When Encryption is not Enough: Memory Encryption is Broken

Technical Report UCSC-WASP-15-03
November 2015

D J Capelis
mail@capelis.dj

Working-group on Applied Security and Privacy
Storage Systems Research Center
University of California, Santa Cruz
Santa Cruz, CA 95064
http://wasp.soe.ucsc.edu/

Abstract

Computer Systems which allow the contents of userspace memory to be protected from view by the operating system often use encryption to implement this security boundary. This technical report shows how rapidly changing memory contents leak information even when an adversary can only read the contents of memory as ciphertext. We use an example to demonstrate that far from providing complete protection from seeing the contents of memory, the patterns of updates to the ciphertext yields information about its contents.

1 Introduction

Systems which provide protected contexts within an untrusted system, such as HARES [9], Overshadow [1], AEGIS [8], XOMOS [7] and others use memory encryption to protect the contents of memory. The goal of this protection is memory opacity, or the property that code which can only see the ciphertext of memory cannot determine anything about the contents of that memory. Unfortunately, memory encryption does not fully provide memory opacity. It can be shown any system relying on this assumption loses some degree of memory opacity over time. This class of analysis can be called *temporal cryptanalysis*.

The starting assumptions include that an attacker can read only ciphertext, all encryption technologies are correctly and competently implemented and provide sufficient integrity checks to guard against writes. These checks are assumed to be durable over time, in that it is assumed that an attacker cannot use a previous block of ciphertext to revert memory to a prior value. These attacks are all problems, but fundamentally, there are cryptographic solutions to them.

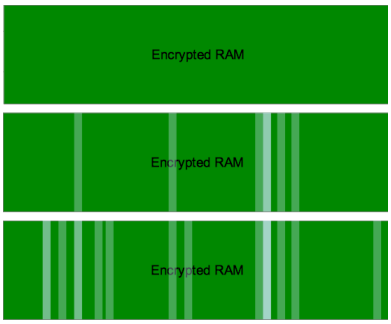


Figure 1: The ciphertext contents of RAM change as updates become visible

Temporal Cryptanalysis on the other hand, is driven by the understanding that data in systems isn't staic. While one observation of ciphertext yields little information about the contents of memory, additional observations rapidly begin to leak information about how the trusted code is changing, updating and interacting with memory. This information is useful.

None of this is particularly new. *Traffic analysis* is a very similar technique which often applies on networks and has been effectively used to break or weaken systems for decades. Traffic analysis is not unknown when it comes to protecting RAM either. It has been implemented using FPGAs on live memory busses to break real systems, notably in the reverse engineering of the Nintendo DS. This exact type of analysis has been noted in a previous work which built a system to try and mitigate these effects to some degree [5]. But somehow when it comes to trusted computing technologies, these risks are rarely discussed when the security properties of the system depend on memory opacity.

2 Proof of Concept

There doesn't seem to be a realization among the broader community that memory encryption cannot provide complete protection to memory contents. So maybe we need a better proof of concept. The goal of a proof of concept is a clear and easily demonstrated example of the flaw.

If we demonstrate a concrete example of an application's essential functions being seen and analyzed through the *changes* in memory data alone, we can show this problem is real in a more tangible way. Then we can simply rely on the hard won understanding the crypto community has

demonstrated and embraced for decades: weaknesses only become more serious with time. Here we briefly present a proof of concept in a game of chess.

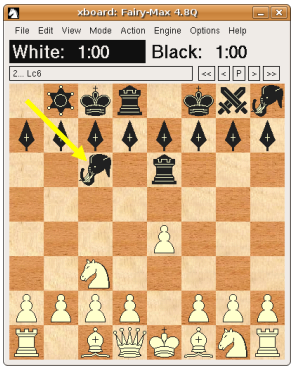
2.1 Tools for Temporal Cryptanalysis

One of the challenging aspects of developing a new proof of concept involves developing the tooling required for examining the problem. Often, new tools are needed to make things better. While plenty of existing memory snapshot tools do exist, we built our own targeting this particular issue:

- `memsnap` [4] is a memory snapshotting tool that provides the ability to dump the entire memory space of a process at a programmable interval. This allows us to compare observations at particular granularities. You can find it here: `http://github.com/djcapelis/memsnap`
- `memdiff` [2] is a memory differencing tool that outputs only the *differences* between subsequent memory snapshots with a programmable block size. You can find it here: `http://github.com/djcapelis/memdiff`
- `memxamine` [3] is a triage tool which provides a basic way of narrowing down the sections of memory that are most likely to be interesting to examine. You can find it here: `http://github.com/djcapelis/memxamine`

These tools work together to provide a toolset for exploring memory for these types of issues. If someone can show that `memsnap` and friends can produce a successful cryptanalysis, then specifically written tools will be able to do even better with lower overhead.

3 Would You Like To Play A Game?



So how can we show that playing chess is vulnerable to this issue on systems which encrypt memory? In this case, we analyzed memory snapshots from xboard, [10] a gnuchess [6] frontend, running on a Linux system. Once we recorded snapshots of memory with `memsnap`, the problem became an issue of figuring out which sections were important to examine. Thankfully, there are some easy rules to triage our memory regions:

- Regions of memory that *never* change are uninteresting.
- Regions of memory that change *sometimes* are the most interesting.
- Regions of memory that *always* change are less useful.

In the version of xboard we examined, after looking closely at the regions which changed periodically and seemed to correlate with the times players made moves in the memory snapshots, multiple regions seemed ripe for exploitation:

- Offset 0x016e1a0 in memory region 0.
- Offset 0x01702a0 in memory region 0.
- Offset 0x016d6f0 in memory region 0.

These sections tended to change each time a move was made in our traces, but another section proved even more interesting. At offset 0x007fe20, a datastructure over 64kb long lies in memory. This structure updates exactly once per move and each move is recorded array style linearly in this memory section. Not only can one determine how many moves were made by seeing when this structure updates, if the memory writeback granularity is low, a clever attacker can recover the number of moves since the last observation of the ciphertext merely by looking at how much ciphertext changed. This means even if an attacker fails to make an observation after every move, they can still determine how many moves have been played.

And of course, given the rules of chess, an attacker that knows how many moves were made in a chess game also knows which player couldn't have won the game. (Technically they can't say the other player won, because the game might have concluded in a draw.)

You can see a youtube video of this proof of concept here: `https://www.youtube.com/watch?v=Eqrtn7LKuoE`

While this section of memory was particularly easy to exploit, it's important to note that our analysis highlighted *multiple* other memory ranges vulnerable to this type of analysis. This is not a problem with one particular datastructure in xboard. This is a problem with how transparent memory write patterns are to analyze while looking at changes in ciphertext.

4 Game Over: Attacker Wins

It is important for system designers to realize that memory encryption is a weaker form of protection than denying access to memory outright. While this may seem like a simple and naive proof of concept, it is especially important to note that this didn't require sophisticated machine learning algorithms, complex analysis or even memory snapshotting tools that are selective in what they capture. If it took sophisticated tools to show information can leak in memory encryption, that might be more comforting.

But it does not. With a general purpose memory snapshotting tool and a shell script, some of the most sophisticated trusted computing systems in the world can't protect the confidentiality of a chess game. Memory encryption is not sufficient.

5 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1018928. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Bibliography

[1] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–13, 2008.

[2] D J Capelis. Memdiff. [Online]. Available: https://github.com/djcapelis/memdiff

[3] ——. Memdiff. [Online]. Available: https://github.com/djcapelis/memxamine

[4] ——. Memsnap. [Online]. Available: https://github.com/djcapelis/memsnap

[5] G. Duc and R. Keryell, "Cryptopage: an efficient secure architecture with memory encryption, integrity and information leakage protection," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual.* IEEE, 2006, pp. 483–492.

[6] GNU Chess Team. GNU Chess. [Online]. Available: https://www.gnu.org/software/chess/

[7] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 178–192.

[8] G. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th annual international conference on Supercomputing.* ACM New York, NY, USA, 2003, pp. 160–171.

[9] J. Torrey, "HARES: Hardened Anti-Reverse Engineering System," in *SyScan*, 2015.

[10] xboard team. xboard. [Online]. Available: https://www.gnu.org/software/xboard/