Key Extraction Attack Using Statistical Analysis of Memory Dump Data

Yuto Nakano(✉), Anirban Basu, Shinsaku Kiyomoto, and Yutaka Miyake

KDDI R&D Laboratories Inc., 2-1-15 Ohara, Fujimino, Saitama 356-8502, Japan
{yuto,basu,kiyomoto,miyake}@kddilabs.jp

Abstract. During the execution of a program the keys for encryption algorithms are in the random access memory (RAM) of the machine. Technically, it is easy to extract the keys from a dumped image of the memory. However, not many examples of such key extractions exist, especially during program execution. In this paper, we present a key extraction technique and confirm its effectiveness by implementing the Process Peeping Tool (PPT) – an analysis tool – that can dump the memory during the execution of a target program and help the attacker deduce the encryption keys through statistical analysis of the memory contents. Utilising this tool, we evaluate the security of two sample programs, which are built on top of the well-known OpenSSL library. Our experiments show that we can extract both the private key of the RSA asymmetric cipher as well as the secret key of the AES block cipher.

Keywords: Memory dump · Key extraction · OpenSSL · RSA · AES

1 Introduction

The growth of various services on the Internet has given rise to a dramatic increase in the information that is exchanged over Internet protocols. Sensitive information in private e-mails, confidential documents, e-commerce and other financial transactions need to be guarded against eavesdropping. In order to protect the communication between two network hosts, the Secure Sockets Layer [4] and Transport Layer Security [3] (SSL/TLS) are commonly used. OpenSSL[1] is the one of most commonly used open source libraries for the SSL and TLS protocols. The core library offers implementations for various cryptographic algorithms and other utility functions. Recently, a critical bug, referred to as CVE-2014-0160, has been found in OpenSSL TLS Heartbeat extension [10], which makes for the attacker to recover cryptographic keys by reading 64 kilobytes of memory at a time [1,2].

Any unauthorised access to cryptographic keys constitutes a security breach. Tamper-proof devices and obfuscation cannot be used during program execution. It is well-known that the cryptographic keys are present in the random access

[1] See: https://www.openssl.org/.

© Springer International Publishing Switzerland 2015
J. Lopez et al. (Eds.): CRiSIS 2014, LNCS 8924, pp. 239–246, 2015.
DOI: 10.1007/978-3-319-17127-2_17

memory (RAM) during the execution of a program; a knowledge that an adversary can use to extract the keys from the RAM [7–9]. Protecting the keys or any other valuable information from unauthorised access during program execution is an important area of on-going research. Oblivious RAM schemes and related works [5,6] can protect the RAM access patterns of programs from unauthorised access. However, these schemes require trustworthy and secure CPUs for the protection, and cannot prevent attacks where the attacker can access the CPU and extract information such as operations, access to memory addresses of operations and values stored in those addresses.

In this paper, we present a new attack method that can extract a private key for RSA and a secret key for AES from dumped memory image. We have implemented a tool called the Process Peeping Tool (PPT) to demonstrate the attacks. PPT enables us to analyse the structure and behaviour of the target program by observing its memory use. It can also statistically analyse the data that is acquired from the RAM, thus enabling the attacker to determine cryptographic keys. We use PPT to extract cryptographic keys (both for RSA and for AES) from sample programs, which use the OpenSSL library. In the key recovery attack against RSA, we iterate through cycles of encryption and decryption. Assuming that the private key remains fixed during the execution of the program the key can be extracted by observing the memory accesses of the decryption function. In case of AES, we encrypt a random number with one fixed secret key. We demonstrate that the secret key can be extracted by observing the memory access patterns of the encryption process. The address of any shared libraries has to be public for the PPT to be able to analyse the program. Using a static library and deleting symbol information makes it harder for the attacker to obtain the keys. However, the keys can still be extracted once the attacker determines the functions that need to be observed. Although, adding dummy data and/or dummy operations also make the attack harder, these can be distinguished from the actual data since the dummy data and accesses do not affect the output of the program.

2 Aquiring Data from Memory

In this section, we introduce two methods which can extract data from memory.

2.1 Memory Dump

Memory dump is usually used for debugging programs especially to detect buffer overflows. It also can be used to attack programs. During the execution of the program, its data is temporally stored in RAM and any process with the same privilege as the user executing the program or a root privilege can access that region. If the program is an encryption/decryption program, the decryption key must be stored in RAM and it is possible for the adversary to dump the contents in the RAM and search for the key. On Linux systems, `dd` or `ptrace()` can be used to dump memory. We can also access the memory of a running process

that runs with different privileges than the user, assuming root is not involved, by using Linux capabilities. By giving the user a capability for the particular program, the user will be able to execute it without the normally required privilege. However, setting capabilities to a program/process requires, initially, root privileges or an appropriate capability.

