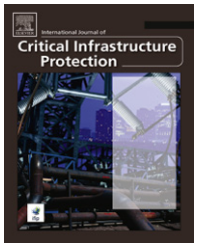
Available online at www.sciencedirect.com
SciVerse ScienceDirect
journal homepage: www.elsevier.com/locate/ijcip



Intrusion detection for resource-constrained embedded control systems in the power grid

Jason Reeves[a,*], *Ashwin Ramaswamy*[a], *Michael Locasto*[b], *Sergey Bratus*[a], *Sean Smith*[a]

[a] *Department of Computer Science, Dartmouth College, Hanover, New Hampshire 03755, USA*
[b] *Department of Computer Science, University of Calgary, Calgary, Alberta T2N 1N4, Canada*

ARTICLE INFO

Article history:
Received 16 April 2011
Accepted 25 January 2012

Keywords:
Embedded control systems
Power grid
Intrusion detection

ABSTRACT

The power grid depends on embedded control systems or SCADA systems to function properly. Securing these systems presents unique challenges—in addition to the resource restrictions inherent to embedded devices, SCADA systems must accommodate strict timing requirements that are non-negotiable, and their massive scale greatly amplifies costs such as power consumption. Together, these constraints make the conventional approach to host intrusion detection – using a hypervisor to create a safe environment from which a monitoring entity can operate – too costly or impractical for embedded control systems in the critical infrastructure.

This paper discusses the design and implementation of Autoscopy, an experimental host-based intrusion detection mechanism that operates from within the kernel and leverages its built-in tracing framework to identify control-flow anomalies, which are most often caused by rootkits that hijack kernel hooks. The paper presents the concepts underlying the original Autoscopy prototype, highlights some of the issues that arose from it, and introduces the new system, dubbed Autoscopy Jr., which addresses the issues. Tests on non-embedded systems demonstrated that the monitoring scope could be managed to limit Autoscopy Jr.'s performance impact on its host to under 5%. The paper also describes the use of an optimized probe framework to reduce overhead and the test results obtained for a hardened kernel. The results demonstrate that Autoscopy Jr.'s design and effectiveness render it uniquely suited to intrusion detection for SCADA systems.

1. Introduction

The world's critical infrastructure has become increasingly dependent on embedded control systems—computers implanted in larger devices to serve as controllers and perform many of their important tasks. The power grid has not been immune from this trend. One study [1] predicts that the number of smart electric meters deployed worldwide – and by extension the embedded control systems inside these meters – will increase from 76 million in 2009 to roughly 212 million by 2014.

The need to secure software that expresses complex process logic is well understood, and is particularly important for devices operating as part of a SCADA system, where this logic applies to the control of potentially hazardous physical processes such as power generation. Any failure to secure these important devices can have grave consequences, as demonstrated by Stuxnet [2]. As a general exploit alone, Stuxnet's credentials are frighteningly

* *Corresponding author.*
E-mail address: Jason.O.Reeves.GR@dartmouth.edu (J. Reeves).

1874-5482/$ - see front matter © 2012 Published by Elsevier B.V.
doi:10.1016/j.ijcip.2012.02.002

impressive. The program attempted to subvert targets using four zero-day vulnerabilities and two compromised digital certificates, and incorporated rootkit functionality that enabled it to hide its behavior. However, rather than attacking systems indiscriminately, Stuxnet specifically targeted devices within an industrial control system. In particular, the program looked for Windows computers used to configure programmable logic controllers that operate uranium-enriching centrifuges [3]. Because industrial control systems are often found within critical facilities such as power plants, the consequences of such sabotage could be severe and potentially life-threatening. Therefore, ensuring the integrity of these devices and others within the critical infrastructure is essential.

A number of malware protection proposals (see, e.g., [4–9]) address the issue of device integrity using virtualization, relying on a hypervisor to create a trusted space to use for monitoring the potentially-compromised system. These proposals, however, fail to account for some key attributes of embedded control systems used in the power grid:

- The space and storage constraints of embedded devices may render the use of a hypervisor impractical. For example, Petroni and Hicks [7] found that running the Xen hypervisor on a test platform (a laptop with a 2 GHz dual-core processor and 1.5 GB RAM) imposes an overhead of nearly 40%.
- Embedded systems in the power grid must deal with strict application timing requirements, some of which require a message delivery time of no more than 2 ms [10].
- The extra costs associated with security computations (i.e., computations performed solely to achieve device security goals) do not scale well in a power grid environment. For example, LeMay and Gunter [11] note that, in a planned rollout of 5.3 million electric meters, including a trusted platform module (TPM) with each device would incur an added power cost of more than 490,000 kWh per year, even assuming that the TPMs sit idle at all times.

On the whole, the collective price (in terms of maintenance, patching, energy, etc.) [12] of hypervisor-based approaches obviates their use in industrial control environments. However, this conclusion leaves the door open for non-virtualized alternatives. A particularly promising approach is to use a kernel protection mechanism that resides at the same privilege level as the kernel to defend against malware. Such approaches have proven to be effective in the past. For example, kernel hardening efforts (e.g., grsecurity/PaX [13] and OpenWall [14]) that implement a variety of security mechanisms in the code of the Linux kernel itself (by creatively leveraging the MMU hardware of x86 and other architectures and the ELF binary format features) are successful at reducing the kernel attack surface without resorting to a separate implementation of a formal reference monitor.

This paper describes the Autoscopy system, which we developed as a prototype in-kernel intrusion detection mechanism [15] and recently refined to protect embedded control systems [16]. Instead of being separated from its host via a hypervisor, Autoscopy demonstrates the possibilities of an intrusion detection system working inside the operating system kernel to reduce the overhead on its host [15]. To do so, the system leverages Kprobes [17,18], a tracing framework included in the Linux kernel, to place probes in indirectly-called functions within the kernel to dynamically monitor the control flow of running programs for anomalies [15].

