to answer these questions using our example implementation of heap clearing and a variety of common workloads.

5.1 Measurement Tool

We created a tool for measuring data lifetime related events in standard user applications. It works by dynamically instrumenting programs to record all accesses to memory: all reads, writes, allocations, and deallocations. Using this information, we can generate precise numbers for ideal, natural, and secure deallocation lifetimes as well as other data properties like holes.

We based our tool on Valgrind [18], an open source debugging and profiling tool for user-level *x*86-Linux programs. It is particularly well-known as a memory debugger. It also supports a general-purpose binary instrumentation framework that allows it to be customized. With this framework, we can record timestamps for the events illustrated in Figure 1. We can compute various lifetime spans directly from these timestamps.

5.2 Application Workloads

We performed our experiments on a Linux *x*86 workstation, selecting applications where data lifetime concerns are especially important, or which lend insight into interesting dynamic allocation behavior:

- *Mozilla*, a popular graphical web client. We automated Mozilla v1.4.3 to browse through 10 different websites chosen for their mix of images, text layouts, CSS, and scripting.
- *Thunderbird*, a graphical mail client included as part of the Mozilla application suite. We set up Thunderbird 1.0 to automatically iterate through over 100 email messages that include text and images.
- `ssh`, a secure remote shell. Using OpenSSH 3.9p1, we scripted our `ssh` workload using `expect` to log into a `ssh` server, read mail in `pine`, edit some text with `emacs`, and walk through various directories.
- `sshd`, the secure shell daemon from OpenSSH v3.9p1. This is the server side of the `ssh` client test.
- *Python*, an interpreted, object-oriented language with garbage collection. Python and similar managed language runtimes are increasingly important for running applications, and they have data-lifetime properties characteristically different from applications that manually manage data. We used Python 2.4 to run a program that computes large primes.
- *Apache*, a web server. Our Apache workload serves a small corpus of static HTML and images using Apache 2.0.52. We automated a client to spend about an hour hitting these objects in sequence.
- `xterm`, a terminal emulator for X11. Inside XFree86's `xterm` 4.3.99.5(179), we ran a small client process that produces profuse output for about 45 minutes.
- `ls`, the canonical directory lister. Although `ls` does not obviously handle much sensitive data, its data lifetime characteristics give us some insight into how more non-GUI-centric applications might be expected to behave. This workload performs a recursive, long-formatted directory listing starting from the root of a large file system using GNU `ls` 5.0.

We omit detailed performance testing for these applications, due to the difficulty of meaningfully characterizing changes to interactive performance. Any performance penalties incurred were imperceptible. Performance of heap zeroing is analyzed in section 7.

5.3 Results

Table 1 summarizes the results of our experiments. We ran each application through our modified Valgrind, recorded timings for various memory events, and computed the resultant data lifetimes.

The table contains several statistics for each experiment. *Run Time* is the time for a single run and *Allocated* is the total amount of heap memory allocated during the run. *Written* is the amount of allocated memory that was written, and *Ideal Lifetime* is the ideal lifetime of the written bytes, calculated from first write to last read for every byte written. *Written & Freed* is the allocated memory that was first written and later deallocated, and *Secure Deallocation Lifetime* is the data lifetime obtained by an allocator that zeros data at time of free, as the time from first write to deallocation. Finally, *Written, Freed, & Overwritten* is the allocated bytes that were written and deallocated and later overwritten, with *Natural Lifetime* the data lifetime obtained with no special effort, as the time from first write to overwrite.

The GUI workloads Mozilla and Thunderbird are visually separated in the table because their data lifetime characteristics differ markedly from the other workloads, as we will discuss further in section 5.5 below.

One thing to note about the binary instrumentation framework Valgrind provides is that it does tend to slow down CPU-bound programs, dilating the absolute numbers for the lifetime of data. However, the relative durations of the ideal, secure deallocation, and natural lifetimes are still valid; and in our workloads, only the Python experiment was CPU-bound.

5.4 Natural Lifetime is Inadequate

Our results indicate that simply waiting for applications to overwrite data in the course of their normal execution (i.e. natural lifetime) produces extremely long and unpredictable lifetimes.

