Protecting Cryptographic Keys from Memory Disclosure Attacks*

Keith Harrison and Shouhuai Xu
Department of Computer Science, University of Texas at San Antonio
{kharriso,shxu}@cs.utsa.edu

March 23, 2007

Abstract

Cryptography has become an indispensable mechanism for securing systems, communications, and applications. While offering strong protection, cryptography makes the assumption that cryptographic keys are kept absolutely secret. In general this assumption is very difficult to guarantee in real life because computers, on which cryptographic keys are stored and utilized, may be compromised relatively easily. In this paper we investigate a class of attacks, which exploit *memory disclosure vulnerabilities* to expose cryptographic keys. We demonstrate that the threat is real by formulating an attack that exposed the private key of an OpenSSH server within **1** minute, and exposed the private key of an Apache HTTP server within **5** minutes. We propose a set of software-based techniques to address such attacks. Experimental results show that our techniques are efficient (i.e., imposing no performance penalty) and effective — unless a large portion of *allocated* memory is disclosed.

Keywords: cryptographic key, OS security, memory disclosure.

1 Introduction

The utility of cryptography is based on the assumption that cryptographic keys are kept absolutely secret. This assumption is very difficult to guarantee in real-life systems due to various bugs in operating systems and applications. In this paper we focus on a class of attacks that exploit memory disclosure vulnerabilities, called *memory disclosure attacks*. Such an attack can expose the content of (a portion of) computer memory, and thus cryptographic keys in the disclosed memory. As an example (see Section 2 for details), we mention that the vulnerability reported in [17] can be exploited to expose the RSA private key of an OpenSSH server within **1** minute, and the RSA private key of an Apache HTTP server within **5** minutes. This motivates us to explore the attacks as well as solutions to countering them.

1.1 Our Contributions

In this paper we make three contributions. First, we thoroughly assess (Section 2) the damage of memory disclosure attacks against the private keys of OpenSSH servers and Apache HTTP servers. The attacks exploit two reported vulnerabilities that do *not* requiring the `root` privilige on a victim machine. Our experiments show that such attacks effectively expose the private keys of the servers.

Second, we propose a method for helping understand the attacks (e.g., why are they so powerful?). The core of the method is a software tool (Section 3.1) we developed to help analyze the content of computer memory (Section 3.2). Through our software tool, we found that disclosure a portion of either *allocated memory* or *unallocated memory* would effectively expose cryptographic keys. This is interesting because existing literature often emphasized the importance of clearing unallocated memory (cf. Viega et al. [22, 23] and Chow et al. [7]), but not necessarily taking care of allocated memory.

*This is the full version of [16], which appeared in the proceedings of DSN'07.

Third, our analyses on the attacks suggest the following countermeasures: We should ensure (i) a cryptographic key only appears in allocated memory a minimal number of times (e.g., one) as long as this does not significantly degrade the system performance, and (ii) unallocated memory does not have a copy of cryptographic keys. We thus proceed to propose a set of software-based solutions. Our solutions can eliminate attacks that disclose unallocated memory, and can mitigate the damage due to attacks that disclose portions of allocated memory. In particular, our method for minimizing the number of copies of a private key in allocated memory, to our knowledge, is novel in the sense that it takes full advantage of the operating system "copy on write" memory management policy [21] — a technique that was not originally motivated for security purpose. While our solutions are general enough to help protect cryptographic keys of many applications, we conduct case studies by applying our solutions to protect the private keys of OpenSSH servers and of Apache HTTP servers. Experimental results show that our solutions do not incur any performance penalty, and can eliminate attacks that disclose unallocated memory and mitigate the damage due to attacks that disclose a small portion of allocated memory. It is stressed, however, that if the portion of disclosed memory is large (e.g., about 50% as shown in our case study), the key is still exposed in spite of the fact that our solutions can minimize the number of key copies in memory. Therefore, our investigation may serve as an evidence that in order to completely avoid key exposures due to memory disclosures, special hardware is necessary.

1.2 Related Work

The problem of ensuring the secrecy of cryptographic keys (and their functionalities thereof) has been extensively investigated by the cryptography community. There have been many novel cryptographic methods that can mitigate the damage caused by the compromise of cryptographic keys. Notable results include the notions of threshold cryptosystems [9], proactive cryptosystems [18], forward-secure cryptosystems [1, 2, 3, 14], key-insulated cryptosystems [10], and intrusion-resilient cryptosystems [15]. The present paper falls into an approach that is orthogonal to the cryptographic approach. Clearly, our mechanisms can be deployed to secure traditional centralized cryptosystems, as evidently shown in the present paper that the keys of OpenSSH servers and of Apache HTTP servers can be better protected by utilizing our mechanisms. Equally, our mechanisms can be utilized to provide another layer of protection for the aforementioned advanced cryptosystems.

It has been deemed as a good practice in developing secure software to clear the sensitive data such as cryptographic keys, promptly after use (cf. Viega et al. [22, 23]). Unfortunately, as confirmed by our experiments as well as an earlier one due to Chow et al. [6], this practice has not been widely or effectively enforced. Chow et al. [6] investigated the propagation of sensitive data within an operating system by examining all places the sensitive data can reside. Their investigation was based on whole-system simulation via a hardware simulator, namely the open-source IA-32 simulator Bochs v2.0.2 [4]. More recently, Chow et al. [7] presented a strategy for reducing the lifetime of sensitive data in memory called "secure deallocation," whereby data is erased either at deallocation or within a short, predictable period afterwards in general system allocators. As a result, their solution can successfully eliminate attacks that disclose unallocated memory. However, their solution has no effect in countering attacks that may disclose portions of allocated memory. Whereas, our solutions can not only eliminate attacks that disclose unallocated memory, but also mitigate the damage due to attacks that disclose portions of allocated memory. That is, our solutions provide strictly better protections.

There are some loosely related works. Broadwell et al. [5] explored the core dump problem, and inferred which data in a system is sensitive based on programmer annotations. This was motivated to facilitate the shipment of crash dumps to application developers without revealing users' sensitive data. Provos [19] investigated a solution to use swap encryption for processes in possession of confidential data. Gutmann [13] showed the difficulty of removing all remnants of sensitive data once written to a disk. A cryptographic treatment on securely erasing sensitive data via a small erasable memory was presented by Jakobsson et al. [8].

Outline. The rest of the paper is organized as follows. In Section 2 we evaluate the severity of the memory disclosure problem. In Section 3 we show how to understand the attacks in detail based on our software tool. In Section 4 we present a general solution to countering memory disclosure attacks, whose concrete instantiations to protect private keys of OpenSSH servers are explored in Section 5, and concrete instantiations to protect private keys of Apache servers are explored in Section 6. We conclude this paper in Section 7.

2 Threat Assessment: Initial Experiments

In this section we report our initial experiments that exploit two specific memory disclosure vulnerabilities to expose the RSA [20] private keys of an OpenSSH Server and of an Apache HTTP server. The first vulnerability was reported in [17], which states that Linux kernels prior to 2.6.12 and prior to 2.4.30 are vulnerable to the following attack: directories created in the `ext2` file systems could leak up to 4072 bytes of (unallocated) kernel memory for every directory created. The second vulnerability was reported in [12], which states that a portion of memory of Linux kernels prior to 2.6.11 may be disclosed due to the misuse of signed types within `drivers/char/n_tty.c`. The disclosed memory may have a random location and may be of a random amount. Both vulnerabilities can be exploited *without* requiring the `root` privilege.

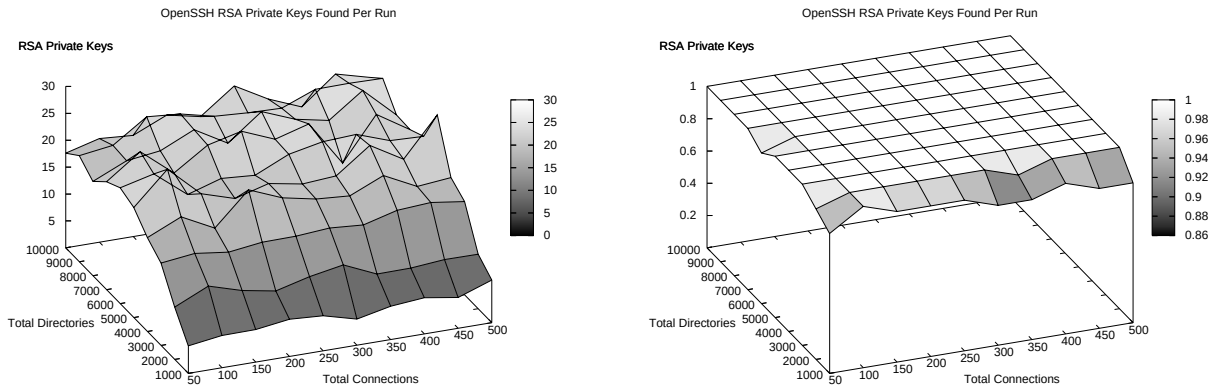
Recall that the RSA cryptosystem has a public key (e, N) and a private key (d, N), where $N = PQ$ for some large prime P and Q (e.g., $|P| = |Q| = 512$). In practice, a variation of the Chinese Remainder Theorem (CRT) is utilized to speed up the signing/decryption procedure, meaning that a RSA private key actually consists of 6 distinct parts: d, P, Q, $d \mod (P-1)$, $d \mod (Q-1)$, and $Q^{-1} \mod P$. Notice that there is a special PEM-encoded private key file, which contains the whole private key. For simplicity, we only consider d, P, Q, and the PEM-encoded file in the sense that disclose of any of them immediately leads to the compromise of the private key. Therefore, we call any appearance of any of them "a copy of the private key."

Our experiments ran in the following setting: the server machine has a 3.2GHz Intel Pentium 4 CPU and 256MB memory; the operating system is Gentoo Linux with a 2.6.10 Linux kernel; the OpenSSH server is OpenSSH 4.3_p2; the Apache HTTP server is Apache 2.0.55 (compiled using the `prefork` MPM); the OpenSSL library version is 0.9.7i.

On the power of attacks exploiting the vulnerability reported in [17]. Our experimental attacks proceeded as follows. (i) We plugged a small 16MB USB storage device into the computer running OpenSSH (or Apache HTTP) server. (ii) We wrote a script to fulfill the following. In the case of OpenSSH server, it first created a large number of SSH connections to `localhost`; whereas in the case of Apache HTTP server, it first instructed a remote client machine to create a large number of HTTP connections to the server. Then, the script immediately closed all connections. Finally, the script created a large number of directories on the USB device, where each directory created revealed less than 4,072 bytes of memory onto the USB device. (iii) We removed the USB device, and then simply searched the USB device for copies of the private key. Experimental results are summarized as follows.

- The case of OpenSSH server: Figure 1(a) depicts the average (over 15 attacks) number of copies of private keys found from the disclosed memory on the USB device, with respect to the number of `localhost` SSH connections and the number of created directories. For example, by establishing 500 total connections and creating 1,000 directories (i.e., disclosing up to about 4 MBytes memory), we were able to recover about 8 copies of the private key. From a different perspective, Figure 1(b) depicts the average success rate of attacks (i.e., the rate of the number of successful attacks over the total number of 15 attacks), which clearly states that an attack almost always succeeds. In this case, an attack took *less than one minute*.

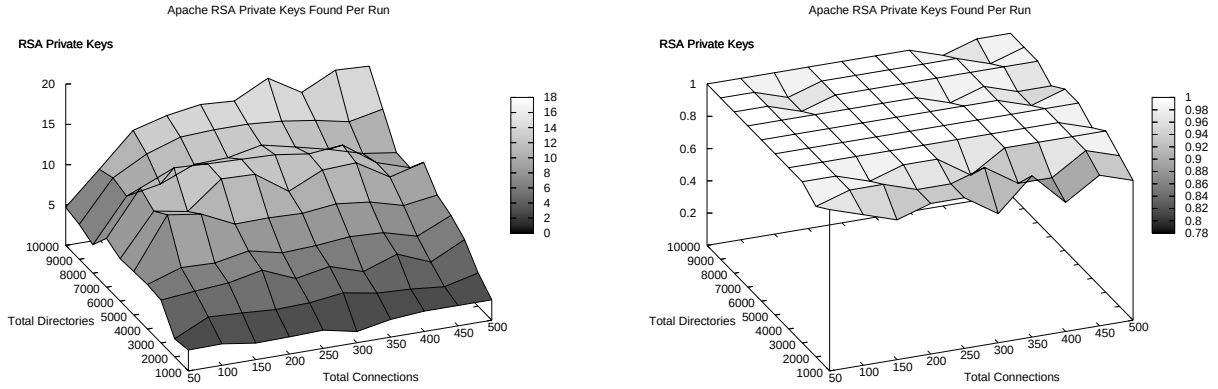
- The case of Apache HTTP server: Figure 2(a) shows the average (over 15 attacks) number of copies of private keys found on the USB device, with respect to the number of connections and the number of



(a) # of copies of the private key found (b) Success rate of attacks

Figure 1: OpenSSH case

created directories. For example, by establishing 500 connections and creating 1,000 directories (i.e., disclosing up to 4 MBytes memory), we were able to recover about 5 copies of the private key. From a different perspective, Figure 2(b) depicts the average success rate of attacks, which clearly states that an attack almost always succeeds. In this case, an attack took *less than five minutes*.

