Gone, But Not Forgotten: The Current State of Private Computing

Aseem Rastogi*
University of Maryland, College Park
aseem@cs.umd.edu

Jun Yuan, Rob Johnson
Stony Brook University
{junyuan,rob}@cs.stonybrook.edu

***Abstract*—Private data comes in many forms: web browser histories, chat logs, sensitive word processor documents, network proxy logs, and many more. Some applications – primarily web browsers – now support private modes that aim to prevent sensitive information leaks. There are two problems with this application-level approach. First, there are many software engineering challenges in implementing correct and complete private modes. More fundamentally, applications cannot always tell which data is private – this is up to the user. As a result, applications that do support private modes may not implement the user's desired policy, and many other applications that process private data do not have a private mode at all, because the developers did not consider that use case.**

In this paper, we present a case for private computing mode as a system service, rather than a per-application feature. We specify the threat model and the goals of the private computing mode, and argue that the applications alone cannot achieve these goals. We briefly describe our ongoing work on developing a private computing mode service.

***Keywords*-WWW; Privacy; Operating Systems**

I. Introduction

Private data no longer consists of just passwords and cryptographic keys; it now includes other forms of data such as chat transcripts, web browser history, application logs, and word processor documents, among others. However, traditional applications and operating systems are not designed to treat such data as sensitive. For example, chat clients often save chat transcripts to the disk, web browsers save browsing history and other websites data to the disk, applications often log user activities in persistent log files, and operating systems do not clear page caches when the pages containing the documents contents are freed. Applications may also send data to other processes on the system. Web browsers pass domain names to the DNS daemon, which in turn may store the domain-to-IP address mapping in its local cache to serve future requests. Thus, in their traditional form, applications and operating systems result in wide-spread sensitive data leaks.

Some applications – primarily web browsers [2] – now support private modes of computation. In the private mode, the application does not save sensitive data to the disk. In addition, once the user exits the private mode, the sensitive data is erased from the application memory.

*Work done while the author was at Stony Brook University.

Providing private mode as a per-application feature has several problems. First, there are many applications which call for a private computing mode – word processors for editing sensitive documents, chat clients for private chat, terminals for executing sensitive commands, Dropbox for uploading sensitive documents to the cloud, etc. Implementing private computing mode in each application would require a lot of (possibly duplicated) effort.

Then, there are software engineering challenges involved in retrofitting security on a large code base – developers could easily overlook data leaks in the code. Furthermore, applications alone cannot seal all the leaks. Sensitive data could propagate to parts of the system that the application can't access, such as swap space, kernel buffers, and the memory of other processes.

More fundamentally, the applications cannot always tell which data is private. The precise semantics of data is known only to the user. A user, for example, might want to retain pdf files downloaded during a private web browsing session but discard all the video files once he quits the session. As a result of this uncertainty about semantics of data, applications cannot enforce a one-size-fits-all policy. In other cases, application developers simply do not provide a private mode because they do not consider it central to their application's functionality.

Existing research systems that aim to prevent data leaks are either incomplete, or use heavy-weight mechanisms such as hypervisors. Non-hypervisor based solutions mostly focus on specific, short-lived, and easily-identifiable forms of private data, such as passwords, and do not fully address issues such as IPC and long-term storage. Hypervisor based solutions, although complete, incur usability and performance cost.

In this paper, we explain why private computing mode must be implemented as a system service and outline an implementation that is complete, does not incur the usability and performance costs associated with hypervisors, and is driven by a user policy to retain or discard persistent data.

II. Related Work

Aggarwal et. al. [2] provide a comprehensive analysis of private browsing mode implementations in major web browsers. They point out that the implementations leak a lot of sensitive data and are inconsistent with each other.

None of the implementations handles downloaded files. Some implementations even retain bookmarks added during a private session. They also analyze popular Firefox plug-ins and find that out of the 32 analyzed plug-ins, 16 undermine the private mode implementation. As expected, none of the implementations prevent domain names from being leaked to a local DNS cache.