In tests run on a standard laptop system, Autoscopy was able to detect every one of the published control-flow hooking rootkit techniques it was tested against, while imposing an overhead of 5% or less on a wide range of performance benchmarks [15]. Our second iteration of the program, dubbed Autoscopy Jr., includes a system profiler, which permits the location and removal of "heavy" probes that generate too much overhead, helping balance security with performance and allowing the customization of the mediation scope to keep the overhead under the 5% limit [16]. These results indicate that, unlike virtualized intrusion detection solutions, Autoscopy's design and performance make it well-suited to the task of protecting embedded control devices, including those that are used within the critical infrastructure.

2. Background

This section discusses embedded systems in the power grid, frames the debate between virtualized and in-kernel security solutions, and introduces the tracing framework used by Autoscopy for monitoring its host. The section also provides details about one of the more successful projects in the area of kernel hardening.

2.1. *Embedded control systems in the power grid*

The electrical grid contains a variety of intelligent electronic devices (IEDs), including transformers, relays and remote terminal units. The capabilities of these devices can vary widely. For example, the ACE3600 RTU sports a 200 MHz PowerPC-based processor and runs a VX-based real-time operating system [19], while the SEL-3354 computing platform has an option for a 1.6 GHz processor based on the ×86 architecture and can support operating systems such as Windows XP or Linux [20].

In addition to the issues that arise from restricted resources, embedded control systems in the power grid are often subject to strict timing requirements when passing data in a network. For example, IEDs within a substation require a message delivery time of less then 2 ms to stream transformer analog sampled data, and must be able to exchange event notification information for protection within 10 ms [10]. Given these small timing windows, introducing even a small amount of overhead could affect a device so that it cannot meet its message latency requirements, prohibiting it from performing its task—an outcome that may well be worse than a malware infection. Therefore, it is vital to limit the amount of overhead imposed on a device, especially as its availability takes precedence over its security.

Another important issue is the evolution of the technologies used to power critical embedded systems. While companies have historically used customized proprietary products in SCADA and other critical systems, the current trend is to deploy commercial off-the-shelf (COTS) products, including operating systems, applications and communication protocols [21]. (The COTS trend was confirmed in a conversation

Please cite this article in press as: J. Reeves, et al., Intrusion detection for resource-constrained embedded control systems in the power grid, International Journal of Critical Infrastructure Protection (2012), doi:10.1016/j.ijcip.2012.02.002

with David Whitehead from Schweitzer Engineering Laboratories [22].) Our partners in the power hardware industry indicate that time-to-market concerns drive the movement towards COTS solutions; also, the fact that control systems tend to stay in operation for decades after their initial adoption [23] amplifies both the benefits of gaining acceptance in the marketplace and the costs of being left behind.

2.2. *Virtualization vs. self-defense*

In the computer security community, virtualization often means simulating a specific hardware environment that can function as if it were an actual physical system. Typically, one or more of these simulations, or virtual machines (VMs), are executed such that they are isolated from the actual system and other VMs, with a virtual machine monitor (VMM) in place to moderate VM access to the real hardware.

Virtualization has become a commonly-used security measure. The basic idea is to employ a hypervisor that imposes a strict barrier between a VM and the underlying hardware, preventing a program running on a VM to affect the host system or other VMs executed by the host. Several recent intrusion detection system proposals [4,5,7] leverage this feature to separate the detection program from the system being monitored. However, these isolation assumptions have been challenged by Bratus et al. [12], who argue that there is no good way to discuss policies concerning how information is allowed to pass between boundaries. In particular, Bratus et al. state:

"*... [because] little thought has been given to what the best way is to combine the twin roles of resource provider and reference monitor ... virtualization environments can find themselves attempting to measure security-relevant properties of a system in ways that are both creative and convoluted.*"

In addition, such a setup is computationally expensive (a hypervisor can add 40% overhead [7]) and an embedded control system may not have the available resources to support such a configuration and still perform its duties in a timely manner. Also, hypervisors often found in virtual configurations are not immune to attack themselves [24,25]. Moreover, Bratus et al. [12] note that adding a hypervisor means updating the guest software and the host operating system, and that remotely-deployed machines require remote management systems that rely on less-than-secure technologies.

To avoid the trouble and overhead of a virtualized solution (or other external solution), we propose using an internal approach to intrusion detection, one that allows the kernel to monitor itself for malicious behavior. The idea of giving the kernel a view of its own intrusion status dates at least as far back as 1996, when Forrest et al. [26] proposed building a system-specific view of "normal" behavior, which could then be used for comparisons with future process behavior. The approach underlying Autoscopy can be viewed through the same lens, as the kernel is provided with a module that allows it to perform intrusion detection using its own structures and to determine whether an action is trustworthy or not.

2.3. *Kprobes*

In recent years, several operating systems have introduced tracing frameworks to give authorized users standard and easy access to the internals of the system at the granularity level of kernel symbols. Examples include DTrace [27] for Solaris and Kprobes [17,18] for Linux.

Once properly configured, one or more Kprobes can be inserted into the kernel at any arbitrary address within kernel text (including multiple probes at the same address), although some exceptions exist [17]. Upon receiving a signal from the appropriate trap, the system first verifies that it is indeed a Kprobe breakpoint [18], then passes control to the Kprobe mechanism, which executes the pre-handler associated with the probe, then single-steps the probed instruction, and finally executes the post-handler of the probe [17].

A recent enhancement to the Kprobe infrastructure is the concept of "direct jump probes" or Djprobes, which were introduced by Hiramatsu [28]. A Djprobe uses a `jmp` instruction to move to the corresponding Kprobe code rather than using a breakpoint instruction [29]. After some safety checking to determine whether or not it is appropriate to overwrite the bytes needed for the `jmp` instruction, the system prepares a "detour buffer" that handles saving/restoring registers, provides a path to the probe handlers, and returns the flow back to the original execution path [17]. After further safety checking, the `jmp` to the detour buffer is inserted into the kernel.

The primary benefit of using Djprobes is speed: Hiramatsu's initial testing showed that Djprobes were at least ten times faster than other probes [28]. However, Djprobes introduce several restrictions when performing safety checks, which limit where an optimized probe can be inserted [17].

