RAM is Key
Extracting Disk Encryption Keys From Volatile Memory

Brian Kaplan
bfkaplan@alumni.cmu.edu

Advisor: Matthew Geiger
mgeiger@cert.org

Carnegie Mellon University
May 2007

Thesis Report

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science in Information Security Policy and Management

Abstract

The increasing mobility of computing devices combined with frequent stories of privacy breaches and identity theft has thrust data encryption into the public eye. This heightened awareness of, and demand for, encryption has resulted in the development of a number of strong encryption solutions that emphasize usability. While encryption can help mitigate the threat of unintentional data exposure, it is equally capable of hiding evidence of criminal malfeasance. The increasing accessibility and usability of strong encryption solutions present new challenges for digital forensic investigators, whose traditional response methodologies leave them largely unprepared to deal with pervasive strong encryption.

In this paper we address the shortcomings of the traditional forensic response methodology with respect to encryption. We develop and discuss a variety of practical techniques for dealing with the use of encryption to conceal evidence. Our research highlights the virtues of volatile memory analysis by demonstrating how key material and passphrases can be extracted from memory to facilitate the analysis of encrypted media in a forensically sound manner. We also present a proof of concept tool capable of automatically extracting key material from a volatile memory dump and using it to decrypt an encrypted disk image.

Table of Contents

Introduction 3
Background 3
  Disk Encryption 3
  Shortcomings of the Traditional Methodology 4
  Live Acquisition 5
  Volatile Memory 6
Key Identification 6
  Previous Work 7
  Applicability 7
  Brute Force 8
  Real World Obstacles 8
  Cryptosystem Specific Techniques 10
  Key Schedule Technique 11
  Defeating Key Identification 11
Making Use of Volatile Memory Structure 12
  Background 12
  Reducing the Virtual Address Search Space 12
  Pool Memory 14
    Background 14
    Previous Work 16
    Small Pool Allocation Header 16
    Pool Tagging For Key Recovery 17
  Operating System Structures as Key Pointers 18
    Related Work on Linux 18
    Adapting to Windows 19
Putting it all Together 20
  Volatile Memory Incentives 20
  A Usable Tool: Disk Decryptor 20
Related Work: Passphrase Recovery 22
  Background 22
  Case Study: Truecrypt 22
  Full-Disk Encryption Passphrase Recovery 23
Future Research 23
Conclusion 24
References 26

Introduction

In the wake of a recent spate of high-profile privacy breaches, the need for better ways to protect private data has become widely apparent. The United States government is currently conducting a review of full-disk encryption solutions as a result of a June 2006 mandate from the Executive Office of the President recommending the use of encryption on mobile computers and devices [1]. While strong encryption can be used constructively for protecting against unintentional data exposure, it has also long been embraced by criminals who use it to conceal evidence of their illicit activity [2]. Once considered a tool for only the technically savvy, push-button encryption solutions have made strong encryption accessible to the masses. Seen as a boon by privacy activists, the increasing accessibility and usability of strong encryption solutions troubles digital forensic practitioners. As has been apparent since Phil Zimmerman introduced strong, publicly available encryption with PGP in the 1990's, one cannot prevent criminals from using strong encryption. One can, however, equip investigators with the tools and techniques necessary for effectively dealing with and responding to its use.

Background

Disk Encryption

Before discussing the implications of encryption as it pertains to forensics, it is helpful to highlight some of the defining characteristics of disk encryption. This research focuses on software-based, "on the fly" encryption (OTFE). In this approach, data is transparently decrypted as it is requested from the disk and encrypted before it is written to the disk. Although OTFE can happen at the file level, as is the case with Microsoft's Encrypting File System (EFS) [3], we are primarily concerned with encryption at the sector level on the storage device.

Sector level OTFE is accomplished by inserting a cryptographic "filter" that intercepts access to the disk or other storages device and allows upper layers to read and write to the disk without being concerned about decrypting the data being read or encrypting the data being written (see figure 1). When a user successfully authenticates to the cryptosystem, a symmetric encryption key is typically unlocked and passed to the cryptographic filter. Once this filter receives the key, it can begin transparently decrypting and encrypting all data that passes through it "on the fly." Without the correctly keyed filter layer, however, all data read from the disk is unintelligible ciphertext. For this reason, it is often said that OTFE is only meant to protect data "at rest." That is, once an encrypted volume is mounted on a running system it usually remains accessible to the system's users as if it was plaintext until it is unmounted or the system is shut down.

OTFE at the sector level often takes the form of full-disk encryption where essentially every sector of a disk is encrypted. It can also take the form of container-based (virtual volume) encryption where an encrypted container file is mounted and accessed as a logical device. We focus primarily on full-disk and virtual volume encryption, which arguably pose the greatest challenge to digital forensic practitioners.

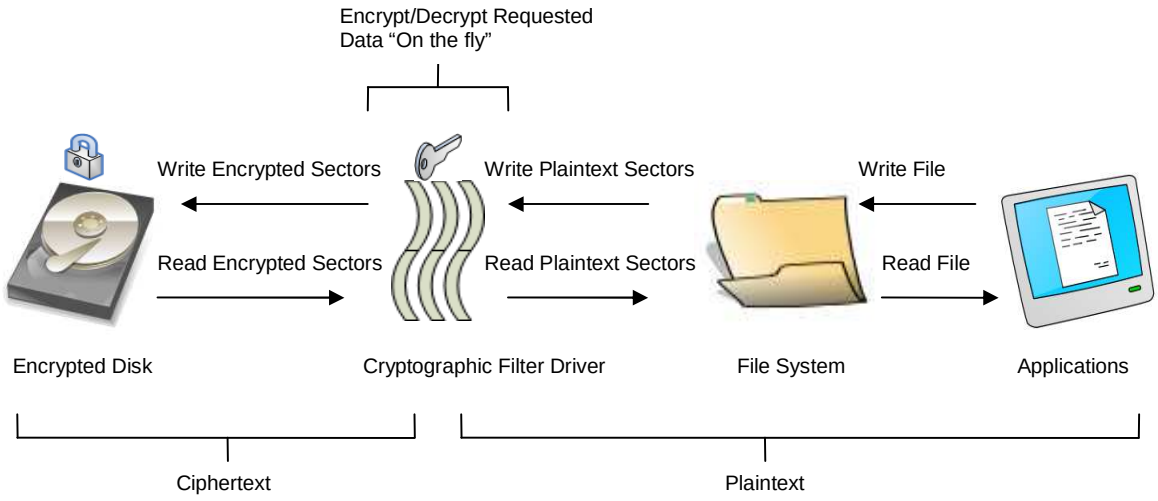

Figure 1: Simplified Sector Level "On the Fly" Encryption

Shortcomings of the Traditional Methodology

The traditional "disk centric" approach to forensic acquisition and analysis falls short when dealing with encryption. An investigator confronted with a machine to seize typically "pulls the plug" and creates a bit-for-bit image of the physical hard drive [4]. The problem with this technique is that if OTFE was being used on that system, the investigator has probably destroyed the best chance of recovering the plaintext data. While the system is running, encryption keys are loaded and the software is transparently decrypting data requested from the disk before it is processed by applications or presented to the user. However, once the machine is powered off, that transparent decryption layer with the loaded encryption key is purged from volatile memory and the investigator is left to image an unintelligible encrypted disk.

With just an encrypted disk image, an investigator's only real option is to recover the suspect's authentication credentials in order to proceed with their analysis. The credentials can be used to extract the data encryption key, which in turn, can be used to decrypt the disk. These credentials are typically in the form of a passphrase but can also include USB tokens, smartcards, and even biometrics. Of course, if the data encryption key can be extracted directly, it does not matter what methods of authentication are in use. In some cases, particularly within corporate environments, alternative decryption keys can be recovered from either a key escrow or disaster recovery disk [5,6]. More often than not, however, an investigator will be forced to recover credentials by other means.

Assuming standard passphrase-based authentication is in use, the investigator has the option of performing a computationally expensive and time-consuming brute force or dictionary attack to guess the passphrase. Some encryption software packages now warn its users if they try to create weak passphrases that are susceptible to such attacks, so this technique is becoming much less practical. Applicable laws permitting, a second, perhaps

less time-consuming, option is to recover the user's credentials by forcing the suspect to give them up. The British government, for example, plans to implement a controversial portion of the Regulation of Investigatory Powers (RIPA) act making it a crime to refuse an order to release encryption keys to authorities [7]. Depending on one's interpretation of the fifth amendment to the U.S. constitution, one could be compelled to produce their encryption key or passphrase under U.S. law as well [8].

