JYX JYVÄSKYLÄN YLIOPISTO UNIVERSITY OF JYVÄSKYLÄ

This is a self-archived version of an original article. This version may differ from the original in pagination and typographic details.

Author(s): Kiperberg, Michael; Zaidenberg, Nezer

Title: Efficient remote authentication

Year: 2013

Version: Published version

Copyright: © Peregrine Technical Solutions, LLC, 2013

Rights: In Copyright

Rights url: http://rightsstatements.org/page/InC/1.0/?language=en

Please cite the original version:

Kiperberg, M., & Zaidenberg, N. (2013). Efficient remote authentication. Journal of Information Warfare, 12(3), 49-55. https://www.jinfowar.com/journal/volume-12-issue-3/efficient-remote-authentication

Efficient Remote Authentication

M Kiperberg, N Zaidenberg

Faculty of Information Technology
University of Jyväskylä, Finland,
E-mail: michael@trulyprotect.com; nezer@trulyprotect.com

Abstract: *In 2003, Kennel and Jamieson described a method of remote machine authentication. By authentication, the authors meant that the remote machine is non-virtual, and the operating system on the remote machine is not malicious. The described method does not consider the variety of versions of each operating system. The description completely ignores the existence of modules that can be plugged into the operating system. The authors of this paper adapt the method described by Kennel and Jamieson to the real world so that it can be applied without prior knowledge of the operating system or the modules on the remote machine.*

Keywords: *Virtual Machine, Digital Rights Management, Remote Authentication, Database*

Introduction

In 2011, Averbuch, Kiperberg, and Zaidenberg showed how a content distributor can verify that a remote machine is non-virtual and that the operating system is authentic. This verification procedure allows the distributor to execute her game on the remote machine while keeping the game's code hidden. The verification procedure consists of two interleaved processes.

The first process, the goal of which is to verify that the operating system is authentic, computes a checksum of the memory locations that hold the code of the operating system. The idea of computing and sending the checksum of some memory region is not new. For example, the AOL Instant Messenger was sending to a server a hash of a requested memory region, and the client was blocked if this hash was not the expected one (AOL 2012; PyxisSystemsTechnologies 2002).

The second process, the goal of which is to verify that the remote machine is non-virtual, is executed in parallel with the first process. The second process combines the checksum computed so far with a value of some performance counter. The last idea is based on the assumption that performance counters cannot be easily simulated in a timely manner (Bedichek, R 1990; Magnusson, SP & Werner, B 1994; Witchel, E & Rosenblum, M 1996).

Therefore, if the remote machine is a virtual machine, the result produced by the verification procedure will be either erroneous or computed after a long period of time. The distributor rejects all the erroneous and the late results, thus guaranteeing the two properties of the remote machine: the machine is non-virtual and the operating system is authentic.

The authors of this paper concentrate on the second property, that is, the authenticity of an operating system. The phrase 'authenticity of an operating system' denotes that the code that is executed in kernel mode is trusted by the distributor. The reason authenticity is important lies in the way the game is executed on the remote machine. In order to be able to execute the

ISSN 1445-3312 print/ISSN 1445-3347 online

game on a remote machine without revealing the game's code, the distributor supplies the game in an encrypted form. The decryption key is transferred to the remote machine that passes the verification procedure. The transferred key is stored in one of the CPU registers that can be accessed only in kernel mode. That is why it is so important to be sure that no malicious code gets executed in kernel mode. **Figure 1** depicts the internal structure of a CPU memory.

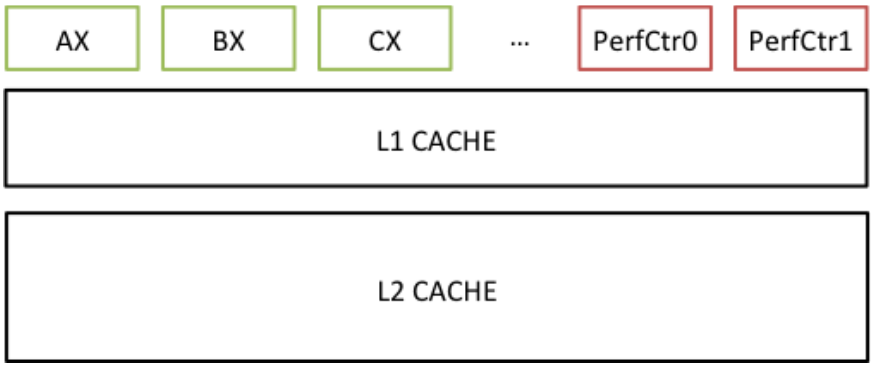

Figure 9: CPU Internal Memory. The registers AX, BX, and CX can be accessed in non-kernel mode. The registers PerfCtr0 and PerfCtr1 can be accessed only in kernel mode. The L1 and L2 caches' access policy is determined by access policy of the underlying memory.

