4 NetWatch: System Management Mode is not just for Governments.

by Joshua Wise and Jacob Potter

Neighbors, by now you have heard of a well known state's explorations into exciting and exotic malware. The astute amongst you may have had your ears perk up upon hearing of SCHOOLMONTANA, a System Management Mode rootkit. You might wonder, *how can I get some of that SMM goodness for myself?*

Before we dive too deeply, we'll take a moment to step back and remind our neighbors of the many wonders of System Management Mode. Our friends at Intel bestowed SMM unto us with the i386SL, a low-power variant of the '386. When they realized that it would become necessary to provide power management features without modifying existing operating systems, they added a special mode in which execution could be transparently vectored away from whatever code be running at the time in response to certain events. For instance, vendors could use SMM to dynamically power sound hardware up and down in response to access attempts, to control backlights in response to keypresses, or even to suspend the system!

On modern machines, SMM emulates classic PS/2 keyboards before USB drivers have been loaded. It also manages BIOS updates, and at times it is used to work around defects in the hardware that Intel has given us. SMM is also intricately threaded into ACPI, but that's beyond the scope of this little article.

All of this sounds appetizing to the neighbor who hungers for deeper control over their computer. Beyond the intended uses of SMM, what *else* can be done with the building blocks? Around the same time as the well known state built SCHOOLMONTANA and friends, your authors built a friendlier tool, NetWatch. We bill NetWatch as a sort of lights-out box for System Management Mode. The theory of operation is that by stealing cycles from the host process and taking control over a secondary NIC, NetWatch can provide a VNC server into a live machine. With additional care, it can also behave as a GDB server, allowing for remote debugging of the host operating system.

We invite our neighbors to explore our work in more detail, and build on it should you choose to. It runs on older hardware, the Intel ICH2 platform to be specific, but porting it to newer hardware should be easy if that hardware is amenable to loading foreign SMM code or if an SMM vulnerability is available. Like all good tools in this modern era, it is available on GitHub.[1]

We take the remainder of this space to discuss some of the clever tricks that were necessary to make NetWatch work.

4.1 A thief on the PCI bus.

To be able to communicate with the outside world, NetWatch needs a network card of its own. One problem with such a concept is that the OS might want to have a network card, too; and, indeed, at boot time, the OS may steal the NIC from however NetWatch has programmed it. We employ a particularly inelegant hack to keep this from happening.

The obvious thing to do would be to intercept PCI configuration register accesses so that the OS would be unable to even prove that the network card exists! Unfortunately, though there are many things that a System Management Interrupt can be configured to trap on, PCI config space access is not a supported trap

[1] `https://github.com/jwise/netwatch`

on ICH2. ICH2 does provide for port I/O traps on the Southbridge, but PCI peripherals are attached to the Northbridge on that generation. This means that directly intercepting and emulating the PCI configuration phase won't work.

We instead go and continuously "bother" PCI peripherals that we wish to disturb. Every time we trap into system management mode—which we have configured to be once every 64ms—we write garbage values over the top of the card's base address registers. This effectively prevents Linux from configuring the card. When Linux attempts to do initial detection of the card, it times out waiting for various resources on the (now-bothered) card, and does not succeed in configuring it.

Neighbors who have ideas for more effectively hiding a PCI peripheral from a host are encouraged to share their PoC with us.

4.2 Single-stepping without hardware breakpoints.

In a GDB slave, one of the core operations is to single-step. Normally, single-step is implemented using the TF bit in the FLAGS/EFLAGS/RFLAGS register, which causes a debug exception at the end of the next instruction after it is set. The kernel can set TF as part of an IRET, which causes the CPU to execute one instruction of the program being debugged and then switch back into the kernel. Unfortunately Intel, in all their wisdom, neglected to provide an analog of this feature for SMM. When NetWatch's GDB slave receives a single-step command, it needs to return from SMM and arrange for the CPU to execute exactly one instruction before trapping back in to SMM. If Intel provides no bit for this, how can we accomplish it?

Recall that the easiest way to enter SMM is with an I/O port trap. On many machines, port `0xB2` is used for this purpose. You may find that MSR SMI_ON_IO_TRAP_0 (`0xC001_0050`) has already been suitably set. NetWatch implements single-step by reusing the standard single-step exception mechanism chained to an I/O port trap.

Suppose the system was executing a program in user-space when NetWatch stopped it. When we receive a single step command, we must insert a soft breakpoint into the hard breakpoint handler. This takes the form of an OUT instruction that we can trap into the #DB handler that we otherwise couldn't trap.