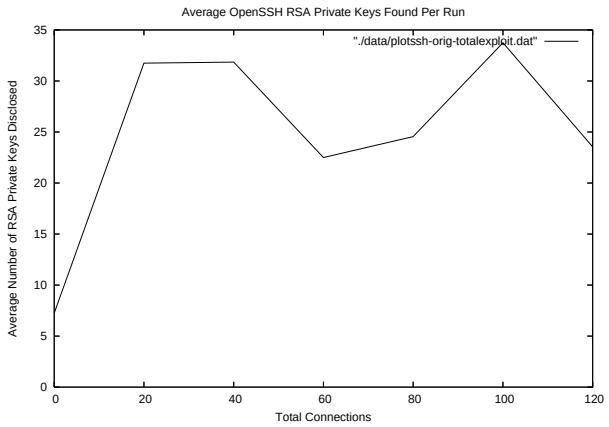


(a) # of copies of the private key found (b) Success rate of attacks

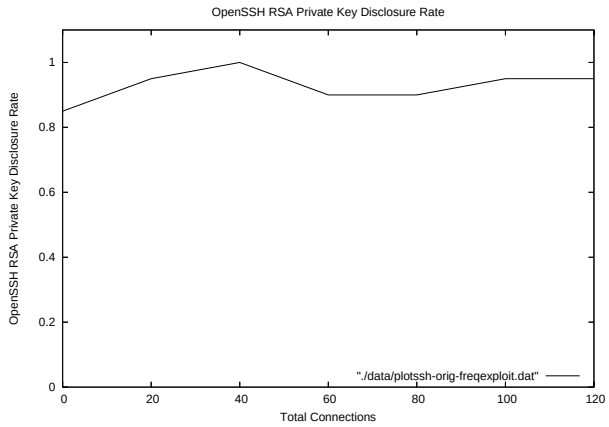
Figure 2: Apache case

On the power of attacks exploiting the vulnerability reported in [12]. Our experimental attack was orchestrated by a script that fulfills the following: (i) In the case of OpenSSH server, it created a large number of SSH connections to `localhost`. In the case of Apache HTTP server, it instructed a remote computer to establish a large number of HTTP connections to the server. (ii) The script executed a program (due to [12]) to dump a piece of memory to a file, which was then searched for the private key. The size and location of the disclosed memory varied, dependent on the terminal running the exploit. The exploit disclosed about 50% of the memory (i.e., 128 MBytes) on average. Experimental results are summarized as follows.

- The case of OpenSSH server: Figure 3(a) shows the average (over 20 attacks) number of copies of private keys found from the dumped memory with respect to the number of connections. From a different perspective, Figure 3(b) shows the success rate of attacks (i.e., the rate of the number of successful attacks over the total number of 20 attacks), which clearly states that an attack almost always succeeds. In this case, an attack took *less than 1 minute*.



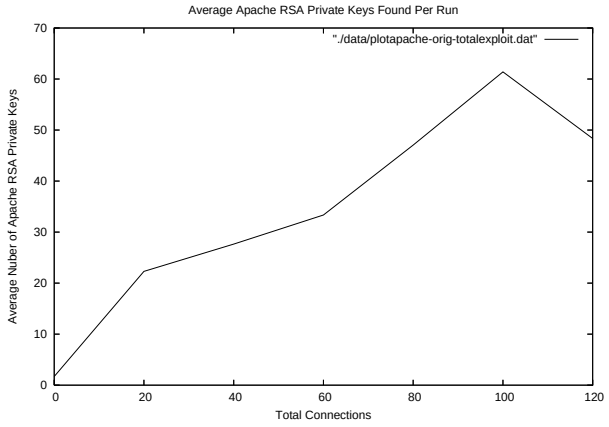
(a) # of copies the private key found



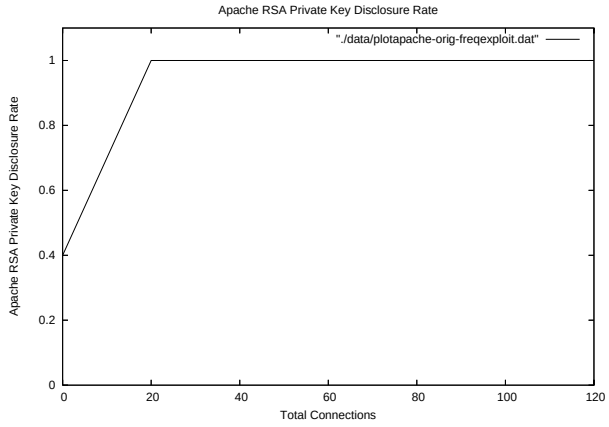
(b) success rate of attacks

Figure 3: OpenSSH case

- The case of Apache HTTP server: Figure 4(a) shows the average (over 20 attacks) number of private keys found in the dumped memory. From a different perspective, Figure 4(b) shows the success rate of attacks, which clearly states that an attack always succeeds when 30 or more connections are established. In this case, an attack takes *less than 1 minute*.



(a) # of copies the private key found



(b) success rate of attacks

Figure 4: Apache case

In summary, our initial experiments showed that cryptographic keys can be easily compromised by attacks that exploit memory disclosure vulnerabilities. Since the attacks are so powerful, we suspect that copies of the cryptographic keys were somehow flooding the memory to some extent. This motivates us to thoroughly examine, in Section 3, the behavior of cryptographic keys in memory.

3 Understanding the Attacks

Motivated by the results of our initial experiments, we aimed to understand why the attacks are so powerful. For this purpose, we developed a software tool to locate cryptographic keys in memory (Section 3.1). The tool was then used to analyze the behaviors of cryptographic keys in memory (Section 3.2).

3.1 Supporting Tool: Locating Cryptographic Keys in Memory

In order to understand the attacks, we needed a tool to capture the "snapshots" of memory, and to bookkeep information such as "which processes have access to which memory pages that contain copies of private keys". We developed a software tool for this purpose. The C code of our tool, called `scanmemory`, is about 260 lines. The detail of the code is deferred to the full version of this paper. As highlighted in the pseudocode below, its functionality is to search for copies of a private key, `privkey`, within memory in a linear fashion. Thus, its time complexity is $O(n)$, where n is the size of memory. In our experiments, it took about 5 seconds to scan the 256MB memory.

```
procedure scanmemory(string array privkey, integer numkeys) {
   foreach address from 0 to MEMSIZE
      foreach i from 0 to numkeys
      if *address equals privkey[i]
         Print 'Key (privkey[i]) found at address (*address) owned by processes:';
         printOwingProcesses(address)
         Print newline
}
procedure printOwningProcesses(void pointer address){
   pageFrameNumber := address << PAGE_SHIFT
   page := page_frame_number_to_page(pageFrameNumber)
   foreach Anonymous_VMA associated with the page
      foreach process in the system
         if the given process has the given vma in its memory mapping
            Print '(process pid) '
   if no vma's are associated with the page, but page_count(page) is > 0
      Print '0' #For the kernel
}
```

The `scanmemory` is implemented as a loadable kernel module (LKM). In addition to fulfilling the functionality of `scanmemory`, the LKM creates a `/proc` file system entry to facilitate communications between `scanmemory` and a user process. The `scanmemory` is invoked whenever the newly created `/proc` file system entry is read. For every match found, `scanmemory` further searches through all processes to determine which processes, if any, have this physical memory page in their logical address space. To speed up this search, `scanmemory` takes full advantage of the reverse mapping functionality introduced into the 2.6 Linux kernel series. The output of `scanmemory` to the `/proc` file system includes locations of the `privkey` in memory, and identities of the processes that have the `privkey` in their logical address space.

3.2 Understanding the Attacks

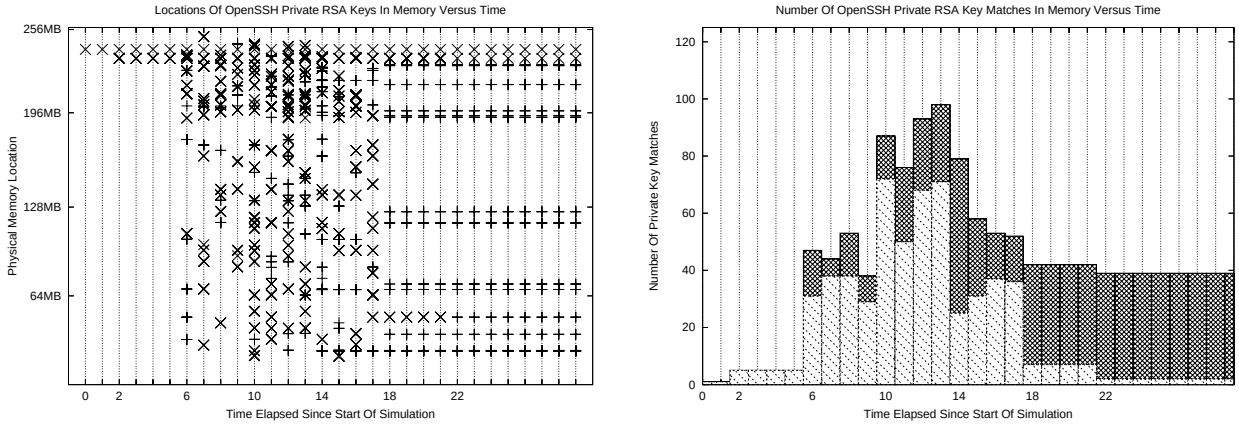
The case of OpenSSH server. Equipped with our software tool, we conducted another experiment with the same hardware and software setting as in our initial experiments, except that the operating system was replaced by Gentoo Linux with a 2.6.16.1 Linux kernel. The intent of experimenting with a newer version of operating system, which was *not* known to be subject to the aforementioned two vulnerabilities, was due to the following two considerations: (1) To confirm or deny the suspected phenomenon – cryptographic keys were somehow flooding in the memory to some extent. (2) To validate whether the suspected phenomenon is still relevant in newer operating systems.

Specifically, we let two other machines act as clients for issuing SSH requests to the server via a 100Mb/s switch network. We wrote a Perl script to automatically trigger events at the following predefined points in time (unit: 2 minutes).

- Time t=0: The simulation is started without OpenSSH running.

- Time t=2: The OpenSSH server is started via the command `/etc/init.d/sshd start`.
- Time t=6: The first client machine begins issuing SSH requests and maintains 8 concurrent `scp` transfers. Each transfer lasts about 4 seconds.
- Time t=10: The second client machine initiates an additional 8 concurrent `scp` transfers. This brings the number of concurrent connections to 16 in total.
- Time t=14: The first client machine stops all file transfers. This reduces the total number of concurrent file transfers to 8.
- Time t=18: The second client machine stops all file transfers, and thus all network traffic ceased.
- Time t=22: The OpenSSH server is stopped via the command `/etc/init.d/sshd stop`.
- Time t=29: The experiment is finished.

Corresponding to the above events, outputs of the LKM are plotted in Figures 5(a) and 5(b) from two different perspectives. Both pictures have the time as the x axis.



(a) Locations of keys in memory

(b) # of copies of the key in memory

Figure 5: OpenSSH case

Figure 5(a) shows the locations of copies of the private key in memory, where "×" represents a copy in allocated user or kernel space, and "+" represents a copy in unallocated memory. From this picture we draw the following observations. (1) The OpenSSH private key is in memory at time t=0, even though the OpenSSH server is not started until time t=2. This is because the PEM-encoded file has been loaded into memory by the Reiser file system. (2) When the OpenSSH server is started at time t=2, the newly appearing ×'s are actually the d, P, and Q of the private key. (3) When OpenSSH client requests begin to be issued at time t=6, the number of copies of the private key increases abruptly. We also begin to see copies of the private key in unallocated memory. (4) When the client machines stop issuing requests at time t=18, the number of copies of the private key in allocated memory drops abruptly. We also observe that many copies of the private key are not erased before entering unallocated memory. (5) When the OpenSSH server stops at time t=22, d, P and Q exist only in unallocated memory, except the PEM-encoded private key file that remains in the Linux kernel's page cache.

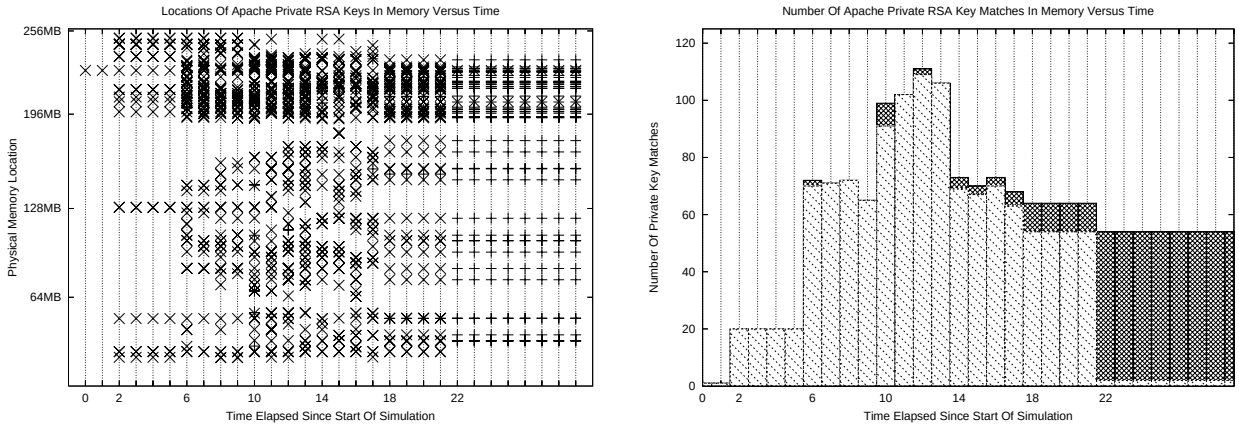
Figure 5(b) shows the total number of copies of the private key in memory, where lightly shaded bars correspond to copies of the private key in allocated memory, and dark shaded bars correspond to copies of the private key in unallocated memory.