It should be noted that not only the code of the operating system is executed in kernel mode, but also the code of device drivers and other kernel modules is executed in kernel mode. **Figure 2** depicts the layout of the process virtual memory on 64-bit machine running Linux. The kernel and kernel modules are located in the 'Kernel Space' segment depicted in **Figure 2**. However, the exact location in memory of each kernel module or even its presence can differ from machine to machine, thus making it difficult for the distributor to reproduce the result of a verification procedure.

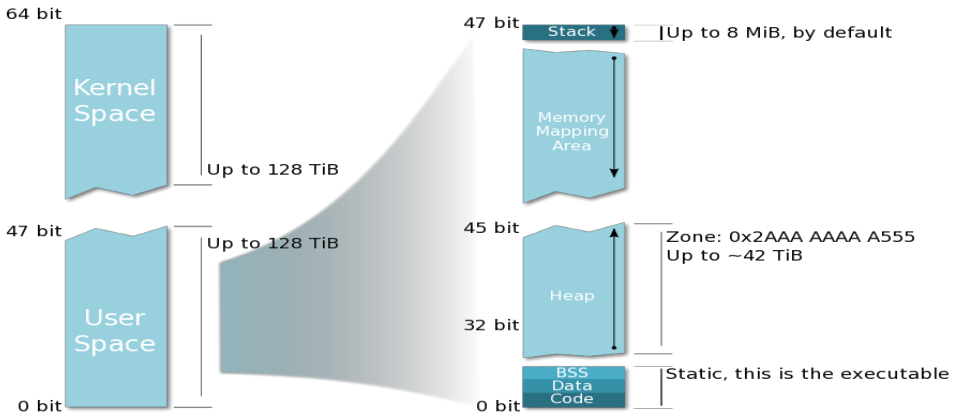

Figure 10: Memory Layout of a Process on 64-Bit Machine Running Linux.

This paper attempts to overcome the above difficulties by pre-calculating the results. The paper is organized as follows. The next section gives a detailed description of the problems that are then solved in the section following it. The last section summarizes the paper and outlines future work.

Meaning

Since CPUs today cannot execute program in an encrypted form as was proposed by Best (1980), the instruction of any encrypted program should be decrypted prior to its actual execution in a CPU. Therefore, the decryption key should be held in the remote machine that executes the program. Since the components on the remote machine cannot be trusted, in the sense that a malicious user can analyze externally the content of any component, it seems to be impossible to allow program execution on a remote machine without revealing the code of this program. That is why the trust policy is refined to include a single component – the CPU.

The approach to executing a program without revealing its code is as follows. A user wishes to play some game. The user asks the distributor to send him this game. The distributor requests the user to validate his machine; the user's machine will be designated the 'remote machine'. The distributor generates a test, which is a code to be executed on the remote machine. This code computes a checksum of some memory region. The distributor sends the test to the remote machine and waits for the result. If the result is correct, the distributor encrypts the game using some random key and sends the key and the game to the remote machine. The remote machine stores the key in a CPU register that can be accessed only in kernel mode (see **Figure 1**). This key is used to decrypt the game function-by-function on-demand. Since only small portions of the game are decrypted at any time, there is a small probability that they will be evicted from the CPU cache.

The only problem with this approach is that any code running in kernel mode has access to the register that contains the decryption key. This code can read the content of the register and print it to the screen, which will be enough to decrypt the entire program. That is why a guarantee that no such code is executed in kernel mode is necessary. On Linux, there are only two types of code that are executed in kernel mode. The first type is the kernel itself, and the second type is the code of kernel modules that are plugged into the kernel.

In order to understand the difficulties that arise from the variety of kernels, modules and their combinations, we should explain first, how the distributor verifies the result of the test that is sent to the remote machine. The distributor verifies the correctness of the result by executing exactly the same test on some of its local machines. Then, it compares the result on the local machine with the result on the remote machine. The test is qualified as correct if and only if the results are equal. **Figure 3** (below) depicts the verification procedure. Obviously the results will differ if the two machines have different CPUs or contain different content in the memory region under test (see **Figure 2**). Note, that the latter does not imply that the kernel or the device drivers are malicious; the memory may, as well, contain exactly the same drivers that for some reason were loaded in a different order and thus reside in different locations of the memory.