- Track down the location of the IDT and the target of the #DB exception handler.
- Replace the first two bytes of that handler with `E6 B2`, "`out %al, $0xb2`"
- Save the `%cs` and `%ss` descriptor caches from the SMM saved state area into reserved spots in SMRAM.
- Return from SMM into the running system.

Now that SMM has ceded control back to the regular system, the following will happen.

- The system executes one instruction of the program being debugged.
- A #DB exception is triggered.
- If the system was previously in Ring 3, it executes a mode switch into Ring 0 and switches to the kernel stack. Then it saves a trap frame and begins executing the #DB handler.
- The #DB handler has been replaced with `out %al, $0xb2`.

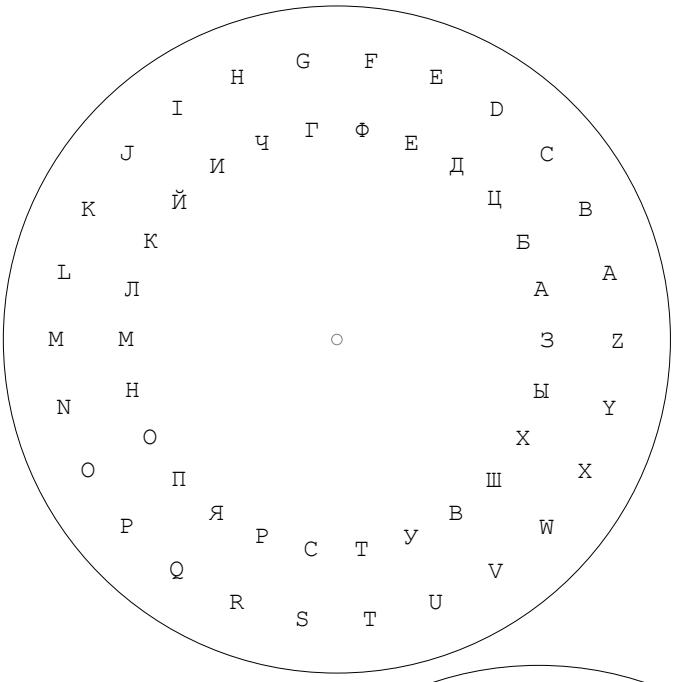
Finally, the OUT instruction triggers a System Management Interrupt into our SMM toolkit.

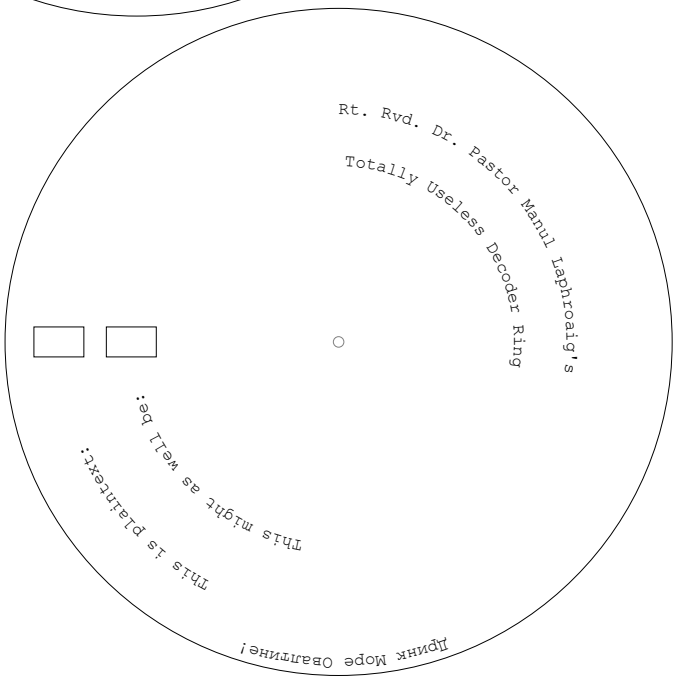
- The SMI handler undoes the effect of the exception that just happened: it restores RIP, CS, RFLAGS, RSP, and SS from the stack, and additionally restores the descriptor caches from their saved copy in SMRAM. It also replaces the first two bytes of the #DB handler.
- NetWatch reports the new state of the system to the debugger. At this point, a single X86 instruction step has been executed outside of SMM mode.