The case of Apache HTTP server. The experiment setting is the same as in the case of OpenSSH server, except that the server being tested is Apache HTTP Server 2.0.55 with SSL enabled. and compiled

without threading support using the default `prefork` MPM. We also use two other machines to act as the clients for issuing requests to the server. All the machines are connected via a 100MB/s switch network. We also wrote a Perl script to automatically trigger events at the following predefined points in time (unit: 2 minutes).

- Time t=0: The simulation is started without Apache running.
- Time t=2: The Apache server is started via the command `/etc/init.d/apache2 start`.
- Time t=6: The first client machine begins issuing HTTPS requests and maintains 8 concurrent `wget` transfers. Each transfer lasts about 4 seconds.
- Time t=10: The second client machine initiates an additional 8 concurrent `wget` transfers. This brings the number of concurrent connections to 16 total.
- Time t=14: The first client machine generates stops all file transfers. This returns the total number of concurrent file transfers to 8.
- Time t=18: The second client machine generates stops all file transfers. All network traffic has ceased at this point.
- Time t=22: The Apache server is stopped via the command `/etc/init.d/apache2 stop`.
- Time t=29: The experiment is finished.

Figure 6(a) plots the locations of copies of the private key in memory, where "×" represents a copy in allocated memory, and "+" represents a copy in unallocated memory. From this picture we draw the following observations: (1) When the Apache server is started at time t=2, the private key appears multiple times in memory. (2) When client requests begin to be issued at time t=6, the number of copies of the private key increases abruptly. At the same time, we begin to see copies of the private key in unallocated memory. (3) When the clients stop issuing requests at time t=18, the number of copies of the private key drops abruptly. Although the total number of copies of the private key in memory decreases, the number of copies in unallocated memory increases. (4) When the Apache server stops at time t=22, there are many copies of the private key residing in unallocated memory. This remains the case until the simulation ends at time t=29.



(a) Locations of keys in memory

(b) # of copies of the key in memory

Figure 6: Apache case

Figure 6(b) plots the total number of copies of the private key in memory, where lightly shaded bars correspond to the copies in allocated memory, and dark shaded bars correspond to the copies in unallocated memory.

Summary. In both OpenSSH and Apache HTTP servers, copies of the private key can be found in both allocated memory and unallocated memory. While it is well known that unallocated memory may contain secret data because its content was not cleared, our experiment found an equally interesting phenomenon, namely that allocated memory also contains many copies. Moreover, the experiment confirmed our suspicion that copies of cryptographic keys actually somewhat flooded in memory when the number of SSH / HTTP connections increases — even in newer operating systems. Retrospectively, this may explain why the previously investigated memory disclosure attacks were so powerful.

4 Countering Memory Disclosure Attacks

Analyses in the last section naturally suggest the following countermeasures: We should ensure (i) a cryptographic key only appears in allocated memory a minimal number of times (e.g., one) as long as this does not significantly degrade system performance, and (ii) unallocated memory (or any other place with a disclosure potential such as swap space) does not have a copy of a cryptographic key. For this purpose, now we present a set of concrete solutions at different layers, from application down to operating system kernel. The strengths and limitations of each of them will be examined. Specifically, our solutions are:

* **Application level solution:** At the application level, the following can be enforced.

  1. Utilize the "copy on write" memory management policy [21] to avoid unnecessary duplications of cryptographic keys. This policy says that when a process is forked, physical memory will be copied only after one process attempts to write to that memory region. Specifically, we propose placing the private key into a special memory region, and guaranteeing that no process will write to that memory region. This ensures that the private key will only exist once in physical memory (in addition to the PEM-encoded private key file), no matter how many processes are forked.
  2. Avoid appearances of cryptographic keys as follows: (1) Ensure the private key is not explicitly copied by the application or any involved libraries. (2) Disable swapping of the memory that contains the key using the appropriate system calls. This is because when memory is swapped to disk, the memory is not immediately cleared and the private key may appear in unallocated memory.

  On one hand, the application level solution can prevent private keys from appearing in memory other than the aforemention special region (which exists in userspace) and the PEM-encoded private key file (which exists in kernelspace). On the other hand, the application level solution alone cannot guarantee that there are always no keys appearing in unallocated memory, unless special care has been taken to clear the aforementioned special memory region before the application itself dies.

* **Library level solution:** At the library level, we suggest eliminating unnecessary duplications of cryptographic keys in allocated memory using the same measures suggested in application level solution. This suffices to prevent private keys from appearing in memory other than the aforemention special region and the PEM-encoded private key file. However, the library level solution alone cannot guarantee that there are always no keys will appear in unallocated memory, unless special care has been taken to clear the aforementioned special memory region before the application itself dies.

* **Kernel level solution:** At the kernel level, we propose ensuring that the unallocated memory does not contain any private keys. This can be fulfilled by having the kernel zero any physical pages before they become unallocated. However, kernel level solution alone cannot prevent unnecessary duplications of cryptographic keys within *allocated* memory.

* **Integrated library-kernel solution:** We propose integrating the library and kernel level mechanisms together to obtain strictly stronger protection. This way, unnecessary duplications of private keys in allocated memory and any appearances of private keys in unallocated memory are simultaneously

eliminated. Moreover, this solution can even remove the PEM-encoded private key from allocated memory, provided that the library instructs the kernel not to cache the PEM-encoded private key file. Therefore, whenever possible, this solution should be adopted.

5 Protecting Private Keys of OpenSSH Servers

In this section we show how to protect private keys of OpenSSH servers (with respect to Linux operating systems and OpenSSL library) by adopting the above general methods. In Section 5.1 we elaborate the solutions with features tailored to OpenSSH servers. In Section 5.2 we evaluate the effectiveness of our recommended solution.

5.1 Solutions

Application level solution. At the application level, we instantiate the above general solution with a function, `RSA_memory_align()`, which should be called as soon as OpenSSL's `RSA` data structure contains the private key. This ensures that exactly one copy of the private key appears in allocated memory, in addition to the PEM-encoded file. Notice that we need to start OpenSSH with the undocumented `-r` option to prevent the OpenSSH server from re-executing itself after every incoming connection. Specifically, this function operates as follows:

1. `RSA_memory_align()` takes advantage of the "copy on write" memory management policy as follows. First, it uses `posix_memalign()` to request one or more memory pages for fulfilling the aforementioned special memory region. Then, it copies the private key into the special memory region, and zeros and frees the memory originally containing the private key. Then, it updates the pointers in the `RSA` data structure to point to the new location of the private key. Finally, it sets the `BN_FLG_STATIC_DATA` flag to inform OpenSSL that the private key is now exclusively located at the special region.[1]

2. `RSA_memory_align()` prevents OpenSSL's `RSA_eay_mod_exp()` from caching the private key by unsetting the `RSA_FLAG_CACHE_PRIVATE` flag in the `flags` member of the associated RSA data structure. This function is normally called once per connection when establishing a session key. OpenSSH re-executes itself after handling each connection so the cache appears to be of no use to OpenSSH. Moreover, `RSA_memory_align()` disables swapping of memory that contains the private key by calling `mlock()` on the memory allocated by `posix_memalign()`. This is necessary for dealing with memory disclosure attacks because memory that is swapped out is not immediately cleared. As an added benefit this measure helps prevent swap space based attacks.

Library level solution. At the library level, we modify the OpenSSL function `d2i_PrivateKey()`. This function is responsible for translating a PEM-encoded private key file into the RSA key parts by calling `d2i_RSAPrivateKey()` to fill in the RSA data structure. The modification is that when the `d2i_RSAPrivateKey()` method returns, we immediately call the function `RSA_memory_align()` mentioned above.

Kernel level solution. At the kernel level, we modify the kernel function `free_hot_cold_page()` to enforce that memory pages are cleared, via `clear_highpage()`, before they are added to one of the lists of free pages. This way, it is guaranteed that there are no private key appearances in unallocated memory. As mentioned before, this countermeasure was well known but somehow not widely adopted in practice.

Integrated library-kernel solution. In addition to the modifications made in the library level solution and in the kernel level solution mentioned above, the PEM-encoded private key file can be removed from allocated memory. In order to do this, we introduce a new flag, `O_NOCACHE`, to allow an application to instruct the kernel to immediately remove this file from the "page cache". Specifically this is implemented as follows.

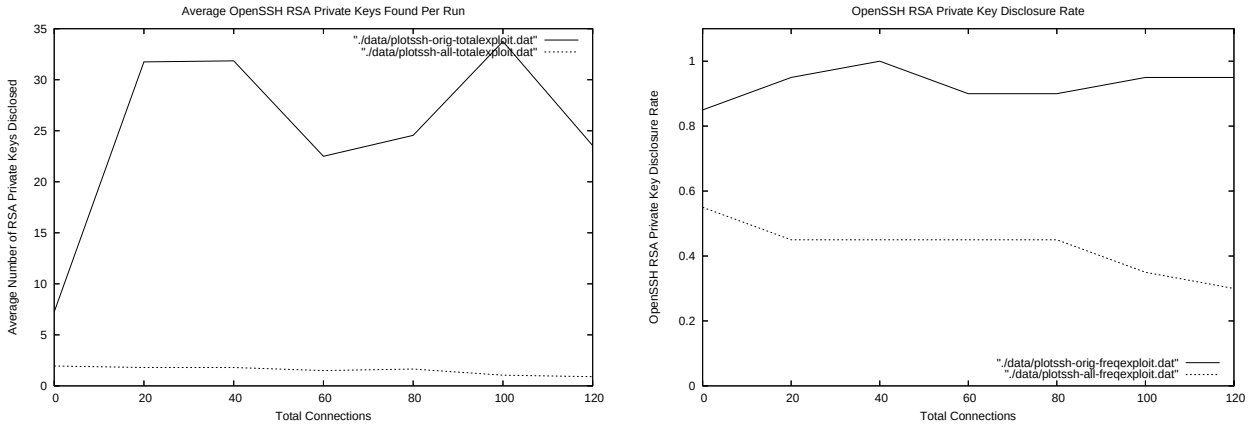
[1] It should be noted that the functionality of `RSA_memory_align()` cannot be fulfilled by OpenSSL's `RSA_memory_lock()`, which may seemingly fulfill what `RSA_memory_align()` is doing at a first glance.

Whenever the PEM-encoded private key file is read, the kernel gives the file contents to the requester and then checks if the `O_NOCACHE` flag is specified. If so, the kernel immediately deletes the corresponding "page cache" entry by calling `remove_from_page_cache()` before calling `free_page()`.

5.2 Experimental Result

Re-examination of the power of the aforementioned two attacks against vulnerable operating systems after adopting our solutions. First, we re-examined the attack based on [17] against the same vulnerable 2.6.10 Linux Kernel, except that the system is now patched with our respective solutions. In no case were we able to recover any portion of the private key. However, this does necessarily mean that our solutions are equally powerful; indeed, there is some significant difference. On one hand, both our kernel level solution and integrated library-kernel solution can completely eliminate this attack. This is because they ensure that no copies of a private key appear in unallocated memory, portions of which can be exposed by the vulnerability [17]. On the other hand, neither our application level solution nor our library level solution by itself can completely eliminate this attack.

Second, we re-examined the attack based on [12] against the same vulnerable 2.6.10 Linux kernel, except that the system is now patched with our respective solutions. For conciseness, we only consider our integrated library-kernel solution. Figure 7(a) compares the average (over 20 attacks) number of copies of the private key found in the USB device *before* and *after* deploying our solution. It clearly shows that the number of copies of the private key recovered is significantly reduced by our solution. This is because our solution ensured that only one copy of the private key appears in allocated memory, and no copies of the private key appear in unallocated memory.



(a) Comparison of # of private keys found before and after adopting our library-kernel solution: OpenSSH case

(b) Comparison of success rates of attacks before and after adopting our library-kernel solution

Figure 7: OpenSSH case

Figure 7(b) compares the success rate of attacks *before* and *after* deploying our integrated library-kernel solution. While our solution significantly reduces the success rate of attacks, the attack still succeeds with a probability about 50%. The reason is that the attack discloses on average about 50% of the memory, which means that with about 50% probability an attack would still succeed. As mentioned before, completely eliminating such powerful attacks might have to resort to some special hardware devices.

Impact of our integrated library-kernel solution on system performance. It is important that any changes to kernel or operating system does not significantly downgrade the performance of the whole system, at least from the perspective of the users. Since we are not aware of any benchmark for measuring the performance of OpenSSH servers, we wrote a simple perl script to perform a benchmark. The perl script is run on a client machine and maintains 20 concurrent `scp` connections to the OpenSSH Server. The 20 concurrent connections repeatedly transfer 10 different files until 4000 file transfers have been completed.

The 10 different files vary in size from 1 KBytes to 512 KBytes, with an average size of 102.3 KBytes. This benchmark was repeated 16 times in order to obtain average measurements. We considered two metrics: *transaction rate*, namely the number of files transferred per second, and *throughput* in terms of Mbits per second transferred. Figure 8 plots the comparison. In each metric case, the left bar corresponds to the measurement before adopting our solution, and the right bar corresponds to the measurement after adopting our solution. It is clear that our solution does not impose any performance penalty.

