Data Lifetime is a Systems Problem

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

Abstract

As sensitive data lifetime (i.e. propagation and duration in memory) increases, so does the risk of exposure. Unfortunately, this issue has been largely overlooked in the design of most of today's operating systems, libraries, languages, etc. As a result, applications are likely to leave the sensitive data they handle (passwords, financial and military information, etc.) scattered widely over memory, leaked to disk, etc. and left there for an indeterminate period of time. This greatly increases the impact of a system compromise.

Dealing with data lifetime issues is currently left to application developers, who largely overlook them. Security-aware developers who attempt to address them (e.g. cryptographic library writers) are stymied by the limitations of the operating systems, languages, etc. they rely on. We argue that data lifetime is a systems issue which must be recognized and addressed at all layers of the software stack.

1 Introduction

System compromise is an inevitable part of computer security. Despite our best efforts, software will continue to have exploitable bugs and misconfigurations, and malicious parties will continue to gain physical access to system hardware. Given this, it is essential that we build systems to minimize the impact of compromises.

Reducing the risk of sensitive data exposure is an essential part of this process. We can reduce risk by minimizing the amount of sensitive data (e.g. encryption keys, passwords, military and financial documents) in a system at any point in its execution. This also reduces the risk of accidental leaks.

We refer to the duration of time given sensitive data remains in a system and how widely it is propagated as *data lifetime*. As lifetime increases so does the risk of exposure.

Data lifetime related problems are more subtle than simple vulnerabilities. Unlike other software weaknesses (e.g. buffer overflows), data lifetime problems are often not immediately exploitable. Instead, they tend to increase the impact of other exploits. Consequently, data lifetime issues receive much less attention than more obvious vulnerabilities.

Due to this lack of attention, data lifetime is rarely addressed in operating systems, programming languages, etc. Thus, the task of dealing with data lifetime issues is left to application developers. Unfortunately, only the most sophisticated developers take these issues into account. Even then, the limitations of today's languages and operating systems make their solutions incomplete, error prone, and fragile. In practice, software that handles most of the world's sensitive data is developed in complete ignorance of data lifetime issues.

We argue that data lifetime problems cannot be addressed solely by application developers. Instead they must be addressed by design in every layer of the software stack by providing safer defaults and flexible mechanisms for dealing with sensitive data. Only a whole system approach can provide an effective solution to data lifetime problems.

We begin our discussion with a simple model of data lifetime, an example of data lifetime in a real system, and an examination of potential threats. Next, we consider how current systems fail to address data lifetime problems. Finally, we discuss how data lifetime problems can be addressed, and the design trade-offs involved with reducing data lifetime.

2 The Data Lifetime Problem

We define the lifetime of sensitive data in terms of two components: *where* copies of this data end up, and *how long* these copies survive before being cleared or otherwise destroyed (i.e. their durations). Location dictates potential threats, i.e. the methods by which an attacker can eventually recover the data. Duration defines the window of opportunity for an attack. Increased duration also increases opportunity for propagation, e.g. longer-lived data is more likely to be paged to disk.

We can define the *lifetime* L of a piece of sensitive data as a set of tuples in which each tuple represents a unique copy of the data. Copies need not be identical, e.g. a password may have a variety of encodings in its lifetime: Unicode, ASCII, UTF-8, keyboard scan codes, base-64 encoded, etc. Each tuple $\ell \in L$ takes the form $\langle address, birth, death \rangle$, where *address* is the copy's location, *birth* is the copy's time of creation, and *death* is the time when a copy becomes inaccessible e.g. for an encrypted copy of data either the key was destroyed or the data was overwritten. We can then define the *duration* of a data lifetime L as

$$\max_{\ell \in L} \ell_{death} - \min_{\ell \in L} \ell_{birth},$$

and the *propagation* of a data lifetime L to be $|L|$, that is, the number of copies of L's data.

2.1 Propagation

Sensitive data propagates through many parts of a system. For example, in a recent study we traced a password typed into Mozilla under Linux on its journey through a system to a wide range of locations [2]:

1. *Interrupt context keyboard queue:* Linux reads keystrokes during a hardware interrupt and appends them to a circular queue for processing.
2. *Process "tty" buffer:* The OS copies characters from the interrupt-context queue into a tty buffer.
3. *Window system event queue:* The X server reads characters from the tty buffer into its own event queue.
4. *Network buffer:* The X server sends the characters to the X client over a Unix domain socket. The kernel copies them into a kernel network buffer.
5. *Widget buffer:* The GTK+ library reads the characters into a buffer in the password field widget on Mozilla's behalf.
6. *String buffers:* Mozilla and its windowing library make many copies of password data on the heap as they pass it around and eventually to the remote host.