On operating systems which use virtual memory, part of or entire memory contents of a program are sometimes moved from main memory to secondary storage (i.e., the hard disk drive). If the adversary can access the region of a disk where the pages are stored, it is easy to read the content of memory. Another possibility that the memory content can be stored on the disk is core dump. When a process is unexpectedly stopped, the memory image of that process is saved as a core file in order for debugging. The adversary can access the core file and try to analyse the memory image.

2.2 `ptrace` System Call

The `ptrace()` system call enables one process (the "tracer") to observe, control the execution of another process (the "tracee"), examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing. Other than `ptrace()` system call, there are several other system calls that may help the adversary to monitor the process and its access to memory, such as `ltrace()` for monitoring dynamic libraries and `strace()` for system calls.

There is a mode called `PTRACE_SINGLESTEP` in `ptrace`, which can load a program or can be attached to an existing program. The `PTRACE_SINGLESTEP` mode allows the attacker to execute the program step-by-step. We can also acquire the values stored in the CPU registers in each step such as program addresses, register values of operand and values in RAM pointed to by the registers by using a `PTRACE_PEEKDATA` mode.

3 Attack Scenarios and Key Extraction Attack

The private key of RSA and the secret key of AES are assumed to be of fixed values, or at least fixed during the execution of the program. Such a key can be hidden inside the program with some protection. However, when the program is initiated, the key must be loaded in the RAM in plaintext. If the attacker can dump the memory when the key is loaded in the RAM, it is possible to extract the secret key from the dumped image. We assume that the adversary has the access to the target program and memory dump data.

Attack Scenario 1. A service provider provides various services such as hosting, web application and file sharing. The clients connect to the server through secure channels established with SSL/TLS or any equivalent protocols. When the clients connect to the server, the secret keys have to be stored in the RAM. While the connection is active, the malicious operator can attach his attacking process on

the server-side to the target and dump the related area in the memory without being detected by the clients; and thereby extract the secret key. Even when there is no malicious operator, the server may be compromised by malware, which acquires the root privilege and mounts similar attacks on the processes handling secure connections.

Attack Scenario 2. Several users share the same physical server (i.e., public cloud) on which they operate their own separate virtual machines (VMs). However, once the attacker can login as the administrative user, it is possible to attach the attacking process to the victims' VM processes and extract keys from memory dumps.

Attack Scenario 3. Last but not the least is the user-as-the-attacker scenario, where the aim of the user is to extract the secret keys or other valuable information from the target program by attaching the attacking process to the target process.

3.1 Attack Against RSA

The exponentiation of RSA decryption involves variable length arithmetic, it is expected that decryption process uses shift operations. The Chinese Remainder Theorem (CRT), which involves division operations, is often used for exponentiation operations of encryption and decryption. Therefore, shift and division operations deal the private key and we can recover the key by dumping and analysing memory region used by these operations.

Assume that both the public and private keys are fixed while the plaintext and the ciphertext keep changing. Every time the encryption or decryption functions are executed, the program's accesses to the key which is the fixed value, while its accesses to the plaintext or the ciphertext which are keep changing. Therefore, as the number of executions increase, the accesses to keys can be distinguished from other accesses by counting the number of accesses to each value. The procedure can be summarised as: (a) iterate encryption and decryption of random numbers with the fixed key, (b) dump values which are accessed by shift and division operations, and (c) output key candidate values which are accessed considerably more than other values. If the multiplication of two recovered values matches the modulo N, the key can be correctly recovered.

3.2 Attack Against AES

AES consists of four functions, and one of them is called AddRoundKey. The AddRoundKey takes two inputs – the round key and the internal state. Hence, we can recover the round keys and the secret key, by dumping and analysing memory region accessed by the AddRoundKey function. As one of the inputs of AddRoundKey is the fixed key while the other is the variable internal state, the key can be distinguished from the internal state by counting the number of accesses during the iteration of encryption. The procedure can be summarised as:

(a) iterate the encryption of random numbers with the fixed key (b) dump values which are accessed by AddRoundKey operations, and (c) output key candidate values which are accessed considerably more than other values.