Encryption software vendors have responded to the legal trend by providing their users with "plausible deniability." These features allow users to ostensibly comply with key disclosure orders by creating two sets of encrypted data: a benign one for which the passphrase can be shared for purposes of compliance, and an undetectable or "plausibly deniable" one containing the real data the user wishes to protect. Truecrypt, for example, boasts the ability to create hidden volumes within an encrypted outer volume. If a user is ever forced to divulge his or her passphrase, he or she can simply give the passphrase that unlocks the outer volume. Their adversary would have no way to know whether another hidden volume exists. Some full-disk encryption software such as Drive Crypt Plus Pack enable users to hide an entire operating system within the free space of another innocuous operating system in order to achieve the same type of plausible deniability. Clearly one cannot depend upon credential-based data recovery even where forced key disclosure is permitted by law.

Live Acquisition

A better solution for dealing with encryption is to perform a "live acquisition" in which the investigator creates a bit-for-bit image of the logical device while the system is still running. In this scenario, the investigator simply reads every block of data on the logical device, each of which is transparently decrypted by the encryption software, resulting in a plaintext representation of the suspect's disk. Creating an image of a running system, however, results in a "smeared" representation of the disk over the course of the imaging process due to the data changes associated with natural disk activity that occurs on a running system. This disk activity has the potential to overwrite data with evidentiary value, causing some to call its virtues into question. Furthermore, this technique relies on an untrusted operating system to present what is actually on the disk and could fall prey to anti-forensic techniques [9].

Some of these factors cause live acquisition techniques to be viewed as less forensically sound than the traditional "dead disk" acquisition techniques. The same view suggests that live acquisition should only be used when necessary, such as when disk encryption is known to be in place. This, of course, necessitates a reliable, unobtrusive, way to determine if encryption is in use and the requisite modifications to the standard forensic response methodology. The result is a more complicated and error-prone acquisition process that requires more highly skilled forensic acquisition technicians. Despite these issues, live acquisition is considered by many to be the only practical way to deal with the growing problem of encrypted data, so some are willing to overlook its faults until better techniques are developed.

Volatile Memory

When forensic practitioners "pull the plug" on a system, they are not only potentially locking themselves out of an encrypted system, they are also losing troves of volatile information contained in physical memory (RAM). Data such as running processes, network connections, open files and fragments, logged in users, et cetera, all provide context about the runtime state of the system that can be used for corroboration with evidence found during traditional disk analysis. To date, this volatile data has been largely ignored by forensic practitioners due to a lack of effective tools, training, and time [10]. It has, however, been an active area of recent research resulting in a variety of tool releases, each of varying degrees of usability [11,12,13,14]. It remains to be seen whether any of these tools will provide enough return on training and time investment to encourage their integration into standard forensic methodologies.

While it is beyond the scope of this paper to discuss all of the virtues of incorporating volatile memory acquisition and analysis into the standard forensic response methodology, we do demonstrate how volatile memory can be leveraged in the analysis of encrypted disks. That is, by capturing volatile memory prior to performing the standard "dead disk" imaging response, an investigator can extract the cryptographic keys from the memory image to decrypt and analyze the disk image using standard tools and procedures. This even applies for "plausibly deniable" encrypted volumes that were mounted at the time the memory image was created. It is our hope that by providing a practical and forensically sound alternative to live disk imaging for dealing with encrypted media that the requisite return on investment for incorporating volatile memory acquisition into standard forensics practices will be achieved. With such change, we believe the forensics community will be better equipped to deal with the growing threat posed by strong data encryption in the hands of criminals.

Key Identification

The basic principle that makes it possible to extract encryption keys from memory is that any program, including both its data and instructions, must first be loaded into physical or main memory before being run by the processor (see figure 2). Therefore any software that performs encryption must, at some point, have the instructions and requisite data (in this case, the key material) loaded into the system's main memory. If one has access to that memory, one presumably also has access to the key material. The trick, of course, is being able to uniquely identify it among the hundreds of megabytes or even gigabytes of other data.

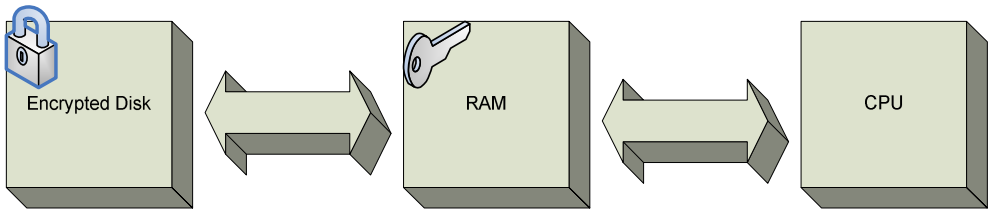

Figure 2: Key Loaded in Main Memory

Previous Work

The notion of extracting cryptographic material from a large body of plaintext is not a new one. In 1998, Adi Shamir and Nicko van Someren discussed a “lunchtime attack” for efficiently locating cryptographic key material in large amounts of data [15]. The attack focused on locating asymmetric plaintext RSA keys on a plaintext disk. Although such keys are often encrypted when stored on disk, the idea of exploiting the cryptographic properties of the keys to efficiently find them on disk was rather novel.

Eight years later, Tobias Klein published a paper and proof of concept tool that facilitates the extraction of RSA keys from a process memory dump. Klein took a different approach than Shamir and van Someren, exploiting the standard storage formats for private keys and SSL certificates as opposed to the mathematical properties of the keys themselves. The standard storage formats for RSA private keys and SSL certificates, as described in PKCS #8 and x509 v3 respectively, were used to create a signature for locating them in memory. Using this signature, a simple pattern match could be done to extract the candidate asymmetric keys in their plaintext form, which could then be verified using an external tool such as openssl [16]. This signature-based search technique could be combined with Shamir and van Someren’s heuristics to create an efficient and effective way to extract plaintext asymmetric key material from full volatile memory dumps.

Applicability

While there are some practical techniques for extracting asymmetric key material from volatile memory, it is important to understand why these techniques cannot be applied directly to key extraction for disk decryption. The principal reason is that these techniques exploit properties of, and thus apply solely to, asymmetric cryptography whereas disk encryption is done almost exclusively with symmetric cryptography. Asymmetric algorithms are on the order of 1,000 times slower than their symmetric counterparts and are thus unacceptably slow for the frequent cryptographic operations required for high-throughput encrypted disk access [17].

The inherent differences between asymmetric and symmetric keys are what make extracting the latter so much more challenging. Asymmetric key pairs are not only related to each other but also have measurable mathematical properties unto themselves. Symmetric keys, however, are relatively small pseudorandom numbers. By definition, they do not have any measurable mathematical properties other than being cryptographically random. For this reason, the techniques outlined by Shamir and van Someren for testing the mathematical properties of candidate asymmetric keys do not apply to the symmetric space. For similar reasons, Klein’s signature-matching technique also does not apply. Symmetric keys have no real structure other than being pseudorandom numbers; there is no standard format for storing them and thus no signature on which to do pattern matching.

The one technique used by Shamir and van Someren to identify asymmetric keys that, at least ostensibly, has applicability to symmetric key identification is entropy analysis. Since symmetric keys are essentially random numbers, it makes sense to try to identify

them by testing for their entropy or randomness. Unfortunately this technique falls short because symmetric keys are so much smaller than asymmetric keys. It is much more difficult to get a meaningful entropy value for a typical 128 or 256 bit symmetric key block than it is for a common 1,024 or 2,048 bit asymmetric key block. In our experiments, the false positive rates for entropy analysis were much too high to be useful for discriminating between keys and other data. This can be attributed to both the relatively small symmetric key sizes, as well as the presence of other highly entropic sequences such as compressed or encrypted data resident in volatile memory. With few, if any, distinguishing characteristics, one can see that extracting symmetric key material from a large body of data can be a challenging problem. Clearly different techniques than those used for asymmetric key extraction are necessary.

Brute Force

The theoretical strength a cryptosystem is often quantified by its ability to resist brute force attacks. As such, it makes sense to consider this avenue of attack first. While a brute force attack on the entire keyspace of a typical asymmetric or symmetric key would be computationally infeasible, a brute force attack within the search space consisting of all possible key-sized blocks of contiguous bits in volatile memory would not.