None of the copies listed above were erased when no longer needed. Instead, they were erased only incidentally as memory was reclaimed and reused for some other purpose.

Data can propagate to persistent storage through a wide range of mechanisms largely outside of programmer control, including the following:

1. *Core dump:* Under Unix-like systems, core dumps contain the entire virtual memory image of a process, including sensitive data. Under Windows, "Dr. Watson" produces similar output.
2. *Hibernation:* Laptops and some desktop systems support "hibernation," a kind of sleep mode that writes the machine's entire physical memory to disk, sensitive data included. Hibernation data is particularly worrisome because it is usually written to a dedicated disk partition, making erasure unlikely.
3. *Checkpointing:* Systems for process checkpointing (and migration) typically write the entire memory image of a process to disk. VMware Workstation and other virtual machine monitors can also suspend the state of a virtual machine to a "checkpoint" file. Sensitive data in the checkpoint can be read from disk, or from network traffic if the checkpoint is transmitted in cleartext.
4. *Paging:* Virtual memory systems can leak significant amounts of data to disk through paging. Even "pinned" kernel memory can be paged if the OS is run inside a virtual machine monitor such as VMware Workstation. Due to increasing memory sizes, the amount of data paged to disk is typically decreasing, but this also implies that paged out data is being overwritten less frequently.
5. *Application Specific:* A wide range of application specific mechanisms for checkpointing, logging, serialization etc. can leak sensitive data to disk. Often this occurs through unintended feature interactions within an application or across components, or through use of code for an application unanticipated by developers.

Propagation to persistent storage greatly increases risk as data is less likely to be overwritten, persists beyond system reboots, and can be subject to recovery through direct physical access even if overwritten [4]. Network storage can lead to leaks over the network and propagation across machines.

2.2 Threats

Data lifetime issues are an important consideration in a wide range of threat models:

1. *Online remote attacks:* Remote exploits that can expose the data of a single user or the entire system (i.e. memory and disk) are common. This class of attack is increasingly important as system uptimes increase and increased memory sizes reduce the frequency that data is overwritten. A

scenario where data sits in memory for days or even weeks for a given allocator or workload is not inconceivable.

2. *Offline physical attack:* Improper disposal is one common means. Companies and individuals often neglect to properly erase disks before selling or disposing of them [3, 6]. Even when all disk data has been overwritten, previously stored data can be recovered [4]. Thus preventing sensitive data from reaching disk unencrypted is essential.

3. *Online physical attack:* Tamper-resistant devices are becoming increasingly commonplace. These devices are attacked by attempting to directly tap buses, read memory, etc. Limiting data lifetime can prevent an attacker from gaining access to sensitive data processed prior to device compromise.

4. *Accidental leakage:* Data can leak from systems (or to a lower privilege level) through a variety of channels, e.g. core dumps. For instance, exploits have been observed in the wild in which an attacker forces a privileged application to dump core, disclosing the contents of the shadow password file [12]. Some operating systems [15] are actually programmed to ship private application memory dumps to the OS vendor. (Broadwell et al. [1] studied the problem of sensitive data in core dumps.)

Often data lifetime problems are dismissed entirely when only a particular threat model has been addressed. For example, a typical encrypted loopback device uses a single key that remains constant between reboots. Using such a device for swap space can leave data vulnerable for days or even weeks, limited only by the system's uptime.

3 What's Wrong in Today's Systems

The implementations and interfaces of today's operating systems, programming languages, libraries and other software components generally overlook data lifetime issues. This creates a variety of problems when attempting to build systems that minimize data lifetime:

Virtual memory isn't. Some properties of virtual memory differ from those of physical memory, with significant lifetime implications. Physical memory is generally non-persistent, but virtual memory may be written to storage through paging, hibernation, checkpoint, core dump facilities etc. Data written to storage can persist even if a system is rebooted or data overwritten. Physical memory is also generally isolated, i.e. data will only be accessible to code running in the address space of the process. Core dumps, process migration, swapping etc. can violate isolation.

The lack of definite persistence and isolation properties of virtual memory makes it very difficult for application designers to control or reason about data lifetime. This is also a problem for operating systems as all memory becomes virtual when an OS is hibernated, runs under a virtual machine monitor, etc.