To begin, many of our test applications free most of the memory that they allocate, yet never overwrite much of the memory that they free. For example, the Mozilla workload allocates 135 MB of heap, writes 96 MB of it,

Application	Run Time	Allocated	Written	Ideal Lifetime mean	Ideal Lifetime stddev	Written & Freed	Secure Deallocation Lifetime mean	Secure Deallocation Lifetime stddev	Written, Freed, & Overwritten	Natural Lifetime mean	Natural Lifetime stddev
Mozilla	23:04	135 MB	96 MB	11 s	68 s	94 MB	21 s	83 s	80 MB	40 s	105 s
Thunderbird	44:20	232 MB	155 MB	5 s	86 s	153 MB	10 s	120 s	143 MB	34 s	162 s
ssh	30:55	6 MB	6 MB	0 s	0 s	6 MB	0 s	0 s	6 MB	7 s	73 s
sshd	46:19	6 MB	6 MB	0 s	0 s	6 MB	0 s	0 s	6 MB	5 s	120 s
Python	46:14	352 MB	232 MB	24 s	53 s	232 MB	23 s	53 s	214 MB	59 s	131 s
Apache	1:01:21	57 MB	5 MB	0 s	0 s	5 MB	0 s	0 s	5 MB	0 s	0 s
xterm	46:13	8 MB	8 MB	1 s	2 s	0 MB	1 s	2 s	0 MB	3 s	53 s
ls	46:02	86 MB	23 MB	1 s	13 s	22 MB	2 s	15 s	20 MB	65 s	326 s

Table 1: Data lifetime statistics for heap allocated memory. *Allocated* is the total amount of heap memory allocated during each run. *Written* is the amount of allocated memory that was actually written, and *Ideal Lifetime* is the lifetime this written data would have if it were zeroed immediately after the last time it was read. *Written & Freed* is allocated bytes that were written and later freed, with *Secure Deallocation Lifetime* the lifetime of this data when it is zeroed at deallocation time. Finally, *Written & Freed & Overwritten* is allocated bytes that were written and freed, then later reallocated and overwritten by the new owner, with *Natural Lifetime* the lifetime of this data.

frees about 94 MB of the data it wrote, yet 14 MB of that freed data is never overwritten.

There are several explanations for this phenomenon. For one, programs occasionally free data at or near the end of their execution. Second, sometimes one phase of execution in a program needs more memory than later phases, so that, once freed, there is no need to reuse memory during the run. Third, allocator fragmentation can artificially prevent memory reuse (see 3.2 for an example).

Our data shows that *holes*, that is, data that is reallocated but never overwritten, are also important. Many programs allocate much more memory than they use, as shown most extremely in our workloads by Python, which allocated 120 MB more memory than it used, and Apache, which allocated over 11 times the memory it used. This behavior can often result in the lifetime of a block of memory extending long past its time of reallocation.

The natural lifetime of data also varies greatly. In every one of our test cases, the natural lifetime has a higher standard deviation than either the ideal or secure deallocation lifetime. In the `xterm` experiment, for example, the standard deviation of the natural lifetime was over 20 times that of the secure deallocation lifetime.

Our experiments show that an appreciable percentage of freed heap data persists for the entire lifetime of a program. In our Mozilla experiment, up to 15% of all freed (and written to) data was never overwritten during the course of its execution. Even in programs where this was not an appreciable percentage, non-overwritten data still amounted to several hundred kilobytes or even megabytes of data.

5.5 Secure Deallocation Approaches Ideal

We have noted that relying on overwrite (natural lifetime) to limit the life of heap data is a poor choice, often leaving data in memory long after its last use and providing widely varying lifetimes. In contrast, secure deallocation very consistently clears data almost immediately after its last use, i.e. it very closely approximates ideal lifetime.

Comparing the *Written* and *Written & Freed* columns in Table 1, we can see that most programs free most of the data that they use. Comparing *Ideal Lifetime* to *Secure Deallocation Lifetime*, we can also see that most do so promptly, within about a second of the end of the ideal lifetime. In the same cases, the variability of the ideal and secure deallocation lifetimes are similar.

Perhaps surprisingly, sluggish performance is not a common issue in secure heap deallocation. Our Python experiment allocated the most heap memory of any of the experiments, 352 MB. If all this memory is freed and zeroed at 600 MB/s, the slowest zeroing rate we observed (see section 7.1), it would take just over half a second, an insignificant penalty for a 46-minute experiment.

GUI Programs Table 1 reveals that GUI programs often delay deallocation longer than other programs, resulting in a much greater secure deallocation lifetime than others.

One reason for this is that GUI programs generally use data for a short period of time while rendering a page of output, and then wait while the user digests the information. During this period of digestion, the GUI program must retain the data it used to render the page in case the window manager decides the application should refresh its contents, or if the user scrolls the page.

Consequently, the in-use period for data is generally quite small, only as much to render the page, but the deallocation period is quite large because data is only deallocated when, e.g., the user moves on to another webpage. Even afterward, the data may be retained because, for user-friendliness, GUI programs often allow users to backtrack, e.g. via a "back" button.

6 Kernel Clearing: A Case Study

The previous section examined data lifetime reduction for a single allocator, the heap, and showed it provided a significant quantitative reduction in lifetime for data in general. In contrast, this section takes a more qualitative approach, asking whether our implementation promptly removes *particular* sensitive data from our entire system. In answer, we provide an in-depth case study of data lifetime reduction in several real applications' treatment of passwords, as they pass through our kernel.

