Exe-Guard Project

Final Technical Report

January 30, 2016

Secure Control Systems for the Energy Sector

Funding Number:	DE-OE0000537
Project Director:	Rhett Smith, SEL
Principal Investigator:	Tim Marshall, DVP
Principal Investigator:	Adrian Chavez, SNL
Principle Investigator:	Sergey Bratus, Dartmouth
Project Status:	Completed
Reporting Period:	October 2010- December 2015

Project Team:

Schweitzer Engineering Laboratories, Inc.
Sandia National Laboratory
Dominion Virginia Power
Dartmouth University

Executive Project Summary

The exe-Guard Project is an alliance between Dominion Virginia Power (DVP), Sandia National Laboratories (SNL), Dartmouth University, and Schweitzer Engineering Laboratories (SEL). SEL is primary recipient on this project. The exe-Guard project was selected for award under DE-FOA-0000359 with CFDA number 81.122 to address *Topic Area of Interest 4: Hardened platforms and Systems.* The exe-Guard project developed an antivirus solution for control system embedded devices to prevent the execution of unauthorized code and maintain settings and configuration integrity.

This project created a white list antivirus solution for control systems capable of running on embedded Linux® operating systems. White list antivirus methods allow only credible programs to run through the use of digital signatures and hash functions. Once a system's secure state is baselined, white list antivirus software denies deviations from that state because of the installation of malicious code as this changes hash results.

Black list antivirus software has been effective in traditional IT environments but has negative implications for control systems. Black list antivirus uses pattern matching and behavioral analysis to identify system threats while relying on regular updates to the signature file and recurrent system scanning. Black list antivirus is vulnerable to zero day exploits which have not yet been incorporated into a signature file update. System scans hamper the performance of high availability applications, as revealed in NIST special publication 1058 which summarizes the impact of blacklist antivirus on control systems: *Manual or "on-demand" scanning has a major effect on control processes in that they take CPU time needed by the control process (Sometimes close to 100% of CPU time). Minimizing the antivirus software throttle setting will reduce but not eliminate this effect. Signature updates can also take up to 100% of CPU time, but for a much shorter period than a typical manual scanning process.*

Control systems are vulnerable to performance losses if off-the-shelf blacklist antivirus solutions aren't implemented with care. This investment in configuration in addition to constant decommissioning to perform manual signature file updates is unprecedented and impractical. Additionally, control systems are often disconnected or islanded from the network making the delivery of signature updates difficult.

Exe-Guard project developed a white list antivirus solution that mitigated the above drawbacks and allows control systems to cost-effectively apply malware protection while maintaining high reliability. The application of security patches can also be minimized since white listing maintains constant defense against unauthorized code execution. Security patches can instead be applied in less frequent intervals where system decommissioning can be scheduled and planned for. Since control systems are less dynamic than IT environments, the feasibility of maintaining a secure baselined state is more practical. Because upgrades are performed in infrequent, calculated intervals, it allows a new security baseline to be established before the system is returned to service.

Exe-Guard built on the efforts of SNL under the Code Seal project. SNL demonstrated prototype Trust Anchors on the project which are independent monitoring and control devices that can be integrated into untrustworthy components. The exe-Guard team

started with the lessons learned under this project then designed commercial solution for white list malware protection.

Malware is a real threat, even on islanded or un-networked installations, since operators can unintentionally install infected files, plug in infected mass storage devices, or infect a piece of equipment on the islanded local area network that can then spread to other connected equipment. Protection at the device level is one of the last layers of defense in a security-in-depth defense model before an asset becomes compromised.

This project provided non-destructive intrusion, isolation and automated response solution, achieving a goal of the Department of Energy (DOE) Roadmap to Secure Control Systems. It also addressed CIP-007-R4 which requires asset owners to employ malicious software prevention tools on assets within the electronic security perimeter. In addition, the CIP-007-R3 requirement for security patch management is minimized because white listing narrows the impact of vulnerabilities and patch releases.

The exe-Guard Project completed all tasks identified in the statement of project objective and identified additional tasks within scope that were performed and completed within the original budget. The cost share was met and all deliverables were successfully completed and submitted on time. Most importantly the technology developed and commercialized under this project has been adopted by the Energy sector and thousands of devices with exe-Guard technology integrated in them have now been deployed and are protecting our power systems today.