2.4. *grsecurity and PaX*

The grsecurity project was originally a part of a project focused on hardening the 2.4 Linux kernel [30]. grsecurity provides a number of additional protection features for the Linux kernel, including such features as a full-fledged role-based access control system and enhanced `chroot` protection [31]. Its most notable feature, however, is the inclusion of the PaX Project [13], which contains a number of its own kernel hardening measures, including address space layout randomization (ASLR) [32] and memory-page execution protection (using a no-execution bit if available [33] or simulating it if one is not available [34]).

The payoff of using grsecurity/PaX is a kernel that is much more difficult to exploit [35]. Very little public work exists on the subject of exploiting grsecurity/PaX-hardened kernels and even Rosenberg and Oberheide's exploit technique [35], which leveraged vulnerabilities in kernel code and not in the grsecurity/PaX protections, was promptly squashed by the PaX Team [35]. On the whole, the grsecurity project is a testament to the security benefits that can be achieved by an in-kernel protection scheme.

3. Related work

A large portion of the work on kernel rootkit technique analysis that defined the threat space and informed defenders originated in hacker publications such as *Phrack*

Please cite this article in press as: J. Reeves, et al., Intrusion detection for resource-constrained embedded control systems in the power grid, International Journal of Critical Infrastructure Protection (2012), doi:10.1016/j.ijcip.2012.02.002

and public forums such as the Bugtraq mailing list. The discussion of system call hijacking and countermeasures can be traced back to at least 1997 (see the classic hacker guide [36] for a summary of this early work). A complete survey of this research is beyond the scope of this paper; however, interested readers are encouraged to examine *Phrack* from issue #50 [37] onward.

Much work in the intrusion detection field has been based on the availability of a hypervisor or some other virtualization primitive. SBCFI [7] uses VMs to create a separate, secure space for their control-flow monitoring program, from which it is possible to validate both the kernel text and any control-flow transfers in the monitored operating system. Patagonix [5] and VMWatcher [4] use hypervisors to protect their monitoring programs, but take different approaches to bridging the semantic gap between the hypervisor and the operating system. Patagonix relies on the behavior of the hardware to verify the code being executed, while VMWatcher simply reconstructs the internal semantics of the monitored system for use by an intrusion detection system within the secured VM. NICKLE [8] and HookSafe [9] use trusted shadow copies of data to protect against rootkits. NICKLE uses a VMM to create a copy of VM memory space that only contains authenticated kernel instructions; this helps ensure that unauthenticated code is not allowed to run in kernel space. HookSafe copies the kernel hooks of an operating system into a page-aligned memory area, where it can take advantage of the page-level protection within the hardware to moderate access.

Several malware detection proposals do not require a hypervisor, but they suffer from other drawbacks that affect their utility in an embedded control system. For example, Kolbitsch et al. [38] build a behavior graph of individual malware samples using system calls invoked by malware, and then attempt to match unknown programs to the graph. However, much like traditional antivirus systems, this approach requires prior analysis of malware samples; deploying updates to embedded devices – which are often remotely deployed in areas with questionable network coverage – is also a challenge. Integrating a security policy into programs has also been investigated, but considerable effort is required to adapt this solution to new systems. As an example, the proposal of Hicks and colleagues [39] to bring together a security-typed language with operating system services that handle mandatory access control would most likely require rewriting a large number of legacy applications.

Kprobes have been used for a number of different tasks. Most of them focus on debugging the kernel or analyzing kernel performance (e.g., SystemTap [40]), but some other novel uses for Kprobes have recently been identified. For example, ACAP [41] uses Kprobes to capture network packets by probing important functions in the INET socket layer while Atom LEAP [42] leverages Kprobes to place "energy calipers" at arbitrary kernel code locations for measuring and characterizing the energy usage of a system. However, to the best of our knowledge, the Autoscopy effort [15,16,43] is the first to leverage Kprobes as a tool for system protection.

4. Autoscopy and Autoscopy Jr.

This section provides a high-level overview of the original Autoscopy system, discusses some of its shortcomings and shows how they are addressed in Autoscopy Jr. Also, the section discusses how the Autoscopy setup makes it uniquely suited for embedded control systems. Interested readers are referred to [15] for additional details about Autoscopy and to [16] for details about Autoscopy Jr. and its new features.

4.1. *Autoscopy design*

Autoscopy does not search for specific instances of malware on its host—instead, it watches for a specific type of control-flow alteration commonly associated with malicious programs. The control flow of a program is defined as the sequence of code instructions that are executed by the host system when the program is executed. Diverting control flow within a system has been a favored tactic of malware authors for some time, and, as such, using control-flow constraints as a security device has been well-explored (see [44] for a good discussion of the topic).

Specifically, Autoscopy looks for a certain type of pointer hijacking, where a malicious function interposes itself between a function pointer and the original function to which it pointed. The hijacking causes the pointer to point to the malicious function, which then calls the original target function of the pointer at some point within itself. In this manner, the malicious program can use the original target function to preserve an illusion of normalcy by giving the user the expected output while allowing the malware to perform whatever actions it desires (e.g., scrubbing the output to hide itself and its activities).

Autoscopy has two phases: the learning phase and the detection phase.

Learning phase. In this phase, Autoscopy scans the kernel for function pointers to protect and collects information about "normal" behavior on the system. First, Autoscopy scans kernel memory for function pointers by dereferencing every address it finds, looking for an address that could point to another location within the kernel. (This list can be verified against the `System.map` file of the kernel if desired.) Next, Autoscopy places a Kprobe on every potential function pointer it finds, then silently monitors the probe as the system operates, collecting all the control-flow information it requires for the detection method being used. (Multiple rounds of probing may be necessary and probes that are never activated are removed from consideration.) The end result is a list of functions that Autoscopy tags as being called by a function pointer, complete with the necessary detection information.

To obtain a complete picture of trusted behavior, the Linux Test Project [45] was used to exercise as much of the kernel as possible, to attempt to bring rarely-used functions under the protection scope and to reduce the number of false positives from frequently-used ones. Since this method can leave out some of the more task-specific behavior, it is recommended to engage actual use cases in the learning phase in addition to test suites.