6.1 Identifying Sensitive Data

We used TaintBochs, our tool for measuring data lifetime, to evaluate the effectiveness of our kernel clearing. TaintBochs is a whole-system simulator based on the open source *x*86 simulator Bochs, version 2.0.2. We configured Bochs to simulate a PC with an 80386 CPU, 8 GB IDE hard disk, 32 MB RAM, NE2000-compatible Ethernet card, and VGA video.

TaintBochs provides an environment for tagging sensitive data with "taint" information at the hardware level and propagating these taints alongside data as it moves through the system, allowing us to identify where sensitive data has gone. For example, we can taint all incoming keystrokes used to type a password as tainted, and then follow these taints' propagation through kernel tty buffers, X server event queue, and application string buffers.

TaintBochs and its analysis framework is fully described in our previous work [5].

6.2 Augmenting Kernel Allocators

To augment kernel allocators to provide secure deallocation, we began with large, page-granular allocations, handled by the Linux page allocator. We added a bit to the page structure to allow pages to be individually marked *polluted*, that is, containing (possibly) sensitive data. This bit has an effect only when a page is freed, not while it is still in use.

Whereas an unmodified Linux 2.4 kernel maintains only one set of free pages, our modified kernel divides free pages into three pools. The *not-zeroed pool* holds pages whose contents are not sensitive but not (known to be) zeroed. The *zeroed pool* holds pages that have been cleared to zeros. The *polluted pool* holds free pages with sensitive contents. The code for multiple pools was inspired by Christopher Lameter's prezeroing patches for Linux 2.6 [17].

Data lifetime is limited by introducing the *zeroing daemon*, a kernel thread that wakes up periodically to zero pages that have been in the polluted pool longer than a configurable amount of time (by default 5 seconds). Thus, our clearing policy is a "deadline" policy, ensuring that polluted pages are cleared within approximately the configured time. This policy is easy to understand: after a polluted page is freed, we know that it will be cleaned within a specified amount of time. It is also simple to implement, by maintaining a linked list of freed polluted pages ordered by time of deallocation.

Appendix A describes in detail how allocation requests are satisfied from these page pools. It also describes our changes to clear kernel I/O buffers as soon as they are no longer needed.

6.3 Application Workloads

6.3.1 Apache and Perl

We tracked the lifetime of the password through the Apache web server to a Perl subprocess. Our CGI script uses Perl's CGI module to prompt for and accept a password submitted by the user. The script hashes the password and compares it to a stored hash, then returns a page that indicates whether the login was successful.

With an unmodified kernel, we found many tainted regions in kernel and user space following the experiments. The kernel contained tainted packet buffers allocated by the NE2000 network driver and a pipe buffer used for communication between Apache and the CGI script. Apache had three tainted buffers: a dynamically allocated buffer used for network input, a stack-allocated copy of the input buffer used by Perl's CGI module, and a dynamically allocated output buffer used to pass it along to the CGI subprocess. Finally, Perl has a tainted file input buffer and many tainted string buffers. All of these buffers contained the full password in cleartext (except that some of the tainted Perl string buffers contained only hashed copies).

Our modified kernel cleared all of the Perl taints following Perl's termination. When the Apache process terminated, those taints also disappeared. (Apache can be set up to start a separate process for each connection, so kernel-only support for limiting data lifetime may even, in some cases, be a reasonable way to limit web server data lifetime in the real world.)

A few tainted variables did remain even in our modified kernel, such as:

- The response from the CGI process depends on the correctness of the password, so the response itself is tainted. Perl allocates a buffer whose size is based on the length of the response, and the size of the buffer factors into the amount of memory requested from the system with the `sbrk` system call. Therefore, the kernel's accounting of the number of committed VM pages (`vm_committed_space`) becomes tainted as well.

- The Linux TCP stack, as required by TCP/IP RFCs, tracks connections in the `TIME_WAIT` state. The tracking info includes final sequence numbers. Because the sending-side sequence number is influenced by the length of the tainted response, it is itself tainted.

- Apache's log entries are tainted because they include the length of the tainted response. Thus, one page in Linux's page cache was tainted.

Assuming that the length of the response is not sensitive, these tainted variables cannot be used to determine sensitive information, so we disregarded them.

6.3.2 Emacs

Our second effectiveness experiment follows the lifetime of a password entered in Emacs's shell mode. In shell mode, an Emacs buffer becomes an interface to a Unix shell, such as `bash`, running as an Emacs subprocess. Shell mode recognizes password prompts and reads their responses without echoing. We investigated the data lifetime of passwords entered into `su` in shell mode. We typed the password, then closed the root shell that it opened and the outer shell, then exited from Emacs and logged off.