4.3 Places to go from here.

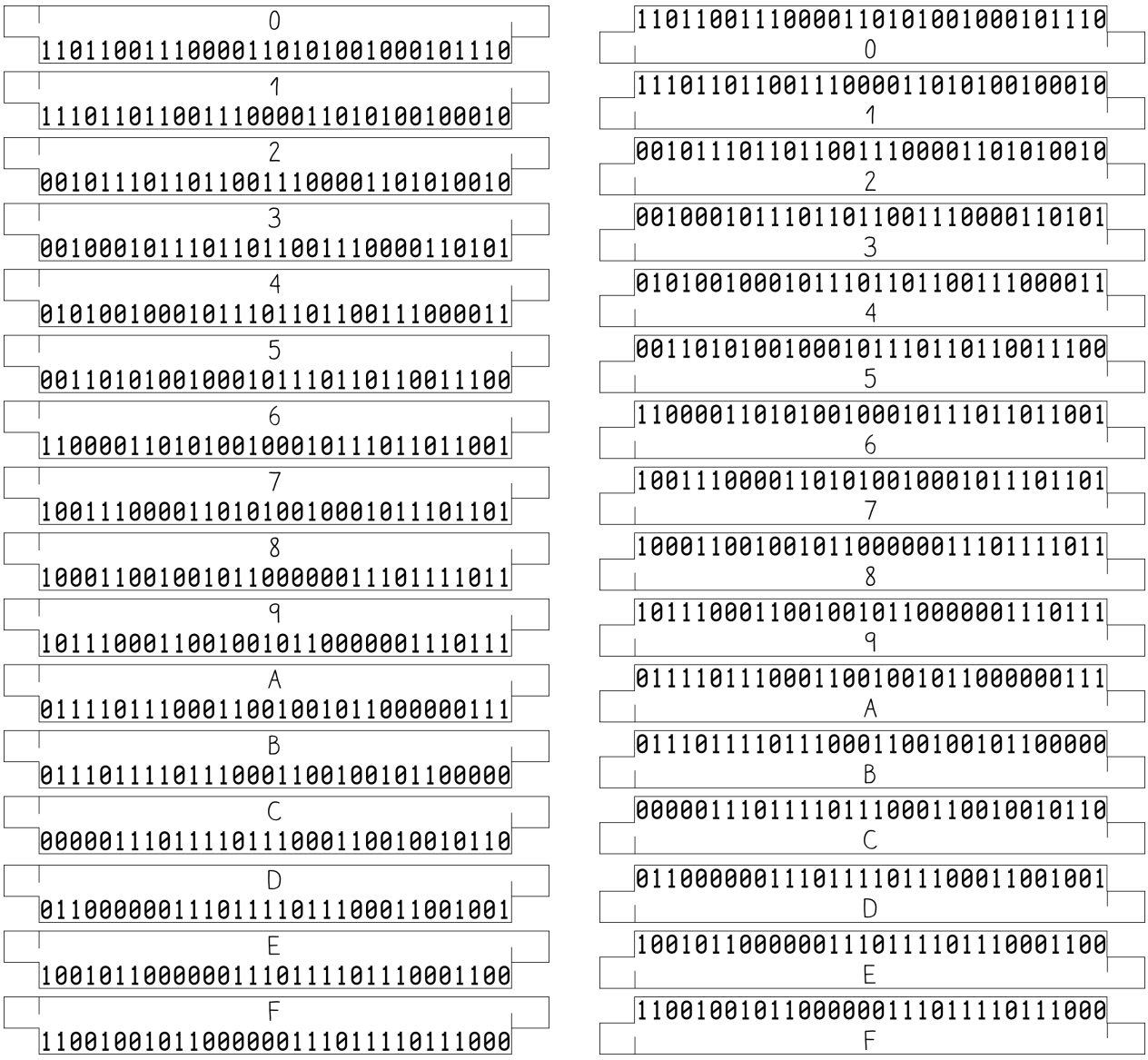
NetWatch was written as a curiosity, but having a framework to explore System Management Mode is damned valuable. Those with well-woven hats will also enjoy this opportunity to disassemble SMM firmware on their own systems. SMM has wondrous secrets hidden within it, and it is up to you to discover them!

The authors offer the finest of greets to Dr. David A. Eckhardt and to Tim Hockin for their valuable guidance in the creation of NetWatch.





Hey kids!
Xerox this page and cut out the crypto wheel.
You can write your own secret messages that only idiots can't read!



Hey kids!
Xerox this page and cut the paper strips apart.
You can write your own odd-alignment packet-in-packet injection strings!