Detection phase. In this phase, Autoscopy places Kprobes on the functions tagged in the learning phase. However, instead of collecting information about system behavior, it verifies the information against the "normal" behavior data compiled earlier. Anomalous control-flows that are identified can be announced immediately or can be logged for collection.

Please cite this article in press as: J. Reeves, et al., Intrusion detection for resource-constrained embedded control systems in the power grid, International Journal of Critical Infrastructure Protection (2012), doi:10.1016/j.ijcip.2012.02.002

4.2. Shortcomings

Upon re-examining Autoscopy in the hopes of leveraging it to protect critical infrastructure assets, we discovered some problems with its implementation that needed to be addressed.

1. *Viewing kernel memory using* `mmap`. Using `mmap` is problematic because the `sys_mmap` system call is itself vulnerable to hijacking and cannot be trusted to provide the desired access to memory [46].
2. *Overhead imposed by the disassembler*. Autoscopy relies on a separate disassembler library (`udis86` [47]) for digging into the assembly code of the host. While the original rounds of testing reported in [15] indicated that using the disassembler inside the probes was efficient enough to be feasible, later tests suggested that the exact opposite was the case. This was confirmed via personal communication with the author of [15]. A more lightweight method is needed to determine whether or not a function is called indirectly.
3. *Edge cases introduced by the detection metric*. Autoscopy identifies control flow anomalies by observing the argument similarity between function calls within the current flow (argument similarity is defined as the number of parameters in the same position and possessing the same value between two function calls) [15]. Simply stated, if more than half of the arguments of a probed function and those of a function discovered above or below it in the current control flow match, then Autoscopy flags it as suspicious behavior.

   However, this metric is problematic for indirectly-called functions with less than two arguments (see [16] for a sampling of such functions found inside the kernel). If we assume that a malicious program changes at least one parameter of a function it hijacks, in the case of a one-argument function, it would change the only parameter, causing the argument similarity check to fail and allowing the malicious program to continue to operate without being detected.
4. *Lack of space for storing kernel control flow information*. During the information-gathering portion of the learning phase, Autoscopy collects the current return address, which it uses later to determine whether or not the probed function is called indirectly [15]. However, the system only sets aside enough space to store a single return address per probe, which means that the slot would be overwritten every time a probe is hit [16]. Furthermore, the check for indirect function calls is only performed after data from the probes is gathered, meaning that the decision about whether or not a function was called indirectly is based on the return address written during the final firing of the probe. Thus, two problematic scenarios arise:
   - If a probe is called indirectly and then called directly at a later point in time, Autoscopy only sees the direct function call and ignores the function believing it was never called indirectly.
   - If a probe is called indirectly and then called indirectly from a different location within the kernel, Autoscopy only collects the context information about the second indirect function call. If the first indirect call appeared during the detection phase, its arguments may be dissimilar enough from the collected context that, even if the first indirect call was subverted, Autoscopy would not recognize it as a similar function call.

4.3. Autoscopy Jr.

We embarked on a significant redesign of Autoscopy to address the problems described above. The resulting program, dubbed Autoscopy Jr., incorporates several new features.

Memory access using a character driver. The original Autoscopy system used a character driver to deploy Kprobes and gather information from them. However, in addition to the usual read and write operations, Linux character drivers support an `ioctl` method as a catch-all for more esoteric functions such as those that perform hardware-control tasks [48]. Autoscopy Jr. takes advantage of this fact by defining an `ioctl` function within the learning phase kernel module that allows values to be read directly from memory, without taking the chance of calling an untrustworthy function.

Simplified checks for indirect function calls. To work around the inability to use the disassembler library [47] inside the probes, a simple assembly checker was developed that looked for specific bytes that signaled the start of a `CALL` instruction. In researching the byte makeup of x86 assembly instructions, we discovered that `CALL` instructions in 32-bit mode begin with one of three potential byte values: `9a`, `e8` and `ff` [49]. Of the three, only `ff` signifies an indirect `CALL` [49], indicating that the first byte of the instruction could be examined to differentiate indirect and direct function calls.

Trusted location lists. This change is the most important and the most substantial portion of the redesign. Instead of gathering context information for argument similarity checks, all the return addresses associated with indirect calls seen during the learning phase are collected and compiled into trust lists that are used as whitelists for validating control flows. While location-based verification is not a particularly groundbreaking approach (e.g., the technique has been used in [50–52]), it helps determine whether or not a current control flow is trustworthy.

To address the single-return-address-slot issue discussed above, 200 return address slots are allocated for each probe. The vast majority of probes use fewer address slots; the probes that exceeded this number were associated with functions that were never called indirectly. Thus, the switch to trusted location lists simplifies the learning-phase logic while correcting two problems with Autoscopy (#3 and #4 in Section 4.2).

Profiler for managing mediation scope. To increase the flexibility of Autoscopy Jr., a system profiler was constructed to divide the probes into groups based on their location within the kernel, thereby providing more information about where probes are located within the operating system and the amount of overhead they impose. Using the kernel source files, object files and `System.map` file, the profiler produces a set of files, where each file corresponds to a top-level directory in the kernel source and contains the probes that fall within that directory. With this information in hand, it is possible

Please cite this article in press as: J. Reeves, et al., Intrusion detection for resource-constrained embedded control systems in the power grid, International Journal of Critical Infrastructure Protection (2012), doi:10.1016/j.ijcip.2012.02.002

to benchmark each group of probes individually to see which groups impose the most overhead and may require additional adjustments to enable Autoscopy Jr. to run effectively.

4.4. Advantages

As mentioned above, Autoscopy Jr. incorporates the following improvements over the original Autoscopy system:

- A more trustworthy path to kernel memory.
- The ability to handle argument-similarity edge cases.
- Enough space to capture the return addresses for probes called indirectly from multiple places.
- A lightweight assembly checker that identifies indirect function calls even from inside probes.

Autoscopy Jr. offers two additional advantages:

Elimination of a disassembler library. Incorporating assembly checking minimizes the dependence on the actual system architecture, meaning that a disassembly library is not needed. Also, assembly checking is simple enough that the system can be adapted to other architectures simply by making a few changes to the heuristics.