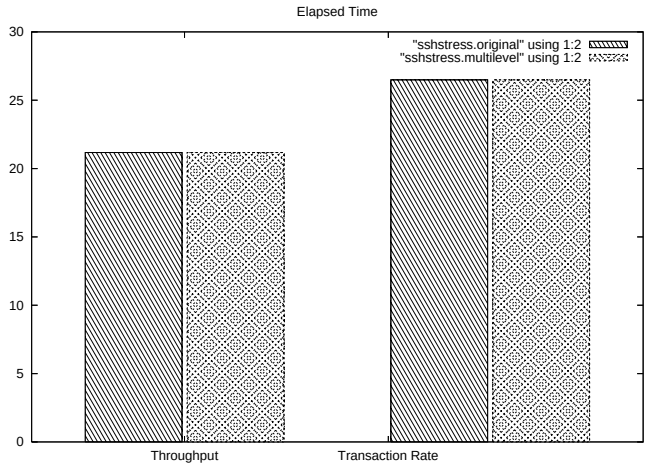


Figure 8: Comparison of performances before and after adopting our library-kernel solution: OpenSSH case

5.3 Detailed Analysis of the Experimental Results

The effectiveness of our application level solution. Figure 9 plots the locations of copies of the private key in memory, where "×" represents a copy in allocated memory and "+" denotes a key in unallocated memory. It is clear that no copies of the private key appeared in unallocated memory, which is partially due to an extra effort to store the PEM-encoded file on an `ext2` file system (so as to avoid the additional caching that may occur with the Reiser file system). Figure 10 further plots the number of copies of the private key, all in allocated memory. We stress that the number of copies of the private key in memory remains almost constant during the server running time. In Figure 9, the line of ×'s between `t=2` and `t=21` are the d, P, and Q in the special memory region allocated by `posix_memalign()`. The other line of ×'s between `t=2` and `t=29` is the PEM encoded file which stored in the "page cache".

The effectiveness of our library level solution. Figure 11 plots the locations of copies of the private key in memory, where "×" represents a copy in allocated memory and "+" denotes a key in unallocated memory. It is clear that no copies of the private key appear in unallocated memory, which is partially due to an extra effort to store the PEM-encoded file on an `ext2` file system (so as to avoid the additional caching that may occur with the Reiser file system). Figure 12 depicts the number of copies of the private key, all in allocated memory. The result is the same as in the case of deploying our application level solution. In particular, the number of private keys within memory are kept small and independent of the number of connections.

The effectiveness of our kernel level solution. Figure 13 plots the locations of copies of the private key in memory, where "×" represents a copy in allocated memory and "+" denotes a key in unallocated memory. It is clear that no copies of the private key appear in unallocated memory. Figure 14 depicts the number of copies of the private key, all in allocated memory. Notice that there are a large number of keys within allocated memory, but no copies of the private key appear in unallocated memory. This is because the kernel level solution cannot avoid unnecessary duplications of keys. Notice also that the PEM-encoded file remains in the page cache until the end of the simulation.

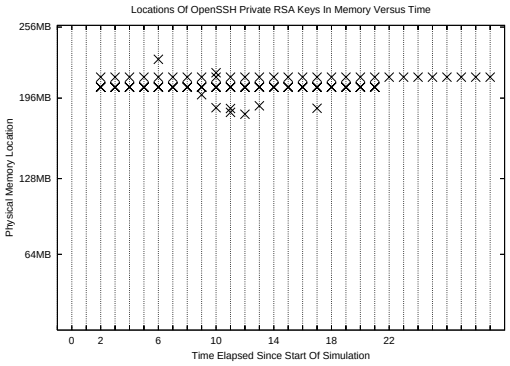


Figure 9: Location of keys in memory after deploying our application level solution

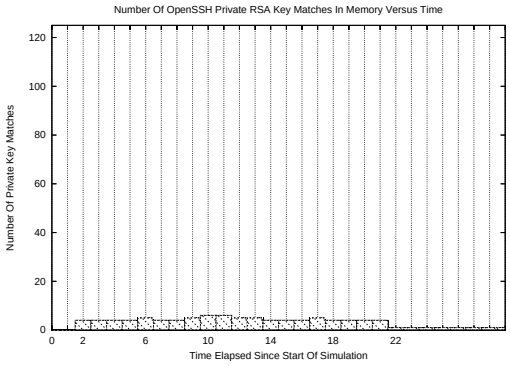


Figure 10: # of keys in memory after deploying our application level solution (actually, all in allocated memory)

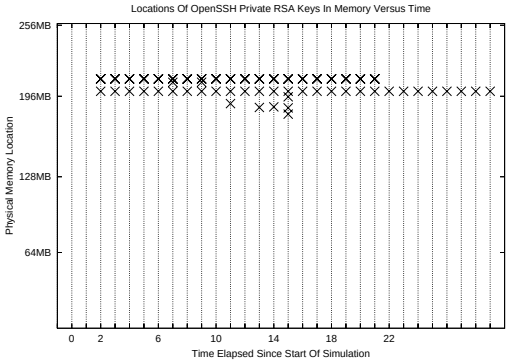


Figure 11: Location of keys in memory after deploying our library level solution

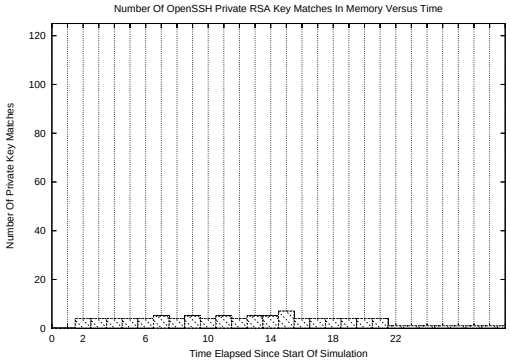


Figure 12: # of keys in memory after deploying our library level solution (actually, all in allocated memory)

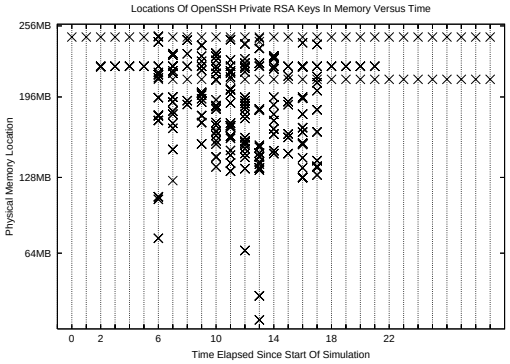


Figure 13: Location of keys in memory after deploying our kernel level solution

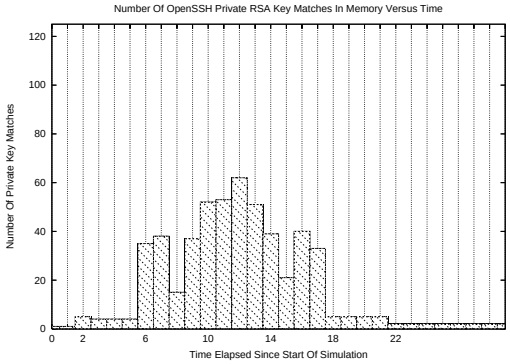


Figure 14: # of keys in memory after deploying our kernel level solution (actually, all in allocated memory)

The effectiveness of our library-kernel solution. Figure 15 plots the locations of copies of the private key in memory, where "×" represents a copy in allocated memory and "+" denotes a key in unallocated memory. It is clear that no copies of the private key appear in unallocated memory. Figure 16 depicts the number of copies of the private key, all in allocated memory. This is the most effective, and thus recommended, solution. Basically, the kernel modifications guarantee that unallocated memory will never contain copies of the private key, whereas library level measures avoid duplications of keys (including even the removal of the PEM-encoded file from allocated memory).

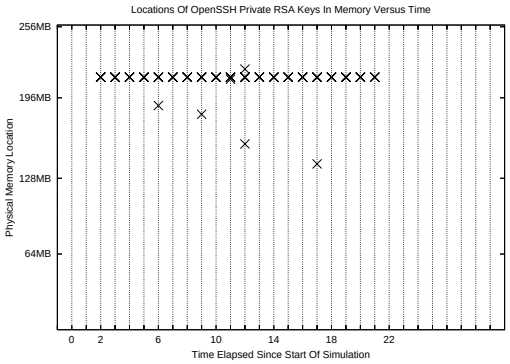


Figure 15: Location of keys in memory after deploying our library-kernel solution

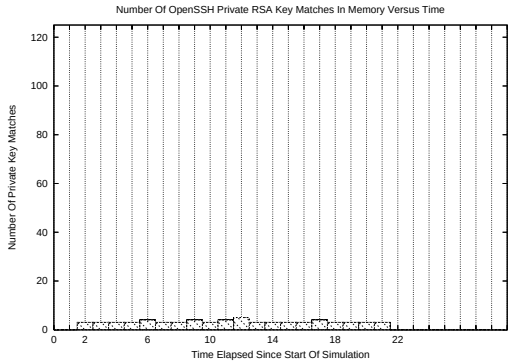


Figure 16: # of keys in memory after deploying our library-kernel solution (actually, all in allocated memory)

6 Protecting Private Keys of Apache HTTP Servers

In this section we show how to protect private keys of Apache HTTP servers (with respect to the Linux operating system and the OpenSSL Library) by adopting the above general methods. In Section 6.1 we elaborate the solutions tailored to Apache HTTP servers, which are actually quite similar to their counterparts in the case of OpenSSH servers. In Section 6.2 we evaluate the effectiveness of our recommended solution.

6.1 Solutions

Application level solution. This is the same as in the application level solution for OpenSSH server. Our function `RSA_memory_align()` is used just as before to align the RSA private keys within memory as soon as they are allocated by OpenSSL. Recall that this function clears the bit flag `RSA_FLAG_CACHE_PRIVATE` and aligns all the keys on a single page of memory allocated via `posix_memalign()`, after that it locks the page in memory via a call to `mlock()`. These techniques were presented earlier and will not be further repeated. The `RSA_memory_align()` function was added to the Apache mod SSL source code, and called from the appropriate place.

Library level solution. This is the same as its counterpart in the case of OpenSSH servers. Specifically, the RSA private key is kept on a single page, the `RSA_FLAG_CACHE_PRIVATE` flag was cleared so as to disable replications of the P and Q parts of the RSA private key, the keys are prevented from being swapped out by a call to `mlock()`.

Kernel level solution. This is the same as its counterpart in the case of OpenSSH server. Specifically, the `free_hot_cold_page()` function is modified to ensure that memory released by a process is cleared before it can be allocated again.

Integrated library-kernel solution. This is even simpler than its counterpart in the case of OpenSSH servers. This is because no modifications to Apache servers are required.

6.2 Experimental Results

Re-examination of the power of the aforementioned two attacks against vulnerable operating systems after adopting our solutions. First, we re-examined the attack based on [17] against the same vulnerable 2.6.10 Linux Kernel, except that the system is now patched with our respective solutions. In no case were we able to recover any portion of the private key. However, this does necessarily mean that our solutions are equally powerful; indeed, there is some significant difference. On one hand, both our kernel level solution and integrated library-kernel solution can completely eliminate this attack. This is because they ensure that no copies of private keys appear in unallocated memory, portion of which can be exposed by the vulnerability [17]. On the other hand, neither our application level solution nor our library level solution by itself can completely eliminate this attack. This is true even though we did not successfully find a copy of the private key in our experiments.

Second, we re-examined the attack based on [12] against the same vulnerable 2.6.10 Linux kernel, except that the system is now patched with our respective solutions. For conciseness, we only report our integrated library-kernel solution. Figure 17 compares the average (over 20 attacks) number of copies of the private key found in the USB device *before* and *after* deploying our solution. It clearly shows that the number of copies of the private key recovered is significantly reduced by our solution. This is because our solution ensured that only one copy of the private key appears in allocated memory, and no copies of the private key appear in unallocated memory.

Figure 18 compares the success rate of attacks *before* and *after* deploying our integrated library-kernel solution. While our solution substantially reduces the success rate of attacks, an attack still succeeds with a probability about 38%. The reason is that an attack discloses on average about 50% of the whole memory.

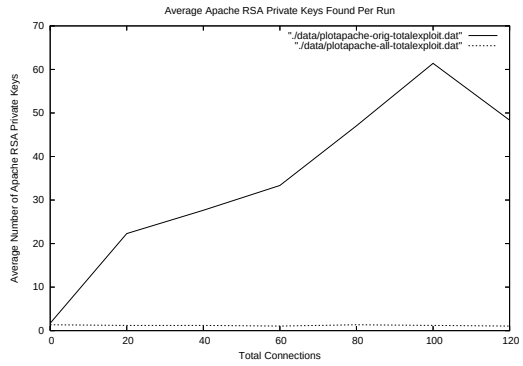


Figure 17: Comparison of # of private keys found before and after adopting our library-kernel solution: Apache case

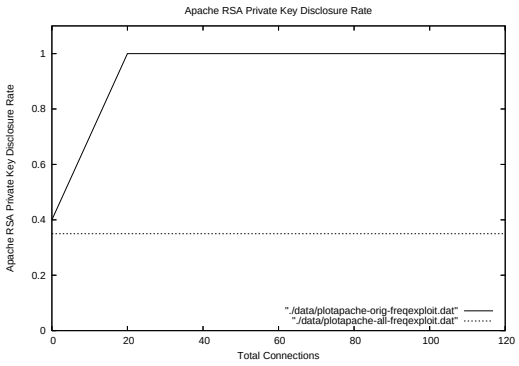


Figure 18: Comparison of success rate of attacks before and after adopting our library-kernel solution: Apache case

Impact of integrated library-kernel solution on system performance. Benchmarks were taken using the open source Siege benchmarking utility, which ran on a separate client machine to initiate 4000 HTTPS transactions while attempting to maintain 20 concurrent connections throughout the experiment. CPU utilization was observed to guarantee that the benchmarks were not influenced by the client machine. We considered four metrics: *response time*, namely the average time the Apache HTTP server took to respond to each request; *throughput*, namely the average number of bytes transferred by Apache every second; *transaction rate*, namely the average number of transactions the Apache was able to handle per second; *concurrency*, namely the average number of concurrent connections throughout the experiment. Figure 19 plots the average response time (seconds) and throughput (bytes per second) metrics output by the Siege stress tester. Figure 20 plots the average transaction rate (transactions per second) and concurrency (processes) output by the Siege stress tester. These pictures clearly show that our modifications to the library and kernel do not incur any performance penalty.

