Stuxnet: Dissecting a Cyberwarfare Weapon

Ralph Langner
Langner Communications

Last year marked a turning point in the history of cybersecurity—the arrival of the first cyberwarfare weapon ever, known as Stuxnet. Not only was Stuxnet much more complex than any other piece of malware seen before, it also followed a completely new approach that's no longer aligned with conventional confidentiality, integrity, and availability thinking. Contrary to initial belief, Stuxnet wasn't about industrial espionage: it didn't steal, manipulate, or erase information. Rather, Stuxnet's goal was to physically destroy a military target—not just metaphorically, but literally. Let's see how this was done.

SCADA and Controllers 101

Much has been (and still is) written about Stuxnet being an attack on SCADA systems, but this simply isn't true. In a nutshell, a SCADA system is a Windows application that allows human operators to monitor an industrial process and to store and analyze process values. True, a SCADA application played a small role in the Stuxnet attack, but mainly as a means of distribution. The real attack wasn't against SCADA software—it was aimed at industrial controllers that might or might not be attached to a SCADA system. To get rid of another misconception, the attack wasn't remotely controlled, either—it was completely stand-alone and didn't require Internet access. The command-and-control servers that Stuxnet contacted appear to have been used primarily for evidence of compromise.

We can think of a controller as a small, real-time computer system that manipulates electrical outputs based on the condition of electrical input signals and program logic. Devices such as pumps, valves, drives (motors), thermometers, and tachometers are electrically connected (hardwired) to a controller, either directly or by what's called a *fieldbus connection*. While a computer program only operates on information, a controller program—sometimes called *ladder logic*—operates on physics. This is an interesting and important point, especially for cybersecurity considerations: manipulations of a controller have less to do with the confidentiality, integrity, and availability of information and more to do with the performance and output of a physical production process. In the worst case, controller manipulations could lead to physical damage. Stuxnet was the worst case, as it was carefully designed by experts to do just that with the utmost determination.

Distribution and Target Determination

The attackers tried their best to limit Stuxnet's spread. They didn't choose conventional worm technology but instead relied on local distribution, mainly via USB sticks and local networks. While Stuxnet infected any Windows PC it could find, it was much pickier about controllers. It targeted only controllers from one specific manufacturer (Siemens); upon finding them (attached to an infected Windows box via Ethernet, Profibus, or Siemens' proprietary communication link called MPI), it went through a complex process of fingerprinting to make sure it was on target. This process included checking model numbers, configuration details, and even downloading program code from the controller to check if it was the "right" program. Stuxnet did all of this by exploiting the vendor's driver DLL, which both the SCADA product and the programming software use to talk to the controller. When it found a match, Stuxnet's dropper loaded rogue code to the controller. Because this is done only if an exact fingerprint match is found, we haven't seen controller infections anywhere but those confirmed by Iran in its Natanz uranium enrichment plant. In contrast, the dropper itself spread to roughly 100,000 infections worldwide. It is even possible to infer by this fact that Natanz must have been Stuxnet's one and only target, because this is the only installation globally where controller infections have been reported. The rogue driver DLL contained three controller code sets: two destined for a Siemens 315 controller, and the third looking for a 417 controller. The first two were almost identical and will be treated as one digital warhead in the following discussion. The 417 attack code was much more complex and four times the size of the 315 attack code. For starters, the 417 is Siemens' top-of-the-line controller product, whereas the 315 is a small, general-purpose controller.

It's just like in a Hollywood movie, where the bad guys feed observation cameras with unsuspicious prerecorded input. In the same manner, Stuxnet replayed prerecorded input to the legitimate code during the attack.

Controller Hijacking

Once Stuxnet found a matching controller target, it loaded rogue code on the controller from one of its three code sets, along with any data blocks that the rogue code was working on. The rogue code was logically structured in a set of subfunctions; the attackers' goal was to get any of these functions called. Stuxnet achieved this goal by injecting code into an executive loop. Both the 315 and the 417 attack codes used the main cycle for this purpose—the 315 also injected code into the 100-ms timer. (The 100-ms timer is like an interrupt handler that the operating system calls automatically every 100 milliseconds.) We can think of a controller's main cycle as the `main()` function in a C program, with the notable difference being that the operating system automatically executes the main cycle, hence the name.

