Dan Bednar

Article Review 1

01/08/12

IEEE Transaction on Security Journal

This article discussed how hardware can be configured to prevent malicious users from exploiting the hardware’s circuits. This peaked my interests because I had never considered the possibility of a hacker using my hardware as a means to break in and/or receive data from my computer. The article covered 3 methods of protecting the computer: power resets, data obfuscation, and sequence breaking.

Hardware resets sound very simple and for the most part it is. Hardware resets are designed to prevent “time bombs” from executing after a pre-determined number of clock cycles. The resets are designed to power cycle the hardware to clear their cache and lookup table which flushes any pending time bombs. Even if the bomb gets reloaded its counter will never reach the threshold needed to trigger the attack.

The problem with hardware resets is preventing errors and data loss when the component is resetting. The article acknowledges this and points to device drivers as the solution. When a device goes down for the few clock cycles the driver would need to replay the transaction. There is also a problem of the non-volatile memory. All of the registers cache memory and anything else that is in a volatile system gets cleared but the non-volatile remains. The solution is to burn-out, or write past the flash’s write endurance, a cell of memory that is suspected to have malicious code.

The results that are displayed in the chart on the top of page 10 in the article show that the power resets have almost no affect on the performance of the system. The graph shows that when compared to the baseline results some tests performed faster while others were slightly slower. In the end, the tests average out to no noticeable impact to the user.

The second option for preventing a hardware back door was data obfuscation. This involves encrypting data before it’s processed by the unit. It’s designed to prevent what the article considers to be “single shot cheat codes”. In non-technical terms these are malicious instructions that are disguised to look legitimate, but end up triggering some recovery or permission changing mechanism in the hardware. What makes a cheat code a “single shot” is when it can all fit into one instruction set.

Obfuscation works on two different types of units, computational and non-computational. The non-computational units are the busses or registers that simply pass and store values. Because the values only need to remain secret for a couple of clock cycles there isn’t a need for a high-tech encryption. The addition of some random value to the register or running an XOR scheme on the value is all that would be required.

When working with computational units, the authors report the need for a more complex encryption scheme. The computational units must be handled on a unit-by-unit basis to preserve performance. To help share the load, dedicated encryption and decryption units can be installed. This means the important unit only sees encrypted data that cannot trigger any malicious event.

The authors did a lot of tests on obfuscation. Not only did they test just encrypting memory, but also all the caches and a combination of both caches being encrypted. Just memory and L-Cache encryptions did not have much of a visible performance impact. However when the D-Cache got encrypted the performance impact was a lot more of a noticeable performance hit.

The final method to prevent the backdoors is sequence breaking. Sequence breaking targets what is known as a sequence cheat code. This differs from the single shot cheat codes above in that it needs multiple instruction sets to run. In order to run properly the code must be run in a very specific order. The order can be broken in two ways, re-ordering of the commands and memory values. The second is to insert dummy data in between the commands and memory values. Both of these can turn a bad set of code that would have disrupted the system into something that would no longer trigger a response.

The article mentions one way to ensure that sequences are broken. This is to have a memory controller randomly assign it to a physical location. This virtually guarantees that a malicious code will not be accessed or stored in the malicious order.

The Results for reordering and the insertion of dummy data was very similar to the results for the reset tests. Most of the re-ordering tests performed just as well as the power resets. The memory insertion tests were slightly worse but not by much according to the graph. Even then, the insertion tests performed better than most of the encryption tests. What surprises me with the reordering results is how it appears to decrease the impact of the performance hit of encryption. This can be seen by looking at the red bars on the “Memory Encryption” and “Memory Encryption with Reordering”

The article was an interesting read. It made me wonder how much of these techniques my current computers use. It’s an intriguing thought to think that every few seconds every piece of hardware in my laptop is resetting itself even as I type this paper. The paper does say that they hope to see this implemented in “real hardware” which implies that none of this is currently in use. However, it sounded like they were using a regular DDR3 memory controller to run the tests. Does this mean that RAM controllers are doing this now but nothing else is? Again, it’d be fun to know if all the information stashed in 8-gigs worth of ram is getting reset every few seconds.

**Appendix**

Graph taken from the article