For an asymmetric key, each trial decryption is relatively expensive, but the attacker has the advantage of being able to generate a known plaintext/cipher text pair using the public key and an arbitrary block of plaintext. The attacker can use this known plaintext to check each candidate private key in the brute force attack. This is not always possible when dealing with symmetric keys. Still, if the target ciphertext is a disk, the attacker often knows, for example, the plaintext and standard location for the boot loader. File system structures and zero-filled sectors can also serve as useful known plaintext values. While finding a known plaintext/cipher text pair for symmetric disk encryption is often possible, it is not guaranteed. Without one, it becomes much more difficult and error prone to check candidate keys in the search space.

A brute force attack on the limited search space represented by volatile memory, while feasible, can still be very time consuming. One could significantly reduce the search time by adding in a lower-bound entropy constraint. This, of course, assumes that the entropy calculation is less computationally expensive than the trial decryption and known plaintext comparison operations, but that should typically be the case.

Real World Obstacles

The brute force approach on a limited search space is, in theory, an attractive option for extracting keys from memory. In practice, however, it is not as straightforward as it may seem. Symmetric cryptosystems consist of more than just a block cipher and secret key. For example, understanding the details of the mode of operation, that is, how blocks of ciphertext output by the block cipher are dependent upon, or chained to, one another can be a crucial detail. Similarly, the specific details about how initialization vectors (IV) are calculated can be important. Without fully understanding these details, the challenge of brute forcing the keys is compounded by effectively needing to brute force the encryption implementation at the same time.

While some vendors do expose this level of detail about their cryptosystems, it has been our experience that the vast majority do not. Some use well known block modes while others modify them to varying degrees. We can only trust that this is not done in the false hope that obscurity will improve the security of their systems, but rather as an inherent improvement to the implementation of the cryptosystem. The relative secrecy and reluctance by vendors to use well known block cipher modes may be attributed to recent "watermarking" attacks that allow an attacker to craft a specific file that has the potential to give away some information about the existence of plaintext on the disk, without knowledge of the secret key [18,19]. As a stopgap measure, some vendors may have made modifications to the standard block modes and IV calculation techniques they were using in an attempt to prevent these types of attacks [20].

Stronger block modes designed specifically for disk encryption have since been developed by the cryptographic community. Some vendors appear to be moving toward these standards and away from rolling their own solutions. This standardization is helpful because it makes decryption easier, but it is not a necessity. As Kerckhoffs' principle states, any good cryptosystem should remain secure even if every detail about the system is made public except for the key itself [21]. While obscuring details of a cryptosystem does present real world challenges, all of the cryptographically important data and instructions must nonetheless be loaded into memory before they are used. Given enough time, both the secret key and the exact details of each cryptosystem's operation can be discovered and used to decrypt the ciphertext generated by even the most proprietary and closed implementations.

Until either a standard is agreed upon for disk encryption or vendors decide to release full details about their cryptosystems, there will always be some level of reverse engineering required to successfully decrypt a disk with a known key. It should be mentioned, however, that knowing all details of the cryptosystem will not always be needed to identify the key in a brute force attack. The formula used in the initialization vector calculation, for example, may not be necessary, depending on the block mode being used. In some modes, such as cipher block chaining (CBC), the initialization vector only affects a single block of plaintext during decryption (see figure 3). Thus, one can use an arbitrary IV and properly decrypt all blocks except for the ones dependent upon the IV. For disk decryption this typically means that one block (typically 16 or 32 bytes) in each sector (typically 512 bytes) will be unintelligible garbage but the rest will be properly decrypted. This should be sufficient to verify the candidate keys and facilitate a brute force attack assuming, of course, that the known plaintext string is contained in the properly decrypted portion of the sector. While understanding all details of a cryptosystem may not be necessary for key extraction, it is necessary for making practical use of the key for full ciphertext decryption.

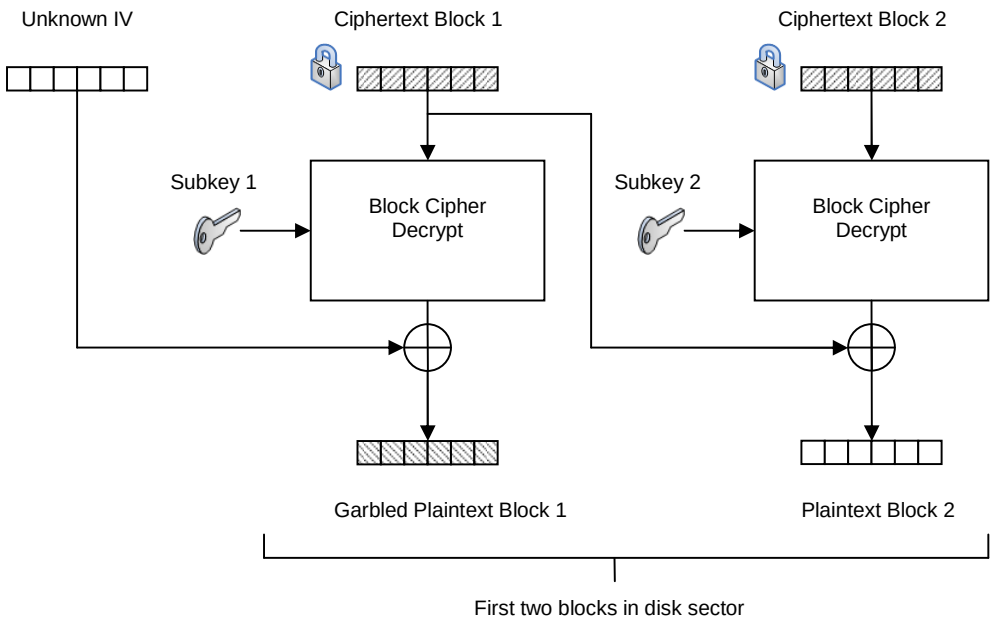

Figure 3: CBC mode decryption of first two blocks in a sector with an unknown IV

Cryptosystem Specific Techniques

While brute forcing all key-sized blocks in a volatile memory dump may be a viable option for disk decryption when compared with brute forcing the key space, it can still be computationally expensive. Since cryptosystem-specific details are already required to perform the validity checks in a brute force attack, we looked at specific details about each individual cryptosystem implementation that could potentially be exploited to locate a key in memory more efficiently.

PGP, for example, boasts that their PGP Desktop product protects against “memory static ion migration”, an attack in which a static charge “burned into” memory by storing the key in the same location for long periods can be read by an attacker using special hardware. This threat is mitigated by, “keeping two copies of the key in RAM, one normal copy and one bit-inverted copy, and inverting both copies every few seconds [22].” This type of information can be used to uniquely identify a key in a memory dump. By taking each key-sized window of consecutive bits in the memory dump, inverting the bits, and checking the rest of the memory dump for that sequence of inverted bits, one can fairly reliably extract candidate PGP disk keys.

This attack can be done more efficiently and with fewer false positives by first using entropy analysis as a filter because checking the entropy of a block is much less expensive than checking the rest of a memory dump for a particular sequence of inverted bits. Despite any optimizations, this technique is still a quadratic-time function and is thus significantly slower than a linear-time brute force attack on the volatile memory keyspace. It does, however, have the advantage of possibly not requiring a known

plaintext/ciphertext pair to identify a key. Other cryptosystem specific attacks exist as well, but few provide great advantages over brute force.

Key Schedule Technique

Since neither brute force nor cryptosystem specific attacks are ideal, we looked into developing a more generic way to identify key material in a memory dump. As previously mentioned, symmetric keys are random numbers and have no intrinsic, testable mathematical properties. The way in which they are used and implemented, however, could be exploited to identify them. Most modern symmetric key cryptosystems are implemented as product ciphers where several iterations or "rounds," each consisting of a series of transformations are done to achieve the desired properties of confusion and diffusion. Each round uses a different subkey derived from the master key using a derivation function called the key schedule algorithm. The master key combined with its subkeys for each round is collectively known as the expanded key or key schedule. This is useful for identifying keys in volatile memory for two main reasons:

- The key schedule provides a testable mathematic relationship between the master key and the subkeys.
- The key schedule is often pre-computed and stored with the original key for performance reasons.

This means that by simply knowing the encryption algorithm being used (and thus the key schedule algorithm being used) one can go through the memory dump computing a key schedule for each key-sized block checking if the computed subkeys appear anywhere in the memory dump. We found that this provides a convenient and accurate way to validate whether a random number is a valid key for a particular block cipher algorithm without knowing any information other than the algorithm itself and key size being used. Because these basic details are provided by nearly all of the cryptosystem vendors, this technique is an attractive one for extracting symmetric keys from memory.