The code injections got Stuxnet in business—it could then do its thing and prevent legitimate controller code from doing anything useful. However, it didn't. Stuxnet was a stealth control system that resided on the controller alongside legitimate code, which continued to be executed; Stuxnet only occasionally took over. Both on the 315 and on the 417, the attacks were implemented as state machines that didn't require any interaction with command-and-control servers. Instead, the attacks were triggered by complex timer and process conditions. In the initial state, attack code just stayed put and let the legitimate controller code do its thing, although it monitored it closely. When strike time came, rogue code took control without the legitimate controller code noticing. During the attack, the legitimate code was simply disabled. How this happened differed considerably for the 315 and the 417 attacks.

For the 315 attack, execution of legitimate code simply halted during the strike condition, which can take up to 50 minutes. This is the infamous DEADFOOT condition that I first discovered and published on 16 September 2010 (www.langner.com/en/2010/09/16/stuxnet-logbook-sep-16-2010-1200-hours-mesz). DEADFOOT is an aviation adage; if an engine quits in a twin-engine airplane, the pilot can determine which one died by using the concept "dead foot = dead engine." Determining the dead engine in propeller-driven aircraft isn't easy because the propeller keeps spinning; the only option is to cut fuel to the dead engine and feather the propeller. In an engine-out situation, more than one pilot has turned off the good engine and ended up crashing.

The 417 attack code followed a different concept. At strike time, legitimate controller code kept running, but it was isolated from real I/O. The 417 attack implemented a man-in-the-middle attack on the controller, intercepting the interaction between the legitimate control program and physical I/O. In a modern controller, control logic operates on a "process image" of physical I/O. This is similar to a computer application that does not directly access any physical interface registers of an I/O chip (such as Ethernet, USB, or serial), but would use data from the driver's buffer. The 417 attack code exploited this, intercepting physical I/O and providing the legitimate program code with a fake process image. In computer terms, it acted like a fake driver. But it gets even weirder: the fake data that the 417 attack code fed to the legitimate controller program was recorded from real, unsuspicious inputs. It's just like in a Hollywood movie, where the bad guys feed observation cameras with unsuspicious prerecorded input. In the same manner, Stuxnet replayed prerecorded input to the legitimate code during the attack.

Threat Outlook and Mitigation Opportunities

Many people who followed the IT press closely in December were under the impression that with Microsoft's last Stuxnet-related security patch, the threat was gone. Unfortunately, this is a severe misconception. The Microsoft patches only affect the dropper part of Stuxnet; the digital warheads are a completely different story. The vulnerabilities that they exploit can't be patched because they aren't software or firmware defects, but legitimate product features. In other words, these vulnerabilities are here to stay until the vendor comes out with a new product generation and asset owners replace the installed base with such new products before the scheduled end of lifetime, a scenario that will take many, many years. Within this timeframe, attackers can use the same digital warheads again, given they use a new dropper, and new droppers are comparatively easy to obtain. The digital warheads as such can't be disarmed if they make it on controllers. Industrial controllers don't have antivirus software, and the controller exploits that Stuxnet uses can't be "patched." They're fully functional as long as the present product generation is in use, which could be another 20 years.

Although many argue that Stuxnet makes a point for better defense in depth, this is actually a bit misleading. True, defense in depth never hurts, but it's no recipe for a world that's immune against Stuxnet-inspired malware. The major vulnerability that Stuxnet exploits is that present controllers don't allow for digital code signing. A controller treats code—as long as it's syntactically correct—as legitimate, no matter where it came from. A digital signature would enable the controller to make sure that the code loaded originated from a legitimate engineering station. Because this check needs only to be done once after load time, it isn't time critical (runtime is real time on a controller, prohibiting compute-intense calculations).

The next best solution to verify code and configuration integrity is to monitor controllers for any change. For the products attacked by Stuxnet, this can be done easily via the network. The essential requirement here is not to use the vendor's driver DLL, which might be compromised. If the monitoring solution uses an independent driver, it can check for changes reliably by fingerprinting the controllers' configuration in a way similar to Stuxnet. Once a change is detected, the operator or maintenance engineer can then determine if the change is legitimate. This way, even changes can be detected that originate from authorized engineering stations but occurred accidentally or weren't reported, which is a frequent problem with contractors in real production environments. ☐

Ralph Langner *is the founder and CEO of Langner Communications, a German software and consulting company that focuses on control system security. He has more than 20 years' experience with computing and control systems and became recognized worldwide as the first researcher who determined that Stuxnet was a directed cyberwarfare attack against the Iranian nuclear program. Contact him at info@langner.com.*

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.