Lacuna [7] develops a notion of *ephemeral channels*, an implementation technique for controlling the lifetime and leakage of private data inside a virtual machine monitor. While it shares design goals with our project, our design principle is fundamentally different. Lacuna uses hypervisor based heavy-weight mechanisms in order to achieve a high level of assurance against multiple data copies made by the OS. We, on the other hand, believe that OSes are carefully designed on the zero-copy principle. Hence, we propose to explore the feasibility of implementing a private computing service within the OS, without the overhead and inconvenience of virtualization.

Researchers have proposed building blocks for different PCM components – such as data in kernel memory, application memory, swap space, and disk – but none completely addresses the challenges of PCM.

Viega [12] talks about how applications should handle sensitive data in RAM. Chow et. al. [6] and Garfinkel et. al. [8] focus on the problem of in-memory data lifetime. They present a case for secure deallocation of in-memory data in all the layers of software stack – application, runtime, and all the way down to the kernel. SWIPE [9] transforms C programs to erase sensitive data immediately after its intended use. Chen et. al. [5] discuss residual objects in browsers. None of these projects consider data leaks through disk and in-memory kernel page caches, nor do they take into account data leaks via inter-process communication.

Provos [11] describes encrypted swap and Blaze [3] proposes an encrypted file system for Unix. However, encrypted file-systems, block devices, and swap, all use long-term keys, so simply running an application on a system with an encrypted file-system and swap will not render the data unreadable by an attacker that later gains access to the computer (Section III). Although some encrypted swap specifications support re-keying, none that we know of implement this feature.

Borders et. al. [4] consider the problem of working with private data on a host with an untrusted OS. They solve this problem by placing a trusted VMM below both the untrusted OS and the privacy-sensitive application, and make numerous other performance and functionality sacrifices to seal various covert channels. The threat model of private computing mode assumes that the host OS is trustworthy while the private computation takes place.

Kannan et. al. [10] propose to remove all traces of private data by snapshotting the system before beginning a sensitive computation, restoring the snapshot afterwards, and then replaying all non-sensitive inputs. Although this can give a high level of assurance that all private state is gone, the implementation is complex and incurs high overhead.

III. Threat Model

We refer to any program that the victim runs on her computer (system) as a computation.

Attacker capabilities. We consider a passive, local attacker. Such an attacker cannot make any changes to the victim's computer, but may inspect every component of the system before and after the private computing session. Thus, the attacker cannot install key-loggers or other malware to monitor the user's behavior during the private computing session, but may inspect and compare the contents of all memory – including kernel and user-space memory and device buffers – and all disks in the system. This is the standard threat model assumption for PCM ([2], [7]).

The attacker has only *local* privileges on the system. He cannot, for example, inspect the network packets on the wire. However, he can scan through the kernel network buffers (among other memory areas) on the system.

Attacker goals. The goal of the attacker is to glean any information about the data involved in the victim's private computing session. For a chat client program, the data could be the chat recipient and transcripts; for a word processor or Dropbox, the data could be the document names and contents; for a web browser, the data could be the URLs visited, cookies stored by the web sites, etc.

PCM. PCM aims to hide such computation details from the attacker. It may reveal that the computation was performed, but not its details. For example, it may reveal that the victim used the word processor but no more details about the documents it accessed.

IV. Challenges

Data can leak through multiple channels on a system. We discuss each of these in detail.

In-memory data. Chow et. al. [6] show that application data can linger in memory for days after the application exits. Application data may reside in the application stack, heap, and data segment, and also kernel buffers such as page caches and network buffers, kernel stack and heap. For example, if the victim edits a sensitive document with a word processor, the kernel page cache could contain the document contents even after the processor exits.

Many of these locations, such as kernel buffers, are not accessible to the application. Thus applications cannot implement private computing modes without OS assistance.

Persistent data. Data written to disk during the computation is also a challenge for the private computing mode. For example, during a private browsing session, the user might download and view her medical records and then would like to get rid of those records upon exiting the session.