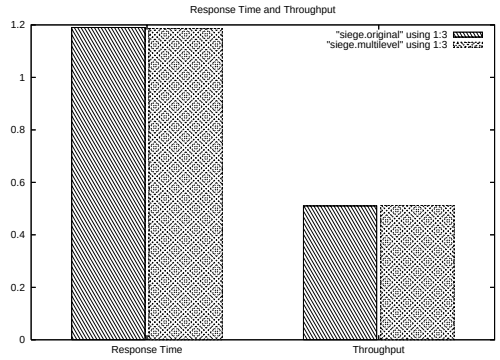


Figure 19: Comparisons of *average response time* and *throughput* before and after adopting our library-kernel solution

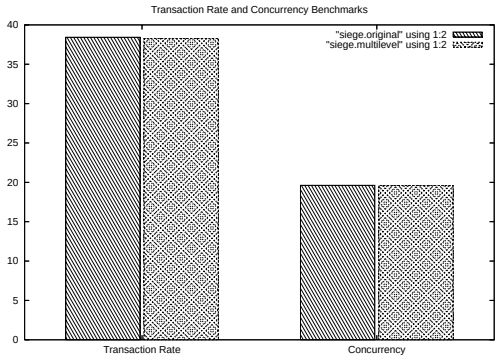


Figure 20: Comparisons of *average transaction rate* and *concurrency* before and after adopting our library-kernel solution

6.3 Detailed Analysis of the Experimental Results

The effectiveness of our application level solution. Figure 21 show the results after deploying our application level solution, where "×" represents a copy in allocated memory and "+" denotes a key in unallocated memory. It is clear that no copies of the private key appear in unallocated memory. Figure 22 plots the number of copies of the private key in memory, actually all in allocated memory. Notice that the number of keys in memory are no longer dependent on the number of processes running, and the d, P, and Q exist in one single page in memory throughout the experiment. The PEM-encoded file is loaded into the Linux kernels page cache and remains there through the end of the experiment.

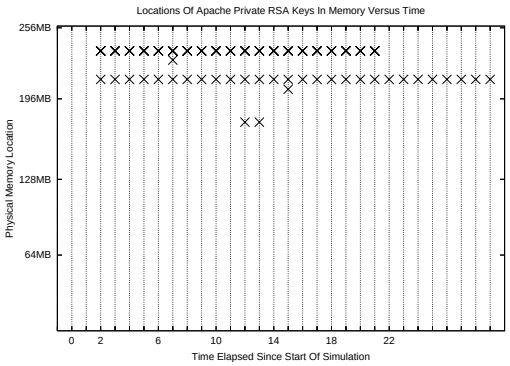


Figure 21: Location of keys in memory after deploying our application level solution

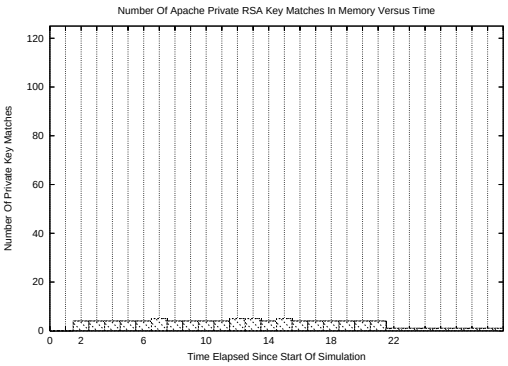


Figure 22: # of keys in memory after deploying our application level solution (actually, all in allocated memory)

The effectiveness of our library level solution. Figure 23 show the results after deploying our library level solution, where "×" represents a copy in allocated memory and "+" denotes a key in unallocated memory. It is clear that no copies of the private key appear in unallocated memory. Figure 24 plots the number of copies of the private key in memory, actually all in allocated memory. The result is the same as in the case of deploying our application level solution.

The effectiveness of our kernel level solution. Figure 25 show the results after deploying our kernel level solution, where "×" represents a copy in allocated memory and "+" denotes a key in unallocated memory. It is clear that no copies of the private key appear in unallocated memory. Figure 26 plots the number of copies of the private key in memory, actually all in allocated memory.

The effectiveness of our integrated library-kernel solution. Figure 27 show the results after deploying

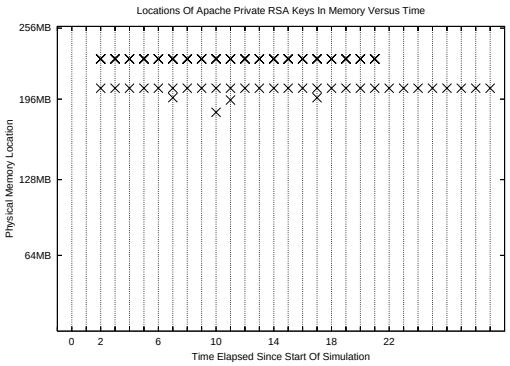


Figure 23: Location of keys in memory after deploying our library level solution

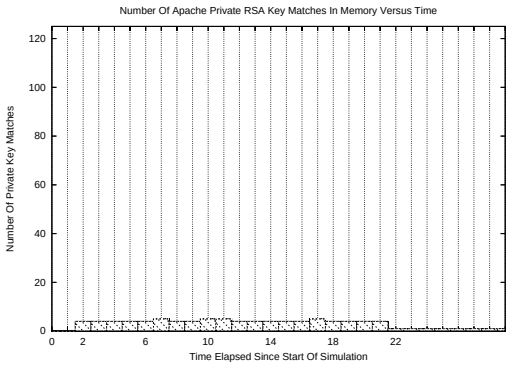


Figure 24: # of keys in memory after deploying our library level solution (actually, all in allocated memory)

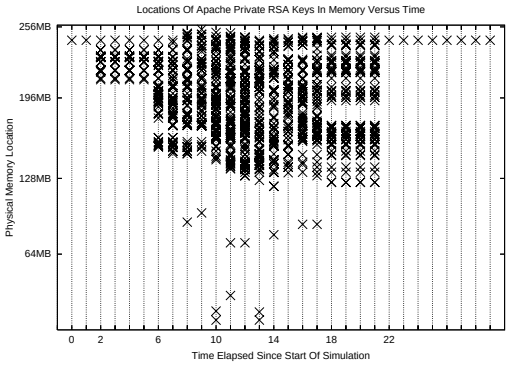


Figure 25: Location of keys in memory after deploying our kernel level solution

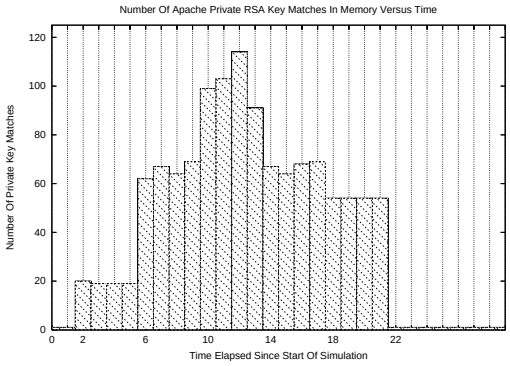


Figure 26: # of keys in memory after deploying our kernel level solution (actually, all in allocated memory)

our library-kernel solution, where "×" represents a copy in allocated memory and "+" denotes a key in unallocated memory. It is clear that no copies of the private key appear in unallocated memory. Figure 28 plots the number of copies of the private key in memory, actually all in allocated memory. Again, the kernel level modifications guarantee the private key will never appear in unallocated memory, and the library level mechanisms eliminate the unnecessary duplication of the private key. Moreover, the PEM-encoded file is even removed from memory.

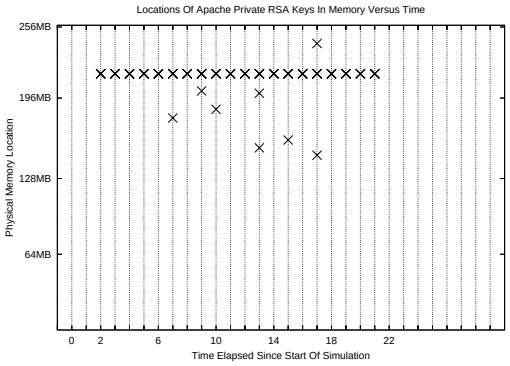


Figure 27: Location of keys in memory after deploying our library-kernel solution

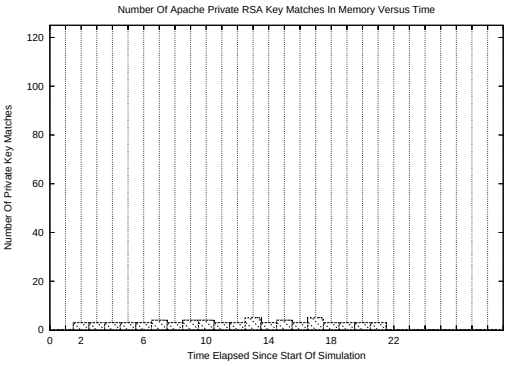


Figure 28: # of keys in memory after deploying our library-kernel solution (actually, all in allocated memory)

7 Conclusion

We showed that memory disclosure attacks are a severe threat to real-life systems. We then explored a set of mechanisms to deal with such attacks. The mechanisms have been successfully integrated into real-life systems including the OpenSSL Library and the Linux kernel. Through simulation and benchmarking, we showed that our mechanisms are effective in eliminating memory disclosure attacks that disclose unallocated memory, and in mitigating the damage due to attacks that disclose a small portion of allocated memory. Our result suggests that in order to eliminate powerful attacks that can disclose a large portion of memory, one may have to resort to special hardware devices.

Acknowledgement. We thank the anonymous reviewers of DSN'07 for their valuable comments, and our DSN'07 shepherd, Luigi Romano, for his constructive suggestions that improved the paper.

References

[1] R. Anderson. On the forward security of digital signatures. Technical report, 1997.

[2] M. Bellare and S. Miner. A forward-secure digital signature scheme. In *Proc. Crypto'99*, pages 431–448.

[3] M. Bellare and B. Yee. Forward-security in private-key cryptography. In *Cryptographer's Track - RSA Conference (CT-RSA)*, pages 1–18.

[4] Bochs. the bochs ia-32 emulator project. `http://bochs.sourceforge.net/`.

[5] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proceedings of Usenix Security Symposium 2003*, 2004.

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

[7] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime. In *Proc. 14th USENIX Security Symposium*, August 2005.

[8] G. Di Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson. How to forget a secret. In *Proceedings of STACS 1999*, pages 500–509. 1999.

[9] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Proc. Crypto'89*, pages 307–315.

[10] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *Proc. EUROCRYPT'02*, pages 65–82, 2002.

[11] Trusted Computing Group. https://www.trustedcomputinggroup.org/.

[12] Georgi Guninski. linux kernel 2.6 fun. windoze is a joke. `http://www.guninski.com/where_do_you_want_billg_to_go_today_3.html` (dated 15 February 2005).

[13] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of Usenix Security Symposium 1996*, pages 77–90, 1996.

[14] G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In *Proc. Crypto'01*, pages 332–354. 2001.

[15] G. Itkis and L. Reyzin. Sibir: Signer-base intrusion-resilient signatures. In *Proc. Crypto'02*, pages 499–514. 2002.

[16] K. Harrison and S. Xu. Protecting cryptographic keys from memory disclosure attacks. In *Proc. DSN'07*.

[17] Mathieu Lafon and Romain Francoise. Information leak in the linux kernel ext2 implementation. `http://arkoon.net/advisories/ext2-make-empty-leak.txt` (Arkoon Security Team Advisory - dated March 25, 2005).

[18] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *Proc. ACM PODC '91*, pages 51–59. 1991.

[19] N. Provos. Encrypting virtual memory. In *Proceedings of Usenix Security Symposium 2000*, 2000.

[20] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM*, 21(2):120–126, 1978.

[21] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts (sixth ed.)*. John Wiley & Sons, 2001.

[22] J. Viega. Protecting sensitive data in memory. `http://www.cgisecurity.com/lib/protecting-sensitive-data.html`, 2001.