Allowance of legitimate pointer hooking. If desired, Autoscopy Jr. can be used in conjunction with other programs that alter the control flow of a system for security or other legitimate reasons. This is because it can simply tag program behavior as trusted during the learning phase.

4.5. Disadvantages

For all of the advantages that Autoscopy Jr. offers, certain shortcomings exist that need to be taken into account.

Malware target. By operating within the kernel, Autoscopy Jr. is as open to compromise as the host system itself. While additional measures can be taken to protect the integrity of the program and kernel, these measures may run up against the resource constraints of the embedded control system.

Trusted base state requirement. Because trusted lists are created by simply whitelisting every return address seen in a probed function, the behavior of any malware that is installed before the learning phase would be classified as trusted. Therefore, the system that hosts Autoscopy Jr. must be in a trusted state before the learning phase to ensure that malicious behavior is classified correctly.

Kernel memory mapping dependence. The contents of kernel page tables are highly dependent on the amount of system RAM. Specifically, the kernel attempts to map as much of the available physical memory as it can, up to a limit of 896 MB [53]. This limitation, however, means that memory locations above an unmapped memory address would be missed by the scan because the addresses would never be reached. (The learning module in one of our test kernels resided in such memory.) Addressing this issue is important if Autoscopy Jr. is to be used in production power systems.

False positive identification. The number of false positives depends on the comprehensiveness of the test suites used in the learning phase. If an indirect control-flow path that is not seen in the learning phase appears during the detection phase, it will always be reported as an anomaly, whether or not the flow actually indicates a malicious hijacking. It is important to make a finer-grained distinction between a flow that is "malicious" and one that is simply "new".

4.6. Threats

At this point, it is important to consider the potential threats to Autoscopy Jr.

Data modification. An attacker with the ability to read and write to arbitrary locations on the system could conceivably modify the underlying data structures to defeat the defenses of Autoscopy Jr. For example, a malicious program could modify a Kprobe or trusted location list to include the addresses of its own functions; or it could disable individual probes.

Program circumvention. Autoscopy Jr. detects malware by checking for the use of legitimate kernel functions from illegitimate locations. However, an attacker who uses his/her own code to duplicate the functionality of a kernel function could avoid any probed functions and completely bypass Autoscopy Jr.

Kprobe-specific hijacking. As discussed in Section 2.3, regular Kprobes are triggered when the kernel hits the breakpoint placed by the probe. However, if a piece of malware interferes with the breakpoint-handling code, it could bypass the Kprobe notification setup, once again working around Autoscopy Jr.

While these threats are a concern, the design nevertheless raises the bar for a malicious program to subvert the system. Specifically, malware is forced to increase its footprint on the host in terms of either processor cycles (more cycles are needed to locate the appropriate data structures) or code size (extra functions are needed to duplicate kernel behavior and adapt it to the Kprobe architecture). The larger footprint would, of course, increase the chances of the malware being detected on the host system.

Other techniques could also be used to protect Autoscopy Jr. data—for example, the trusted location lists could be stored on a read-only memory chip. However, the embedded host constraints may hinder the complete implementation of such techniques.

5. Experimental results

The original Autoscopy system was tested on a standard laptop system running Ubuntu 7.04 and using Linux kernel version 2.6.19.7. While a different underlying system – a Pentium 4 desktop with a 2 GHz processor and 768 MB RAM – was used to test Autoscopy Jr., the same Linux distribution and kernel version were used to provide a means for comparing the results obtained using the two Autoscopy systems.

5.1. Autoscopy results

We evaluated the ability of Autoscopy to detect common control-flow altering techniques. Also, we examined the overhead imposed on the host system in terms of additional time required and bandwidth reduction. The tests demonstrate that Autoscopy performs well in both areas.

Malware detection. We tested Autoscopy against a collection of control-flow-altering rootkits that employ representative kernel hook hijacking techniques; two of the rootkits were

Please cite this article in press as: J. Reeves, et al., Intrusion detection for resource-constrained embedded control systems in the power grid, International Journal of Critical Infrastructure Protection (2012), doi:10.1016/j.ijcip.2012.02.002

Table 1 – Autoscopy detection results.

Technique	Malware	Detected
Syscall table hooking	`superkit`	Yes
Syscall table entry hooking	`kbdv3`, Rial, Synapsys v0.4	Yes
Interrupt table hooking	`enyelkm v1.0`	Yes
Interrupt table entry hooking	DR v0.1	Yes
`/proc` entry hooking	DR v0.1, Adore-ng 2.6	Yes
VFS hooking	Adore-ng 2.6	Yes
Driver hooking	Custom network driver rootkit [15]	Yes
Kernel text modification	Phantasmagoria	No

developed by us as proofs-of-concept (see [15] for additional details). Most of the rootkits tested are prototypes that incorporate hooking techniques rather than actual stealth malware captured in the wild. Nevertheless, they were written to showcase a broad range of control-flow-altering techniques and the corresponding control flow behaviors.

Table 1 lists several techniques used by malware to subvert an operating system, examples of text and/or code that incorporate these techniques, and whether or not Autoscopy was able to detect these techniques. Autoscopy was able to detect every one of the hooking behaviors listed. Interested readers are referred to [15] for a complete list of the rootkits used for testing.

Performance overhead. We measured the impact of Autoscopy using five benchmark programs: two standard benchmark suites (SPEC CPU2000 [54] and `lmbench` [55]), two large compilation projects (compiling a version of the Apache web server and the Linux kernel), and one test involving the creation of a large file. In the vast majority of these tests, Autoscopy imposed an additional time cost of no more than 5% on the system. In fact, some of the tests indicated that the system ran faster with Autoscopy installed, which we interpreted to mean that Autoscopy had no noticeable impact on the system. Only one test (bandwidth measurement during the reading of a file) showed a large discrepancy between the results with and without Autoscopy installed. In [15], Ramaswamy hypothesized that this was the result of I/O path preemption or disk caching interference on the part of the kernel.