Project Actuals Compared to Goals

Estimated vs. Actual Accomplishments Milestone Description	Estimated Completion	Actual Completion
Project Start Date	*12/*2010	12/2010
Complete revision of the project management plan.	*1/2011*	1/2011
SEL, SNL, and DVP to author high-level white list antivirus concept document	3/2011	3/2011
Author high-level product requirements in a system specification.	6/2011	6/2011
Bi-annual Review #1	6/2011	6/2011
Complete identification of useable open source technology and interoperable opportunities	7/2011	7/2011
Complete product design and use cases	7/2011	7/2011
Gate 1 Exit - Go/No-Go Decision Point	**8/2011**	8/2011
Bi-annual Review #2	*12/2011*	
Complete all low-level implementation requirements	3/2012	3/2012
Bi-annual Review #3	*6/2012*	7/2012
Linux–based product prototype with preliminary code completed.	10/2012	10/2012
Bi-annual Review #4	*12/2012*	
SEL product code completes unit testing, reviews, and approvals.	07/2013	6/2013
Sandia National Labs and Dominion Virginia Power provide prototype feedback to SEL development team.	09/2013	11/2013
Commercial product development complete.	10/2013	11/2013
Gate 2 Exit - Go/No-Go Decision Point	**10/2013**	Dominion Meeting 11/2013
PHASE 2: TESTING & DEMONSTRATION		
Sandia National Labs begins product security robustness testing.	10/2013	9/2013
Dominion Virginia Power begins field verification.	10/2013	11/2013
Sandia National Labs provides product security test report results.	01/2014	3/2014
Dominion Virginia Power provides field verification results.	01/2014	12/2013
Gate 3 Exit - Go/No-Go Decision Point	**12/2013**	12/2013
Release of commercial product.	01/2014	12/2013
Dominion Virginia begins deployment of the white list antivirus solution	02/2014	3/2014

Author Deployment Guide	02/2014	3/2014
Complete research of inter-process memory access policies	8/2015	8/2015
Transfer research results to Open source community	9/2015	12/2015
Complete testing on inter-process memory access control policies	10/2015	12/2015
Project Closeout Review	11/2015	1/2016
Project Closeout Reports Submitted	12/2015	1/2016

Project Summary

The team, which is made of a system owner, national laboratory, academia, and technology supplier gathered to capture the functional requirements required to have a successful deployment of malware protection on energy delivery systems. This effort documented the system requirements and came up with a set of top level features that would need to be met to allow industry adoption:

1) A software solution capable of operating with embedded devices capable of being installed retroactively
2) Automated response and rejection of unauthorized attempts to inject malicious code or alter settings with reduced impact to system performance
3) Digitally signed firmware to prevent installation of modified firmware and have a known good starting point
4) Detailed logging and situational awareness of unauthorized attempts
5) Zero settings
6) Minimal impact to boot times and processor burden when in operation

With the design goals clearly identified the team searched for open source software solutions and evaluated if the Code Seal or Trust Anchor technology could be used. It was determined that the hardware requirements for Trust Anchor were not able to meet the backward compatible industry requirement but the team did identify the open source use of the following:

1) Security enhanced Linux SELinux for application of mandatory access controls
2) Autoscopy from Dartmouth and University of Illinois Urbana Champagne for root kit prevention

Next the team worked together to develop and commercialize the technology. SEL lead this development and selected the SEL-3620 as the first product to be commercialized with the exe-Guard technology. The SEL-3620 is a security gateway developed under the Lemnos Project which is another DOE sponsored projects under the CEDS program. During this effort for commercialization the Energy sector requested similar products to

be included in this effort. The exe-Guard team expanded the commercial release to include six products SEL-3620, SEL-3622, SEL-3610, SEL-3530, SEL-3505, and SEL-3555 without requiring project schedule or budget to change. This expansion was due to the simple design and scalability of the technology integration and the overwhelming industry demand for the technology in this products. The team completed the development meeting all end user expectations releasing a zero setting malware protection solution protecting root kit, application whitelisting, digitally signed firmware, and memory mandatory access controls. The boot times were less the 5% additional and processor burden is less than 4% additions. This is accomplished by compiling the whitelist technology into the firmware build of the product and whitelisting the released functionality of the product.

After commercial release the team provided industry education and outreach through normal SEL sales channel and industry conferences. Exe-Guard was presented at two conferences. This industry education generated lots of interest which were addressed by the team authoring a white paper explaining the functional concepts of the malware protection and whitelist architecture and application notes detailing how to deploy and monitor the technology.

