Protecting Drive Encryption Systems Against Memory Attacks

Leo Dorrendorf
Safend Ltd
Ha-Barzel st. 32, Tel Aviv, Israel
leo.dor@gmail.com

May 6, 2011

Abstract

Software drive encryption systems are vulnerable to memory attacks, in which an attacker gains physical accesses to the unattended computer, obtains the decryption keys from memory and consequently decrypts the drive. We reviewed the currently existing mitigations and have found that they provide only partial protection, and none of them protect against the full range of memory attacks. We propose a new method for protecting encryption systems against memory attacks, by converting them to use two tiers of keys, a single Master Key and a set of File or Sector keys. When the computer is unattended, the Master Key and part of the second-tier keys are erased from memory. The method is secure against any type of memory attack, including attackers who gain complete control of the unattended system. Compared to previous methods of protection, which erase keys and shut down the computer, our method allows to keep the computer operational by a combination of cryptographic and operating systems techniques. Applications may continue running, and can access any unencrypted data as well as a chosen subset of the encrypted data, at the cost of leaving that data unsecured against memory attacks. We first describe the application of the method to file-based encryption systems, where we have implemented and tested it in practice, and then describe a possible adaptation to disk-based encryption systems.

1 Introduction

Drive encryption systems attempt to provide data confidentiality against an attacker with physical access. Memory attacks break drive encryption systems by extracting the decryption keys from RAM on a running or recently turned off computer. This paper proposes a new method of protection against memory attacks on drive encryption systems. The method erases encryption keys to protect data from any attacker. Contrary to simpler methods of protection, where the computer is shut down after erasing keys, we describe cryptographic and operating system methods to keep the computer operational. The administrator may choose a trade-off between functionality and security, selectively making encrypted data available to running programs at the cost of making it vulnerable to memory attacks. We have converted an existing drive encryption system to implement the method. In this paper we describe the theoretical background and practical considerations that determine its design.

Based on the range of existing memory attacks we reviewed in this paper, we have concluded that an attacker can gain full control of the unattended computer. Therefore, we designed our method of protection to withstand an attacker with unlimited access, without depending on the properties of any particular memory attack.

1.1 Requirements

Our design requires two conditions to protect an encryption system from memory attacks. First, the computer must be able to detect when it is left unattended by the authorized user. This is necessary because our protection limits the computer's functionality, and it should be inactive in the user's presence, while the computer is safe against physical access by an attacker. When the computer is left unattended, it switches over to the protected mode. In practice, there is a short period after the authorized user leaves and before the computer recognizes his absence. The computer will be unprotected during that time. For the purposes of this paper, we ignore this period, as it can be made arbitrarily short at the cost of some usability. We will address this topic in detail in Section 3.1.

The second condition is that the authorized user must log on to the computer with a password or some recognizable digital token that can be used as input to a cryptographic function. The possession of this token is the difference between an authorized user and an attacker. Although this requirement is satisfied in most authentication systems, some systems that fail this requirement are biometric techniques that use approximate authentication tokens,and techniques that rely solely on performing zero-knowledge proofs (normally using secure devices such as smart cards). Such systems can also be adapted to produce a recognizable digital token.

The protection method requires no hardware additions and no modifications to the operating system, except for the installation of intermediate drivers, as expected in an encryption system. In our experience, the method of protection incurs a negligible performance cost during transitions to and from the protected state. The changes to the encryption system's architecture incur some computational costs due to longer key setup, but these are negligible when compared to the time complexity involved in regular disk I/O operations.

Our method of protection against memory attacks is platform-independent. The requirements and design considerations we present are independent of the operating system and the file system involved. Our experience is based on implementing the protection in a real-world, commercially deployed, software-only drive encryption system with a file-based architecture that works under the Windows operating system and the NTFS file system. We have covered the design considerations and requirements in detail, and hope to make it obvious that none of them depend on the platform. In fact, since in our implementation we were limited to acting as an independent software vendor, installing software but never modifying the core operating system, and only using official operating system APIs, it is possible that implementers who can modify the OS for better cooperation, may make better use of operating system facilities. We only used the official facilities provided by the OS to encryption systems.

1.2 Benefits

Compared to previous attempts to defend against memory attacks, this paper's main contribution is the protection against memory attacks in the powered-on, unattended state. In contrast with the simpler method of erasing encryption keys and shutting down the computer, our method is designed to protect a working computer. The encryption system can be customized to leave some processes fully functional, at the cost of exposing their data to the memory attacker. The remainder of the encrypted data is secured. Beside the working state, the protection also helps erase and secure keys during shutdown, hibernation, and sleep states. During shutdown, the encryption keys are simply erased to protect them from theft. For hibernation and sleep, the protection is activated during the power-down stage and resumes when the computer returns to power, protecting the encryption keys and sensitive data until the authorized user logs on. Simpler methods of protection do not cover hibernation and sleep.

Any process, daemon or application that can run usefully when unattended can benefit from our method. Examples include keeping web servers and LAN shares available on-line, and keeping long processing jobs running including audio processing, video rendering, mathematical and cryptographic computations. On

the other hand, programs which are unproductive when unattended, such as text editors, e-mail clients, and other interactive applications need not continue running in the user's absence. Such applications handle confidential information more commonly than non-interactive applications, and they can benefit from the added level of protection.

1.3 Hardware encryption systems

We consider hardware-only encryption systems secure against memory attacks, and therefore have not covered them in this article. Hardware encryption systems may become more widespread with time, but for now we have focused on securing the software systems.