We can also use Maartmann-Moe et al.'s idea [8], which uses the character of the key expansion, to confirm if the key is correctly recovered. As the round keys are derived from the secret key, we can apply the key expansion to the recovered candidate values and see if derived values appear on RAM.

4 Process Peeping Tool (PPT)

The core component of the PPT is the `ptrace()` system call, which enables one process (the "tracer") to observe, control the execution of another process (the "tracee"), examine and change the tracee's memory and registers. PPT can analyse the structure of the target program, including which shared libraries it uses and which functions are used in each library. It can also analyse memory addresses that the target process accesses and values that the target process uses. These addresses and values are recorded and statistically analysed. Dumped data can be used efficiently for the key extraction attack as, unlike existing memory dump tools, we can specify libraries and functions of interest.

One can attach the PPT process to the target program by specifying the target's process ID (PID). Once successfully attached to the target PID, the evaluator can browse inside the target as if the target and the analysing tool were respectively a file system directory structure and the shell. In the next step, the child process is executed with `PTRACE_TRACEME`. When the child process executes a function, the parent process receives `SIGTRAP` and pauses the child process. The parent process replaces, keeping a copy of the original values, the function's addresses with breakpoints in Procedure Linkage Table (PLT). Then the original operation is restored to execute a single step of the child process with `PTRACE_SINGLESTEP`, and the parent process obtains information on which libraries the child process accessed. After the single step, the parent process again takes over the control and continues the operation until it encounters the next breakpoint. The evaluator, therefore, can find out which libraries are used and which functions inside these libraries are called.

The evaluator can control how each function can be executed by setting its status. The available statuses are: **watch** – execute the function step-by-step recording its data; **watchdeeply** – in addition to *watch*, this status enables recording the behaviour of other functions called inside the target function; **through** – execute the function step-by-step without recording data; and **skip** – execute the function as usual.

When the function is under surveillance with "watch" status, the function is executed with a `PTRACE_SINGLESTEP`. When the single step of the child process is executed, the addresses and the values from the child task can be read by `PTRACE_PEEKDATA`. The addresses and values are recorded by PPT and used for static analysis. Any function with a "watch" status is skipped if that function is called by one with a "skip" status. In order for the "watch" status to work with a function, it should be ensured that its caller function has a status set to "through".

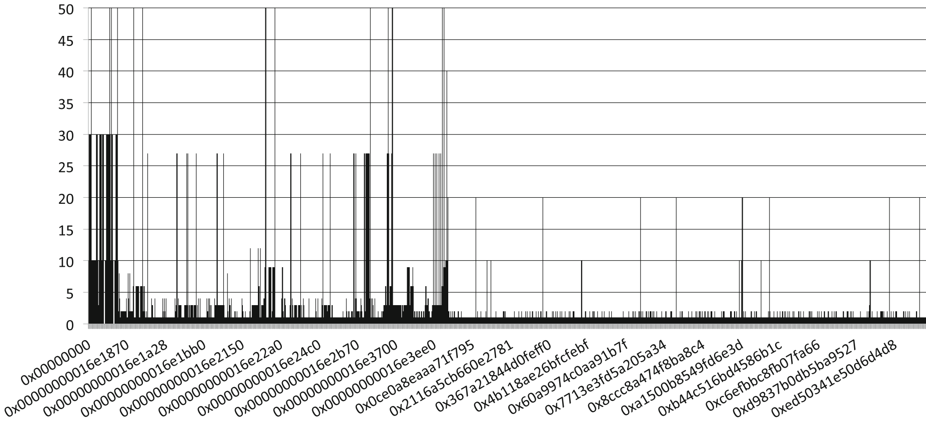

Fig. 1. Values referred from RAM when we iterate RSA encryption and decryption ten times

5 Key Extraction from Memory Dump

In this section we demonstrate how the Process Peeping Tool can help extract the private key and the secret key of RSA and AES respectively from two separate target sample programs. The attack procedure is summarised as follows:

1. analyse the structure of the target program including libraries and functions,
2. specify which library or function to monitor,
3. execute the target program while specified libraries and functions are executed step-by-step,
4. record addresses and values,
5. recover the key by statically analysing data acquired in step 4.

The experimental environment is following; CPU: Intel Core i7 4930 K, RAM: 24 GB, OS: Ubuntu 13.10 64-bit, Library: OpenSSL 0.9.8.