Free isn't the same as dead. Once memory is deallocated, it is out of the hands of application implementers. Under some circumstances the developers can manually clear memory prior to deallocation, but often they have no control. For example, in most OSes and programming language runtimes there are no guarantees about data lifetime after a program crash. In many high level languages the programmer has limited control over deallocation and allocators in general take no measures to minimize data lifetime, e.g. by clearing deallocated data.

There is no cooperation. Even if an application's code is well behaved with respect to data lifetime, it relies on an OS, language runtime and libraries that typically are not. Software implementers must assume that such systems will make extraneous copies of data passed to them that are never erased, or perhaps even leaked, e.g. through error reporting or logging features. This also means that systems provide no defense in depth: if a developer makes a mistake, such as forgetting to clear out a buffer, there is no safety net. Finally, as lifetime semantics of components that implementers rely on are not well specified, it is extremely difficult to reason about data lifetime in a whole system.

You can't say what you mean. Most programming languages and operating systems don't provide complete or portable mechanisms for expressing basic semantics needed to limit data lifetime, e.g. "don't write this to disk in plaintext form" or "clear this from memory."

To express the former, at best programs can rely on functions that "lock" a range of pages in memory. Unfortunately, these function were designed to provide predictable performance for real-time applications, not for security [10]. As a result they don't provide the semantics that are required and that most programmers expect.

Unix-like operating systems provide the `mlock` call. `mlock` defines "lock" to mean "will always be in memory" and does not prevent pages from being written to disk, although it may be implemented that way [10]. The Windows `VirtualLock` function has a similar problem: "locked" pages can be evicted as long as they are read back in before the program regains the CPU. Windows' AWE API also obtains locked memory, but imposes numerous restrictions [9, 8, 14].

Clearing all memory into which sensitive data propagate is essentially impossible in modern high level languages (e.g. Java, Perl, etc.). In C, where it is possible, it is fraught with pitfalls. The most common and recommended practice for clearing memory is to use `memset` or `bzero` to overwrite sensitive data. Unfortunately, optimizing compilers can remove this code by detecting that a local variable's value is not used after it is cleared [7, 5]. Manually clearing memory is also error prone, especially in light of abnormal change of control flow, such as `longjmp`, signals, and program crashes.

4 Reducing Data Lifetime

There is no silver bullet for solving data lifetime problems. However, a variety of techniques can significantly improve the data lifetime properties of software systems.

Make data clearing automatic. As previously noted in section 3 manually identifying and clearing sensitive data can be complicated and error prone. Ideally, it should instead be done automatically. One simple way to do this is by leveraging allocators e.g. garbage collectors, the C `free` function, and C++ destructors to recognize memory no longer in use that should be cleared. Of course this is not optimal in all situations. For example, garbage collectors that are only invoked at a high water mark rarely deallocate objects if a program does little allocation, clearly undesirable for zeroing sensitive data.

Provide policy trade-offs. Mechanisms for limiting data lifetime like encryption, zeroing, or pinning often involve costly overheads. The performance trade-off afforded by providing only a single policy can make the difference between an application using a data lifetime reducing mechanism and ignoring it. Given this, it is important to support a range of policies that afford different lifetime/performance trade-offs.

For example, zeroing large regions of memory such as an entire process address space can add significant overhead and pollute the cache (without use of a "store uncached" machine instruction, such as `MOVNTI` on $x86$). Thus, zeroing memory immediately is not always an option. An alternative is to schedule zeroing. For example, Windows 2000 zeroes unused pages during idle time [14], and a common Unix practice is to zero memory immediately before reuse. Unfortunately, both these approaches make it difficult to reason about when memory will be cleared as they depend on workload. A more desirable clearing schedule from a data lifetime standpoint could provide explicit guarantees about when data would be cleared, and allow certain storage to be prioritized for clearing.

Storing data in encrypted form offers another way to trade performance against data lifetime. This increases the overhead of accessing data, but clearing data is fast because one merely has to discard the key. This is especially valuable when there is a very large sensitive data set, or when clearing is performance intensive e.g. securely clearing disk data can require multiple write passes [4]. When data is on persistent storage this provides the additional benefit that system crashes fail safe as the key is automatically discarded.

Design for data lifetime. Program design can significantly influence data lifetime. For example, a program that caches a server password (e.g. `ssh-agent`, web browsers) can be redesigned to prompt for it each time, reducing data duration at a cost in usability. However, if care is not taken to clear the password when it is not in use, the extra copies produced by re-entering passwords could offset the reduction in duration.