The rest of the paper is laid out as follows. Section 2 is a technical overview, revisiting memory attacks, describing a combined model of the attacker's capabilities, examining the limitations of existing mitigation techniques, and reviewing the architectures of disk-based and file-based encryption systems. Section 3 describes in detail the implementation of the protection method, as applied to a file-based encryption system. Section 4 deals with applying the protection to disk-based systems. We propose several items for future research in Section 5.

2 Technical Overview

2.1 Memory Attacks on Drive Encryption Systems

A memory attack is a type of side-channel attack on drive encryption systems in which an attacker with physical access to a computer obtains the decryption key from RAM, and proceeds to decrypt the complete contents of the hard drive. Memory attacks destroy the ability of drive encryption systems to protect the data on a computer even after an attacker gains physical access to it. There are several types of memory attacks, each with a different set of requirements and effects, as we discuss below.

The Cold Boot Attack received wide attention in 2008, with the publication by [11]. It is the first memory attack which requires no specialized hardware, whose minimum requirement to obtain memory contents is a reboot. The authors provide downloadable software, which boots the computer into a small program that dumps the contents of the physical memory to external storage. Additional tools then search the dump for encryption keys and key schedules, and help recover keys even in the presence of some bit flip errors. The attack relies on the memory remanence properties of RAM: although power to the RAM chips is temporarily lost, their most recent contents can be recovered. The longer the power loss, the more degradation occurs in the data. The remanence property has been the subject of earlier research in several types of memory, including Static RAM and Dynamic RAM [24, 10].

At lower temperatures, memory contents last longer. Optional improvements to the attack involve cooling the RAM chips by spraying them with compressed air, then physically removing them from the target PC and further cooling them with liquid nitrogen before analyzing them in the attacker's computer.

Other attacks use memory remanence to create a snapshot of the user session active at the time of reboot [5], then resume that session live with modifications giving the attacker full visibility and control.

Memory attacks devised before Cold Boot used peripherals with Direct Memory Access (DMA) capabilities, such as Firewire [3, 4, 20] or USB [2] to dump the target computer's memory. The DMA attacks cause no power interruptions and therefore no data degradation, and in fact do not rely on memory remanence. On the downside, they require connecting a specially programmed external device. Once connected, the device can manipulate physical memory directly. As a result, the attacker gains full visibility and control, including the ability to run his code with system privileges.

In another attack scenario, the attacker induces either hibernation or a crash on the target computer. When entering hibernation or creating a crash dump, the computer makes an orderly snapshot of all virtual

memory to a file. The attacker can recover the file and obtain the original memory contents, including drive decryption keys, without corruption. Making the situation worse, on Microsoft Windows the hibernation and crash dump files are written using a special I/O path that bypasses most drivers, including disk encryption drivers, and is therefore difficult to encrypt[1]. All this makes computers that have hibernation and crash dumps enabled more vulnerable to the attacker.

Together, the scenarios give rise to a more powerful model of the attacker with physical access to the computer. The attacker can obtain an exact snapshot of all memory contents, without degradation. He or she may observe the system and choose the time to make the snapshot, and even scan memory repeatedly.

We have found that all the existing mitigations, which we have detailed in the next section, are defeated by at least one of attacks above. The only encryption systems safe against memory attacks are the ones fully implemented in hardware, since they never place the key in RAM. Our goal is to protect software encryption systems that place the keys in RAM against an attacker who can obtain the complete and undegraded contents of memory, can run arbitrary code with full privileges, and can attack at the timing of his or her choice as long as the computer is unattended.

We note that there is a different class of attacks, which may begin with memory intrusion but work by modifying the system [22, 15]. In those scenarios, the attacker installs a hidden software agent on the computer, waiting for the authorized user to enter his credentials, then stealing the credentials and sending them over the network to the attacker. The attacker can use the credentials to decrypt a previously made copy of the drive. These attacks affect hardware encryption and software encryption equally, and generally cannot be prevented by the encryption system alone. Advanced system integrity mechanisms are needed to counter these attacks. Those are not memory attacks, and are outside the scope of this article.

2.2 Current Mitigations

Software encryption systems are vulnerable to memory attacks because they place the decryption keys in memory. That can be avoided by implementing the encryption in hardware, but the use of hardware generally incurs a cost and delays the time to market, limiting the current penetration of hardware encryption systems to a small share of the market. In this section, we discuss existing mitigations for software systems. These include mitigation by configuration; relocating, scattering, and obfuscating the keys; leakage-resilient cryptography; hardware-assisted key deletion; and offloading.

Changing the computer's configuration can make some attack scenarios harder to execute. Blocking Firewire and USB ports can help prevent DMA attacks. Limiting boot device options and configuring the BIOS to overwrite memory on reboot makes rebooting less helpful to the attacker. Disabling hibernation denies the attacker access to a high-quality snapshot of RAM. There are hardware-assisted techniques to securely wipe encryption keys during a sudden temperature change or power-off [17, 8]. All those techniques are useful, but each one prevents a specific attack scenario while leaving the other scenarios open. The techniques rely on particular properties of hardware and software, and all of them are liable to be circumvented. Worst of all, an attacker who gains control of a system can reconfigure it to remove those mitigations before proceeding with memory attacks.