With an unmodified kernel, several regions in kernel and user space were tainted. The kernel pseudo-random number generator contained the entire user name and password, used for mixing into the PRNG's entropy pool but never erased. Kernel tty buffers did also, in both the interrupt-level "flip" buffer and the main tty buffer, plus a second tty buffer used by Emacs to pass keyboard input to its shell subprocess.

In the Emacs process, tainted areas included: a global circular queue of Emacs input events, the entire password as a Lisp string, a set of individual Lisp character strings containing one password character each, one copy on the Emacs stack, and a global variable that tracks the user's last 100 keystrokes. (Much of Emacs is implemented in Lisp.) No copies were found in the `su` process, which seems to do a good job of cleaning up after itself.

Our modified kernel cleared all of these taints when Emacs was terminated. (The PRNG's entropy pool was still tainted, but it is designed to resist attempts to recover input data.)

7 Performance Overhead

Programmers and system designers seem to scoff at the idea of adding secure deallocation to their systems, supposing the overheads to be unacceptable. However, these fears are largely unfounded.

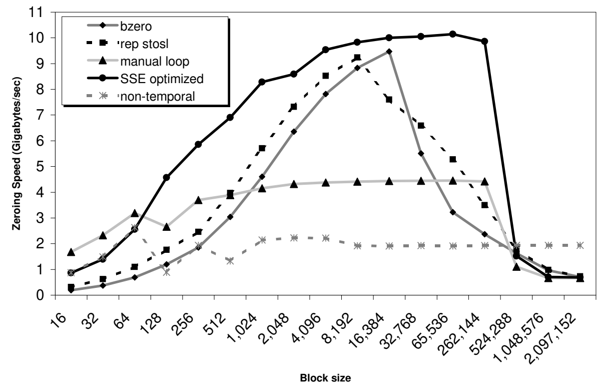

Figure 5: Comparison of speed for different zeroing techniques on a 2.40 GHz Pentium 4. This chart shows the speed in GB/s of zeroing a single block of n bytes repeatedly until 4 GB of zeros have been written. Block sizes are powers of two and blocks are aligned on block boundaries. Refer to the text for a description of each zeroing method.

Data that are in cache and properly aligned can be zeroed rapidly, as we will show. In fact, applications seldom allocate and free enough data to make any appreciable change to their running time.

In this section we show that with careful implementation, zeroing can generally be done with nominal overhead. We show experimentally that user level clearing can be achieved with minimal impact on a wide range of applications. And similarly, that kernel clearing can be performed without significantly impacting either CPU or I/O intensive application performance.

All experiments were run on an *x*86 Linux platform with 1 GB of memory and a 2.40 GHz Pentium 4.

7.1 Implementing Zeroing

All the allocators on our system dole out blocks of data aligned to at least 4-byte boundaries. `malloc` by default aligns blocks to 8-byte boundaries. Also by default, GCC aligns stack frames on 16-byte boundaries. Given common application allocation patterns, most heap and stack data freed and reallocated are recently used, and thus also likely in cache.

These alignment and cache properties allow even modest machines to zero data at blindingly fast speeds. Figure 5 illustrates this fact for five different methods of zeroing memory:

`bzero`, an out-of-line call to glibc's `bzero` function.

`rep stosl`, inline assembly using an *x*86 32-bit string store instruction.

	Running Time	Max In-Flight `mallocs`	Total `mallocs`	Total `frees`	Total Bytes `freed`	Free Rate (bytes/sec)
164.gzip	3m:16s	299	436,222	436,187	110,886,084	565,745
175.vpr	5m:12s	68,036	107,659	103,425	5,355,300	17,164
176.gcc	2m:27s	25,196	110,315	93,823	545,877,388	3,713,451
197.parser	4m:42s	7	153	147	1,111,236	3,940
252.eon	11m:25s	2,397	5,283	4,125	380,996	556
253.perlbmk	3m:26s	2,397,499	31,361,153	30,847,487	6,368,737,804	30,916,202
255.vortex	3m:37s	472,711	4,622,368	4,400,552	1,934,800,720	8,916,132
300.twolf	9m:15s	105,210	574,572	492,729	16,759,620	30,197
firefox	6s	90,327	218,501	219,936	74,545,704	12,081,962

Table 2: Non-trivial allocations by programs in our zero-on-free heap experiment. *Max In-Flight* `mallocs` gives the maximum number of memory allocations alive at the same time. All other numbers are aggregates over the entire run. Runs with under 100 K of `freed` data are not shown.

manual loop, inline assembly that stores 32 bits at a time in a loop.

SSE optimized, a out-of-line call to our optimized zeroing function that uses a loop for small objects and SSE 128-bit stores for large objects.

non-temporal, a similar function that substitutes non-temporal writes for ordinary 128-bit SSE stores. (Non-temporal writes bypass the cache, going directly to main memory.)