Please see attached whitepaper authored by the exe-Guard team explaining the technology in more detail, “Whitelist Malware Defense for Embedded Control System Devices”, Josh Powers and Rhett Smith, Schweitzer Engineering Laboratories, Inc.

The project team identified work within scope and added these tasks to the SOPO while still remaining on budget to the original budget. This allowed more work than originally planned to be accomplished without any budget adjustments. The team did request no cost extensions of the project management plan but did not cut any tasks from scope.

Please see additional technical details on this work within scope in the attached whitepaper, “Implementing a Vertically Hardened ICS/SCADA Control Stack: from kernel to application runtime”, Sergey Bratus Dartmouth

Products Developed Under Award

The exe-Guard project is a cooperative R&D project to accelerate commercialization of advanced cybersecurity technology focused on the Energy sector. The technology developed under this contract is fully commercialized in six products from Schweitzer Engineering Laboratories, Inc. to date and there are plans for more products to be released with the technology integrated in the near future. These products can be found on the SEL website located here

https://selinc.com/products/3620/

https://selinc.com/products/3622/

https://selinc.com/products/3610/

https://selinc.com/products/3530/

https://selinc.com/products/3505/

https://selinc.com/products/3555/

The literature required to support these commercial products including datasheets, manuals, product flyers, and installation guides can be found at these links as well.

The exe-Guard project published one whitepaper titled, “Whitelist Malware Defense for Embedded Control System Devices” and can be found at https://cdn.selinc.com/assets/Literature/Publications/Technical%20Papers/6676_WhitelistMalware_JP_20150120_Web2.pdf

This paper was presented at the 2015 Power and Energy Automation Conference in Spokane Washington.

SEL also published an application guide titled, “Incident Response Planning for exe-Guard” to answer the question industry asked, when I get an exe-Guard log what should I do. This application guide can be downloaded from https://selinc.com/

This project fostered lots of new industry collaboration as well as strengthen existing partnerships. SEL participated in the development of more commercial ready Autoscopy technology using the Dartmouth and UIUC code base and streamlining it to reduce the burden on the processor. This makes it easier for other suppliers to use it in their products. SEL and Sandia strengthened their partnership that started in the Lemnos DOE CEDS project and had the red team from Sandia onsite at SEL for a week working with the developers to make more secure code.

There were no patents applied for under the project and the efforts were structured to be fed back to the open source on the relevant packages like ELFbac and Autoscopy.

Industry Validation Testing

The exe-Guard team collected the industry requirements as one of the first tasks of the project. This was done by communicating with Dominion Virginia Power (DVP). They then participated in the project development monthly calls to help the team weigh the technical trade-offs and select the best priorities. Once development design was over SEL and Sandia traveled to DVP to present the vision for the final product. With approval from DVP the development team set of the complete the commercialization. Once the product released a firmware image was sent to DVP for their lab testing to upgrade an existing SEL-3610 product they had with previous firmware without exe-Guard to the newly released firmware that included exe-Guard for the SEL-3610. DVP tested that the same functionality was supported as they had previously and that the communications through the product was at or better than before. They tested that the firmware upgrade process was easy and the settings were preserved. Then finally they tested the burden on the product to make sure it could stand up to the operation requirements they had at worst case and reliably worked in all conditions. Once all of this past they accepted the engineering change order and started deploying the technology at a pilot site. This pilot site ran for many months before expanding the deployment to the other facilities. All stages of this testing was successful. SEL worked with many other customers and they have successfully repeated this testing and

deployed the technology on their power systems. Exe-Guard is successful because industry has tested and deployed it. Exe-Guard is broadly deployed protecting our nation's power systems today.

Conclusion

The exe-Guard project completed all tasks in the original statement of project objectives and some additional work within scope. The budget covered all work and no additional federal funds were required. The team collaborated to design the most scalable, economical, and safest malware protection solution for embedded control systems on the market today. This was accomplished by having no settings, able to be applied to equipment already deployed and is offered at no additional charge in the products commercialized. The industry has tested and deployed the resulting exe-Guard technology in large scale. This effort awarded in the CEDS program accelerated the development and deployment of advanced cybersecurity technology that is actively protecting our critical infrastructure today.