Obfuscation techniques make it difficult for the attacker to locate a secret. Cryptographic keys have higher entropy (wider statistical byte distributions) than code, strings, and other non-random data, so an attacker can normally use an entropy scan to locate keys [23]. Obfuscations make the key less obvious by hiding it behind a layer of complex code, which distributes it across memory, stores it in an altered representation, and collects and recomputes it when needed. Obfuscation forces an attacker to use reverse-engineering methods to locate and extract the keys from a memory dump. Unskilled attackers may be deterred by obfuscation, but skilled attackers will at best be delayed. Since attackers with access to memory

[1]Before Vista, Windows did not provide third parties with official APIs to encrypt the hibernation file. See an account by Truecrypt at `http://www.truecrypt.org/docs/?s=hibernation-file`.

can obtain the obfuscated data along with the de-obfuscation code, any secret protected by obfuscation can be exposed by simulating the de-obfuscation code.

Key scattering works by inflating the amount of key material needed to restore the actual key. For example, a k-byte key may be stored as a thousand pseudo-random k-byte buffers, which yield the original key when XOR-ed together. If the attacker faces memory degradation, as with the Cold Boot attack, then scattering decreases his chances of successfully computing each bit of the key. In all the other attack scenarios, the attacker obtains the information without degradation, along with the code needed to reconstruct the key. The attacker can then simulate the reconstruction and compute the original key.

Obfuscation and scattering can only protect the keys *most* of the time. Because the encryption system needs to use the keys, it must at some point collect the relevant data and reconstruct the keys in memory. An attacker who happens to dump the RAM at the right time will obtain the keys in cleartext. An attacker who can time his attack to coincide with the reconstruction, or repeat it as needed until successful, can simply bypass both obfuscation and scattering.

Leakage-resilient cryptography can strengthen encryption protocols against the leakage of a limited part of the key [1, 18], but cannot protect against memory attacks because they expose the whole key to the attacker.

Hardware-assisted key deletion techniques [8] allow erasing specific areas of RAM on sudden power loss or temperature drop. While they help against the Cold Boot attack, they do not prevent DMA attacks.

Offloading techniques try to protect encryption keys by moving them outside of RAM. One approach is to use the regular CPU registers for temporary storage [17]. The disadvantage of that approach, as with key scattering, is that the key has to be reconstructed in memory before use. Another approach is to place the keys in special purpose registers where they are used directly, as enabled by the Intel AES Instruction Set [9]. The weakness of that approach is that copies of the keys still end up in RAM, when spilled from the registers during context switches. Another option, demonstrated in an ongoing research project, is to use one of the CPU's caches for key storage [19]. Currently, the CPU cache must be put in a special mode of operation, and the impact on the computer's performance is crippling. Other techniques offload the encryption from the CPU to other commodity hardware, such as the graphical processing unit [14]. None of those techniques include a mechanism to keep an informed attacker from retrieving the keys from the offloaded storage. In order to secure encryption keys, they should be placed in a secure hardware module that would never expose the key to the CPU. The hardware would then have to perform the encryption itself, which brings us to hardware-only encryption systems.

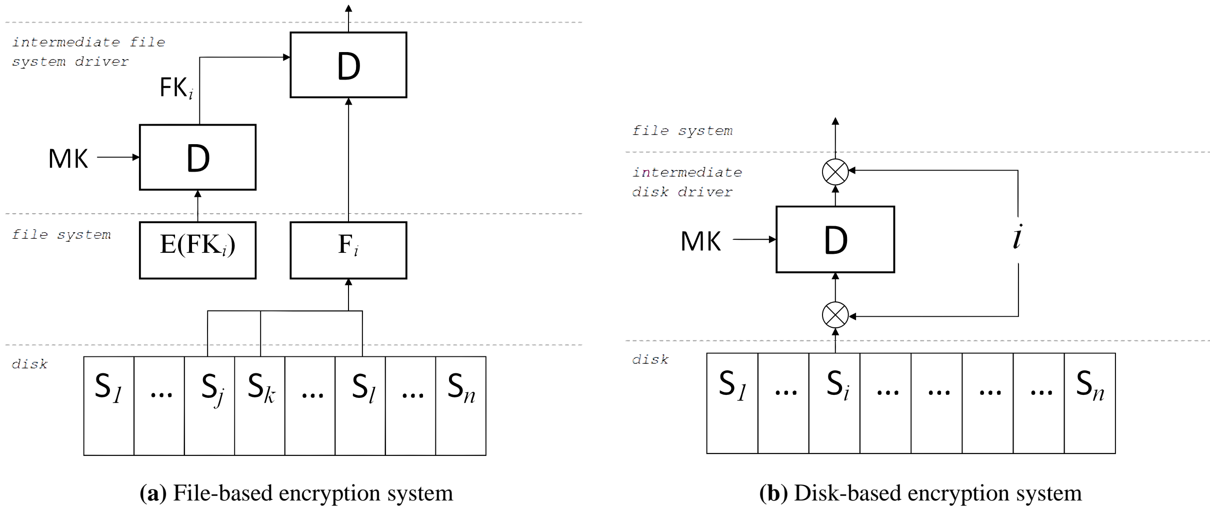
To summarize, every method of protection short of hardware encryption or erasing the encryption key, fails to protect against the combined abilities of the informed attacker.

2.3 Drive Encryption System Architectures

Most drive encryption systems deployed today are implemented in software, although hardware solutions are available. We make the distinction between software and hardware according to the placement of the encryption key: in hardware systems, the key is located in secure hardware and cannot be accessed by the CPU, and consequently the encryption system is secure against memory attacks. If the key is accessed by the CPU during encryption or decryption, we consider the system a software system, and vulnerable to memory attacks.