For small block sizes, fixed overheads dominate. The manual loop is both inlined and very short, and thus fastest. For block sizes larger than the CPU's L2 cache (512 kB on our P4), the approximately 2 GB/s memory bus bandwidth limits speed. At intermediate block sizes, 128-bit SSE stores obtain the fastest results.

Zeroing unused data can potentially pollute the CPU's cache with unwanted cache lines, which is especially a concern for periodic zeroing policies where data is more likely to have disappeared from cache. The *non-temporal* curve shows that, with non-temporal stores, zeroing performance stays constant at memory bus bandwidth, without degradation as blocks grow larger than the L2 cache size. Moreover, the speed of non-temporal zeroing is high, because cleared but uncached data doesn't have to be brought in from main memory.

When we combine these results with our observations about common application memory behavior, we see that zeroing speeds far outpace the rate at which memory is allocated and freed. Even the worst memory hogs we saw in Table 1 only freed on the order of hundreds of MB of data throughout their entire lifetime, which incurs only a fraction of a second of penalty at the slowest, bus-bandwidth zeroing rate (2 GB/s).

7.2 Measuring Overhead

To evaluate the overheads of secure deallocation, we ran test workloads from the SPEC CPU2000 benchmark suite, a standardized CPU benchmarking suite that contains a variety of user programs. By default, the tests contained in the SPEC benchmarks run for a few minutes (on our hardware); it lacks examples of long-lived GUI processes or server processes, which have especially interesting data lifetime issues.

However, we believe that because the SPEC benchmark contains many programs with interesting memory allocation behavior (including Perl, GCC, and an object-oriented database), that the performance characteristics we observe for SPEC apply to these other programs as well. In addition to this, we ran an experiment with the Firefox 1.0 browser. We measured the total time required to startup a browser, load and render a webpage, and then shut-down.

7.2.1 Heap Clearing Overhead

We implemented a zero-on-free heap clearing policy by creating a modified libc that performs zeroing when heap data is deallocated. Because we replaced the entire libc, modifying its internal memory allocator to do the zeroing, we are able to interpose on deallocations performed within the C library itself, in addition to any done by the application. To test heap clearing, we simply point our dynamic linker to this new C library (e.g. via `LD_LIBRARY_PATH`), and then run our test program.

For each program, we performed one run with an unmodified C library, and another with the zero-on-free library. Figure 6 gives the results of this experiment, showing the relative performance of the zero-on-free heap allocator versus an unmodified allocator. Surprisingly, zero-on-free overheads are less than 7% for all tested applications, despite the fact that these applications allocate hundreds or thousands of megabytes of data during their lifetime (as shown in Table 2).

An interesting side-effect of our heap clearing experiment is that we were able to catch a use-after-free bug in one of the SPEC benchmarks, `255.vortex`. This program attempted to write a log message to a stdio `FILE`

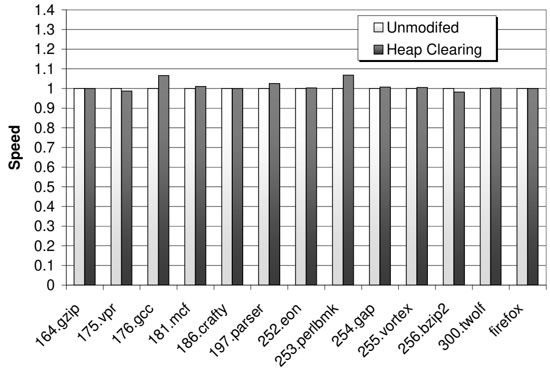

Figure 6: Heap clearing has little performance impact. This chart shows the relative performance of an unmodified glibc 2.3 heap allocator versus the same allocator modified to zero at free time in a set of user programs. The unmodified runs are normalized to 1.0. Zero-on-free overheads are less than 7% for all tested applications.

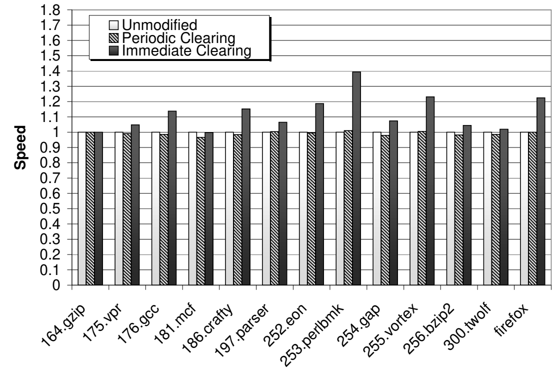

Figure 7: Comparing stack clearing overheads. This chart shows the relative performance of our workload with three strategies: an *unmodified* run with no stack clearing used as a baseline, a *periodic* run with OS scheduled stack zeroing (configured to 5 second intervals) as well as our other kernel zeroing features, and a *immediate* run with immediate stack zeroing on every function return. Periodic zeroing has little performance overhead. Immediate zeroing has more of a penalty, which may be acceptable to security-conscience applications.

after it had closed the file. Removing the `fclose` call fixed the bug, but we had to touch the sources to do this. We don't believe this impacted our performance results.