Defeating Key Identification

All of the techniques for key extraction outlined thus far are fairly effective, but unfortunately they are also easily defeated. Both brute force and the key schedule test assume that the key data being tested is stored contiguously in memory. Both techniques would fail or become prohibitively expensive to perform if the key and or key schedule was stored in parts and there was no easy way to predict the positions of each chunk of key material relative to the first one. The key schedule attack is particularly susceptible to this as it is quite possible that the key schedule subkeys would be stored separately from each other or from the master key. It is also possible that they could be stored in a complicated structure that breaks up their assumed contiguity in memory. Some tools may not even store the full key schedule, re-computing it when necessary. This is not a likely scenario for disk encryption where it would greatly hinder performance, but it is an effective means of preventing key verification using the key schedule test. Of course, inserting "red herring" key schedules into memory would also be an effective deterrent for the key schedule verification technique. With a known plaintext/ciphertext pair, however, the fake keys could be quickly identified and discarded.

Making Use of Volatile Memory Structure

Background

Each of the key extraction techniques discussed thus far treat memory as a blob of unstructured data. In reality, however, it is highly structured. Ignoring this structure is the traditional forensics equivalent of treating a disk image as an unstructured mass of data and performing data carving or string searches, rather than first making use of all of the existing file system structures. Instead, by understanding both the structure of memory and how encryption packages use it, we can better understand where the keys might be stored. Unless otherwise specified, we focus primarily on IA-32 systems running Microsoft Windows, but the general concepts should be applicable on other systems.

Microsoft Windows supports two processor access modes, user mode and kernel mode. This distinction is made to prevent standard user mode applications from accessing or modifying critical operating system data thereby affecting the stability or security of the system. The virtual address space is similarly divided into kernel and user space ranges. The user space is where Windows maps the user mode processes, their data, and user mode libraries. The kernel space is occupied by the operating system itself and kernel mode drivers. An important distinction is that kernel mode code is trusted and can access both user and kernel space, whereas user mode code can only access its own private user memory space or context [23].

Reducing the Virtual Address Search Space

Virtually all full-disk and container encryption software have both user mode and kernel mode components. The user mode component is typically the administrative interface with which the user interacts to control the behavior of the cryptosystem. The kernel component typically takes the form of a device driver that handles all of the encryption and decryption operations. The driver essentially intercepts read and write requests made to a particular device and decrypts or encrypts the requested data on the fly before passing it on to the next level in the device chain. This allows the operating system and applications to operate as if the disk was not encrypted because the cryptographic operations happen transparently. Because the driver is the component that typically handles the cryptographic operations, the first assumption that could be made about the location of the key is that it probably resides in kernel memory. This means that on a 32 bit machine, the lower two gigabytes of virtual address space (or three gigabytes if the /3GB switch is applied in the boot.ini) can be eliminated because these memory locations are reserved for user space (see figure 4) [23]. This reduction of at least 50 percent of the search space is certainly a good start.

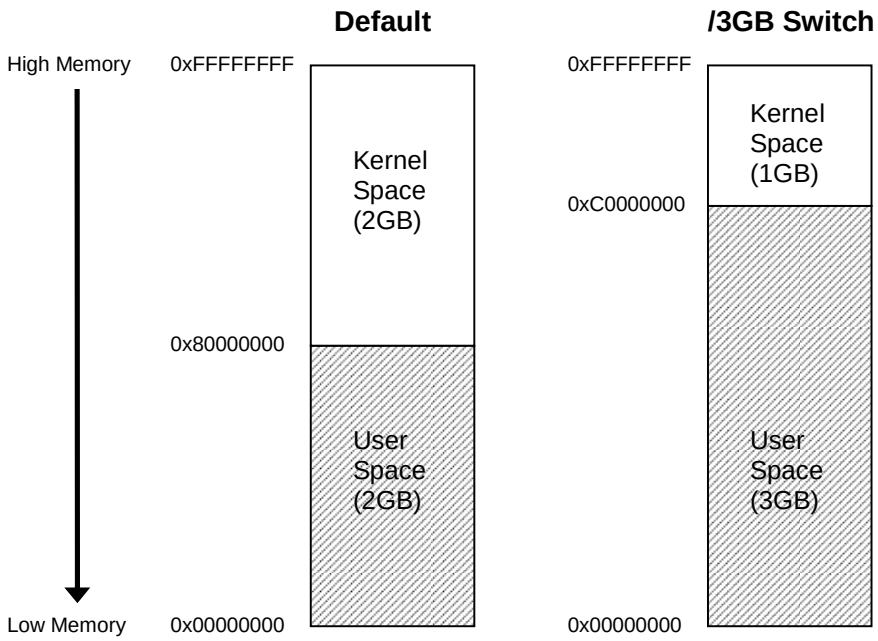

Figure 4: Simplified Virtual Address Space Layout for 32-bit Windows

It is important to note that on a full-disk encrypted system, the entire virtual address space will not typically be available to the investigator. This is because the page file, a structure on the disk into which the operating system stores some memory contents to free up physical memory and improve system performance, resides on disk and is thus encrypted. An encrypted page file cannot be combined with the physical memory dump to reconstruct the full virtual address space, which means that the investigator only has access to the pages loaded into physical memory at the time the memory dump was created. Because physical memory is often much smaller than virtual memory, a significant portion of the virtual address space will be unavailable to the investigator.

Fortunately, since the encryption code and key material are used so frequently, they are rarely, if ever, paged out to the disk. This means that the encrypted data in the page file is not likely to contain the key material. Furthermore, most cryptosystem vendors incorporate even the small probability of key material being paged out to the disk into their threat models and take precautions to ensure that it never happens. This is a much more serious threat for container-based encryption software because the page file is not typically encrypted in this case. If the key was present in the page file, it could be extracted from a standard forensic disk image and used to decrypt an encrypted container without the passphrase. The makers of Truecrypt, a popular open-source virtual volume encryption product, even state in their user guide that sensitive data is stored in non-paged kernel memory to avoid the data being leaked to the page file [24]. Whether for performance or security reasons, one can be reasonably certain that the key material will be stored in the pages marked for kernel mode access and always reside in physical memory, further narrowing the search space.

Clearly the search for keys can be focused using some high-level properties of memory, but it is still essentially a brute force search on a reduced key space. One would ideally like to find the key in memory with as few false positives as possible and without resorting to a computationally intensive brute force approach. To do so, the memory structure needs to be broken down to a more granular level. Klein's signature-based key finder uses user mode process memory dumps as opposed to full memory dumps to extract digital certificates. This targets the search to just the memory owned by the process using the certificates, resulting in much more efficient searches. Using a similar technique, one might try extracting the driver's in-memory image from volatile memory and focus the search on that small chunk of data. However drivers, like user mode programs, are subject to being paged out to disk, making the local memory of the driver's in memory executable image an unlikely place to store sensitive data such as key material. With an understanding of how sensitive data is allocated in drivers, however, techniques that lead even more directly to keys can be developed.

Pool Memory

Background

There are a number of different ways to allocate kernel mode memory in a driver. For discrete, long-term storage allocations, such as relatively static encryption keys, Microsoft recommends allocating memory from the pool [25]. Memory is normally managed and allocated in units called pages, which are commonly four kilobytes in size. This large allocation size can waste significant portions of scarce kernel memory because memory requests are rarely page aligned and will often be much smaller than a page, resulting in internal fragmentation. To solve this problem, a range of pages are combined to form a pool from which requests smaller than a page are separately managed and allocated (see figure 5). The pool can be thought of as a kernel mode equivalent of the user mode heap. Just as one might allocate a block of memory in user mode using `malloc`, one can call one of the `ExAllocatePoolXxx` functions to allocate a block of kernel memory.