Some systems use hardware to a small extent. For example, the Microsoft Windows Bitlocker [7] can use the Trusted Platform Module (TPM) for authentication and key management; however, the TPM cannot deal with the computational load imposed by encryption, so after authentication the encryption keys are placed in RAM where they are accessible to an attacker.

The next most important distinction is the encryption system architecture. There are disk-based and file-based encryption systems. Both architectures operate by inserting intermediate drivers in the appropriate



(a) File-based encryption system

(b) Disk-based encryption system

Figure 1: Comparison of encryption system architectures, showing a decryption operation. On the left, the file $\mathbf{F}_i$, stored in several disk sectors $\mathbf{S}_i$, is decrypted using the File Key $\mathbf{FK}_i$. $\mathbf{E(FK}_i\mathbf{)}$ is the stored File Key, encrypted by the Master Key. On the right, a disk sector $\mathbf{S}_i$ is decrypted using the Master Key **MK** and transformed by a tweak, denoted $\otimes$, based on the sector number i.

driver stack. The intermediate drivers process read and write operations, so that the data resides encrypted on the storage layer below, but appears unencrypted to any software layer above. The difference is that file-based solutions work on the higher file operation layer and process file system operations such as file open, read and write requests, while disk-based systems work on the underlying disk operations layer and process disk operations such as sector read and write requests.

Disk-based systems typically encrypt the whole drive, leaving unencrypted only a small piece of code that initializes the encryption system. This code must authenticate the user, install itself as part of the disk services, and then pass control to the operating system's boot loader. Once the OS loads the regular disk drivers, encryption duties are handed off to an intermediate disk driver, along with the encryption keys. From that point on, the driver intercepts and processes sector read and write operations.

File-based encryption systems intercept and process file read and write operations. They can choose which files to encrypt and which to leave unencrypted. Most implementors leave the operating system unencrypted, including the boot loader, any system drivers, and the programs and configuration files necessary for the operating system to initialize up to the user authentication prompt. Since the operating system's binaries are non-secret, there is no loss of security. The benefit of this approach is that the user does not need to authenticate during boot, but only needs to give his credentials as usual on the log-on prompt.

Encryption systems typically employ one key, which we denote the *Master Key*, the access to which is granted to authorized users. To connect the key with authorization credentials they either derive it directly from user credentials, or decrypt a stored copy of it with a temporary key derived from the credentials. The latter method easily accommodates for multiple users with changing credentials.

Using a single key for encryption could expose similarities and correlations in the encrypted data. The ciphertext of files or sectors with identical prefixes would begin identically. Therefore, a unique token, associated with each file or sector, must be mixed into the encryption process. The extra token may be used to modify the encryption key, the plaintext, the ciphertext, the Initialization Vector in cipher-chaining or feedback modes, or any combination of these.

Disk-based systems use the sector number as the extra token, applying it through a "tweak" function to the plaintext and ciphertext. We discuss tweaks in more detail in Section 4.

File-based systems normally cannot use the file paths as the extra token, because files may be moved or hard-linked, destroying the one-to-one relationship between file path and contents. Accordingly, file systems attach unique tokens to each file, in a hidden header, footer, or a separate data stream. The tokens are copied and moved along with the file, and also preserve their values through linking operations. File-based systems use the token to derive the encryption key for a file or to set the IV. We have not encountered in practice any file-based systems that use the token to directly transform the plaintext or ciphertext.

We have implemented our protection in a system that uses the per-file token to derive the encryption key for file contents. In order to tie the access to file contents to user authentication, the per-file token stores the *File Key* encrypted with the Master Key. A benefit of this architecture is that once the File Key is decrypted, the Master Key is no longer required to access the file. We use that to our advantage, as shown in the next sections, erasing the Master Key to protect it from theft, then choosing a set of File Keys to keep in memory in order to allow access to chosen files.

For a file-based encryption system to be protected, we explicitly require that the file encryption key be derived from a unique per-file token. For the purposes of authorization, the encryption key should be a function of the Master Key. For reasons of security, it must not reveal the Master Key or other tokens. Finally, we require that File Keys for open files be retained in memory, so that the Master Key is not needed to access an already open file.

Formally:

- Each File Key FK_i must be a function of the stored file token and the Master Key.
- Each File Key FK_i must not reveal the Master Key.
- Each File Key FK_i must not reveal FK_j, for $i \neq j$.
- Each File Key FK_i must be retained in memory while its file F_i is open.

Any existing file-based encryption system can be converted to use this architecture, which is illustrated in Figure 1a. Disk-based systems can also be adapted to use two tiers of keys, with Sector Keys instead of File Keys, but they present additional challenges. We discuss their adaptation in section 4.

3 The Protection Method in Detail

Our method of protection can be summarized as follows. It activates in the unattended state when it detects that all authorized users have stopped interacting with the computer. It erases the master encryption key, securing it and any unopened files against the attacker. It then uses a second tier of decryption keys, one key per file, to keep the open files available. To keep the system stable, it blocks file open requests to unopened encrypted files. To secure some of the open files, it selectively erases their keys from the second tier, suspending non-vital processes and taking other preventive measures to ensure that those files not be accessed, and the keys not be needed. Once an authorized user returns and logs in to the computer, the encryption system uses his credentials to restore the master decryption key; it then resumes suspended processes and returns the computer to regular functionality.

3.1 Defining the Unattended State

Memory attacks require physical access, and assume the owner has left the target computer unattended. A common target is a stolen laptop in the locked, sleep or hibernation states. More rarely, the target is a laptop, desktop or server computer under full power. The target computer may have been performing tasks the user defined for it, precluding the option of powering it down.