Table 2 lists the benchmarks and the performance overhead imposed by Autoscopy and Autoscopy Jr. The kernel compilation test was only run on Autoscopy; interested readers are referred to [15] for the raw Autoscopy test data. Note that the system was not heavily inconvenienced by the presence of Autoscopy.

5.2. *Autoscopy Jr. results*

In the case of Autoscopy Jr., the tests focused on the performance aspects of the system and concentrated on keeping the overhead under the 5% threshold. We tested three scenarios: (i) an unoptimized-probe setup similar to the original Autoscopy configuration (both with and without the profiler); (ii) an optimized-probe setup (with Ubuntu 10.04 and kernel version 2.6.34) to test the effect of direct-jump probes [28,29] on the imposed overhead; and (iii) a version of the Linux kernel secured using the grsecurity kernel patch [56].

5.2.1. *Unoptimized probes*

Using `lmbench` [55] as the baseline test suite, we first measured the overhead imposed by the full set of probes discovered in the learning phase. While most of the Autoscopy Jr. results paralleled those obtained with Autoscopy, several benchmark tests produced overhead amounts that were substantially larger than expected, greatly exceeding the 5% threshold. However, by using the probe profiler, we identified the probes responsible for the increased overhead (as well as some of the excessive overhead from the previous tests) and removed them from the mediation scope.

Performance tests using the post-profiling scheme confirmed that the remaining probes did not hinder the system enough to exceed the 5% threshold. Table 2 shows the results obtained for Autoscopy Jr. using a full set of probes and Autoscopy Jr. using only the low-overhead probes identified by the profiler. Note that we only tested the full Autoscopy Jr. probe set using `lmbench` [55]. Interested readers are referred to [16] for the raw Autoscopy Jr. performance data and details about the probe locations.

5.2.2. *Optimized probes*

For the optimized probes, we again turned to `lmbench` [55] to examine how the introduction of a newer kernel sporting "faster" probes would impact system performance. Table 3 presents the *lmbench* benchmark results for Autoscopy Jr. on the 2.6.19.7 Linux kernel with unoptimized probes and the 2.6.34 Linux kernel with optimized probes. The table shows that, while the overhead observed earlier has decreased or disappeared, new sources of overhead have popped up, indicating that even with an optimized probe framework in place, the profiler would be required to bring the overhead down to acceptable limits. Interested readers are referred to [16] for the raw performance data related to the optimized probes.

The experiments also demonstrate that direct-jump probes create a conflict with another Linux tracing framework, `ftrace`, which includes a function stub that is inserted at the beginning of every function in the kernel [57]. This violates the optimized-probe restrictions, which state that the instructions being replaced by the direct jump cannot include a `CALL` instruction [17]. In summary, the testing shows that optimized probes do little good in terms of improved performance.

5.2.3. *Autoscopy Jr. and grsecurity/PaX*

Finally, we tested Autoscopy Jr. on a hardened kernel, with the hope of showing that the system could offer some value to kernels with preexisting in-kernel security measures. However, combining Autoscopy Jr. and grsecurity/PaX is no easy task, since grsecurity/PaX introduces a number of changes to the kernel that interfere with the operation of Autoscopy Jr. In particular, grsecurity/PaX appears to rearrange the load addresses of many kernel symbols (even with grsecurity/PaX set to a low security level with no

Please cite this article in press as: J. Reeves, et al., Intrusion detection for resource-constrained embedded control systems in the power grid, International Journal of Critical Infrastructure Protection (2012), doi:10.1016/j.ijcip.2012.02.002

Table 2 – Autoscopy and Autoscopy Jr. results.

lmbench latency measurements	Autoscopy overhead (%)	Autoscopy Jr. overhead (*unprofiled*) (%)	Autoscopy Jr. overhead (*profiled*) (%)
Simple syscall	−0.2	−0.1	−0.1
Simple read	+1.4	+143.2	−2.5
Simple write	−2.4	−3.5	−4.6
Simple fstat	+0.5	−3.4	+0.1
Simple open/close	+10.6	+46.4	+1.7
lmbench bandwidth measurements			
Mmap read	+0.1	+1.6	−0.1
File read	+21.1	+23.7	+1.0
`libc bcopy` unaligned	+0.1	−2.1	−3.0
Memory read	−0.1	+0.7	+0.7
Memory write	+0.3	+1.6	+1.1
SPEC CPU2000 benchmark			
`164.gzip`	+0.6	N/A	−0.8
`168.wupwise`	−0.4	N/A	−1.3
`176.gcc`	−0.8	N/A	−0.7
`256.bzip2`	−0.3	N/A	0.0
`254.perlbmk`	+0.5	N/A	0.0
`255.vortex`	+1.3	N/A	+0.4
`177.mesa`	+2.0	N/A	+0.8
Custom benchmark			
Random 256 MB File creation	+1.9	N/A	+0.6
Apache `httpd` 2.2.19 compilation	+4.1	N/A	+1.0
Linux kernel 2.6.19.7 compilation	+4.9	N/A	N/A

Table 3 – Autoscopy Jr. `lmbench` benchmark results.

Latency measurements	Autoscopy Jr. overhead (*unprofiled*) (%)	Autoscopy Jr. overhead (*unprofiled and optimized*) (%)
Simple syscall	−0.1	+0.2
Simple read	+143.2	+33.6
Simple write	−3.5	+31.0
Simple fstat	−3.4	+21.7
Simple open/close	+46.4	+12.0
Bandwidth measurements		
Mmap read	+1.6	+0.3
File read	+23.7	+4.2
`libc bcopy` unaligned	−2.1	+3.6
Memory read	+0.7	+0.4
Memory write	+1.5	+1.0

additional options) such that they do not correspond to the `System.map` file. This means that an additional mechanism – or at the very least, a variation of the current mechanism – is required to properly identify the true symbols and hooks. (Of course, any hook-searching rootkit would run into the same problems.) Since any mechanism that is added to gather symbol information should not leak any address information to an attacker, it is important to take great care in adapting the hook locator mechanism.