Identify sensitive data explicitly. Today's OSes, libraries, language runtimes, etc. do not allow sensitive data to be explicitly identified. Thus, components that care about data lifetime must treat all data as sensitive. While desirable from a security standpoint, this may impose unacceptable performance overheads. By identifying data at a more granular level e.g. individual pages or variables vs. whole programs, we can reduce this cost. Providing the system with this information can also shift the burden of clearing memory, etc. from the programmer to the infrastructure.

Paradoxically, while identifying sensitive data explicitly permits more secure handling, it also allows attackers to identify sensitive data. An attacker may therefore be able to easily filter sensitive data from other data, a task which is more difficult today.

Provide secure defaults. Changes that improve data lifetime greatly but impose minimal overhead should be defaults. For example, in the course of examining data lifetime in Mozilla we found that immediately zeroing data in the string class destructor significantly reduced the lifetime of passwords entered into web forms, with negligible overhead. Treating all data as sensitive seems the best default for features that write to persistent storage such as check-pointing/migration, swapping, etc., while using finer grained hints about sensitivity as optimizations.

Measure it! Our experience with Mozilla revealed that even small changes can have a big impact on data lifetime. However, analyzing data lifetime in large systems can be a daunting, error-prone task. A variety of existing tools could help address this problem. Memory profilers could identify long lived dynamically allocated data and memory leaks, which can point to lifetime problems. Static analysis tools such as meta-compilation and type qualifier inference "tainting" analysis could identify variables containing sensitive data [1, 13]. They could also check for errors in implementing memory clearing, e.g. identifying code paths that fail to clear a buffer.

5 Related Work

Past work on system level solutions to data lifetime problems, such as encrypted swap [11] or secure core dump mechanisms [1], just considered piecemeal solutions to specific threats. Texts on secure design/implementation have treated data lifetime issues only as a class of application bug, such as failing to zero crypto keys or lock memory containing passwords.

We believe these views of data lifetime are woefully incomplete. First, although addressing individual threats is important, it fails to prevent future threats either caused or exacerbated by extended data lifetime. Second, data lifetime problems are qualitatively different from bugs.

Bugs are generally local errors in software, often artifacts of a language or API, but data lifetime is a quantifiable property of systems at every level of system design and implementation, from device drivers to language runtimes. Unlike particular bugs, data lifetime cannot be simply eliminated, only reduced, sometimes at the cost of making other design trade-offs. Finally, fixing bugs typically only fixes known exploits, whereas reducing data lifetime can make software more robust against unknown threats.

6 Conclusion

In today's systems compromises are inevitable. Limiting data lifetime reduces the risk of sensitive data exposure when compromises occur. Previously, data lifetime problems have been largely overlooked, left as a problem for application implementers. We have argued that data lifetime is a whole system problem that must be addressed at many levels, from the tool chain to the operating system. We have characterized the data lifetime problem and examined how current operating systems, languages, etc. provide inadequate support for reducing data lifetime. We have explored solutions for reducing data lifetime in a simple, efficient, and flexible manner.

7 Acknowledgments

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

References

[1] 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.

[2] 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.

[3] S. L. Garfinkel and A. Shelat. Remembrance of data passed: A study of disk sanitization practices. 1(1):17–27, Jan./Feb. 2003.

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

[5] P. Gutmann. Software leaves encryption keys, passwords lying around in memory. `http://www.securityfocus.com/archive/82/297827/2002-10-27/2002-11-02/2`, October 2002.

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

[7] M. Howard. Some bad news and some good news. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode%/html/secure10102002.asp`, October 2002.

[8] Microsoft Corporation. Address windowing extensions API. `http://msdn.microsoft.com`, February 2000.

[9] Microsoft Corporation. VirtualLock. `http://msdn.microsoft.com`, January 2004.

[10] Open Group. The single UNIX specification version 3, IEEE standard 1003.1-2001. WWW, 2001. `http://www.unix-systems.org/single_unix_specification/`.

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

[12] R. Rogers. Exploiting the ftp pasv vulnerability. `http://www.securityhorizon.com/whitepapers/hacking/PASV.html`, October 1999.

[13] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. 10th USENIX Security Symposium*, August 2001.

[14] D. A. Solomon and M. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2000.

[15] US Department of Energy Computer Incident Advisory Capability. Office XP Error Reporting May Send Sensitive Documents to Microsoft. `http://www.ciac.org/ciac/bulletins/m-005.shtml`.