None of the web browsers today handle downloaded files in the private mode implementation. More seriously, even if the application did delete the downloaded files, it is well known that on-disk data can be recovered long after it has been deleted. Simply overwriting the contents of the file before deleting it is also not sufficient, since the filename may still be on disk, and even its contents if the file is stored on a log-structured file-system.

Swap area. Application data could be sent to persistent storage when the OS swaps out the application's memory pages. As discussed by Garfinkel et. al. [8], even using `mlock(2)` may not prevent pages from being swapped out.

Inter-process communication. Sensitive data may leak to the memory of other processes in the system. For example, a web browser, even in the private mode, needs to communicate domain names to the local DNS resolver. The DNS resolver may then store the domain name in its local cache for serving future requests. Thus, the domains visited by the victim could remain long stored in the memory of another process – the DNS resolver [2].

As another example, consider a word document that the victim opens in a word processor. Let's say the document contains a URL and when the victim clicks on it, the processor launches a web browser that navigates to the target webpage. Now the document information (the URL) has become the part of browser memory, DNS cache etc., thereby leaking the data. So, even if the word processor implemented a private mode, document contents can leak to other processes that are not running in a private mode.

Clearly, the memory of other processes cannot be accessed by the application and thus, it cannot clear the leaked data.

Peripheral memory. Application data may be copied to memory in external devices, such as video, sound, and network cards. This memory may continue to hold sensitive information, such as the contents of the application window, long after the application has exited. It may not be possible for the application to clear this memory before exiting.

Plug-ins and add-ons. Third party add-ons and plug-ins further complicate prevention of data leaks. The application itself may implement private computing mode correctly by not sending out sensitive data to disk or other processes. However, it has no control over the third party add-ons and plug-ins which usually run in the same address space as the application, and can leak data at their will. Aggarwal, et al., showed that plug-ins are a major source of leaks in private browsing modes [2] and, as a result, many browsers sacrifice the functionality and disable plug-ins in private mode.

Variable privacy preferences. Applications cannot always tell which data is sensitive, and should be discarded after the private computation, and which data the user would like to retain. The semantics of data is known only to the user and therefore, a user policy must decide what to do with the data after the private computation.

Software engineering challenges. Finally, there are software engineering challenges of retrofitting security onto a large software. Since the software was probably not designed to treat certain data as sensitive, there could be many points where data can leak, and developers may miss some of these points when retrofitting a private mode onto the application.

A. Private Computing as a System Service

The above discussion shows that applications cannot prevent all data leaks on their own. Data can propagate to parts of the system that the application cannot clean – kernel buffers, swap space, memory of other processes, deallocated disk blocks – but that a local attacker can access after the private computing session has ended. Thus, a complete private computing mode must be a system service. This will (a) enable users to perform private computations with applications that do not support any private mode, (b) enable users to define their own policies, and (c) give users more consistent private computing semantics across applications.

A system-level private computing mode can provide hooks for applications to specify their own default policies about retention of data from private sessions. This changes the task of sealing data leaks from blacklisting to whitelisting, which is much less likely to result in a bug. Since our threat model assumes applications are benign, this does not compromise the security goals of the private computing mode.

V. Work in Progress

We are developing a system service that will enable users to run any application in a private computing mode.

A. Design Goals

Correctness and completeness. We would like to cover all the channels that can cause data leaks, as described in Section IV. Our service should prevent data leaks via disk, kernel buffers including page caches, inter-process communication, and third-party add-ons.

Pay-as-you-go. Unlike the previous systems [3], [6], [11], we would like to pay for the overheads of data hiding only for applications in the private computing session, thereby minimizing the performance impact. The other applications should run normally, without paying these overheads.

Seamless integration. Applications in a PCM session should integrate with the user's non-private processes. For example, a browser started in a private session should have the same configuration options, plugins, etc., as his usual browser. Furthermore, the PCM system should make it easy for the user and application to specify some state that should be preserved after the PCM session ends.

Low overhead. Processes in private sessions should not be noticeably slower than non-private instances.

User policy driven. We would like to allow users to drive the policy around the on-disk data. Users should decide which data to retain or discard post-computation.