Because of this discovery, we were not able to obtain proper performance measurements using Autoscopy Jr. with the grsecurity/PaX patch. However, we intend to continue our efforts to integrate Autoscopy Jr. with grsecurity/PaX in order to provide another layer of protection.

6. Conclusions

The Autoscopy effort is a practical approach to intrusion detection that operates within the operating system kernel and leverages its built-in tracing framework to minimize the performance overhead on the host system. The original Autoscopy prototype has been refined to create Autoscopy Jr., an intrusion detection system that is specifically targeted towards embedded control systems in the power grid. Given the critical, time-sensitive nature of the tasks performed by embedded devices in the power grid, Autoscopy Jr. offers the flexibility to balance detection functionality with the overhead imposed on the system. Since the intrusion detection functionality is situated in the kernel, other protection measures may be needed at the hardware level (e.g., memory immutability) or software level (e.g., kernel hardening). Still, the Autoscopy effort provides a useful alternative to virtualized security solutions and the overhead they impose.

Experimental results demonstrate the effectiveness of Autoscopy Jr. in standard configurations. However, the testing

Please cite this article in press as: J. Reeves, et al., Intrusion detection for resource-constrained embedded control systems in the power grid, International Journal of Critical Infrastructure Protection (2012), doi:10.1016/j.ijcip.2012.02.002

of special kernel configurations involving optimized probes yielded sub-optimal results, and we were unable to overcome the technical challenges involved in running a hardened kernel. Nevertheless, the in-kernel approach still holds promise for securing embedded control systems where more resource-intensive solutions would not be appropriate.

Note that the views and opinions in this paper are those of the authors and do not necessarily reflect those of the United States Government or any agency thereof.

Acknowledgments

This research was supported by the Department of Energy under Award No. DE-OE0000097 and by the Department of Homeland Security under Award No. 2006-CS-001-000001. The authors also wish to thank David Whitehead, Dennis Gammel, Chris Ewing and David Buehler (Schweitzer Engineering Laboratories [22]) and Tim Yardley (University of Illinois at Urbana–Champaign) for their advice and assistance with the Autoscopy Jr. test plan.

REFERENCES

[1] Transmission and Distribution World, About 212 million smart electric meters in 2014, says ABI Research, February 3, 2010. tdworld.com/smart_grid_automation/abi-research-smart-meters-0210.

[2] N. Falliere, L. O'Murchu, E. Chien, W32.Stuxnet dossier, symantec, mountain view, California, 2011. www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[3] D. Fidler, Was Stuxnet an act of war? decoding a cyberattack, IEEE Security and Privacy 9 (4) (2011) 56–59.

[4] X. Jiang, X. Wang, D. Xu, Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction, in: Proceedings of the Fourteenth ACM Conference on Computer and Communications Security, 2007, pp. 128–138.

[5] L. Litty, H. Lagar-Cavilla, D. Lie, Hypervisor support for identifying covertly executing binaries, in: Proceedings of the Seventeenth USENIX Security Symposium, 2008, pp. 243–258.

[6] B. Payne, M. Carbone, M. Sharif, W. Lee, Lares: an architecture for secure active monitoring using virtualization, in: Proceedings of the IEEE Symposium on Security and Privacy, pp. 233–247, 2008.

[7] N. Petroni, M. Hicks, Automated detection of persistent kernel control flow attacks, in: Proceedings of the Fourteenth ACM Conference on Computer and Communications Security, 2007, pp. 103–115.

[8] R. Riley, X. Jiang, D. Xu, Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing, in: Proceedings of the Eleventh International Symposium on Recent Advances in Intrusion Detection, 2008, pp. 1–20.

[9] Z. Wang, X. Jiang, W. Cui, P. Ning, Countering kernel rootkits with lightweight hook protection, in: Proceedings of the Sixteenth ACM Conference on Computer and Communications Security, 2009, pp. 545–554.

[10] Institute of Electrical and Electronics Engineers, IEEE 1646–2004 standard: communication delivery time performance requirements for electric power substation automation, Piscataway, New Jersey, 2004.

[11] M. LeMay, C. Gunter, Cumulative attestation kernels for embedded systems, in: Proceedings of the Fourteenth European Symposium on Research in Computer Security, 2009, pp. 655–670.

[12] S. Bratus, M. Locasto, A. Ramaswamy, S. Smith, VM-based security overkill: a lament for applied systems security research, in: Proceedings of the New Security Paradigms Workshop, 2010, pp. 51–60.

[13] PaX Team, Homepage. pax.grsecurity.net.

[14] Openwall, Linux kernel patch from the Openwall Project. www.openwall.com/linux.

[15] A. Ramaswamy, Autoscopy: detecting pattern-searching rootkits via control flow tracing, Master's Thesis, Department of Computer Science, Dartmouth College, Hanover, New Hampshire, 2009.

[16] J. Reeves, Autoscopy Jr.: intrusion detection for embedded control systems, Master's Thesis, Department of Computer Science, Dartmouth College, Hanover, New Hampshire, 2011.

[17] J. Keniston, P. Panchamukhi, M. Hiramatsu, Kernel probes (Kprobes), The Linux Kernel Archives. www.kernel.org/doc/Documentation/kprobes.txt.

[18] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, M. Hiramatsu, Probing the guts of Kprobes, in: Proceedings of the Linux Symposium, vol. 2, 2006, pp. 109–124.

[19] Motorola Solutions, ACE3600 specifications sheet, Schaumburg, Illinois, 2009. www.motorola.com/web/Business/Products/SCADA%20Products/ACE3600/%5FDocuments/Static%20Files/ACE3600%20Specifications%20Sheet.pdf?pLibItem=1.

[20] Schweitzer Engineering Laboratories, SEL-3354 embedded automation computing platform data sheet, Pullman, Washington, 2011. www.selinc.com/WorkArea/DownloadAsset.aspx?id=6196.

[21] A. Wright, Secure network architecture for power grid control systems, Presented at the TCIPG Summer School on Cyber Security for Smart Energy Systems, 2011.