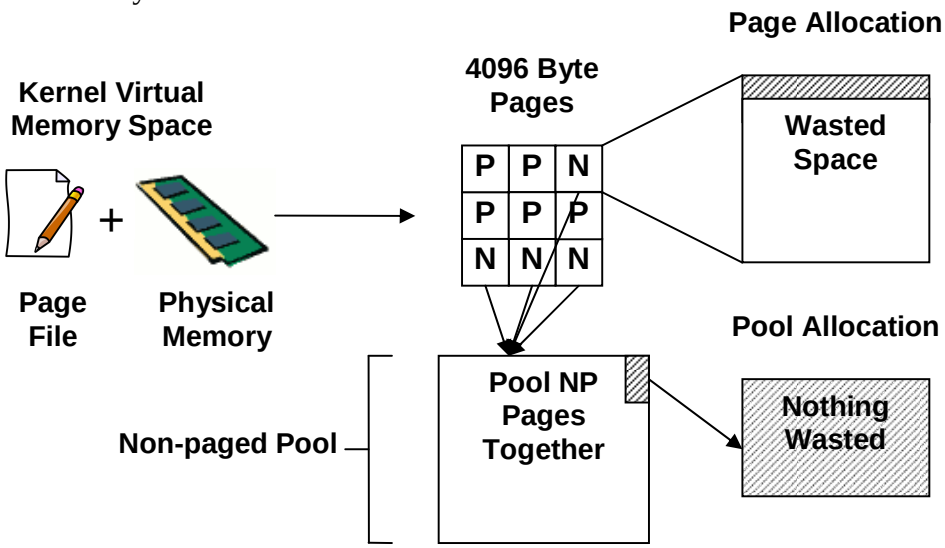

Figure 5: Simplified illustration of space saved by allocating from a pool

It is important to note that the Windows memory manager creates two types of memory pools, a paged pool and a non-paged pool. The non-paged pool consists of a range of virtual addresses that should never be paged out to disk and thus always reside in physical memory. The paged pool is a range of virtual memory pages that can be paged out to disk at any time and therefore may incur a page fault upon access [23]. By specifying the value of the `POOL_TYPE` parameter in the `ExAllocatePoolXxx` function, one can control whether the memory being requested will be allocated from the paged or non-paged pool [26].

As previously mentioned, most cryptosystems take precautions to ensure that sensitive information such as key material does not get paged out to disk. The obvious way to do this is to allocate memory from the non-paged pool by specifying the appropriate `POOL_TYPE` value in the allocation function. We also know that throughput is very important for disk encryption. Page faults are computationally expensive and can be avoided by using non-paged allocations. Of course, non-paged memory cannot be used for everything as physical memory is a finite resource. Non-paged pool allocations always take up space in physical memory, so it is recommended that the paged pool be used whenever possible. The majority of pool allocations made by drivers should therefore reside in the paged pool leaving the sensitive and frequently used ones, such as key material, in the non-paged pool (see figure 6).

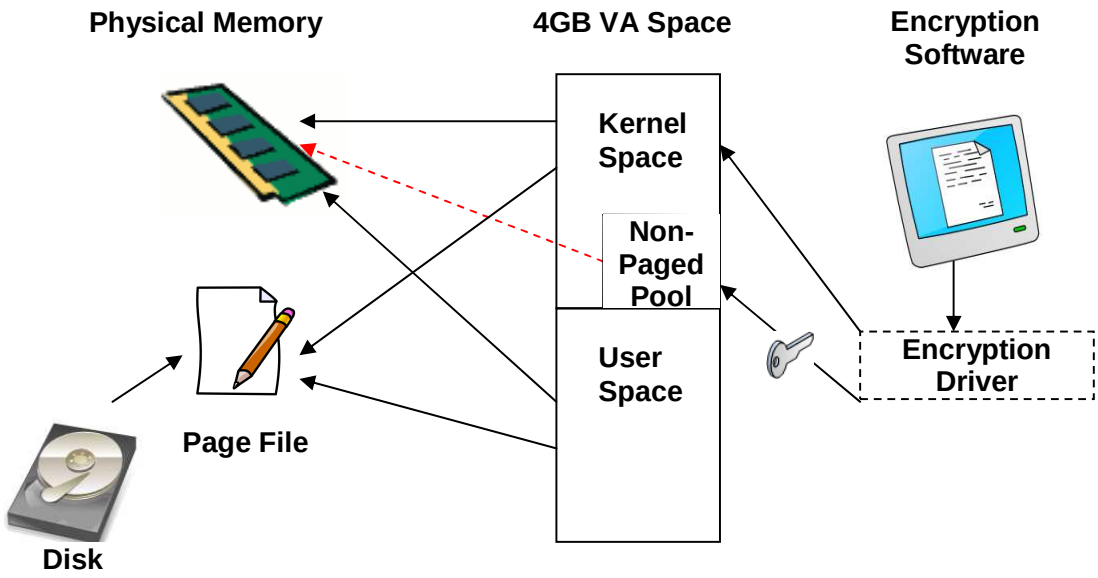

Figure 6: Simplified illustration of key material in non-paged pool memory

Previous Work

Pool memory has been the focal point of prior volatile memory research because it is highly granular and contains many of the important structures, objects, and lists representing the state of the operating system. Pool memory has been used to detect when attempts are made to hide kernel objects and so assist in the cross view detection of rootkits. Chris Carr outlined techniques for finding pool memory ranges as well as the allocations and objects within them. He also released a proof of concept tool called GrepExec that uses the techniques outlined in his paper to identify executive objects in pool memory [27]. Similar techniques, focusing more on their application in forensics, have been developed by Andreas Schuster [28,29]. He released a tool called PoolFinder that uses a related, but more thorough, signature-based pool allocation detection mechanism. PoolFinder performs a brute force scan of the entire memory dump checking every properly aligned sequence of bytes against its signature for what a valid `POOL_HEADER` and allocation should look like. Figure 7 shows the `POOL_HEADER` structure from a Windows XP system.

```
+0x000 PreviousSize     : Pos 0, 9 Bits
+0x000 PoolIndex        : Pos 9, 7 Bits
+0x002 BlockSize        : Pos 0, 9 Bits
+0x002 PoolType         : Pos 9, 7 Bits
+0x000 Ulong1           : Uint4B
+0x004 ProcessBilled    : Ptr32 _EPROCESS
+0x004 PoolTag          : Uint4B
+0x004 AllocatorBackTraceIndex : Uint2B
+0x006 PoolTagHash      : Uint2B
```

Figure 7: `POOL_HEADER` structure for a Windows XP system

Small Pool Allocation Header

This eight-byte header precedes every pool allocation that is less than or equal to `PAGE_SIZE` – sizeof(`POOL_HEADER`) in size and is used by the memory manager to track these allocations [25]. This structure provides quite a bit of useful information about each pool allocation that can be used for locating certain types of data. The `PreviousSize` and `BlockSize` members hold the previous allocation's size and current allocation's size in blocks (including the header itself), where a block is 32 bytes on Windows 2000 and 8 bytes on later versions [28]. These sizes effectively make the pool allocations within a page doubly linked to one another. The `PoolType` member is also useful as this specifies whether the allocation is paged or non-paged. Odd `PoolType`s are non-paged, whereas even ones are allocated from the paged pool. A value of zero indicates that the block is free [29].

Perhaps the most useful part of the `POOL_HEADER` is the four-byte `PoolTag`. The tag is a four character (ASCII values 0-127) identifier that is meant to tie a pool allocation to a particular allocation code path in a driver [26]. Because kernel memory is a shared resource, this mechanism was devised to establish accountability for misbehaving drivers. For example, a driver might be leaking memory by never freeing it, causing system

instability and performance issues. By monitoring the pool usage either on the live system or in a crash dump, one can easily identify which tag is associated with the problem. The tag can then be mapped back to a particular driver or even a particular allocation routine within the offending driver so the problem can be quickly identified and fixed.

Pool Tagging For Key Recovery

While pool tags are clearly useful for debugging, they are also useful for recovering specific pieces of data allocated by drivers. The tags act as discriminators for pool allocations adding yet another level of granularity to volatile memory. Using this information, we postulated that if one can figure out the unique pool tag used for the key allocation code path in an encryption driver, one can theoretically extract that memory, and hence the key, from the pool.

In the authors' experience, many driver writers do not follow Microsoft's recommendation to use unique pool tags for each allocation code path within their drivers. Some drivers use a single pool tag for all allocations within the driver, while others even duplicate the pool tags used by other drivers. Though discouraged and considered to be poor practice, this is fairly common – particularly among older third-party drivers. Obsolete, tagless, versions of the pool allocation functions exist in the driver development API for compatibility reasons so many older drivers that still use the obsolete functions end up with the same default tag of ' kdD' for all of their allocations. Since there is no official registry for pool tags and no mechanism to protect against tag collisions, driver writers are free to use any tag they want, including tags used by other drivers. Microsoft provides a listing of tags used by Windows drivers and components in the Windows Debugging Tools package, which makes it easier to avoid tags used by Windows. It is, however, much more difficult avoid tag collisions with third-party drivers. The duplication of tags not only makes debugging difficult by obscuring accountability in the shared kernel memory space, but it also erodes a level of granularity in volatile memory that can be exploited to find useful data in memory.