B. Service Components

Encrypted storage. Our service uses an encrypted loopback device as the persistent storage during the computation. To support unmodified application binaries, we setup a union-filesystem [1] such that (a) the application can access the actual disk filesystem normally, using regular path names, and (b) all the application writes go to the loopback device. The private mode container (see below) destroys the device encryption key when the computation ends.

Policy engine. The policy engine runs in user-space and implements the user's policy for on-disk data. Once the computation finishes, it copies to-be-retained files from encrypted loop device to the underlying filesystem. The to-be-discarded files are automatically discarded once the encryption key of the loop device is destroyed.

Encrypted swap. The swap area used for swapping out the pages of the computation also lies on the encrypted loop device. We modify the kernel code such that it uses the encrypted swap for the private computation memory pages.

In-memory clearing. In-memory clearing of Chow et. al. [6] is an important component of our design. In addition, we take care of clearing the kernel page caches for the files written during the computation.

Peripheral memory management. We are pursuing lightweight solutions for peripheral memory management. We are developing a proxy X-windows server that inserts commands to erase client windows immediately before forwarding the client's window-unmap commands. Other peripherals, such as sound cards and network cards, may not have directly overwriteable memory, but they may have finite buffers, so that the private computing system can erase their buffers by writing a sufficient amount of dummy output.

Private computing container. The user starts a PCM session by starting the private computing container, a user-mode process, with the actual application (such as web browser) as an argument. The container sets up the environment (like the storage, swap, etc.) for the actual application and then starts it. The container also starts other processes that the application needs to communicate with, in the same environment. Thus, all IPC from the private application is directed to other processes running in the same container. Some services, such as display or audio servers, may be proxied between the private container and the main instance on the system. The set of supporting programs could be configured statically for each application or can be inferred dynamically when the application tries to connect to them. Once the application exits, the container invokes the policy engine to copy any persistent files to the base file-system (per user policy), unmounts and destroys the encrypted loopback device, and erases ephemeral encryption keys from memory.

C. Evaluation Criteria

We will evaluate our private computing service on the following parameters:

Range of applications. Our service should support unmodified applications, including web browsers, chat clients, terminals, cloud file-systems, and word processors.

Effectiveness. Our service should leave no private data in the entire machine state once the computation ends.

Performance. Finally, for the private computing mode to be usable, we will evaluate that the performance costs for the application and the overall system are acceptable.

VI. Conclusion

In this paper, we have considered the problem of private computing mode. We enumerated the challenges to implementing a complete and efficient private computing mode, and argued that private computing mode must be a system service rather than a per-application feature. Finally, we discussed our ongoing work on developing such a service.

Acknowledgments

Don Porter provided invaluable feedback and advice during numerous conversations.

References

[1] http://aufs.sourceforge.net/aufs2/man.html.

[2] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *USENIX*, 2010.

[3] M. Blaze. A cryptographic file system for unix. In *CCS*, 1993.

[4] K. Borders, E. V. Weele, B. Lau, and A. Prakash. Protecting confidential data on personal computers with storage capsules. In *USENIX*, 2009.

[5] S. Chen, H. Chen, and M. Caballero. Residue objects: a challenge to web browser security. In *EuroSys*, 2010.

[6] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: reducing data lifetime through secure deallocation. In *USENIX*, 2005.

[7] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal sunshine of the spotless machine: protecting privacy with ephemeral channels. In *OSDI*, 2012.

[8] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *ACM SIGOPS European workshop*, 2004.

[9] K. Gondi, P. Bisht, P. Venkatachari, A. P. Sistla, and V. N. Venkatakrishnan. Swipe: eager erasure of sensitive data in large scale systems software. In *ACM CODASPY*, 2012.

[10] J. Kannan, G. Altekar, P. Maniatis, and B.-G. Chun. Making programs forget: enforcing lifetime for sensitive data. In *HotOS*, 2011.

[11] N. Provos. Encrypting virtual memory. In *USENIX*, 2000.

[12] J. Viega. Protecting sensitive data in memory. http://www.ibm.com/developerworks/library/s-data.html?n-s-311.