Our technique protects from memory attacks by taking special measures, which could disrupt normal system functionality, but are acceptable when the authorized users are away. The ideal protection would activate at the moment the owner leaves the computer unattended. In practice, the encryption system must define criteria for identifying the unattended state. The criteria should include all power state changes, as well as active users logging out, locking their sessions, going into screensavers or just timing out. Because a single computer can host multiple concurrent user sessions, some of them remote, the encryption system must keep track of the number of active sessions. As the Master Key is shared by all sessions on a computer, the protection must be activated when no active sessions remain.

To resume normal functionality after the unattended state, our design requires the user to authenticate. As explained below, the authentication token is used to regenerate erased keys, providing the means to resume normal operation. Some unattended states, in particular screensavers, might not end with a log-on prompt, so the encryption system must ensure a log-on prompt by changing system configurations or actively locking the user session.

Our implementation combines a counter of active sessions with power state notifications, a screensaver detector, and an idle-time detector. We have implemented different protections for different power state transitions. For instance, on shutdown, all processes are terminated, whereas on hibernation, the operating system suspends the processes before turning power off, and then resumes them after power-on. In the first case, there is no need to preserve process functionality beyond clean termination. In case of sleep or hibernation, the protective measures are interrupted by power-down and continue after the power is restored, to secure the computer until the authorized user returns. To ensure a log-on prompt after screensavers, the system is configured to lock the session after a screensaver begins.

3.2 Eliminating the Master Key

Once the encryption system detects the unattended state, the first defensive measure is to erase the Master Key from memory. The Master Key must be securely wiped, along with any auxiliary material, such as expanded key schedules, which can help the attacker recover the key. We recommend following the guidelines for key storage and deletion presented in [10].

With the Master Key erased, the computer can continue to read the files that were already open (earlier we stated the requirement that File Keys for open files remain stored in memory). On the other hand, any unopened files become inaccessible both to legitimate users and to the attacker, as their File Keys cannot be derived without the Master Key. The security gains at this stage are that the Master Key and most of the drive's contents are safe against the attacker. All memory attack techniques, including the Cold Boot attack, full DMA access, exploiting hibernation files and crash dumps and so on are useless - the Master Key cannot be stolen.

The computer is now in a safer but limited state. In the next section, we explain how to keep the computer stable and operable when it cannot open additional files; and in Section 3.5, we show a method to further reduce the attacker's access, by purging the File Keys of some of the open files.

The computer recovers from the protected state once the authorized user returns and logs in. The encryption system uses the authentication token to restore the Master Key and resume normal function.

An implicit requirement is that the computer must still be able to present a log-on screen in the absence of the Master Key. This demands special attention, as we discuss in section 3.6.

3.3 Keeping the Computer Operational

In the protected state, the encryption system must impose limitations on file operations it cannot properly process. This section describes two possible ways of handling such file operations, denial and blocking, along with possible exceptions and problems that require further care.

As discussed above, the file-based encryption system installs itself as an intermediate file system driver, and receives requests to open, read and write files. In the protected state, the encryption system must deny requests to open encrypted files, because once a file is open, read and write requests for it may arrive. To handle these requests, the encryption system needs a File Key, which it cannot obtain without the Master Key. Therefore the first solution is for the encryption system to deny file open requests for unopened encrypted files.

The reason why the encryption system must deny file open requests, rather than read or write requests, is application stability. Most applications are prepared to handle a denied file open request, but expect unhindered read and write access once they have the file open. Even so, denying file open requests can have adverse effects on applications and overall system behavior.

To avoid these, the encryption system can *block* the file open request, which means suspending the requesting thread indefinitely (instead of returning an error result). A file system driver can do so without tying down system resources. The requesting application will usually wait as long as needed, because file open requests are not usually associated with a timeout mechanism. Once an authorized user returns, and the encryption system returns to normal, it can satisfy the file open request, letting the application proceeds normally.

There are some exceptions - file open operations that can proceed and need not be blocked. If an application requests to open a file that is already open (in the same application or in any other process), the request can be satisfied since the File Key is already in memory. Another exception can be made for creating new files. A new file cannot be cryptographically tied to the Master Key when the Master Key is unavailable, so instead, it can be encrypted with a random File Key, and the File Key stored in cleartext. Once the user returns and the Master Key becomes available again, the stored File Key can be encrypted with the Master Key. The contents of the file itself do not need to be re-encrypted.

Given the above, the operating system faces undesirable limitations: applications may fail because some of their threads became blocked, and their other threads did not expect that behavior; applications that the user expected to run, may be blocked; and the system may fail to present the log-on prompt, because some of the files involved are unavailable or the necessary processes cannot run. Clearly, such system behavior is unacceptable. In the following sections, we show how to resolve the problems that have arisen so far. Section 3.4 describes the methods needed to stop applications from failing because some of their threads are blocked. Section 3.6 describes the methods of defining essential applications and files, the access to which is guaranteed even in the protected state. The end result is a functioning operating system, with a trade-off between functionality and security: non-essential applications suspended or at the risk of suspension, and essential applications remaining active but vulnerable to memory attacks.

3.4 Suspending Applications

In the previous section, we suggested blocking some threads to prevent file open requests the encryption system cannot satisfy. This creates a stability issue, as it is possible for a process to have some of its threads blocked and some to continue running. The running threads may expect some interaction from the blocked threads, and may break otherwise. We therefore suggest the measure of suspending a process completely, first to deal with the stability issue, and later to derive additional security benefits.