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

8 Appendix: Source Code

8.1 Loadable Kernel Module

```
/* scanapache.c */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/stat.h>
#include <linux/fcntl.h>
#include <linux/tty.h>
#include <linux/syscalls.h>
#include <linux/compat.h>
#include <asm/segment.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <linux/mm.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/pagemap.h>
#include <linux/rmap.h>
#include <linux/swap.h>
#include <linux/swapops.h>

#define MAXWRITE 80
#define MIN 5
#define MEMMB 256
#define MEMMAX (MEMMB*1024*1024-4)
#define MEMOFF PAGE_OFFSET
#define MAXSEARCH (MEMOFF+MEMMAX)
#define NUMPAGES 32
#define BUFFSIZE (PAGE_SIZE*NUMPAGES)
#define MAXLEN (BUFFSIZE-2*PAGE_SIZE)

#define BEGIN_KMEM { mm_segment_t old_fs = get_fs(); set_fs(get_ds());
#define END_KMEM set_fs(old_fs); }
#define sys_write(f, buf, sz) (f->f_op->write(f, buf, sz, &f->f_pos))

struct {
        unsigned int num;
        char **name;
        unsigned int *size;
        unsigned long **data;
        unsigned long *first;
} scan;

static struct proc_dir_entry *infop;

static char *mem_buffer;

static char begindata[BUFFSIZE];

static unsigned long len;

static const char *procname = "apachemem";

inline static struct anon_vma *page_lock_anon_vma(struct page *page)
{
        struct anon_vma *anon_vma = NULL;
        unsigned long anon_mapping;

        rcu_read_lock();
        anon_mapping = (unsigned long) page->mapping;
        if (!(anon_mapping & PAGE_MAPPING_ANON))
                goto out;
        if (!page_mapped(page))
                goto out;

        anon_vma = (struct anon_vma *) (anon_mapping - PAGE_MAPPING_ANON);
```

```
        spin_lock(&anon_vma->lock);
out:
        rcu_read_unlock();
        return anon_vma;
}

/*
 * Takes page frame number which is just the physical page
 * number to get this do: absolute memory location >> PAGE_SHIFT
 */
inline static void print_rev_mapping(unsigned long pfn){
  struct page *page;
  struct anon_vma *anon_vma;
  struct vm_area_struct *vma;
  struct task_struct *p;
  if(pfn_valid(pfn)){
    page=pfn_to_page(pfn);
    lock_page(page);
    if((anon_vma=page_lock_anon_vma(page))!=NULL){
      list_for_each_entry(vma, &anon_vma->head, anon_vma_node) {
        for_each_process(p){
          if(p->mm==vma->vm_mm){
            len+=snprintf(mem_buffer+len,128," %u",p->pid);
          }
        }
      }
      spin_unlock(&anon_vma->lock);
    } else if(page_count(page) == 0){
      len+=snprintf(mem_buffer+len,128," none");
    } else {
      len+=snprintf(mem_buffer+len,128," 0");
    }
    unlock_page(page);
  } else {
    len+=snprintf(mem_buffer+len,128," invalid pfn");
  }
  len+=snprintf(mem_buffer+len,128,"\n");
}

int procfile_read(char *page, char **pagestart, off_t off,
                  int count, int *eof, void *data)
{
  unsigned long loc,num,max,x;
  unsigned long *priv;
  if(off==0){
    mem_buffer[0]='\0';
    len=snprintf(mem_buffer,80,"Request recieved\n");
  } else {
    len=strlen(mem_buffer);
    if(off>=len){
      return 0;
    }
    *pagestart=mem_buffer+off;
    return len-off;
  }

  for(loc = MEMOFF; loc < MAXSEARCH && len < MAXLEN; loc++){
    for(num =0;num<scan.num;num++){
      if(unlikely(*((int *)(loc)) == (scan.first[num]^0xFFFFFFFF))){
        preempt_disable();
        max = scan.size[num];
        priv = scan.data[num];
        x=1;
        while((loc+(x*4))<MAXSEARCH&&(((int *)(loc))[x]==(priv[x]^0xFFFFFFFF))&&(++x<max));
        if(x>=max){
          len+=snprintf(mem_buffer+len,128,
                        "Full match found for %s of size %u bytes at: %09u, in page: %06u, processes:",
                        scan.name[num],(unsigned int)x*4,(unsigned int)(loc-MEMOFF),
                        (unsigned int)((loc-MEMOFF)>>PAGE_SHIFT));
          print_rev_mapping((loc-MEMOFF)>>PAGE_SHIFT);
        } else if(x>=MIN){
```

```
          len+=snprintf(mem_buffer+len,128,
                        "Partial match found for %s of size %u bytes at: %09u, in page: %06u, processes:",
                        scan.name[num],(unsigned int)x*4,(unsigned int)(loc-MEMOFF),
                        (unsigned int)((loc-MEMOFF)>>PAGE_SHIFT));
          print_rev_mapping((loc-MEMOFF)>>PAGE_SHIFT);
        }
        preempt_enable();
      }
    }
  }
  *pagestart=mem_buffer;
  return(len);
}

static int __init scanapache_init(void) {
  char *scandata;
  int i,ret,fd;

  infop = create_proc_entry(procname,0444,NULL);
  if(infop == 0){
    return -EINVAL;
  }
  infop->owner = THIS_MODULE;
  infop->read_proc = (void *)procfile_read;

  scandata=begindata;

  BEGIN_KMEM
  /* open file */
  if((fd=sys_open("/rawork/racode/apachekey",O_RDONLY,0000))<0){
    printk("open in failed: %d\n",fd);
    return -1;
  }

  /* read in a whole page */
  if((ret=(sys_read(fd,scandata,PAGE_SIZE)))<=0){
    printk("read failed: %d\n",ret);
    return -1;
  }

  /* close file */
  if(sys_close(fd)<0){
    printk("close failed: %d\n",ret);
    return -1;
  }
  END_KMEM

  /* get num */
  scan.num=*(unsigned int *)scandata;

  /* fill data structure */
  scandata+=ret;
  scan.name=(char **)scandata;
  scandata+=scan.num*sizeof(char *);
  scan.size=(unsigned int *)scandata;
  scandata+=scan.num*sizeof(unsigned int);
  scan.data=(unsigned long **)scandata;
  scandata+=scan.num*sizeof(unsigned long *);
  scan.first=(unsigned long *)scandata;
  scandata+=scan.num*sizeof(unsigned long);

  /* save end of scan data for /proc mem_buffer */
  mem_buffer=begindata+PAGE_SIZE;

  /* foreach 0..num */
  scandata=begindata+sizeof(unsigned int);
  for(i=0;i<scan.num;i++){
    /* read the name */
    scan.name[i]=scandata;
    while(*(scandata++)!='\0');
    /* read the size */
```

```
    scan.size[i]=*((unsigned int *)scandata);
    scandata+=sizeof(unsigned int);
    /* read the data */
    scan.data[i]=(unsigned long *)scandata;
    scandata+=scan.size[i]*sizeof(unsigned long);
    /* copy first */
    scan.first[i]=scan.data[i][0];
  }

  return 0;
}


static void __exit scanapache_exit(void) {
  remove_proc_entry(procname,NULL);

  /* clear data before exit */
  memset(begindata,0,BUFFSIZE);
}

module_init(scanapache_init);
module_exit(scanapache_exit);

MODULE_LICENSE("GPL");
```

```
/* scanssh.c */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/stat.h>
#include <linux/fcntl.h>
#include <linux/tty.h>
#include <linux/syscalls.h>
#include <linux/compat.h>
#include <asm/segment.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <linux/mm.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/pagemap.h>
#include <linux/rmap.h>
#include <linux/swap.h>
#include <linux/swapops.h>

#define MAXWRITE 80
#define MIN 5
#define MEMMB 256
#define MEMMAX (MEMMB*1024*1024-4)
#define MEMOFF PAGE_OFFSET
#define MAXSEARCH (MEMOFF+MEMMAX)
#define NUMPAGES 32
#define BUFFSIZE (PAGE_SIZE*NUMPAGES)
#define MAXLEN (BUFFSIZE-2*PAGE_SIZE)

#define BEGIN_KMEM { mm_segment_t old_fs = get_fs(); set_fs(get_ds());
#define END_KMEM set_fs(old_fs); }
#define sys_write(f, buf, sz) (f->f_op->write(f, buf, sz, &f->f_pos))

struct {
        unsigned int num;
        char **name;
        unsigned int *size;
        unsigned long **data;
        unsigned long *first;
} scan;

static struct proc_dir_entry *infop;

static char *mem_buffer;
```

```
static char begindata[BUFFSIZE];

static unsigned long len;

static const char *procname = "sshmem";

inline static struct anon_vma *page_lock_anon_vma(struct page *page)
{
        struct anon_vma *anon_vma = NULL;
        unsigned long anon_mapping;

        rcu_read_lock();
        anon_mapping = (unsigned long) page->mapping;
        if (!(anon_mapping & PAGE_MAPPING_ANON))
                goto out;
        if (!page_mapped(page))
                goto out;

        anon_vma = (struct anon_vma *) (anon_mapping - PAGE_MAPPING_ANON);
        spin_lock(&anon_vma->lock);
out:
        rcu_read_unlock();
        return anon_vma;
}

/*
 * Takes page frame number which is just the physical page
 * number to get this do: absolute memory location >> PAGE_SHIFT
 */
inline static void print_rev_mapping(unsigned long pfn){
  struct page *page;
  struct anon_vma *anon_vma;
  struct vm_area_struct *vma;
  struct task_struct *p;
  if(pfn_valid(pfn)){
    page=pfn_to_page(pfn);
    lock_page(page);
    if((anon_vma=page_lock_anon_vma(page))!=NULL){
      list_for_each_entry(vma, &anon_vma->head, anon_vma_node) {
        for_each_process(p){
          if(p->mm==vma->vm_mm){
            len+=snprintf(mem_buffer+len,128," %u",p->pid);
          }
        }
      }
      spin_unlock(&anon_vma->lock);
    } else if(page_count(page) == 0){
      len+=snprintf(mem_buffer+len,128," none");
    } else {
      len+=snprintf(mem_buffer+len,128," 0");
    }
    unlock_page(page);
  } else {
    len+=snprintf(mem_buffer+len,128," invalid pfn");
  }
  len+=snprintf(mem_buffer+len,128,"\n");
}

int procfile_read(char *page, char **pagestart, off_t off,
                  int count, int *eof, void *data)
{
  unsigned long loc,num,max,x;
  unsigned long *priv;
  if(off==0){
    mem_buffer[0]='\0';
    len=snprintf(mem_buffer,80,"Request recieved\n");
  } else {
    len=strlen(mem_buffer);
    if(off>=len){
      return 0;
```

```
    }
    *pagestart=mem_buffer+off;
    return len-off;
  }

  for(loc = MEMOFF; loc < MAXSEARCH && len < MAXLEN; loc++){
    for(num =0;num<scan.num;num++){
      if(unlikely(*((int *)(loc)) == (scan.first[num]^0xFFFFFFFF))){
        preempt_disable();
        max = scan.size[num];
        priv = scan.data[num];
        x=1;
        while((loc+(x*4))<MAXSEARCH&&(((int *)(loc))[x]==(priv[x]^0xFFFFFFFF))&&(++x<max));
        if(x>=max){
          len+=snprintf(mem_buffer+len,128,
                        "Full match found for %s of size %u bytes at: %09u, in page: %06u, processes:",
                        scan.name[num],(unsigned int)x*4,(unsigned int)(loc-MEMOFF),
                        (unsigned int)((loc-MEMOFF)>>PAGE_SHIFT));
          print_rev_mapping((loc-MEMOFF)>>PAGE_SHIFT);
        } else if(x>=MIN){
          len+=snprintf(mem_buffer+len,128,
                        "Partial match found for %s of size %u bytes at: %09u, in page: %06u, processes:",
                        scan.name[num],(unsigned int)x*4,(unsigned int)(loc-MEMOFF),
                        (unsigned int)((loc-MEMOFF)>>PAGE_SHIFT));
          print_rev_mapping((loc-MEMOFF)>>PAGE_SHIFT);
        }
        preempt_enable();
      }
    }
  }
  *pagestart=mem_buffer;
  return(len);
}

static int __init scanssh_init(void) {
  char *scandata;
  int i,ret,fd;

  infop = create_proc_entry(procname,0444,NULL);
  if(infop == 0){
    return -EINVAL;
  }
  infop->owner = THIS_MODULE;
  infop->read_proc = (void *)procfile_read;

  scandata=begindata;

  BEGIN_KMEM
  /* open file */
  if((fd=sys_open("/rawork/racode/sshkey",O_RDONLY,0000))<0){
    printk("open in failed: %d\n",fd);
    return -1;
  }

  /* read in a whole page */
  if((ret=(sys_read(fd,scandata,PAGE_SIZE)))<=0){
    printk("read failed: %d\n",ret);
    return -1;
  }

  /* close file */
  if(sys_close(fd)<0){
    printk("close failed: %d\n",ret);
    return -1;
  }
  END_KMEM

  /* get num */
  scan.num=*(unsigned int *)scandata;

  /* fill data structure */
```

```
  scandata+=ret;
  scan.name=(char **)scandata;
  scandata+=scan.num*sizeof(char *);
  scan.size=(unsigned int *)scandata;
  scandata+=scan.num*sizeof(unsigned int);
  scan.data=(unsigned long **)scandata;
  scandata+=scan.num*sizeof(unsigned long *);
  scan.first=(unsigned long *)scandata;
  scandata+=scan.num*sizeof(unsigned long);

  /* save end of scan data for /proc mem_buffer */
  mem_buffer=begindata+PAGE_SIZE;

  /* foreach 0..num */
  scandata=begindata+sizeof(unsigned int);
  for(i=0;i<scan.num;i++){
    /* read the name */
    scan.name[i]=scandata;
    while(*(scandata++)!='\0');
    /* read the size */
    scan.size[i]=*((unsigned int *)scandata);
    scandata+=sizeof(unsigned int);
    /* read the data */
    scan.data[i]=(unsigned long *)scandata;
    scandata+=scan.size[i]*sizeof(unsigned long);
    /* copy first */
    scan.first[i]=scan.data[i][0];
  }

  return 0;
}


static void __exit scanssh_exit(void) {
  remove_proc_entry(procname,NULL);

  /* clear data before exit */
  memset(begindata,0,BUFFSIZE);
}

module_init(scanssh_init);
module_exit(scanssh_exit);

MODULE_LICENSE("GPL");
```