7.2.2 Stack Clearing Overhead

We implemented stack clearing for applications by modifying our OS to periodically zero the free stack space in user processes that have run since the last time we cleared stacks. We do so by writing zero bytes from the user's stack pointer down to the bottom of the lowest page allocated for the stack.

Figure 7 gives the results of running our workload with periodic stack clearing (configured with a period of 5 seconds) plus our other kernel clearing changes. Just like heap clearing, periodic stack clearing had little impact on application performance, with less than a 2% performance increase for all our tests.

Immediate Stack Clearing For those applications with serious data lifetime concerns, the delay inherent to a periodic approach may not be acceptable. In these cases, we can perform an analog of our heap clearing methodology by clearing stack frames immediately when they are deallocated.

We implemented immediate stack clearing by modifying GCC 3.3.4 to emit a stack frame zeroing loop in every function epilogue. To evaluate the performance impact of this change, we compared the performance of a test suite compiled with an unmodified GCC 3.3.4 against the same test suite compiled with our modified compiler.

Figure 7 gives the results of this experiment. We see that overheads are much higher, generally between 10% and 40%, than for periodic scheduled clearing. Clearly, such overheads are significant, though they may be acceptable for applications where data lifetime is an utmost concern.

7.3 Kernel Clearing Overhead

Batch Workload We used Linux kernel builds to stress our page zeroing changes. A kernel build starts many processes, each of which modifies many heap, stack, and static data pages not backed by files. The kernel considers all of these polluted and zeros them within five seconds of deallocation.

With the ordinary kernel running, three kernel builds took 184, 182, and 183 seconds, for an average of 183 seconds. With the zeroing kernel, the runs took 188, 184, and 184 seconds, for an average of 185 seconds, approximately a 1% penalty.

The kernel build zeroed over 1.2 million pages (about 4.8 GB) per run. The actual number of polluted pages generated was much larger than that, but many of those pages did not need to be zeroed because they could be entirely overwritten by pages brought into the page cache from disk or by copies of pages created when triggering copy-on-write operations. (As described in section A.2, we prefer to overwrite polluted data whenever possible.)

Network Workload We evaluated the overhead of zeroing by benchmarking performance on 1 Gbps Ethernet, achieving up to 500 Mbps utilization for large blocks. We found latency, bandwidth, and CPU usage to be indistinguishable between our zeroing kernel and unmodified kernels.

We evaluated the overhead of zeroing network packets using NetPIPE [20], which bounces messages of increasing sizes between processes running on two machines. We configured it to send blocks of data over TCP, in both directions, between a machine running our zeroing kernel and a machine running an unmodified Linux kernel. We then compared its performance against the same test run when both machines were configured with unmodified Linux kernels.

Considering the performance of zeroing depicted in Figure 5, our results are not too surprising. Assuming we zero a buffer sized at the maximum length of an Ethernet frame (1500 bytes), our performance numbers suggest we should be able to zero one second's worth of Gigabit Ethernet traffic in between about 7 ms and 32 ms, depending on the technique used. Such low overheads are well below the normal variance we saw across measurements.

8 Future Work

We are currently looking at the performance trade-offs involved with kernel zeroing, specifically how to parameterize and tune the scheduling of kernel zeroing to provide predictable latency and throughput overheads under diverse workloads.

Examining the impact of parallelism is an interesting direction for inquiry. The move to multi-core processors will provide a great deal of additional available parallelism to further diminish the impact of zeroing.

Providing explicit OS support for reducing data lifetime, for example "ephemeral memory" that automatically zeroes its contents after a certain time period and thus is secure in the face of bugs, memory leaks, etc., is another area for future investigation.

A wide range of more specialized systems could benefit from secure deallocation. For example, virtual machine monitors and programming language runtimes.

So far we have primarily considered language environments that use explicit deallocation, such as C, but garbage-collected languages pose different problems that may be worthy of additional attention. Mark-and-sweep garbage collectors, for example, prolong data lifetime at least until the next GC, whereas reference-counting garbage collectors may be able to reduce data lifetime below that of secure deallocation.