In the protected state, the encryption system should suspend any process in which it has blocked a thread. Suspending a process preempts all of its threads from the CPU and preserves their states so tjat they can be resumed later. Operating systems provide several ways to suspend processes, including process and thread management APIs, debugging APIs, and kernel-level thread management procedures.

Whatever the method used, suspending all an application is generally safe. In particular, applications should not break when their threads are suspended by the operating system, because that is the normal behavior during context switches. Deadlocks are also not a threat because under normal circumstances,

suspending a user process from inside a driver does not add circular dependencies between any locked resources. The main risk to suspended processes is the loss of time-dependent resources, such as timers or stateful network connections. There is a risk of breaking some application's time-based behavior. We have not encountered such applications in practice, but if a particular application breaks because of suspension, it may be exempted from suspension by the administrator. We discuss the exempted application in section 3.6.

We derive an additional security benefit from our ability to intentionally suspend applications. Suspended applications cannot issue file read and write requests, and therefore we can discard the File Keys for the files they have open. A memory attacker will then be unable to access those files. We describe the exact steps needed in the next section.

3.5 Securing Open Files

In this section, we describe a method to protect open, encrypted files from memory attacks. The method includes three steps: suspending applications, flushing file caches, and erasing File Keys.

In the protected state, the Master Key is erased. We assume open files have their File Keys in memory, making them vulnerable to memory attacks, while unopened files do not. We can improve the system's overall security by forfeiting access to some of the open files, and erasing their File Keys.

File Keys are needed to satisfy read and write requests to open files. To discard File Keys, we must ensure no read and write requests arrive. The two sources of read and write requests are the application holding the file open, and the the virtual memory manager responsible for caching the contents of an open file in memory.

As we have mentioned above, denying read and write requests causes stability issues. Blocking such requests is also impossible, because the virtual memory manager expects its requests to be satisfied immediately and will crash the system otherwise. Therefore, to discard File Keys, we must ensure no read and write requests be sent to the relevant files from any applications holding the file open, as well as the virtual memory manager.

The encryption system first suspend the applications holding a file open, as described in the previous section. Afterwards, the encryption system uses virtual memory APIs to flush the file's cached portions to disk. At that point, and until the applications resume execution, no read or write requests will arrive for the file. The encryption system can now erase the File Key from memory and secure the file's contents against a memory attack.

The choice of files to secure and applications to suspend is up to the administrator. In our implementation, after entering the unattended state the encryption system suspends any application holding encrypted files open. Should any application attempt to open an encrypted file, it is suspended as well. This approach is simple, but it reliably identifies all applications handling confidential data and is well suited to systems where operating system data is unencrypted. A list of essential applications, which are never suspended, is centrally managed.

After suspending applications, the encryption system requests the virtual memory manager to clear the file caches, and then erases the File Keys. Having erased all or most File Keys from memory, the encryption system secures all or most of the data against memory attacks. In the next section, we discuss the exceptions: applications that must continue running and files that must remain available.

3.6 Defining Essential Applications and Files

There are processes and applications that cannot be suspended because they are vital to the operating system, important to the user, or are known to break when suspended. The encryption system must maintain a list of essential processes, and exempts them from suspension. Files belonging to those processes and applications must have their File Keys preserved when entering the unattended state. Ideally, the encryption system

should guarantee access to *any* encrypted file an essential application might need, even in the unattended mode. Managing the list of essential applications and files is therefore challenging.

The first type of essential processes are those needed by the operating system to recover from the unattended state. These are the processes involved in presenting the user with a log-on dialog, and in user authentication. On the Windows operating system, that includes *winlogon.exe* and *lsass.exe*.

The second type of essential processes are the applications the user expects to continue running while he or she is away. The user or administrator should explicitly list them as essential processes. The same treatment is needed for processes known to break when suspended.

Identifying all the files needed by an essential application is theoretically impossible. Simply listing all files an application has open when transitioning to the unattended state is not enough, as an application may request to open any additional file at any time. Applying heuristics, such as listing all files in the application's installation directory or its work directory, provides good results in practice, but leaves the theoretical risk of failing to log-on or suspending an essential application if an unexpected file is requested.

Another complication is the need to guarantee that essential applications are able to access encrypted files that were not open at the time of transition to the unattended state. The encryption system must have the File Keys ready for such a case. Because the Master Key is needed to decrypt the stored File Keys, the encryption system must prepare in advance, collecting all needed File Keys before erasing the Master Key.

Evidently, making every file on the disk available to essential applications is impractical: having the File Keys for every encrypted file in memory would not only completely ruin security by exposing all files to memory attacks, but would waste a significant amount of time and memory. The practical alternative is to keep the list of essential files extremely limited. In our implementation, we have relied on the selective encryption property of file-based encryption systems: core operating system files and applications' working directories are left unencrypted, whereas all data files are encrypted and considered non-essential unless explicitly marked as such. Only the system processes and log-on processes are listed as essential. Any application the user wishes to list as essential, should have its data files exempted from encryption, or listed as essential explicitly. Application recognition methods, such as installer package parsing, help determine the files associated with an application.

We stress that defining applications or files as essential is a security trade-off. Because the keys to essential files must reside in memory in the unattended state, essential files are exposed to memory attacks. If memory attacks were the only type of attack, listing files as essential would be equivalent to leaving them unencrypted. Because that is not so, it is preferable to have a file encrypted and listed as essential. An application defined essential will continue running in the unattended state, but will be vulnerable to memory attacks.