8.2 Simulator

```
#!/usr/bin/perl

# Get Action
$action = shift;

# Get Servers
$servers = shift;

# Machines
$box1 = shift;
$box2 = shift;

$usage = "Usage: runsimulation.pl <action> <servers> <ipaddr>\n".
         "   action - all, plot\n".
         "   servers - apache, ssh, both\n";

if($action ne "all" && $action ne "plot"){
   die "1: ".$usage;
}

if ($servers ne "ssh" && $servers ne "apache" && $servers ne "both"){
   die "2: ".$usage;
}

if($action eq "all" && ($box1 eq "" || $box2 eq "")){
   die "3: ".$usage;
}

# Specify servers
$dossh=0;
$doapache=0;
if($servers eq "ssh" || $servers eq "both"){
   $dossh=1;
}
if($servers eq "apache" || $servers eq "both"){
   $doapache=1;
}

# Specify times
$startservers =  2;
$starttraffic =  6;
$moretraffic  = 10;
$lesstraffic  = 14;
$stoptraffic  = 18;
$stopservers  = 22;
$max = 29;

# Total time between iterations (not exact)
$sleeptime = 2.0;

#start at zero
$iter=0;

goto $action;

all:

system("(cd module && make clean && make && make install)");
system("rmmod scanssh");
system("rmmod scanapache");

if($dossh){
   system("modprobe scanssh");
}
if($doapache){
   system("modprobe scanapache");
}

system("mkdir -p ./data");
```

```
print "\n\n";
print "\n*** Fresh Boot ***\n\n";

foreach $iter (0..$max){
   print "\n";
   print "***************\n";
   print "Iteration: $iter\n";
   print "***************\n";
   print "\n";

   if      ($iter < $startservers){
      if($dossh){
         print "*** SSHD Not Started Yet ***\n";
      }
      if($doapache){
         print "*** Apache Not Started Yet ***\n";
      }
   } elsif ($iter== $startservers){
      if($dossh){
         print "*** Starting SSHD... ***\n";
         print "/etc/init.d/sshd start\n";
         @lines=`/etc/init.d/sshd start`;
         print @lines;
      }
      if($doapache){
         print "*** Starting Apache... ***\n";
         print "/etc/init.d/apache2 start\n";
         @lines=`/etc/init.d/apache2 start`;
         print @lines;
      }
   } elsif ($iter < $starttraffic){
      if($dossh){
         print "*** SSHD Running... ***\n";
      }
      if($doapache){
         print "*** Apache Running... ***\n";
      }
   } elsif ($iter== $starttraffic){
      print "*** Starting A Moderate Amount Of Fake Traffic Now... ***\n";
      foreach $q (1..8){
         print "./apps/startconnect.pl $box1\n";
         system("./apps/startconnect.pl $box1");
         sleep(2);
      }
   } elsif ($iter < $moretraffic){
      print "*** A Moderate Amount Of Fake Traffic Running... ***\n";
   } elsif ($iter== $moretraffic){
      print "*** Starting A Large Amount Of Fake Traffic Now... ***\n";
      foreach $q (1..8){
         print "./apps/startconnect.pl $box2\n";
         system("./apps/startconnect.pl $box2");
         sleep(2);
      }
   } elsif ($iter < $lesstraffic){
      print "*** A Large Amount Of Fake Traffic Running... ***\n";
   } elsif ($iter== $lesstraffic){
      print "*** Stoping Some Fake Traffic Now... ***\n";
      print "./apps/stopconnect.pl $box1\n";
      system("./apps/stopconnect.pl $box1");
      sleep(2);
      print "./apps/stopconnect.pl $box1\n";
      system("./apps/stopconnect.pl $box1");
      sleep(2);
      print "./apps/stopconnect.pl $box1\n";
      system("./apps/stopconnect.pl $box1");
   } elsif ($iter < $stoptraffic){
      print "*** A Moderate Amount Of Fake Traffic Running... ***\n";
   } elsif ($iter== $stoptraffic){
      print "*** Stoping All Fake Traffic Now... ***\n";
      print "./apps/stopconnect.pl $box2\n";
```

```
      system("./apps/stopconnect.pl $box2");
      sleep(2);
      print "./apps/stopconnect.pl $box2\n";
      system("./apps/stopconnect.pl $box2");
      sleep(2);
      print "./apps/stopconnect.pl $box2\n";
      system("./apps/stopconnect.pl $box2");
   } elsif ($iter < $stopservers){
      print "*** All Fake Traffic Stopped... ***\n";
   } elsif ($iter== $stopservers){
      if($dossh){
         print "*** Stoppng SSHD... ***\n";
         print "/etc/init.d/sshd stop\n";
         @lines=`/etc/init.d/sshd stop`;
         print @lines;
      }
      if($doapache){
         print "*** Stoppng Apache... ***\n";
         print "/etc/init.d/apache2 stop\n";
         @lines=`/etc/init.d/apache2 stop`;
         print @lines;
      }
   } elsif ($iter<$max){
      if($dossh){
         print "*** SSHD Stopped... ***\n";
      }
      if($doapache){
         print "*** Apache Stopped... ***\n";
      }
   }
   print "Sleeping for ".($sleeptime*(1/10))." minute(s)... \n";
   sleep(15*$sleeptime);

   print "\n";
   if($dossh){
      print "copying /proc/sshmem to ./data/sshmem$iter\n";
      system("nice -n -20 ./apps/safebackgroundcopy /proc/sshmem ./data/sshmem$iter");
   }
   if($doapache){
      print "copying /proc/apachemem to ./data/apachemem$iter\n";
      system("nice -n -20 ./apps/safebackgroundcopy /proc/apachemem ./data/apachemem$iter");
   }
   print "Sleeping for ".($sleeptime*(9/10))." minute(s)... \n";
   sleep(45*$sleeptime);

   print "\n";
}

plot:

if($dossh){
   system("./plotssh.pl $max");
}
if($doapache){
   system("./plotapache.pl $max");
}
```

8.3 Application Patches

```
diff -urN -X ignore-list openssh-4.3p2-orig/authfile.c openssh-4.3p2/authfile.c
--- openssh-4.3p2-orig/authfile.c  2006-03-31 01:36:01.000000000 -0600
+++ openssh-4.3p2/authfile.c  2006-03-31 03:14:15.000000000 -0600
@@ -451,6 +451,52 @@
   return NULL;
 }

+/* START MYADDITION */
+void *malloc_locked_new(size_t size,unsigned int pgsize){
+  void *ptr;
+  size=pgsize*(1+size/pgsize);
+  if(posix_memalign(&ptr,pgsize,size)!=0)
+     return NULL;
+  mlock(ptr,size);
+  return ptr;
+}
+
+int RSA_memory_align(RSA *r)
+  {
+  int i,j;
+  BIGNUM *t[6],*b;
+  BN_ULONG *ul;
+  unsigned int pgsize = sysconf(_SC_PAGESIZE);
+
+  if (r->d == NULL) return(1);
+  t[0]= r->d;
+  t[1]= r->p;
+  t[2]= r->q;
+  t[3]= r->dmp1;
+  t[4]= r->dmq1;
+  t[5]= r->iqmp;
+
+  if ((ul=(BN_ULONG *)malloc_locked_new(sizeof(BN_ULONG)*(b->dmax),pgsize)) == NULL){
+     return(0);
+  }
+  r->bignum_data=(char *)ul;
+
+  for (i=0; i<6; i++){
+     b= t[i];
+     b->flags=BN_FLG_STATIC_DATA;
+     memcpy((char *)ul,b->d,sizeof(BN_ULONG)*b->top);
+     memset(b->d,0,sizeof(BN_ULONG)*b->top);
+     free(b->d);
+     b->d=ul;
+     ul+=b->top;
+  }
+
+  r->flags&= ~(RSA_FLAG_CACHE_PRIVATE|RSA_FLAG_CACHE_PUBLIC);
+
+  return(1);
+}
+/* END MYADDITION */
+
 Key *
 key_load_private_pem(int fd, int type, const char *passphrase,
     char **commentp)
@@ -475,6 +521,9 @@
     prv = key_new(KEY_UNSPEC);
     prv->rsa = EVP_PKEY_get1_RSA(pk);
     prv->type = KEY_RSA;
+      /* START MYADDITION */
+      RSA_memory_align(prv->rsa);
+      /* END MYADDITION */
     name = "rsa w/o comment";
 #ifdef DEBUG_PK
     RSA_print_fp(stderr, prv->rsa, 8);

diff -urN httpd-2.0.55-orig/modules/ssl/ssl_engine_init.c httpd-2.0.55/modules/ssl/ssl_engine_init.c
--- httpd-2.0.55-orig/modules/ssl/ssl_engine_init.c  2006-03-31 02:19:42.000000000 -0600
+++ httpd-2.0.55/modules/ssl/ssl_engine_init.c  2006-03-31 05:20:30.000000000 -0600
```

```
@@ -736,6 +736,53 @@
     return TRUE;
 }

+/* START MYADDITION */
+void *malloc_locked_new(size_t size,unsigned int pgsize){
+  void *ptr;
+  size=pgsize*(1+size/pgsize);
+  if(posix_memalign(&ptr,pgsize,size)!=0)
+     return NULL;
+  mlock(ptr,size);
+  return ptr;
+}
+
+int RSA_memory_align(RSA *r)
+  {
+  int i,j;
+  BIGNUM *t[6],*b;
+  BN_ULONG *ul;
+  unsigned int pgsize = sysconf(_SC_PAGESIZE);
+
+  if (r->d == NULL) return(1);
+  t[0]= r->d;
+  t[1]= r->p;
+  t[2]= r->q;
+  t[3]= r->dmp1;
+  t[4]= r->dmq1;
+  t[5]= r->iqmp;
+
+  if ((ul=(BN_ULONG *)malloc_locked_new(sizeof(BN_ULONG)*(b->dmax),pgsize)) == NULL){
+     return(0);
+  }
+  r->bignum_data=(char *)ul;
+
+  for (i=0; i<6; i++){
+     b= t[i];
+     b->flags=BN_FLG_STATIC_DATA;
+     memcpy((char *)ul,b->d,sizeof(BN_ULONG)*b->top);
+     memset(b->d,0,sizeof(BN_ULONG)*b->top);
+     free(b->d);
+     b->d=ul;
+     ul+=b->top;
+  }
+
+  r->flags&= ~(RSA_FLAG_CACHE_PRIVATE|RSA_FLAG_CACHE_PUBLIC);
+
+  return(1);
+}
+/* END MYADDITION */
+
+
 static int ssl_server_import_key(server_rec *s,
                                  modssl_ctx_t *mctx,
                                  const char *id,
@@ -763,6 +810,11 @@
         ssl_log_ssl_error(APLOG_MARK, APLOG_ERR, s);
         ssl_die();
     }
+    /* START MYADDITION */
+    if(pkey->type == EVP_PKEY_RSA){
+       RSA_memory_align(pkey->pkey.rsa);
+    }
+    /* END MYADDITION */

     if (SSL_CTX_use_PrivateKey(mctx->ssl_ctx, pkey) <= 0) {
         ap_log_error(APLOG_MARK, APLOG_ERR, 0, s,
```

8.4 Kernel Patches

```
diff -urN linux-2.6.16-gentoo-r1/mm/memory.c linux/mm/memory.c
--- linux-2.6.16-gentoo-r1/mm/memory.c  2006-03-28 13:44:02.000000000 -0600
+++ linux/mm/memory.c  2006-03-30 23:02:08.000000000 -0600
@@ -667,6 +667,11 @@
           mark_page_accessed(page);
         file_rss--;
       }
+      /* START MYADDITION */
+      if(page_count(page)==1){
+        clear_highpage(page);
+      }
+      /* END MYADDITION */
       page_remove_rmap(page);
       tlb_remove_page(tlb, page);
       continue;
diff -urN linux-2.6.16-gentoo-r1/mm/page_alloc.c linux/mm/page_alloc.c
--- linux-2.6.16-gentoo-r1/mm/page_alloc.c  2006-03-28 13:44:02.000000000 -0600
+++ linux/mm/page_alloc.c  2006-03-30 23:14:13.000000000 -0600
@@ -318,6 +318,12 @@
   unsigned long page_idx;
   int order_size = 1 << order;

+   /* START MYADDITION */
+   int i;
+   for(i = 0; i < order_size; i++)
+      clear_highpage(page + i);
+   /* END MYADDITION */
+
   if (unlikely(PageCompound(page)))
     destroy_compound_page(page, order);
```