Whitelist Malware Defense for Embedded Control System Devices

Josh Powers and Rhett Smith
Schweitzer Engineering Laboratories, Inc.

Presented at
Saudi Arabia Smart Grid 2015
Jeddah, Saudi Arabia
December 7–9, 2015

Originally presented at the
Power and Energy Automation Conference, March 2015

Whitelist Malware Defense for Embedded Control System Devices

Josh Powers and Rhett Smith, *Schweitzer Engineering Laboratories, Inc.*

***Abstract*—Malware protection is a necessity for any electric device in modern critical infrastructure. We must all protect our critical cyber assets with antivirus as North American Electric Reliability Corporation (NERC) CIP-007 R4 states, but more broadly, we must protect our assets from malicious code infection regardless of whether they are identified as critical assets or not. Embedded devices and traditional personnel computer devices should be protected. The Stuxnet worm demonstrated that air gaps and unplugged devices are not immune from infection. We must engineer devices and systems to protect against the impact of malware.**

Traditionally, this protection was accomplished by using blacklist technology, where the technology watched for known bad code and blocked it. This resulted in a race to update malware protection technology when new threats were discovered, before infection happened. With malware statistics topping 83 million pieces of code, based on the August 2014 McAfee Labs Threats Report, and growing every day, the administrative task is impossible to keep up with. This design also can put excessive burden on processors, slowing computations and communications.

New malware protection technology is designed using a whitelist architecture that only allows known good code to execute on the device. This simplifies administrative overhead because new updates are not needed when new malware is released. A control system environment is built with application-specific devices that are set to accomplish one or more tasks and left alone to continue accomplishing the same tasks for many years, setting a perfect stage for whitelist malware protection technology.

This paper investigates the benefits that whitelist malware protection provides at the application layer (similar to existing anti-malware technology) and explains why embedded devices need architecture-specific malware protection. The paper shows that correctly combining malware protection and embedded architecture improves the reliability and cost of ownership of the whole system. The paper also highlights the enhanced security that whitelist malware protection provides over traditional solutions and how these principles apply to computers and embedded devices. The paper shows how whitelist malware protection meets and exceeds the NERC CIP requirements in Versions 3 and 5.

I. Introduction

Malicious software, or malware, is a tool often used to compromise the integrity of software or hardware. It is primarily used due to the power of automating the reconnaissance, infection, and compromise of a wide selection of targets. Simply put, malware can automate the exploitation of a system and do it much faster than one person.

Strict laptop computer usage policies and constant malware protection updates are protection methods that already exist in the electric sector. Malware trends have moved from targeting code flaws to enticing people to click links in emails or visit infected websites. Corporate infrastructure protection plans and technology are mature and established for malware protection. So how can we bridge the gap between corporate infrastructure protection plans and malware protection for control systems that consist of embedded, application-specific devices, many of which run on real-time operating systems? This changes the game completely because we are now talking about an infrastructure that is built for machine-to-machine (M2M) communications that have to meet high-reliability and availability requirements with very little downtime tolerance in control systems where physical consequences to cyber exploitation exist. Malware protection solutions have to support safe and reliable operations and work with the attributes of the system they are applied to. This leads us to the conclusion that the solutions designed to protect corporate systems are not a good fit for the control system due to the vast differences in their attributes.

Whitelist malware protection provided at the application layer is a viable solution for control systems. This paper discusses the enhanced security provided by whitelist malware protection compared with traditional malware solutions. It further discusses how the combination of malware protection and embedded architecture can improve the reliability and ownership cost of an entire control system. The paper also discusses the implications of whitelist malware protection on North American Electric Reliability Corporation Critical Infrastructure Protection (NERC CIP) requirements in Versions 3 and 5.

II. Increasing Malware Risks in Control Systems

Increasing demands on power systems today are creating more opportunities for malware infections. Smart grid is a term that has many definitions, but all of those definitions can be boiled down to advancements in control, measurement, and operations to automate new functions or previously manual ones. These advances have increased the number of electronic devices and the amount of code in the devices that make up power systems. They have also increased the communications links between all of those devices. These factors have increased the attack surface and the potential spread of malware by providing more targets, entry points, propagation paths, and potential vulnerabilities.