To muddy matters further, Microsoft's documentation on pool memory states that, "the system tags memory allocations only if pool tagging is enabled, even if you specify a tag with `ExAllocatePoolWithTagXxx`" [30]. Pool tagging is a setting that instructs the operating system to collect various statistics regarding pool usage grouped by pool tag. The collected data such as the number of allocations, frees, and total bytes used can all be displayed and updated in real time using a utility such as Poolmon that comes with the Windows Device Driver Kit. When pool tagging is not enabled, the documentation states that the pool tags specified by the developer are not used – eliminating the ability to attribute an allocation to a specific driver or code path. Of course tracking pool usage by pool tag should also no longer be possible when tags are not used.

This situation would clearly pose problems for identifying interesting kernel pool allocations by their tags, particularly because pool tagging is only enabled by default on Windows Server 2003 and later. On earlier systems , pool tagging must be enabled manually by setting the `HKLM\System\CurrentControlSet\Control\Session`

Manager\GlobalFlag registry key to the value 0x00000400 [31]. System-wide pool tagging can also be enabled using the Gflags utility provided with Debugging Tools for Windows. Enabling pool tagging requires restarting the target system, which of course would destroy volatile data, making this solution unacceptable from a forensics standpoint.

The significant number of machines still running Windows XP and earlier makes using pool tags to extract key material seem like a much less attractive option that is not widely applicable in real-world scenarios. It is the authors' experience, however, that pool tags are not completely ignored when pool tagging is disabled. Pool memory allocations less than or equal to a page in size are still tagged regardless of the pool tagging setting on the system. It is the "large pool" that includes allocations greater than PAGE_SIZE – sizeof (POOL_HEADER) in size whose tags are omitted [25]. These large pool allocations are handled quite differently from the regular ones. They are not preceded by the standard eight-byte POOL_HEADER containing the four-byte pool tag. These allocations are instead tracked in a separate table. Each entry in the table contains the virtual address of the start of the data for that allocation, the pool tag, and size of the allocation. The table does not appear in memory when pool tagging is disabled, so pool tags for large pool allocations cannot be used to attribute an allocation to a specific driver or code path.

Fortunately, since most symmetric keys, even in their fully expanded forms, are much smaller than a typical page size, one can realistically expect most keys to be stored in regular pool allocations as opposed to large ones. This, of course, means that no matter what the pool tagging settings on a particular system happen to be, pool tags can usually be used to recover key material from pool memory. In the authors' experience, a cryptosystem-specific signature, consisting of the driver specific pool tag and pool allocation size, are all that is necessary to extract pool allocations containing key material from a memory dump with an acceptably small number of false positives. Other previously discussed checks such as key expansion or known plaintext can also be used to reliably eliminate false positives making this technique both widely applicable and quite practical.

Operating System Structures as Key Pointers

Related Work on Linux

Recent work on volatile memory analysis by Nick Petroni and Aaron Walters yielded a technique for extracting key material for Truecrypt's container-based volume encryption package on Linux [10]. They observed that on Linux, Truecrypt volumes are implemented as Device Mapper targets. Using knowledge of how the Device Mapper kernel module interfaces with and distinguishes between devices, as well as the module's exported symbols, they were able to locate Truecrypt's Device Mapper target in memory. This target stores local context information including a structure containing Truecrypt's cryptographic information. While this technique relies heavily on the Linux-specific implementation of Truecrypt and on the open source nature of both the operating system and the encryption package, it does demonstrate that operating system structures have the potential to lead directly to useful data such as key material. While the Linux and

Windows versions of Truecrypt are implemented quite differently, the same general concept can be adapted to work on Windows.

Adapting to Windows

On Windows, Truecrypt is implemented as layered driver. Every Truecrypt device has a block of memory called the DEVICE_EXTENSION associated with it. The DEVICE_EXTENSION is similar to the local context information in the device mapper target on Linux. The extension is used to store data that the driver must have resident in system space in order to carry out I/O operations [32]. For an encryption tool, this data could certainly include key material. A pointer to the DEVICE_EXTENSION structure can be found in the executive DEVICE_OBJECT that is created for the Truecrypt device. The device extension structure is driver defined, so for an open source application such as Truecrypt, it is trivial to parse the data in the DEVICE_EXTENSION to find the offsets for all of the relevant cryptographic information.

Since the DEVICE_EXTENSION is allocated from the non-paged pool, this technique is only really advantageous in situations where the large pool is being used to store key material and pool tagging is disabled on the system. Instead of finding the large pool allocation by tag, one can find it through the DEVICE_EXTENSION pointer in the executive DEVICE_OBJECT. The current version of Truecrypt (4.3), for example, uses the large pool to store its cryptographic information, so this technique is useful for recovering Truecrypt key material on systems with pool tagging disabled.

In order to actually make use of this technique on Windows, one must be able to locate the executive DEVICE_OBJECT that corresponds to the target encryption driver in volatile memory. The DEVICE_OBJECT is pointed to by its parent executive DRIVER_OBJECT. Previous work has been done by the rootkit detection community to detect hidden DRIVER_OBJECTs in memory. The aforementioned GrepExec utility as well as Joanna Rutkowska's modGREPER take a signature-based approach to locating valid DRIVER_OBJECTs in memory. Because one would not expect a disk encryption package to intentionally unlink its driver from the list with the intention of hiding it, one could probably use a more traditional list-walking approach, starting from the Kernel Processor Control Block, as opposed to the brute force signature-based search. The advantage of list walking is that it should be faster than brute force and may not be as strongly affected by operating system updates that may alter the object detection signature. Regardless of how the DRIVER_OBJECTs are enumerated, one can check the DriverName member of each DRIVER_OBJECT against the encryption driver's name to find the correct one. From there, the DEVICE_OBJECT and corresponding DEVICE_EXTENSION containing the cryptographic information can be found. It should also be noted on some cryptosystems, key material can also be found within the memory allocated to the DEVICE_OBJECT itself.

Making use of the DEVICE_EXTENSION provides an alternative method for finding non-paged pool allocations containing key material. Key material will not always be stored in the area of pool memory allocated for the DEVICE_EXTENSION, but since it is

common practice to use the `DEVICE_EXTENSION` for storing data necessary for performing I/O operations, it is certainly a strong possibility. This technique does require implementation specific knowledge of the `DEVICE_EXTENSION` structure for a specific driver, which can be gained either by examining the driver source code or by reverse engineering it. While not a widely applicable technique on its own, it does complement the pool tag search technique, filling a void where large pool allocations are used and pool tagging is disabled, as is the case for Truecrypt on default installations of Windows XP and below. When combined with the pool tag search technique, cryptographic key material can be reliably extracted from volatile memory and used to decrypt encrypted disks or containers.

Putting it all Together

Volatile Memory Incentives

To date, volatile memory analysis has not been widely adopted by the community of forensic practitioners. This is largely because it is viewed as a time-consuming extra step that often yields comparatively little additional usable evidence. While tools have been developed to extract useful data from volatile memory, many require a significant amount of training and technical proficiency to be used effectively. It is our belief that the body of existing volatile memory research clearly demonstrates that memory analysis can support the traditional disk-centric investigative process by supplying corroboratory evidence of what was happening on a system at the time of its seizure. In some cases, volatile memory is the only source of evidence, and by ignoring it, investigators are effectively throwing away important parts of the "crime scene." Similarly, when encryption is in use, volatile memory may be the investigator's only hope of recovering the plaintext data.

Extolling the virtues of volatile memory analysis, however, doesn't constitute a solution. The usability of existing tools and techniques for performing volatile memory analysis needs to be improved to achieve the requisite return on investment for incorporating such analysis into existing methodologies. Usability can be improved by both increasing the body of knowledge available to the community and by creating tools that are easier to use. It is likely that the latter approach is what is needed in the short term to get the attention of the community as it provides a low-cost incentive with a high initial payoff.

A Usable Tool: Disk Decryptor

In an effort to demonstrate the practicality of the techniques outlined in this research, as well as the value of incorporating usable volatile memory analysis into existing methodologies, we have developed a prototype tool preliminarily named "Disk Decryptor." Disk Decryptor provides a practical solution for dealing with full-disk encryption. Given a volatile memory dump and encrypted disk image, it is capable of automatically extracting the key material and decrypting the disk image so that it can be analyzed using standard forensics tools and techniques. Disk Decryptor uses cryptosystem specific signatures and an extensible decryptor module framework that tells it how to handle implementation specific details of each cryptosystem. The analyst simply chooses the decryptor module corresponding to the full-disk cryptosystem

installed on the target machine from a dropdown menu, selects the location of the volatile memory and encrypted disk images, and Disk Decryptor takes care of the rest (see figure 8). The methodology used by Disk Decryptor is as follows:

- Pool allocations containing potential keys are located in the volatile memory dump using the extraction techniques developed as a result of this research. All of the data necessary for key extraction, such as the pool tag and allocation size, are included in the cryptosystem configuration information that is associated with the selected decryptor module.
- The decryptor module extracts the key material from the memory image and validates it. Validation is implemented as a check for known plaintext in a particular sector as specified in the cryptosystem configuration information that is associated with the selected decryptor module but could be extended to include techniques such as the key schedule check.
- Using the extracted key material, the decryptor module specified by the user begins decrypting the disk sector by sector, writing the plaintext image to the user's chosen output location.