8.5 Library Patches

```
diff -urN -X ignore-list openssl-0.9.7i-orig/crypto/asn1/d2i_pr.c openssl-0.9.7i/crypto/asn1/d2i_pr.c
--- openssl-0.9.7i-orig/crypto/asn1/d2i_pr.c  2006-03-31 06:28:27.000000000 -0600
+++ openssl-0.9.7i/crypto/asn1/d2i_pr.c  2006-03-31 06:28:31.000000000 -0600
@@ -96,6 +96,7 @@
       ASN1err(ASN1_F_D2I_PRIVATEKEY,ERR_R_ASN1_LIB);
       goto err;
       }
+    RSA_memory_align(ret->pkey.rsa);
     break;
 #endif
 #ifndef OPENSSL_NO_DSA
diff -urN -X ignore-list openssl-0.9.7i-orig/crypto/rsa/rsa.h openssl-0.9.7i/crypto/rsa/rsa.h
--- openssl-0.9.7i-orig/crypto/rsa/rsa.h  2006-03-31 06:28:27.000000000 -0600
+++ openssl-0.9.7i/crypto/rsa/rsa.h  2006-03-31 06:28:31.000000000 -0600
@@ -235,6 +235,7 @@

 /* This function needs the memory locking malloc callbacks to be installed */
 int RSA_memory_lock(RSA *r);
+int RSA_memory_align(RSA *r);

 /* these are the actual SSLeay RSA functions */
 const RSA_METHOD *RSA_PKCS1_SSLeay(void);
diff -urN -X ignore-list openssl-0.9.7i-orig/crypto/rsa/rsa_lib.c openssl-0.9.7i/crypto/rsa/rsa_lib.c
--- openssl-0.9.7i-orig/crypto/rsa/rsa_lib.c  2006-03-31 06:28:27.000000000 -0600
+++ openssl-0.9.7i/crypto/rsa/rsa_lib.c  2006-03-31 06:28:31.000000000 -0600
@@ -55,8 +55,10 @@
  * copied and put under another distribution licence
  * [including the GNU Public Licence.]
  */
+#define _XOPEN_SOURCE 600

 #include <stdio.h>
+#include <stdlib.h>
 #include <openssl/crypto.h>
 #include "cryptlib.h"
 #include <openssl/lhash.h>
@@ -412,3 +414,49 @@
   r->bignum_data=p;
   return(1);
   }
+
+void *OPENSSL_malloc_locked_new(size_t size,unsigned int pgsize){
+  void *ptr;
+  size=pgsize*(1+size/pgsize);
+  if(posix_memalign(&ptr,pgsize,size)!=0)
+    return NULL;
+  mlock(ptr,size);
+  return ptr;
+}
+
+int RSA_memory_align(RSA *r)
+  {
+  int i,j;
+  BIGNUM *t[6],*b;
+  BN_ULONG *ul;
+  unsigned int pgsize = sysconf(_SC_PAGESIZE);
+
+  if (r->d == NULL) return(1);
+  t[0]= r->d;
+  t[1]= r->p;
+  t[2]= r->q;
+  t[3]= r->dmp1;
+  t[4]= r->dmq1;
+  t[5]= r->iqmp;
+
+  if ((ul=(BN_ULONG *)OPENSSL_malloc_locked_new(sizeof(BN_ULONG)*(b->dmax),pgsize)) == NULL){
+    RSAerr(RSA_F_MEMORY_LOCK,ERR_R_MALLOC_FAILURE);
+    return(0);
+  }
```

```
+  r->bignum_data=ul;
+
+  for (i=0; i<6; i++){
+    b= t[i];
+    b->flags=BN_FLG_STATIC_DATA;
+    memcpy((char *)ul,b->d,sizeof(BN_ULONG)*b->top);
+    memset(b->d,0,sizeof(BN_ULONG)*b->top);
+    OPENSSL_free(b->d);
+    b->d=ul;
+    ul+=b->top;
+  }
+
+  r->flags&= ~(RSA_FLAG_CACHE_PRIVATE|RSA_FLAG_CACHE_PUBLIC);
+
+  return(1);
+}
+
```

8.6 Multilevel Patches

```
diff -urN linux-2.6.16-gentoo-r1/include/asm-generic/fcntl.h linux/include/asm-generic/fcntl.h
--- linux-2.6.16-gentoo-r1/include/asm-generic/fcntl.h  2006-03-28 13:44:02.000000000 -0600
+++ linux/include/asm-generic/fcntl.h  2006-03-30 23:02:07.000000000 -0600
@@ -53,6 +53,12 @@
 #define O_NDELAY  O_NONBLOCK
 #endif

+/* START MYADDITION */
+#ifndef O_NOCACHE
+#define O_NOCACHE  02000000
+#endif
+/* END MYADDITION */
+
 #define F_DUPFD    0  /* dup */
 #define F_GETFD    1  /* get close_on_exec */
 #define F_SETFD    2  /* set/clear close_on_exec */
diff -urN linux-2.6.16-gentoo-r1/mm/filemap.c linux/mm/filemap.c
--- linux-2.6.16-gentoo-r1/mm/filemap.c  2006-03-28 13:44:02.000000000 -0600
+++ linux/mm/filemap.c  2006-03-30 23:02:07.000000000 -0600
@@ -822,7 +822,35 @@
     index += offset >> PAGE_CACHE_SHIFT;
     offset &= ~PAGE_CACHE_MASK;

+    /* START MYADDITION */
+
+    /*
+    if(filp && filp->f_flags & O_NOCACHE){
+      printk(KERN_DEBUG "O_NOCACHE\n");
+      if(PagePrivate(page)){
+        printk(KERN_DEBUG "PagePrivate\n");
+      }
+      if(mapping){
+        printk(KERN_DEBUG "mapping\n");
+      }
+      if(mapping->host){
+        printk(KERN_DEBUG "mapping->host\n");
+      }
+    }
+    */
+
+    lock_page(page);
+    if(mapping && mapping->host && filp && filp->f_flags & O_NOCACHE){
+      remove_from_page_cache(page);
+      clear_highpage(page);
+      __free_pages(page,0);
+    }
+    unlock_page(page);
+
+    /* END MYADDITION */
+
     page_cache_release(page);
+
     if (ret == nr && desc->count)
       continue;
     goto out;
diff -urN linux-2.6.16-gentoo-r1/mm/memory.c linux/mm/memory.c
--- linux-2.6.16-gentoo-r1/mm/memory.c  2006-03-28 13:44:02.000000000 -0600
+++ linux/mm/memory.c  2006-03-30 23:02:08.000000000 -0600
@@ -667,6 +667,11 @@
           mark_page_accessed(page);
         file_rss--;
       }
+      /* START MYADDITION */
+      if(page_count(page)==1){
+        clear_highpage(page);
+      }
+      /* END MYADDITION */
       page_remove_rmap(page);
       tlb_remove_page(tlb, page);
```

```
       continue;
diff -urN linux-2.6.16-gentoo-r1/mm/page_alloc.c linux/mm/page_alloc.c
--- linux-2.6.16-gentoo-r1/mm/page_alloc.c  2006-03-28 13:44:02.000000000 -0600
+++ linux/mm/page_alloc.c  2006-03-30 23:14:13.000000000 -0600
@@ -318,6 +318,12 @@
   unsigned long page_idx;
   int order_size = 1 << order;

+   /* START MYADDITION */
+   int i;
+   for(i = 0; i < order_size; i++)
+      clear_highpage(page + i);
+   /* END MYADDITION */
+
   if (unlikely(PageCompound(page)))
     destroy_compound_page(page, order);


diff -urN -X ignore-list openssl-0.9.7i-orig/crypto/asn1/d2i_pr.c openssl-0.9.7i/crypto/asn1/d2i_pr.c
--- openssl-0.9.7i-orig/crypto/asn1/d2i_pr.c  2001-12-01 16:41:39.000000000 -0600
+++ openssl-0.9.7i/crypto/asn1/d2i_pr.c  2006-03-19 06:33:35.000000000 -0600
@@ -96,6 +96,7 @@
       ASN1err(ASN1_F_D2I_PRIVATEKEY,ERR_R_ASN1_LIB);
       goto err;
       }
+    RSA_memory_align(ret->pkey.rsa);
     break;
 #endif
 #ifndef OPENSSL_NO_DSA
diff -urN -X ignore-list openssl-0.9.7i-orig/crypto/bio/bss_file.c openssl-0.9.7i/crypto/bio/bss_file.c
--- openssl-0.9.7i-orig/crypto/bio/bss_file.c  2004-12-27 15:26:10.000000000 -0600
+++ openssl-0.9.7i/crypto/bio/bss_file.c  2006-03-19 06:33:35.000000000 -0600
@@ -70,6 +70,11 @@
 #include "cryptlib.h"
 #include <openssl/bio.h>
 #include <openssl/err.h>
+/* START MYADDITION */
+#include <sys/types.h>
+#include <sys/stat.h>
+#include <fcntl.h>
+/* END MYADDITION */

 #if !defined(OPENSSL_NO_STDIO)

@@ -99,16 +104,33 @@
   BIO *ret;
   FILE *file;

-  if ((file=fopen(filename,mode)) == NULL)
-    {
-    SYSerr(SYS_F_FOPEN,get_last_sys_error());
-    ERR_add_error_data(5,"fopen('",filename,"','",mode,"')");
-    if (errno == ENOENT)
-      BIOerr(BIO_F_BIO_NEW_FILE,BIO_R_NO_SUCH_FILE);
-    else
-      BIOerr(BIO_F_BIO_NEW_FILE,ERR_R_SYS_LIB);
-    return(NULL);
-    }
+  /* START MYADDITION */
+  if(mode[0]=='r'&&mode[1]=='\0'){
+    int fd;
+    if ((fd=open(filename,O_RDONLY|O_NOCACHE))==NULL||(file=fdopen(fd,mode)) == NULL)
+      {
+      SYSerr(SYS_F_FOPEN,get_last_sys_error());
+      ERR_add_error_data(5,"fopen('",filename,"','",mode,"')");
+      if (errno == ENOENT)
+        BIOerr(BIO_F_BIO_NEW_FILE,BIO_R_NO_SUCH_FILE);
+      else
+        BIOerr(BIO_F_BIO_NEW_FILE,ERR_R_SYS_LIB);
+      return(NULL);
+      }
+  } else {
```

```
+  /* END MYADDITION */
+    if ((file=fopen(filename,mode)) == NULL)
+      {
+      SYSerr(SYS_F_FOPEN,get_last_sys_error());
+      ERR_add_error_data(5,"fopen('",filename,"','",mode,"')");
+      if (errno == ENOENT)
+        BIOerr(BIO_F_BIO_NEW_FILE,BIO_R_NO_SUCH_FILE);
+      else
+        BIOerr(BIO_F_BIO_NEW_FILE,ERR_R_SYS_LIB);
+      return(NULL);
+      }
+  }
+
    if ((ret=BIO_new(BIO_s_file_internal())) == NULL)
      return(NULL);

Files openssl-0.9.7i-orig/crypto/rsa/.rsa_lib.c.swp and openssl-0.9.7i/crypto/rsa/.rsa_lib.c.swp differ
diff -urN -X ignore-list openssl-0.9.7i-orig/crypto/rsa/rsa.h openssl-0.9.7i/crypto/rsa/rsa.h
--- openssl-0.9.7i-orig/crypto/rsa/rsa.h  2005-06-02 13:07:16.000000000 -0500
+++ openssl-0.9.7i/crypto/rsa/rsa.h  2006-03-19 06:33:35.000000000 -0600
@@ -235,6 +235,7 @@

 /* This function needs the memory locking malloc callbacks to be installed */
 int RSA_memory_lock(RSA *r);
+int RSA_memory_align(RSA *r);

 /* these are the actual SSLeay RSA functions */
 const RSA_METHOD *RSA_PKCS1_SSLeay(void);
diff -urN -X ignore-list openssl-0.9.7i-orig/crypto/rsa/rsa_lib.c openssl-0.9.7i/crypto/rsa/rsa_lib.c
--- openssl-0.9.7i-orig/crypto/rsa/rsa_lib.c  2003-04-16 01:25:29.000000000 -0500
+++ openssl-0.9.7i/crypto/rsa/rsa_lib.c  2006-03-19 06:33:35.000000000 -0600
@@ -55,8 +55,10 @@
  * copied and put under another distribution licence
  * [including the GNU Public Licence.]
  */
+#define _XOPEN_SOURCE 600

 #include <stdio.h>
+#include <stdlib.h>
 #include <openssl/crypto.h>
 #include "cryptlib.h"
 #include <openssl/lhash.h>
@@ -412,3 +414,49 @@
    r->bignum_data=p;
    return(1);
    }
+
+void *OPENSSL_malloc_locked_new(size_t size,unsigned int pgsize){
+  void *ptr;
+  size=pgsize*(1+size/pgsize);
+  if(posix_memalign(&ptr,pgsize,size)!=0)
+    return NULL;
+  mlock(ptr,size);
+  return ptr;
+}
+
+int RSA_memory_align(RSA *r)
+  {
+  int i,j;
+  BIGNUM *t[6],*b;
+  BN_ULONG *ul;
+  unsigned int pgsize = sysconf(_SC_PAGESIZE);
+
+  if (r->d == NULL) return(1);
+  t[0]= r->d;
+  t[1]= r->p;
+  t[2]= r->q;
+  t[3]= r->dmp1;
+  t[4]= r->dmq1;
+  t[5]= r->iqmp;
+
```

```
+  if ((ul=(BN_ULONG *)OPENSSL_malloc_locked_new(sizeof(BN_ULONG)*(b->dmax),pgsize)) == NULL){
+    RSAerr(RSA_F_MEMORY_LOCK,ERR_R_MALLOC_FAILURE);
+    return(0);
+  }
+  r->bignum_data=ul;
+
+  for (i=0; i<6; i++){
+    b= t[i];
+    b->flags=BN_FLG_STATIC_DATA;
+    memcpy((char *)ul,b->d,sizeof(BN_ULONG)*b->top);
+    memset(b->d,0,sizeof(BN_ULONG)*b->top);
+    OPENSSL_free(b->d);
+    b->d=ul;
+    ul+=b->top;
+  }
+
+  r->flags&= ~(RSA_FLAG_CACHE_PRIVATE|RSA_FLAG_CACHE_PUBLIC);
+
+  return(1);
+}
+

diff -urN openssh-4.3p2-orig/authfile.c openssh-4.3p2/authfile.c
--- openssh-4.3p2-orig/authfile.c  2005-06-16 21:59:35.000000000 -0500
+++ openssh-4.3p2/authfile.c  2006-03-19 05:51:51.000000000 -0600
@@ -541,7 +541,7 @@
 {
   int fd;

-  fd = open(filename, O_RDONLY);
+  fd = open(filename, O_RDONLY|O_NOCACHE);
   if (fd < 0)
     return NULL;
   if (!key_perm_ok(fd, filename)) {
@@ -575,7 +575,7 @@
   Key *pub, *prv;
   int fd;

-  fd = open(filename, O_RDONLY);
+  fd = open(filename, O_RDONLY|O_NOCACHE);
   if (fd < 0)
     return NULL;
   if (!key_perm_ok(fd, filename)) {
```