Figure 11: Naïve Verification Procedure. The remote machine (on the right) requests validation. The distributor (in the middle) generates a test and runs it on a local machine (on

the left). The local machine returns the result to the distributor. The distributor sends the same test to the remote machine. The remote machine runs the test and returns the result to the distributor. The distributor compares the results received from the local and remote machines. If the results are equal the validation succeeded.

Solution

Kennel and Jamieson (2003) noted that the verification protocol should be CPU aware. To accommodate this, their protocol starts by sending a message containing the CPU information from the remote machine to the distributor. The authors of this paper propose to include in this message additional info about the version of the kernel, the list of kernel modules, their layout in memory, etc.

Upon receiving this information, the distributor can set up the same arrangement on one of her local machines and afterwards run the test. The problem with the above approach is that it is very time and resource consuming. The setup can take tens of minutes. For each such setup the distributor has to allocate a machine. This machine cannot be used for other tests. Therefore, in the real world, it would not be feasible to perform such a setup.

In order to solve this problem, tests may be prepared for many different configurations of CPU model, kernel version and kernel modules layout. Each such test and the corresponding result will be stored in the 'test database'. This database will be constantly augmented by new tests. Upon request from a user, the distributor will pick one of the tests that suits this user and send the test to the user. **Figure 4** shows the proposed verification procedure.

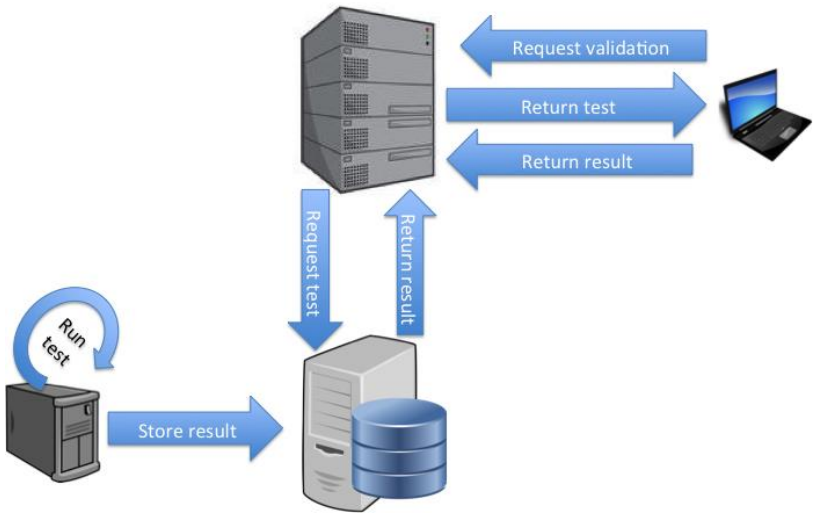

Figure 12: New Verification Procedure. The local machine (on the left) generates and stores test results for various tests in the database (in the middle-bottom). The remote machine (on the right) requests validation. The distributor (in the middle-top) fetches some test result from the database and sends the corresponding test to the remote machine. The remote machine runs the test and returns the result to the distributor. The distributor compares the result of the remote machine to the result fetched from the database. If the results are equal, the validation has succeeded.

It is possible, however, that no tests are available for the configuration of a particular user. There are many reasons for this. It is possible that the user uses a new version of the kernel that was not known to the distributor. It is possible that one of the kernel modules was not known to the distributor. It is also possible that all the components were known to the

distributor; however, it is the first time the distributor encounters the combination of these components together.

The first two cases are relatively rare. Although it is time consuming to augment the database with a new version of kernel, new distributions of Linux are released only once or twice a year. A large percentage of kernel modules are actually device drivers. In spite of the fact that there are many device drivers, an average user uses only a few of them. **Figure 5** demonstrates that there are less than a thousand device drivers. Most of them are present in all versions of the kernel. Therefore, that drivers are not released very often. The remote machine will send every new kernel component (either the kernel itself or one of its modules) to the distributor for investigation. The distributor will assure that the component is not malicious and create a test for the configuration of the remote machine.