In case of RSA, PPT retrieves the private key and a lot of random numbers from RAM. We perform statistical analysis to distinguish the key from the random numbers. On the other hand, the key extraction of AES is simple and we do not need any additional analysis to separate the secret key from other values. Maartmann-Moe et al.'s attack [8] uses the facts that round keys are derived from the initial key and the round keys are stored on RAM right after the initial key. Therefore, their attack cannot be applied when the initial key and round keys are stored on the separate locations on RAM. On the other hand, our attack can recover the key even when the initial key and round keys are stored on the separate locations as we observe the values, which are accessed by the encryption functions.

Table 1. The watch list of functions for RSA decryption and AES encryption

Library	Function	Status
RSA		
`rsao0s_so`	`RSA_private_decrypt`	through
`libcrypto.so.0.9.8`	`BN_div`	watch
`libcrypto.so.0.9.8`	`BN_lshift`	watch
`libcrypto.so.0.9.8`	`BN_rshift`	watch
AES		
`aesopenssl`	`AES_encrypt`	watchdeeply

5.1 RSA

We implemented a simple RSA encryption and decryption program using the OpenSSL library, which repeatedly encrypts random numbers and decrypts the generated ciphertexts. Both keys remain unchanged during the experiment. Table 1 summarises the list of the methods to be observed.

We initiate the sample program and start the encryption and decryption operations. Then, we initiate PPT and attach its process to the sample program. PPT can show the structure of the program, when it is successfully attached to the target. By executing the program while watching the specified functions in Table 1, we record the values and their frequencies in which they are referred to in the RAM. Figure 1 shows relations between the values and their frequencies. The x-axis shows the values and the y-axis shows the number of referred times. As it is unlikely that the private key is a sparse value, we can eliminate the sparse candidates, for instance `0x0000000000000001`. These sparse values are mostly used for controlling the operations such as counters.

For the remaining candidates, we use number of referred times as a clue. In this example, we iterate encryption and decryption 10 times. Thus, the private key has to be used at least 10 times. Even when we do not know how many times encryption and decryption is iterated, the secret key can still be distinguished from other random numbers after sufficient number of iterations.

5.2 AES

We also implemented a simple AES encryption program, named `aesopenssl`, using the OpenSSL library. This sample program continuously encrypts random numbers with a fixed secret key. Table 1 shows the function to be observed.

We execute the program while observing the `aesopenssl` function, and apply the method similar to what we did for RSA to eliminate the non-key values. The secret key we used for the sample program is "THISISSECRETKEY!", which is `0x54, 0x48, 0x49, 0x53, 0x49, 0x53, 0x53, 0x45, 0x43, 0x52, 0x45, 0x54, 0x4b, 0x45, 0x59, 0x21` in ASCII. PPT recovered all these values, and it also recovered `0x0000000a`, which is the number of rounds in AES-128, followed by the round key.

6 Conclusion

In this paper, we introduced a statistical key extraction attack on cryptographic keys using memory dump data, and confirmed the effectiveness of the attack by utilising our Process Peeping Tool. The tool can be attached to the target process and can trace the target's memory usage. We used RSA and AES as example cryptosystems in the target programs, which utilised the OpenSSL library implementations. Thus, it is possible to apply the same approach to other applications using OpenSSL library or similar cryptographic libraries. Although we only applied PPT to RSA and AES implemented in the OpenSSL library, it is possible to apply the same extraction mechanism to other, including non-cryptographic, algorithms or libraries. Although we execute the PPT with the root privilege, we can still apply our method obtaining memory dump data without the root privilege.

References

1. arstechnica.: Critical crypto bug in OpenSSL opens two-thirds of the web to eavesdropping (2014). http://goo.gl/JUm3dq
2. Codenomicon Ltd.: The heartbleed bug (2014). http://heartbleed.com
3. Dierks, T., Rescorla, E.: The transport layer security (TLS) protocol version 1.2. RFC 5246 (2008)
4. Freier, A., Karlton, P., Kocher, P.: The secure sockets layer (SSL) protocol version 3.0. RFC 6101 (2011)
5. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: Aho, A.V. (ed.) STOC, pp. 182–194. ACM (1987)
6. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996)
7. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5), 91–98 (2009)
8. Maartmann-Moe, C., Thorkildsen, S.E., Årnes, A.: The persistence of memory: forensic identification and extraction of cryptographic keys. Digit. Investig. **6**, S132–S140 (2009)
9. Müller, T., Spreitzenbarth, M.: FROST - forensic recovery of scrambled telephones. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 373–388. Springer, Heidelberg (2013)
10. Seggelmann, R., Tuexen, M., Williams, M.: Transport layer security (TLS) and datagram transport layer security (DTLS) heartbeat extension. RFC6520 (2012)