Based on the research results of McAfee Labs in their August 2014 quarterly threats report, malware is increasing on average 100 percent per year, and that trend is accelerating [1]. Malware developers have a lucrative market and are able

to sell their malware for Bitcoin or other currency. In comparison to the average $60,000 starting salary of a software engineer in the United States, and considering that there have been very few successful convictions on malware charges in the court system, illegal software engineering activities do not have strong enough deterrents to stop malware from being created. It is easy to see that ethics are the only thing stopping more people from making a career out of malware development. The writing of malware has gone from the curious and smart just wanting to see what they can do to organized criminals and nation-state actors with financial and political agendas offering advanced training and recruitment programs. Protecting power systems from these motivated and advanced sources is challenging, but we have the advantage when we design and engineer systems with protection capabilities that leverage the core attributes of the power system.

III. Power System Attributes Perfect for Cybersecurity

Power system networks are not like corporate information technology networks because they have a unique set of attributes that make information technology (IT) cybersecurity technology an imperfect fit. Building a cybersecurity program around these unique attributes provides the long-term stability and core foundations that we can use to advance cybersecurity to new levels. The control systems operating power systems are engineered with a specific purpose and are built to the highest levels of reliability. Each piece of technology, communications session, and data set is implemented for a reason. Every device used on the power system is carefully engineered and has a specific task. Each task is carefully programmed and then, in most cases, left alone to run for many years. This provides a baseline behavior for the device (how long it takes to respond, the amount of data served, what other devices talk to it, or what other devices it responds to).

Because the control system is built with M2M applications, baseline behaviors will not change unless the owner changes the services or devices on the system. These changes are rare in comparison to corporate IT infrastructures, so they can be managed with good change control policies and planned for in order to accept a new baseline. This level of understanding is the cybersecurity advantage. The best defense is to know the system, establish methods and means to monitor what is on the system, and react to undesired events. Instead of watching for bad code on the device, operators monitor and confirm that only approved devices and data are on the system. This provides the platform to protect against known and unknown malware. It also provides measurable success criteria for system uptime, reliability, and service provided, giving purpose to the engineers that operate the power systems on a daily basis. Baselining such as this results in metrics for asset management that inform operators what devices are approved on the system and that those devices are operating correctly. Communications outages are captured and unauthorized devices are logged. When systems are understood and monitored to this level, it is extremely difficult for attackers to hide their actions.

Power systems consist of many control and monitoring devices that are application-specific technology or embedded devices. These embedded devices have a variety of microprocessor architectures and operating systems. With a whitelist malware protection approach, we have the device operations and communications that can be monitored to confirm the system is doing only what is desired by the asset owners.

Even better, the operational and administrative management (OAM) costs are very low when technology applies safeguards in a whitelist architecture because it is locked in by the manufacturer. The only time the footprint changes is when a firmware upgrade is performed. There are no requirements for signature or patch updates as new malware is released. The devices are purpose-built, so the running of specific tasks and the communications are consistent. Specific protocols are enabled and turned on for a task and allowed to run continually for that task, enabling a communications baseline to be established. Control system devices on the power system measure the power at various distributed geographic locations, and any change in measurement will be seen by the operators or the automation schemes, triggering an event response action.

Based on the native attributes, the power system is perfect for some of the most advanced cybersecurity ever seen. Two simple protection methods contribute to this level of cybersecurity: whitelisting and deny-by-default. A whitelist approach is the method used to ensure that only desired devices, communications, and data are present on the system. The keys to its success are knowing what is on the system and knowing what each device is doing (this is the baseline). The deny-by-default method requires each device and communication to be off unless explicitly turned on for a purpose. In power systems, the advantage goes to engineers and operators when they know their system, establish a known good baseline, and have methods to ensure that this baseline is preserved.

IV. Power System Attributes Challenging for Cybersecurity

The foundations of the control system architecture enable the industry to advance cybersecurity to greater levels than corporate networks, but there are specific challenges we must address to get there. Power systems are built with many embedded control and measurement devices. Embedded devices are not open computer platforms that allow the end user to install new software. The software running on these devices is produced by the manufacturer, and steps are taken to ensure that no new software can be installed. This is good and bad. The good part is that malware is software trying to install itself on these devices, so the architecture is already safeguarding against this. The bad part is that the end user has limited visibility of what software is running on the device. This is important for patch management procedures. The end user now has to establish monitoring processes for the

manufacturers they have purchased products from and rely on these manufacturers to not only alert them when security vulnerabilities are discovered but release the mitigations in a timely manner.