[22] Schweitzer Engineering Laboratories, Home, Pullman, Washington. www.selinc.com.

[23] R. Anderson, S. Fuloria, Security economics and critical national infrastructure, in: Proceedings of the Eighth Workshop on the Economics of Information Security, 2009.

[24] Mitre Corporation, CVE-2008-0923, Common vulnerabilities and exposures, Bedford, Massachusetts, 2008. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0923.

[25] R. Wojtczuk, Subverting the Xen hypervisor, Presented at Black Hat USA, 2008. www.invisiblethingslab.com/resources/bh08/part1.pdf.

[26] S. Forrest, S. Hofmeyr, A. Somayaji, T. Longstaff, A sense of self for Unix processes, in: Proceedings of the IEEE Symposium on Security and Privacy, 1996, pp. 120–128.

[27] B. Cantrill, M. Shapiro, A. Leventhal, Dynamic instrumentation of production systems, in: Proceedings of the USENIX Annual Technical Conference, 2004, pp. 15–28.

[28] M. Hiramatsu, Overhead evaluation about Kprobe and Djprobe (direct jump probe) 2005. lkst.sourceforge.net/docs/probes-eval-report.pdf.

[29] M. Hiramatsu, S. Oshima, Djprobe—Kernel probing with the smallest overhead, in: Proceedings of the Linux Symposium, vol. 1, 2007, pp. 189–200.

[30] B. Spengler, Detection, prevention and containment: a study of grsecurity, Presented at the Libre Software Meeting, 2002.

[31] Wikibooks, Grsecurity/Overview. en.wikibooks.org/wiki/Grsecurity/Overview.

[32] PaX Team, PaX—address space layout randomization. pax.grsecurity.net/docs/aslr.txt.

Please cite this article in press as: J. Reeves, et al., Intrusion detection for resource-constrained embedded control systems in the power grid, International Journal of Critical Infrastructure Protection (2012), doi:10.1016/j.ijcip.2012.02.002

[33] PaX Team, PaX—non-executable pages design and implementation. pax.grsecurity.net/docs/noexec.txt.
[34] PaX Team, PaX—paging based non-executable pages. pax.grsecurity.net/docs/pageexec.txt.
[35] T. Mittner, Exploiting grsecurity/PaX with Dan Rosenberg and Jon Oberheide, May 18, 2011. resources.infosecinstitute.com/exploiting-gresecuritypax.
[36] pragmatic/THC, (Nearly) complete Linux loadable kernel modules, 1999. dl.packetstormsecurity.net/docs/hack/LKM_HACKING.html.
[37] phrack.org, Phrack, No. 50. www.phrack.org/issues.html?issue=50, April 9, 2007.
[38] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, X. Wang, Effective and efficient malware detection at the end host, in: Proceedings of the Eighteenth USENIX Security Symposium, 2009, pp. 351–366.
[39] B. Hicks, S. Rueda, T. Jaeger, P. McDaniel, From trusted to secure: building and executing applications that enforce system security, in: Proceedings of the USENIX Annual Technical Conference, 2007.
[40] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, B. Chen, Locating system problems using dynamic instrumentation, in: Proceedings of the Linux Symposium, 2005, pp. 49–64.
[41] B. Lee, S. Moon, Y. Lee, Application-specific packet capturing using kernel probes, in: Proceedings of the Eleventh IFIP/IEEE International Conference on Symposium on Integrated Network Management, 2009, pp. 303–306.
[42] D. Singh, W. Kaiser, The atom LEAP platform for energy-efficient embedded computing, Technical Report, Center for Embedded Network Sensing, University of California at Los Angeles, Los Angeles, California, 2010.
[43] J. Reeves, A. Ramaswamy, M. Locasto, S. Bratus, S. Smith, Lightweight intrusion detection for resource-constrained embedded control systems, in: J. Butts, S. Shenoi (Eds.), Critical Infrastructure Protection V, Springer, Heidelberg, Germany, 2011, pp. 31–46.
[44] M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti, Control flow integrity, in: Proceedings of the 12th ACM Conference on Computer and Communications Security, 2005, pp. 340–353.
[45] SourceForge.net, Linux Test Project. ltp.sourceforge.net.
[46] J. Rutkowski, Execution path analysis: finding kernel based rootkits, Phrack, No. 59, 2002. www.phrack.com/issues.html?issue=59&id=10.
[47] V. Thampi, `udis86` Disassembler Library for ×86 and ×86-64, 2009. udis86.sf.net.
[48] J. Corbet, A. Rubini, G. Kroah-Hartman, Linux Device Drivers, O'Reilly Media, Sebastopol, California, 2005.
[49] Intel Corporation, Intel 64 and IA-32 architectures software developer's manual: instruction set reference, A–M, vol. 2A, Santa Clara, California, 2011.
[50] FuSyS, KSTAT—-kernel security therapy anti-trolls (2.4.x version) v1.1-2. www.s0ftpj.org/en/tools.html.
[51] J. Levine, J. Grizzard, H. Owen, A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table, in: Proceedings of the Second IEEE International Information Assurance Workshop, 2004, pp. 107–125.
[52] T. Miller, Detecting loadable kernel modules (LKM). www.s0ftpj.org/docs/lkm.htm.
[53] D. Bovet, M. Cesati, Understanding the Linux Kernel, O'Reilly Media, Sebastopol, California, 2006.
[54] Standard Performance Evaluation Corporation, SPEC CPU2000 benchmark suite, Gainesville, Florida, 2007. www.spec.org/cpu2000.
[55] L. McVoy, C. Staelin, `lmbench`: portable tools for performance analysis, in: Proceedings of the USENIX Annual Technical Conference, 1996.
[56] Open Source Security, grsecurity. grsecurity.net.
[57] S. Rostedt, Ftrace—Function Tracer, The Linux Kernel Archives. www.kernel.org/doc/Documentation/trace/ftrace.txt.

Please cite this article in press as: J. Reeves, et al., Intrusion detection for resource-constrained embedded control systems in the power grid, International Journal of Critical Infrastructure Protection (2012), doi:10.1016/j.ijcip.2012.02.002