9 Related Work

Our previous work explored the problem of data lifetime using whole system simulation with TaintBochs [5]. We focused on mechanisms for analyzing the problem, demonstrated its frequency in real world applications, and showed how programmers could take steps to reduce data lifetime. Whereas this earlier work looked at how sensitive data propagates through memory over relatively short intervals (on the order of seconds), the current paper is concerned with how long data survives before it is overwritten, and with developing a general-purpose approach to minimizing data lifetime.

We explored data lifetime related threats and the importance of proactively addressing data lifetime at every layer of the software stack in a short position paper [7].

The impetus for this and previous work stemmed from several sources.

Our first interest was in understanding the security of our own system as well as addressing vulnerabilities observed in other systems due to accidental information leaks, e.g. via core dumps [15, 16, 14, 13] and programmer error [1].

A variety of previous work has addressed specific symptoms of the data lifetime problem (e.g. leaks) but to the best of our knowledge none has offered a general approach to reducing the presence of sensitive data in memory. Scrash [4] deals specifically with the core dump problem. It infers which data in a system is sensitive based on programmer annotations to allow for crash dumps that can be shipped to the application developer without revealing users' sensitive data.

Previous concern about sensitive data has addressed keeping it off of persistent storage, e.g. Provos's work on encrypted swap [19] and work by Blaze on encrypted file systems [3]. Steps such as these can greatly reduce the impact of sensitive data that has leaked to persistent storage.

The importance of keeping sensitive data off of storage has been emphasized in work by Gutmann [9], who showed the difficulty of removing all remnants of sensitive data once written to disk.

Developers of cryptographic software have long been aware of the need for measures to reduce the lifetime of cryptographic keys and passwords in memory. Good discussions are given by Gutmann [10] and Viega [22].

10 Conclusion

The operating systems and applications responsible for handling the vast majority of today's sensitive data, such as passwords, social security numbers, credit card numbers, and confidential documents, take little care to ensure this data is promptly removed from memory. The result is increased vulnerability to disclosure during attacks or due to accidents.

To address this issue, we argue that the strategy of secure deallocation, zeroing data at deallocation or within a short, predictable period afterward, should become a standard part of most systems.

We demonstrated the speed and effectiveness of secure deallocation in real systems by modifying all major allocation systems of a Linux system, from compiler stack, to `malloc`-controlled heap, to dynamic allocation in the kernel, to support secure deallocation.

We described the data life cycle, a conceptual framework for understanding data lifetime, and applied it to analyzing the effectiveness of secure deallocation.

We further described techniques for measuring effectiveness and performance overheads of this approach using whole-system simulation, application-level dynamic instrumentation, and system and network benchmarks.

We showed that secure deallocation reduces typical data lifetime to 1.35 times the minimum possible data lifetime. In contrast, we showed that waiting for data to be overwritten often produces data lifetime 10 to 100 times longer than the minimum, and that on normal desktop systems it is not unusual to find data from dead processes that is days or weeks old.

We argue that these results provide a compelling case for secure deallocation, demonstrating that it can provide a measurable improvement in system security with negligible overhead, while requiring no programmer intervention and supporting legacy applications.

11 Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. 0121481 and a Stanford Graduate Fellowship.

References

[1] O. Arkin and J. Anderson. Etherleak: Ethernet frame padding information leakage. http://www.atstake.com/research/advisories/2003/atstake_etherleak_repor%t.pdf.

[2] Arkoon Security Team. Information leak in the Linux kernel ext2 implementation. http://arkoon.net/advisories/ext2-make-empty-leak.txt, March 2005.

[3] M. Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[4] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proceedings of the 11th USENIX Security Symposium*, August 2003.

[5] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 12th USENIX Security Symposium*, 2004.

[6] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the Eighteenth ACM symposium on Operating Systems Principles*, pages 57–72, New York, NY, USA, 2001. ACM Press.

[7] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *Proc. 11th ACM SIGOPS European Workshop*, September 2004.

[8] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 313–323. ACM Press, 1998.

[9] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium*, July 1996.

[10] P. Gutmann. Software generation of practically strong random numbers. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[11] T. Hamilton. 'Error' sends bank files to eBay. Toronto Star, Sep. 15, 2003.

[12] S. Hand. Data lifetime bug in VMM. Personal communications.

[13] Coredump hole in imapd and ipop3d in slackware 3.4. http://www.insecure.org/sploits/slackware.ipop.imap.core.html.

[14] Security Dynamics FTP server core problem. http://www.insecure.org/sploits/solaris.secdynamics.core.html.

[15] Solaris (and others) ftpd core dump bug. http://www.insecure.org/sploits/ftpd.pasv.html.

[16] Wu-ftpd core dump vulnerability. http://www.insecure.org/sploits/ftp.coredump2.html.

[17] C. Lameter. Prezeroing V2 [0/3]: Why and when it works. Pine.LNX.4.58.0412231119540.31791@schroedinger.engr.sgi.com, December 2004. Linux kernel mailing list message.