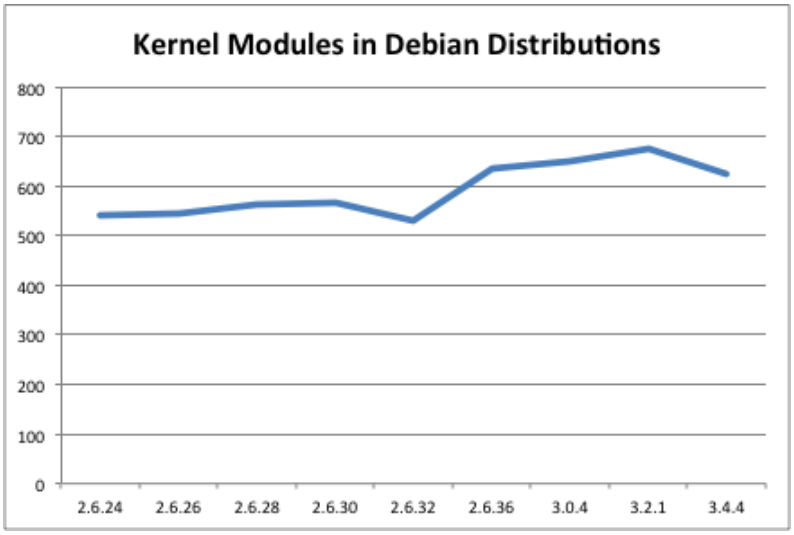

Figure 13: Kernel Modules in Debian Distributions. The X-axis represents the various versions of the Linux kernel that were used in the Debian distributions. The Y-axis represents the number of kernel modules in the directory /lib/modules/A.B.C/kernel/drivers/ where "A.B.C" is the kernel version.

The third case is much simpler since it does not involve a, possibly manual, verification of a kernel component. The test for the new configuration can be generated automatically.
During the construction of tests and results for new configurations, the user having one of these configurations will not be able to start the verification procedure. In this case, the user will be informed of the situation and asked to try again later.

Procedure Entities Role

This section gives a detailed description of the verification procedure. The description covers all the entities participating in the procedure (see **Figure 4**).
The procedure begins when the remote machine requests to perform verification. The request message, which the remote machine sends to the distributor, indicates the configuration of its system. The configuration consists of the following information:

- CPU model,
- Kernel version and location in memory,
- List of kernel modules, their version, and locations in memory.

The verification procedure supports only relatively new CPU models, since newer models can efficiently simulate the older ones. Therefore, after receiving the configuration (C), the

verification procedure fails immediately if the CPU model is not supported. Otherwise, if the configuration was not known previously, the distributor adds it to the list of known configurations and fails the verification procedure. If the configuration was known previously, the distributor fetches the set (S) of tests and results (R) for C. Then a pair of a test and its result (T, R) is picked uniformly at random from S. The distributor sends T to the remote machine and sets a timer. The remote machine runs the test (T) and returns the computed result (R') to the distributor. When the distributor receives the result (R'), it checks whether R' was received within an allowable interval of time. If not, the verification procedure fails. Otherwise, R and R' are compared, and the remote machine is informed accordingly.

Each local machine produces new tests in an infinite loop as follows. It scans the list of all known configurations in the database and fetches only those configurations that correspond to the CPU of the local machine. It then chooses a configuration with the smallest number of tests and produces a test for it. Note that there is at least one local machine for every supported CPU model.

Conclusion

This paper describes a feasible method to verify that a remote machine is non-virtual and its operating system is authentic. The described method allows accommodating for bursts of verification requests, thus making the entire system more scalable. The method breaks the dependency between the test-generation procedure and the remote-machine verification procedure. The latter makes it possible to share the generated tests between multiple verification procedures, thus allowing even greater scalability.

The authors' future work will concentrate on porting the ideas of this paper to other operating systems, specifically Windows and Mac OS X, and on attempting to automate the verification of kernel components.

References

AOL 2002 *The America Online Instant Messenger Application*, viewed 26 September 2013, <http://www.aol.com/>.

Averbuch, A, Kiperberg, M & Zaidenberg N 2011, 'An efficient VM-based software protection', *Network and System Security (NSS) 2011: Proceeding of the 5th International Conference*, Milan, Italy, pp. 121-28.

Bedichek, R 1990, 'Some efficient architecture simulation techniques', *Proceedings of the Winter 1990 USENIX Conference,* Washington, D.C., pp. 53-63.

Best, MR 1980, 'Preventing software piracy with crypto-microprocessors', *Proceedings of IEEE Spring COMPCON 80*, San Francisco, CA, pp. 466-69.

Kennel, R and Jamieson, H 2003, 'Establishing the genuinity of remote computer systems', *Proceedings of the 12th Annual USENIX Security Symposium*, Washington, D.C., pp. 295-308.

Magnusson, PS and Werner, B 1994, 'Some efficient techniques for simulating memory', *Technical Report R94-16,* Swedish Institute of Computer Science, Kista, Sweden.

PyxisSystemsTechnologies 2002, *AIM/oscar protocol specification: Section 3: Connection management*, viewed 26 September 2013, <http://aimdoc.sourceforge.net/faim/protocol/section3.html>.

Witchel, E and Rosenblum, E 1996, 'Embra: Fast and flexible machine simulation', *Sigmetrics: Proceedings of the 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, pp.68-79.