Figure 8: Disk Decryptor Interface

Each decryptor module contains all of the cryptosystem-specific information such as algorithm, key offset within the pool allocation, IV calculation, block chaining implementation, et cetera.

Related Work: Passphrase Recovery

Background

While the focus of this research is on the extraction of symmetric key material from volatile memory, it is also worth mentioning that it is sometimes possible to extract passphrases from memory. The passphrase, or more precisely a key derived from the passphrase, is typically used to encrypt the master symmetric key that is used for encrypting the actual data. In theory, knowing the passphrase is at least as good as knowing the master key. In practice, finding the master key with just the passphrase requires knowledge of the password-based key derivation function (PBDKF) as well as the offset to the encrypted master key on disk, so using the master key directly is oftentimes the most straightforward way to decrypt a disk image. Having the passphrase, however, can be more useful than just having the key if the same passphrase is used repeatedly.

While well designed encryption programs typically seek to wipe passphrases from memory after they are used to unlock the data encryption key, there are some situations in which they may stay resident in memory. Container-based cryptosystems often allow users to cache their passphrases in memory so they do not have to reenter them whenever they mount or unmount their containers in a single session. To a much lesser extent, either memory leaks or compiler optimizations could also result in a passphrase being left in memory.

Case Study: Truecrypt

The current version of Truecrypt (4.3), for example, has passphrase-caching functionality built in, though it is disabled by default. Users can enable it by selecting a checkbox at the passphrase prompt. Cached passphrases are considered to be sensitive data, so many of the techniques discussed for reducing the search scope in the virtual address space apply for passphrases as they do for symmetric keys. Passphrases can also be extracted using a signature-based approach similar to the ones used to locate Windows executive objects and pool allocations. Since passphrases are often stored as part of a larger structure, a signature can be developed to find valid instances of the structure in memory. Truecrypt stores its passphrases in a structure consisting of a four-byte length value followed by a buffer large enough to hold the maximum allowable passphrase. A set of four of these passphrase structures are contained within a larger structure. Using this knowledge, a series of rules can be developed forming a signature that can be used to extract the cached passphrases with a minimal number of false positives. Some of these rules might include (on a little endian architecture):

- The four-byte length will have a value between 1 and the maximum allowable passphrase size (64) in the first of the four bytes
- The next three bytes of the length field will be zero
- The bytes following the three zero values that represent the passphrase must be ASCII printable [0x20 – 0x7E] and must match the length value
- The next byte after the ‘length’ printable bytes should be a null

    - The rest of the bytes after the passphrase up to the maximum passphrase size should also be null depending on how the memory was allocated
- These individual {length, passphrase} tuples occur in a group of four as part of the larger cached passphrase structure

In our experience, a similar set of rules is sufficient for finding Truecrypt passphrases in memory with an acceptable number of human identifiable false positives (on the order of ones or tens of them) depending on both the size of the memory dump and what was running in memory at the time it was created. Armed with the additional knowledge that Truecrypt passphrases are allocated and stored within the local memory of the driver itself, we have often been able to uniquely identify the passphrase in memory. This can be done by locating the executive `DRIVER_OBJECT` in memory and using the aforementioned set of rules to search the virtual addresses space between the `DriverStart` address and `DriverStart` + `DriverSize` address. This type of search on just the small in-memory image of the driver is not only more accurate, but also significantly faster than a search including the entire virtual or physical address space.

Full-Disk Encryption Passphrase Recovery

Recent work by Adam Boileau found that PGP Whole Disk Encryption as well as BIOS passphrases can be found in the real mode keyboard interrupt buffer in the BIOS [33]. This buffer holds the last 16 bytes typed before switching into protected mode. Typically full-disk encryption systems implement some sort of Pre-Boot Authentication (PBA) that occurs prior to loading the operating system in what is known as “real” mode. Once the user is properly authenticated in this PBA environment, the system begins decrypting and loading the operating system moving from real mode to protected mode. We have neither verified these claims for PGP Whole Disk Encryption nor tested this technique on any other full-disk encryption solutions. This does, however, serve as another possible means of using memory contents to attack encrypted disks for the purposes of forensic analysis.

Future Research

While this research has produced some practical techniques for responding to the growing problem of data encryption in digital forensics, much work still needs to be done. In the near term, we would like to extend Disk Decryptor to make it a tool that can be deployed in the field and integrated into standard forensic response methodologies. More extensive signature sets need to be developed that enable it to handle all of the full-disk and container-based encryption packages that might be encountered in the field. Additional features that extract cached passphrases or enable investigators to extract keys directly from a running system without first creating a memory dump would also make the tool more useful and thus be worthy of further research.

Disk encryption does not just include software-based solutions. In the future, disk encryption is likely to shift from software to a hybrid or fully hardware based approach. As such, work needs to be done to investigate the implications of software based cryptosystems that make use of the Trusted Platform Module (TPM) as well as hardware based full-disk encryption solutions before these technologies become widely deployed.

Exploring techniques for dealing with encryption when machines are locked or when access is otherwise blocked to physical memory is also an important area of research. As of Service Pack 1 in Windows Server 2003, user mode access is blocked to the `\Device\PhysicalMemory` object, making volatile memory acquisition a more difficult problem [34]. Without access to a running system or its physical memory there is no practical and reliable way to decrypt the data on disk short of a brute force or cryptanalytical attack. Techniques for extracting physical memory, even from a locked system, using direct memory access (DMA) over FireWire have been developed by Maximillian Dornseif, et al. [35] and later improved upon to run on Windows by Adam Boileau [33]. George Garner also recently released KnTDD, a software solution for overcoming the Windows usermode access restriction on physical memory [11]. For high-value systems, a preinstalled dedicated PCI card, such as Tribble [36] or CoPilot [37], can be pre-installed and used for acquiring volatile memory. A better understanding of the limitations of these techniques [38,39] as well as refining and expanding upon these techniques to make them more widely applicable and easy to use will be important for both key extraction and volatile memory analysis as a whole.

In a similar vein, developing techniques for recovering key material from a system that has been powered off is also important. While most volume based cryptosystems incorporate the paging of sensitive data to disk into their threat models, “hibernation” and “suspend” features may be overlooked. For full-disk solutions, research into the physical properties of volatile memory may hold some promise. In his 1998 paper, Peter Gutmann states, “Contrary to conventional wisdom, ‘volatile’ semiconductor memory does not entirely lose its contents when power is removed” [40]. More recent work by Chow, Pfaff, Garfinkel, and Rosenblum showed that “soft” reboots which do not turn off a machine’s power does not clear most of physical memory. Perhaps more encouraging, on some sets of hardware, old data was retained in memory after thirty seconds without power [41]. While far from being a practical technique, software-based key recovery from a powered off machine may, at least in theory, be possible.

Conclusion

This research underscores the challenges faced by forensic practitioners associated with the increasing prevalence and usability of strong encryption solutions. Just as strong encryption can be effectively used to protect against unintentional data exposure, it can also be used by criminals to hide evidence of their malfeasance. Traditional disk-centric forensic methodologies must be updated to stay ahead of the increasing threat posed by push-button strong encryption solutions.

In this research we have outlined some practical techniques for dealing with software based full-disk and container OTFE. Based on the observation that encryption keys must be loaded into memory for the processor to perform cryptographic operations, we have posited that these keys can be extracted from memory and used to decrypt encrypted media. Utilizing various properties of, and structures contained within, volatile memory, we have demonstrated practical techniques for performing symmetric key extraction from volatile memory. We have also developed a prototype tool capable of automatically

decrypting encrypted disk images using a volatile memory dump from the same system. We hope that by providing a practical and forensically sound approach to the encryption problem, that the requisite incentives for incorporating volatile memory acquisition into standard forensics practices will be achieved and that forensic practitioners will be better prepared to face some of the looming challenges posed by strong disk encryption.