4 Protecting Disk-Based Encryption Systems

So far we have addressed the case of file-based encryption systems, where we require two tiers of keys, with a File Key for every encrypted file. Disk-based encryption systems cannot be protected the same way: as long as a single Master Key encrypts all sectors, it can never be erased. To extend our protection to disk-based encryption system, we isolate the needed properties of file-based systems, and apply them to disk-based systems. Although we have not implemented the protection in a disk-based system, we can define the requirements and propose a realistic design.

We note that all disk encryption modes currently accepted as standard - XTS, XEX, LRW, CMC, and EME [6, 21, 16, 13, 12] directly use the Master Key in all cipher operations. The modes above differ mostly in the tweaking schemes they use to transform the plaintext and ciphertext. The tweak operations involve XOR, multiplication, and sometimes exponentiation modulo a finite field, and their specifics are unimportant for this discussion. To enable erasing the Master Key, we must change the encryption to use

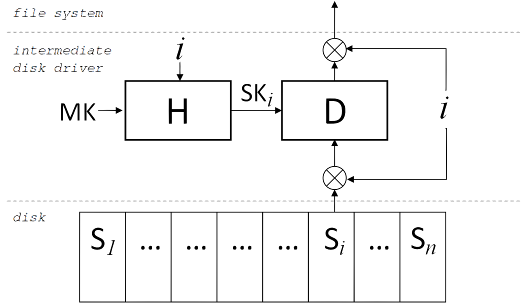


Figure 2: Disk-based encryption system architecture with the proposed modifications.

two tiers of keys instead of one.

We propose to use a set of *Sector Keys* as an extra tier below the Master Key. The following key management requirements must be satisfied to protect against memory attacks:

- Each Sector Key SK_i must be a function of the sector number and the Master Key.
- Each Sector Key SK_i must not reveal the Master Key.
- Each Sector Key SK_i must not reveal SK_j, for $i \neq j$.
- Each Sector Key SK_i must be retained in memory for the duration of the unattended mode, for every needed sector.

These requirements are similar to those of the file-based systems.

The first three requirements are easy to satisfy, if we derive SK_i by using a one-way pseudo-random function H of the Master Key and the sector number, i:

$$\mathrm{SK}_i = \mathrm{H}(\mathrm{MK}, i)$$

Suitable options for H are encrypting i with the Master Key, or using the HMAC construction instantiated with a strong hash function to hash i with the Master Key.

The change above only covers key derivation. Similar requirements apply to tweaking schemes: they must not use the Master Key as a direct input, and they must not expose it. The first requirement makes the tweaking schemes usable in the unattended state. The second requirement means that tweaking schemes which use the Master Key must switch to a one-way function of it, or to another, unrelated key. Some of the existing tweaking schemes do not use the Master Key, and the rest can be adapted easily.

By mandating the use of different encryption keys per sector, we incur a computational cost. For ciphers using a key expansion stage, the standard disk encryption systems set up a single instance of the cipher, and have to go through key expansion only once. We now require one H operation and one key expansion operation per sector processed.

There are several complications which disk-based systems must address. First of all, instead of the issue of essential files as discussed in Section 3.6, disk-based systems must address the issue of essential sectors, the sectors containing data from essential files. The disk-based system can map essential files to a list of their sectors by using file system APIs. At first glance, the disk encryption system must collect the Sector Keys for all essential sectors before entering the unattended state, and retain them for the duration of that state. However, disk-based encryption systems typically encrypt the whole disk, including operating system files. The set of essential files for disk-based systems would by default encompass the whole file system.

Making matters worse, write operations may increase the size of a file, spreading it to new sectors. To support the encryption of new sectors without a Master Key, the encryption system must precompute and store a set of Sector Keys for some or all the free sectors. For example, with AES-256 and a typical hard drive geometry, the space cost is 32 bytes for each 512-byte sector of the hard drive. That would take up roughly 6% of the drive, or 64 gigabytes for a terabyte hard-drive (assuming the whole drive consists of essential files and free space). It would be impossible to hold all Sector Keys in memory and they would have to be kept on disk, occupying some of the disk space.

To avoid the complications, we propose to eliminate the need for storing sector keys for free space and non-essential files.

Note that free sectors are equivalent to newly created encrypted files in the file-based setting. By defining a separate Master Key just for them, it is possible to eliminate the need to keep all of their sector keys in memory. There would be two different Master Keys: one for regular sectors (MK_1), and one for free sectors (MK_2). MK_2 would not be erased when entering the unattended state. For this method to work, the encryption system must distinguish sectors encrypted with MK_1 from those encrypted with MK_2. It could use a bitmap to do so, consuming some disk space. A better solution is possible if the encryption system can cooperate with the disk sector allocation algorithms. For example, by allocating regular sectors from the beginning of the disk and newly occupied sectors from the end, it can use just two counters to keep track of the areas encrypted with each key. In the unattended state, newly written sectors should initially be encrypted with MK_2, and can be moved to the regular area and re-encrypted with MK_1 once the user logs back on.

Similarly, a separate Master Key can be used for essential files. The key would not be erased in the unattended state. The reasoning then proceeds as in the last paragraph, and in fact, the same key used for free sectors (MK_2) can be used for sectors belonging to essential files. As in the file-based setting, essential files are not protected against memory attacks.

The proposed measures reduce the number of sector keys we need to retain. The encryption system would only need to retain sector keys corresponding to open non-essential files, and only if it does not intend to suspend the applications holding those files open. The reduced space requirements should allow for a practical implementation.

5 Open Issues and Future Work

This section discusses two types of data left unsecured against memory attacks: some system files remaining exposed to the attacker, and applications retaining confidential data in their address space.

