Runtime Integrity Measurement and Enforcement with Automated Whitelist Generation

Anna Kornfeld Simpson
University of Washington
aksimpso@cs.washington.edu

Nabil Schear
MIT Lincoln Laboratory
nabil@ll.mit.edu

Tom Moyer
MIT Lincoln Laboratory
thomas.moyer@ll.mit.edu

This poster discusses a strategy for automatic whitelist generation and enforcement using techniques from information flow control and trusted computing. During a measurement phase, a cloud provider uses dynamic taint tracking to generate a whitelist of executed code and associated file hashes generated by an integrity measurement system. Then, at runtime, it can again use dynamic taint tracking to enforce execution only of code from files whose names and integrity measurement hashes exactly match the whitelist, preventing adversaries from exploiting buffer overflows or running their own code on the system. This provides the capability for runtime integrity enforcement or attestation. Our prototype system, built on top of Intel's PIN emulation environment and the libdft taint tracking system, demonstrates high accuracy in tracking the sources of instructions.

1. INTRODUCTION

Trusted computing techniques, such as integrity measurement attestation via a Trusted Platform Module (TPM) [17], can provide guarantees of a known good state when a program is loaded. Sailer et. al. proposed the Integrity Measurement Architecture (IMA) to allow measurement and attestation for all programs executed on the machine by measuring (hashing) each program as it is executed into the TPM, then sending a challenger both the TPM contents and the list of programs executed as an attestation [15]. IMA is part of the Linux kernel, and measurement of all binary files to be executed (both kernel modules and user applications) can be enabled with a kernel command line argument (these measurements can be done upon boot or file system mounting, optimizing performance at runtime) [6]. Therefore, IMA provides the ability for a machine to attest to the *load-time integrity* of its system and running applications.

However, load-time integrity has no view into code once it begins running, so it cannot detect any runtime modifications. Many items executed in a cloud system run for long enough for an adversary to be able to compromise them, so *runtime integrity measurement* is necessary to verify that the program is still in a good state. Work that built upon the ideas of IMA has attempted to achieve some runtime measurements, such as by imaging the runtime memory from a coprocessor [13], or from the hypervisor [11], using contextual inspection to monitor dynamic data structures as well as static code [12], dynamic rewriting techniques to defend against return-oriented programming [4], or by combining integrity measurement with *information flow integrity* via some kind of policy framework [8]. However, all of these techniques are limited and do not provide full runtime integrity.

Dynamic Taint Analysis (DTA) or dynamic taint tracking can address some of the runtime challenges by determining the origin (files on disk, network data, or other input) of executed instructions at the cost of performance at runtime. The performance cost is because taint tracking must be done from within some kind of emulation environment such as QEMU [1] or Intel's PIN [7] and each instruction must be rewritten into several in order to propagate the taint correctly. Several techniques have optimized the emulation process, including reducing switches between the programs and the DTA code [14], direct memory mapping [3], increasing the parallelization of analysis and execution [9], and on-demand emulation [5]. Finally, native hardware support for taint tracking can significantly increase the performance since instructions will not have to be re-written and multiplied [16] [2].

1.1 Threat Model

Our threat model assumes an adversary who is able to compromise some application on a cloud system, via exploitation of some vulnerability such as a buffer overflow attack, which allows the adversary to attempt to run some arbitrary code on the system. We want to prevent our adversary from executing this code. As a defense, we assume that binaries to can be measured in a trusted environment to create a whitelist prior to their installation on the system. Additionally, we assume the presence of a TPM and integrity measurement system on the machine that allows for a full trusted boot process, enabling us to trust everything from the hardware up through and including the kernel because any modifications to the kernel will cause a re-measurement and a failure to attest via IMA.

2. IMPLEMENTATION

We implemented a prototype whitelist generation utility by building on top of the libdft code from Kermerlis et. al. [10] to track taint from all files opened by the program

ACSAC '14 New Orleans, Louisiana USA

This work is sponsored by Assistant Secretary of Defense for Research & Engineering under Air Force Contract FA8721-05-C-0002.
Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

during the measurement phase, and record the filename and hash of a file that was the source of executed code into a whitelist. Libdft works on Linux 32 bit operating systems, and is built on top of Intel's PIN runtime emulation software [7]. For every executable, it creates shadow memory in order to taint track inputs to that executable, and then determine whether tainted information is ever executed.

We modified libdft to support tracking every instruction that was executed, including ones from shared libraries, data read from a file into memory via `mmap`, the contents of the initial binary (found in memory by parsing the loaded ELF file), and a number of other changes. A particular challenge is the dynamic loader, which loads many shared libraries, and the associated VDSO (virtual dynamic shared objects) file. The locations of these items in memory is not easily visible from within PIN, but is fixed when address randomization is turned off for all files tested.

Our taint tracker associates a color with every source of input (binary file, shared library, etc.), and colors all memory containing contents from that input. When the instruction pointer reaches a colored register or memory cell, the associated filename and its hash (read from a file system extended attribute generated by the IMA kernel module) are written to the whitelist file. Execution of memory colored by multiple sources results in all the sources and their hashes being added to the whitelist. Once the whitelist is produced, a very similar workflow can be used for enforcement.

3. EVALUATION

We evaluated the whitelist generation, checking for taint coverage of all JMP/RET calls, on a number of small and medium sized Linux executables ranging from `cat` which had 3,000 JMP/RET calls to the Python interpreter with nearly 700,000. We had 100% coverage for all executables, and saw between 2x and 9x slowdown relative to PIN, mostly due to the overhead of libdft rather than our additions. Further evaluation to ensure the coverage remains accurate on larger binaries is necessary, but runs into a limitation in the number of distinct inputs that can be tracked in libdft. Once larger numbers of inputs can be tracked, we plan to generate whitelists for some cloud applications based on their test suites and use that to test enforcement coverage during runtime.

4. CONCLUSION

The combination of dynamic taint analysis and existing integrity measurement techniques allows for detailed monitoring and attestation of runtime integrity, even for complex cloud applications.

5. REFERENCES

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, page 41âĂŞ46, 2005.

[2] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee. A software-hardware architecture for self-protecting data. In *Proceedings of the 2012 ACM conference on Computer and communications security*, page 14âĂŞ27. ACM, 2012.

[3] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on*, page 749âĂŞ754. IEEE, 2006.

[4] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, page 49âĂŞ54. ACM, 2009.

[5] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. *ACM SIGOPS Operating Systems Review*, 40(4):29âĂŞ41, 2006.

[6] Integrity measurement architecture (IMA) / wiki / home, 2014.

[7] Intel. Pin - a dynamic binary instrumentation tool, 2014.

[8] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, page 19âĂŞ28. ACM, 2006.

[9] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. ShadowReplica: efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, page 235âĂŞ246. ACM, 2013.

[10] V. Kermerlis. libdft: Practical dynamic data flow tracking for commodity systems, 2013.

[11] Y. Kinebuchi, S. Butt, V. Ganapathy, L. Iftode, and T. Nakajima. Monitoring integrity using limited local memory. *Information Forensics and Security, IEEE Transactions on*, 8(7):1230âĂŞ1242, 2013.

[12] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, page 21âĂŞ29. ACM, 2007.

[13] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, page 179âĂŞ194. San Diego, USA, 2004.

[14] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, page 135âĂŞ148. IEEE, 2006.

[15] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, page 223âĂŞ238, 2004.

[16] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Acm Sigplan Notices*, volume 39, page 85âĂŞ96. ACM, 2004.

[17] TCG. PC Client Specific TPM Interface Specification (TIS). Technical Report 1.2 revision 116, Trusted Computing Group, Mar. 2011.