[18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[19] N. Provos. Encrypting virtual memory. In *Proceedings of the 10th USENIX Security Symposium*, pages 35–44, August 2000.

[20] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. NetPIPE: A network protocol independent performance evaluator. http://www.scl.ameslab.gov/netpipe/paper/full.html.

[21] The Mozilla Organization. Javascript "lambda" replace exposes memory contents. http://www.mozilla.org/security/announce/mfsa2005-33.html, 2005.

[22] J. Viega. Protecting sensitive data in memory. http://www-106.ibm.com/developerworks/security/library/s-data.html?dwzo%ne=security.

[23] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[24] C. A. Waldspurger. Memory resource management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

A Kernel Support for Secure Deallocation

This section describes strategies that we found useful for reducing data lifetime in the Linux kernel. Some improve the performance of secure allocation in cases where we have additional semantic knowledge; the rest work to reduce the lifetime of data that is long-lived from the point of view of the kernel allocators, such as data stored in circular queues. We believe that these strategies will prove useful in other long-lived programs.

A.1 What Data is Sensitive?

Section 6.2 described kernel mechanisms for labeling sensitive data. Once these mechanisms are available, we need a policy to distinguish sensitive data from other data. The policy for our prototype implementation was based on a few rules of thumb. First, we considered all user input, such as keyboard and mouse data, all network traffic, and all user process data, to be sensitive.

However, we consider data or metadata read from or written to a file system not sensitive, because its data lifetime is already extended indefinitely simply because it has been written to disk [9]. (For simplicity of prototyping, we ignore the possibility of encrypted, network, in-memory, or removable media file systems, as well as temporary files.) Thus, because pages in shared file mappings (e.g. code, read-only data) are read from disk, they are not considered sensitive even though they belong to user processes. On the other hand, anonymous pages (e.g. stack, heap) are not file system data and therefore deemed sensitive.

We decided that the location of sensitive data is not itself sensitive. Therefore, pointers in kernel data structures are never considered sensitive. Neither are page tables, scheduling data, process ids, etc.

A.2 Allocator Optimizations

Section 6.2 described the division of kernel allocators into pools and the use of a zeroing daemon to delay zeroing. However, the kernel can sometimes avoid doing extra work, or clear polluted pages more quickly, by using advice about the intended use of the page provided by the allocator's caller:

- The caller may request a zeroed page. The allocator returns a zeroed page, if one is available. Otherwise, it zeroes and returns a polluted page, if available, rather than a not-zeroed page. This preference reduces polluted pages at no extra cost (because a page must be zeroed in any case).

- The caller may indicate that it will be clearing the entire page itself, e.g. that the page will be used for buffering disk data or receiving a copy of a copy-on-write page. In this case the allocator returns a polluted page if available, again reducing polluted pages without extra cost. In this case the caller is responsible for clearing the page; the allocator does not zero it.

- If the caller has no special requirements, the allocator prefers not-zeroed pages, then zeroed pages, then polluted pages. If a polluted page is returned, then it must be zeroed beforehand because the caller may not overwrite the entire page in an punctual fashion.

We applied changes similar to those made to the page allocator to the slab allocator as well. Slabs do not have a convenient place to store a per-block "polluted" bit, so the slab allocator instead requires the caller to specify at time of free whether the object is polluted.

A.3 Oversized Allocations Optimizations

Without secure deallocation, allocating or freeing a buffer costs about the same amount of time regardless of the buffer's size. This encourages the common practice of allocating a large, fixed-size buffer for temporary use, even if only a little space is usually needed. With secure deallocation, on the other hand, the cost of freeing a buffer increases linearly with the buffer's size. Therefore, a useful optimization is to clear only that part of a buffer that was actually used.

We implemented such an optimization in the Linux network stack. The stack uses the slab allocator to allocate packet data, so we could use the slab allocator's pollution mechanism to clear network packets. However, the blocks allocated for packets are often much larger than the actual packet content, e.g. packets are often less than 100 bytes long, but many network drivers put each packet into 2 KB buffer. We improved performance by zeroing only packet data, not other unused bytes.

Filename buffers are another place that this class of optimization would be useful. Kernel code often allocates an entire 4 KB page to hold a filename, but usually only a few bytes are used. We have not implemented this optimization.

A.4 Lifetime Reduction in Circular Queues

As already discussed in section 4.3, circular queues can extend data lifetime of their events, if new events are not added rapidly enough to replace those that have been removed in a reasonable amount of time. We identified several examples of such queues in the kernel, including "flip buffers" and tty buffers used for keyboard and serial port input, pseudoterminal buffers used by terminal emulators, and the entropy batch processing queue used by the Linux pseudo-random number generator. In each case, we fixed the problem by clearing events held in the queue at their time of removal.