References

1. Clay, Johnson. United States. Office of Management and Budget. Executive Office of the President. Memorandum for the Heads of Departments and Agencies: Protection of Sensitive Agency Information. 23 June 2006. <http://www.whitehouse.gov/omb/memoranda/fy2006/m06-16.pdf>.

2. Denning, Dorothy E., and William E. Baugh. "Encryption and Evolving Technologies as Tools of Organized Crime and Terrorism." Working Group on Organized Crime (WGOC) (1997). <http://www.cs.georgetown.edu/~denning/crypto/oc-rpt.txt>.

3. "Protecting Data by Using EFS to Encrypt Hard Drives." Microsoft TechNet. Microsoft. <http://www.microsoft.com/technet/security/smallbusiness/topics/cryptographyetc/protect_data_efs.mspx>.

4. Nelson, Sharon D., and John W. Simek. "Electronic Evidence: the Ten Commandments." 2003. Sensei Enterprises. <http://www.senseient.com/article18.asp>.

5. "Configuring Active Directory to Back up Windows BitLocker Drive Encryption and Trusted Platform Module Recovery Information." Microsoft TechNet. Microsoft. Windows Vista Tech Center. <http://technet2.microsoft.com/WindowsVista/en/library/3dbad515-5a32-4330-ad6f-d1fb6dfcdd411033.mspx?mfr=true>.

6. Henry, Allison. Implementing Desktop Encryption Using Pointsec for PC. 2006. Information Services and Technology, University of California at Berkeley. <https://kb.berkeley.edu/jivekb/servlet/KbServlet/download/1026-102-1/pointsecdeploy.pdf>.

7. Espiner, Tom. "British Legislation to Enforce Encryption Key Disclosure." ZDNet. 18 May 2006. <http://news.zdnet.com/2100-1009_22-6073654.html?tag=nl>.

8. Sergienko, Greg S. "Self Incrimination and Cryptographic Keys." The Richmond Journal of Law and Technology (1996). <http://law.richmond.edu/jolt/v2i1/sergienko.html>.

9. Bilby, Darren. Low Down and Dirty: Anti-Forensic Rootkits. Ruxcon 2006. <http://www.security-assessment.com/files/presentations/darrenbilby_ruxcon06_v0_5.pdf>.

10. Walters, Aaron, and Nick Petroni Jr. Volatools: Integrating Volatile Memory Forensics Into the Digital Investigation Process. Blackhat Federal 2007, Komoku,

Inc. <https://www.blackhat.com/presentations/bh-dc-07/Walters/Paper/bh-dc-07-Walters-WP.pdf>.

11. Garner, George M. “KnTTools™ with KnTList™.” GMG Systems, Inc. <http://users.erols.com/gmgarner/KnTTools/>.

12. Burdach, Mariusz. “Windows Memory Forensic Toolkit (WMFT).” Digital Investigation. 2006. <http://forensic.seccure.net/>.

13. “Volatools Basic.” 28 Feb. 2007. Komoku, Inc. <http://www.komoku.com/forensics/basic.html>.

14. Betz, Chris. “Memparser.” Sourceforge. 4 July 2006. <http://sourceforge.net/projects/memparser>.

15. Shamir, Adi, and Nicko Van Someren. “Playing Hide and Seek with Stored Keys.” (1998). <http://www.web.ms11.net/hawaii//keyhide2.pdf>.

16. Klein, Tobias. All Your Private Keys are Belong to Us. 2006. <http://www.trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf>.

17. Workman, Sean. “Selecting Appropriate Cryptographic Keys.” (2005). <http://www.primefactors.com/resources/index.cfm?fuseaction=article&rowid=41>.

18. Chiriliuc, Adal. “BestCrypt IV Generation Flaw.” 7 Nov. 2003. <http://adal.chiriliuc.com/bc_iv_flaw.php>.

19. Saarinen, Markku-Juhani O. Linux for the Information Smuggler. Helsinki University of Technology. <http://mareichelt.de/pub/notmine/diskenc.pdf>.

20. Clemens, Fruhwirth. “The IV Problem of Dm-Crypt : is It Secure Nevertheless?” Gmane. 12 Sept. 2001. <http://article.gmane.org/gmane.linux.kernel.device-mapper.dm-crypt/472>.

21. “Kerckhoffs' Principle.” Wikipedia. <http://en.wikipedia.org/wiki/Kerckhoffs'_principle>.

22. “PGP Desktop 9.0 for Windows User Guide.” PGP Corporation (2006). <https://supportimg.pgp.com/guides/PGP_Desktop_Win_9.0.6_Eng.pdf>.

23. Solomon, David, and Mark Russinovich. Windows Internals. 4th ed. Redmond: Microsoft P, 2005.

24. “Truecrypt User's Guide V3.1a.” (2005). <http://security.ngoinabox.org/Programs/Security/Encryption%20Tools/TrueCrypt/TrueCrypt%20User%20Guide.pdf>.

25. Memory Management: What Every Driver Writer Needs to Know. Microsoft. 2005. <http://www.microsoft.com/whdc/driver/kernel/mem-mgmt.mspx>.

26. "ExAllocatePoolWithTag." MSDN. Microsoft. Windows Driver Kit: Kernel-Mode Driver Architecture. <http://msdn2.microsoft.com/en-us/library/ms796989.aspx>.

27. Carr, Chris. "GREPEXEC: Grepping Executive Objects From Pool Memory." (2006). <http://www.uninformed.org/?v=4&a=2&t=pdf>.

28. Schuster, Andreas. "Searching for Processes and Threads in Microsoft Windows Memory Dumps." Digital Investigation (2006). <http://www.dfrws.org/2006/proceedings/2-Schuster.pdf>.

29. Schuster, Andreas. Searching for Processes and Threads in Microsoft Windows Memory Dumps. DFRWS, Aug. 2006, Deutsche Telekom AG. <http://www.dfrws.org/2006/proceedings/2-Schuster-pres.pdf>.

30. "Who's Using the Pool?" Microsoft. <http://www.microsoft.com/whdc/Driver/tips/PoolMem.mspx>.

31. "How to Use Memory Pool Monitor (Poolmon.Exe) to Troubleshoot Kernel Mode Memory Leaks." MSDN. Microsoft, 2007. Knowledge Base. <http://support.microsoft.com/kb/177415>.

32. "Device Extensions." MSDN. Microsoft. Windows Driver Kit: Kernel-Mode Driver Architecture. <http://msdn2.microsoft.com/en-us/library/ms794734.aspx>.

33. Boileau, Adam. Hit by a Bus: Physical Access Attacks with Firewire. Ruxcon 2006. <http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf>.

34. "Changes to Functionality in Microsoft Windows Server 2003 Service Pack 1: \Device\PhysicalMemory Object." Microsoft TechNet. Microsoft. Microsoft Windows Server TechCenter. <http://technet2.microsoft.com/windowsserver/en/library/e0f862a3-cf16-4a48-bea5-f2004d12ce351033.mspx?mfr=true>.

35. Becher, Michael, Maximillian Dornseif, and Christian N. Klein. FireWire: All Your Memory are Belong to Us. CanSecWest, 2005, Laboratory for Dependable Distributed Systems. <http://md.hudora.de/presentations/firewire/2005-firewire-cansecwest.pdf>.

36. Carrier, Brian D., and Joe Grand. "A Hardware-Based Memory Acquisition Procedure for Digital Investigations." <http://www.digital-evidence.org/papers/tribble-preprint.pdf>.

37. Petroni, Nick L., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – A Coprocessor-Based Kernel Runtime Integrity Monitor. The Thirteenth USENIX Security Symposium, Aug. 2004, University of Maryland, College Park. <http://www.cs.umd.edu/~waa/pubs/USENIX-copilot.pdf>.

38. Adam, Boileau, and George M. Garner. "Projects: Firewire, DMA & Windows." 20 Oct. 2006. <http://www.storm.net.nz/projects/16>.

39. Rutkowska, Joanna. Beyond the CPU: Defeating Hardware Based RAM Acquisition Tools. Black Hat DC, 2007, COSEINC. <https://www.blackhat.com/presentations/bh-dc-07/Rutkowska/Presentation/bh-dc-07-Rutkowska-up.pdf>.

40. Gutmann, Peter. Secure Deletion of Data From Magnetic and Solid-State Memory. The Sixth USENIX Security Symposium, July 1996, University of Auckland. <http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html>.

41. Chow, Jim, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. The Fourteenth USENIX Security Symposium, July-Aug. 2005, Stanford University. <http://www.stanford.edu/~blp/papers/shredding.pdf>.