There are special files to which the encryption system cannot deny access, most obviously the page file, constantly used by the virtual memory manager to swap memory pages out to disk and back. Those memory pages may contain confidential information from an application's memory space. The virtual memory manager is directly in charge of this file, and may access it at any time. Memory-mapped files are treated the same way. The encryption system must keep these files available, and therefore their contents will be vulnerable to memory attackers.

Finally, confidential data may be kept in application memory space, where memory attacks would expose it. An encryption system could suspend an application and evict its address space to disk, where it can encrypt the data. When returning to normal, the encryption system would restore the original memory contents and then resume the application. We have not yet attempted to implement such a system.

6 Conclusion

We have described a technique which protects the keys and data in a drive encryption system against memory attacks. We have defined a combined model of the attacker, and detailed the steps needed to withstand memory attacks in a file-based encryption system, while keeping the computer in a limited but operational state. Finally, we have described the steps needed to implement our protection in disk-encryption systems.

If hardware-based drive encryption systems receive wider acceptance, we expect the impact of memory attacks to decline. Until then, we propose a solution which provides protection against memory attacks at no hardware cost.

Acknowledgements

The author would like to thank Pavel Berengoltz and Adam Carmi of Safend Ltd. for their contributions and reviews. Special thanks to Dr. Benny Pinkas from Haifa University for his review and advice on publication, and to Dr. Yaron Sella for his review and suggestions. Thanks to Julie Feinberg for her language review, and to the anonymous reviewers for providing pointed and motivating feedback.

References

[1] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *TCC '09: Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, pages 474–495, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] Darrin Barral and David Dewey. Plug and root: The USB key to the kingdom. `http://frozencache.blogspot.com/`, 2005.

[3] Michael Becher, Maximilian Dornseif, and Christian Klein. Firewire: all your memory are belong to us. In *Proceedings of CanSecWest*, 2005.

[4] Adam Boileau. Hit by a bus: Physical access attacks with firewire. Presentation at RUXCON 2006, Australia, 2006.

[5] Ellick M. Chan, Jeffrey C. Carlyle, Francis M. David, Reza Farivar, and Roy H. Campbell. Bootjacker: compromising computers using forced restarts. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 555–564, New York, NY, USA, 2008. ACM.

[6] Morris Dworkin. Recommendation for block cipher modes of operation: The XTS-AES mode for confidentiality on block-oriented storage devices. NIST Special Publication 800-38E, 2009.

[7] Niels Ferguson. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista. `http://go.microsoft.com/fwlink/?LinkId=82824`, 2006.

[8] Trusted Computing Group. PC client work group platform reset attack mitigation specification. `https://www.trustedcomputinggroup.org/resources/pc_client_work_group_platform_reset_attack_mitigation_specification_version_10/`, 2008.

[9] Shay Gueron. Advanced encryption standard (AES) instructions set. `http://software.intel.com/en-us/articles/advanced-encryption-standard-aes-instructions-set/`, 2009.

[10] Peter Gutmann. Data remanence in semiconductor devices. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 4–4, Berkeley, CA, USA, 2001. USENIX Association.

[11] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.

[12] Shai Halevi and Phillip Rogaway. A parallelizable enciphering mode. In *Proc. RSA Conference 2004 Cryptographers Track*, pages 292–304. Springer-Verlag, 2003.

[13] Shai Halevi and Phillip Rogaway. A tweakable enciphering mode. In *Advances in Cryptology - CRYPTO 2003*, pages 482–499. Springer-Verlag, 2003.

[14] Owen Harrison and John Waldron. Practical symmetric key cryptography on modern graphics hardware. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 195–209, Berkeley, CA, USA, 2008. USENIX Association.

[15] Peter Kleissner. Stoned bootkit. `http://www.stoned-vienna.com/`, 2009.

[16] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In *Advances in Cryptology CRYPTO 2002*, pages 31–46. Springer-Verlag, 2002.

[17] Patrick McGregor, Tim Hollebeek, Alex Volynkin, and Matthew White. Braving the cold: New methods for preventing cold boot attacks on encryption keys. `https://www.blackhat.com/presentations/bh-usa-08/McGregor/BH_US_08_McGregor_Cold_Boot_Attacks.pdf`, 2008. BitArmorSystems,Inc.

[18] Moni Naor and Gil Segev. Public-key cryptosystems resilient to key leakage. In *CRYPTO '09: Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, pages 18–35, Berlin, Heidelberg, 2009. Springer-Verlag.

[19] Jurgen Pabel. Frozen cache: A blog about the development of a general-purpose solution for mitigating cold-boot attacks on full-disk-encryption solutions. `http://frozencache.blogspot.com/`, 2009.

[20] David R. Piegdon. Hacking in physically addressable memory: A proof of concept. Seminar of Advanced Exploitation Techniques, 2007.

[21] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Advances in Cryptology - ASIACRYPT 2004*, pages 16–31. Springer-Verlag, 2004.

[22] Joanna Rutkowska. Why do I miss Microsoft BitLocker? `http://theinvisiblethings.blogspot.com/2009/01/why-do-i-miss-microsoft-bitlocker.html`, 2009.

[23] Adi Shamir and Nicko van Someren. Playing "hide and seek" with stored keys. In *FC '99: Proceedings of the Third International Conference on Financial Cryptography*, pages 118–124, London, UK, 1999. Springer-Verlag.

[24] Sergei P. Skorobogatov. Data remanence in flash memory devices. In *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.