Another challenge is the availability requirement for control systems. There is very little tolerance for downtime. Any reboot or decommissioning to take a product out of service for updates costs the company money and increases safety hazards. These updates need to be planned and tested well in advance, which will result in a slower deployment time between when security vulnerabilities are fixed and when they are deployed on the control system. The best mitigation to this is to select devices that accomplish the job they are intended to do with as small a code footprint as possible. These devices all work as a larger system, so many times when taking one device out of service for maintenance, the overall system suffers.

It is good that these systems have many channels for monitoring, and that operators watching the system understand event response plans. The challenge comes with change control. When changes are made, alarms and logs generated by these changes need to be expected or operators will waste time investigating them, or worse, will get comfortable seeing alarms and not respond. Most importantly, these systems are built for reliability and use redundancy to meet extreme reliability requirements. The contingencies to any change must be planned and well understood before the change is applied. This mandates that lab testing and system validation testing be performed and that engineering standard documents inform work instructions to prevent any undesired operations.

V. Malware Protection Architectures

There are four common means of protecting a system from malware that we look at in this paper: blacklisting, whitelisting, mandatory access control (MAC), and rootkit prevention.

A. Blacklisting

Blacklisting is the traditional approach used in corporate environments to protect computing resources. In this environment, systems change frequently to support corporate requirements. Blacklisting works well in these environments because updates are easily managed and automated. When new malware is detected, a signature is created and all of the clients receive the new signature. Blacklisting has a long history and has been shown to work reasonably well in many cases. The signatures are stored in large proprietary databases that are updated regularly with the newly detected signatures. Because new signatures are created regularly, a system with a recently updated signature database must scan all files and processes on the device in order to check for possible infections it did not previously know about.

B. Whitelisting

Whitelist anti-malware creates a signature for all of the allowed software on a system and assumes that the system will rarely change. This means that programs cannot be installed or modified without updates to the whitelist. Whitelist protection is fairly unexplored because it is difficult to manage in corporate environments. There are also two kinds of whitelist anti-malware. In some systems, the whitelist can be modified in the field by entering a password. This type of system is used in some corporate environments. The other way to use whitelist anti-malware is to cryptographically sign the files with a public and private key pair and keep the private key elsewhere. This type of system is more difficult to update because any updates have to be signed before they can be brought out into the field. It is more secure in the field because the secret protecting the whitelist security is not kept on the device being secured.

C. Mandatory Access Control

MAC has a lot of support in the open source community and has a strong backer in the National Security Agency (NSA). MAC works by segregating applications into separate domains of execution with very specific permissions granted to those domains. This is in contrast to discretionary access control (DAC), which is the default system used by most operating systems. Fig. 1 shows that with DAC systems, permission levels lower on the list have access to anything above them. Kernel can access root and root can access anything user specific. While the kernel still has access to anything in user space with MAC, each user space application is segregated into separate domains that all have limited, specifically granted permissions to each other. This narrows the scope of an exploit, limiting its reach to only the permissions the original domain had. Before, if the root layer was compromised, the entire user space was compromised.

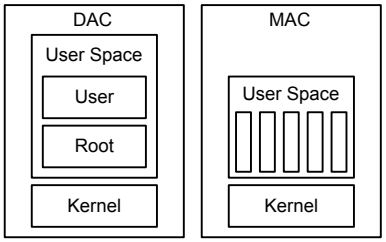

Fig. 1. DAC and MAC Protection Architectures

D. Rootkit Prevention

Rootkit prevention is the newest and most unexplored area of malware protection. It works by attempting to ensure that drivers and kernel modules come from a trusted source and, if not, preventing their use entirely. Drivers and kernel modules can both circumvent many other security measures because of their access to the kernel or operating system. Some rootkit prevention systems also attempt to verify that system calls have not been modified or interfered with. Adding a hook, a piece of code that runs when another function is called, to a system call is a common means of getting a rootkit into the kernel, and it is exceptionally difficult to detect.

VI. Evaluating Security of Malware Technology

Each of the previously described methods that are used to protect a system from malware has advantages and disadvantages.

A. Blacklisting Benefits and Drawbacks

Blacklisting is a reactive approach. It suffers from zero-day vulnerabilities because of this, but it does have the advantage of experience. It also benefits from the fact that specific attacks can be countered once a new signature is created. The biggest disadvantage, however, comes from the requirement to update the blacklist antivirus signature database on a regular basis in order to maintain its effectiveness. This is a poor design in an embedded system where updates are costly and infrequent. As mentioned previously, when a signature database is updated, a scan of the system must be performed to ensure that an infection that was previously undetectable is now detectable. The problem, though, is that embedded systems generally perform a specific task, often with real-time constraints. They are generally not engineered with occasional central processing unit (CPU) spikes and I/O-intensive disk scans in mind. This means that simply updating the signature database could degrade the ability of an embedded system to perform its main function for a period after the install. These limitations make blacklist anti-malware unsuitable for use in embedded systems.

B. Whitelisting Benefits and Drawbacks

Whitelist anti-malware maximizes safeguards while minimizing the administrative overhead in purpose-built M2M infrastructures. Whitelist malware protection is optimized for control systems instead of corporate information systems because of the reduced change management requirements on control systems. When there are frequent changes in what each devices does, whitelist malware protections become administratively burdensome. Each time an update is made to the system, an update must be made to the whitelist signature database as well. This is cost prohibitive in corporate environments, where updates to software are frequent. However, in an embedded system, updates to software are infrequent and it is not infeasible to include updates to the whitelist signature database when updates are made. In fact, it is often the case that updates to software in an embedded system are done via firmware image updates that can include the whitelist signature updates as well. The strong suit of whitelist anti-malware is that it does not require periodic updates to keep up with recent malware activity and does not suffer from zero-day exploits, except those against the whitelist anti-malware software itself.

Another weakness of whitelist anti-malware is that it cannot perform checks on running software. Once a piece of software has been loaded into memory, whitelist anti-malware can no longer say anything meaningful about its integrity. This means that a whitelist anti-malware solution cannot protect against malware that exploits things like buffer overflows, except to contain the exploit to only the running process that was exploited. Whitelist anti-malware does prevent an infection that has been persisted to disk from running.

C. MAC Benefits and Drawbacks

MAC takes a different approach to security than that of either whitelist or blacklist anti-malware. Instead of attempting to block the execution of a program, it attempts to constrain the reach of running software. All resources on a system are placed in a predefined domain and domains are then given specific access to other domains.

There are many MAC implementations, but the most common MAC systems are AppArmor and Security-Enhanced Linux (SELinux), both of which provide a similar result when properly configured. Individual executables are limited to the minimum set of permissions they need to do their job. This means that an exploited process will have limited reach and will be less likely to corrupt a system or prevent it from performing its primary function.

The downside to MAC is that it is very difficult to configure correctly, and mistakes in the configuration may not be detectable without a significant design effort. Fortunately, the effort of setting up MAC for an embedded system falls to the company creating the firmware, and they generally have the information required to correctly configure MAC. In the corporate environment, MAC is much more difficult to configure because small changes in the system can be difficult to adapt to in the MAC policy. Another problem with MAC is that it makes no attempt to verify the integrity of the process being placed into a domain. This means that an exploit that can modify the file system can persist its infection and perhaps spread by infecting other executables on the system, increasing its reach over time.

D. Rootkit Prevention Benefits and Drawbacks

Rootkit prevention provides yet another approach to securing a system. Generally speaking, operating systems are divided into two segments: the user space and the kernel space. User space tools are kept secure largely by software running in kernel space. Requests for access to all resources on a device go through the kernel, so it is the logical place to provide security. However, the kernel can be compromised, so a layer of security to attempt to detect these types of attacks, called rootkits, is needed.

Rootkit prevention is difficult at best because there is no other layer managing and monitoring the kernel, so the kernel must attempt to monitor itself. There are two common kinds of rootkit preventions. The first one attempts to verify that syscalls from user space to the kernel are not tampered with and the other attempts to verify that drivers and kernel modules that are loaded into the system are not malicious. The second kind is a form of whitelist anti-malware for drivers and kernel modules. Something similar is already implemented in Microsoft® Windows® systems, but not in Linux® systems, by default. The first type, however, is more difficult and is the subject of current research.

VII. Layering Malware Prevention Technologies

All of the malware prevention technologies we analyzed have strengths and weaknesses. A solution we identified to prevent the weaknesses from being exploited is to layer malware prevention technologies. As mentioned, a whitelist antivirus system cannot prevent runtime exploitation of things like buffer overflows. Layering on MAC to limit the scope of access that an exploited running application has is a good solution. MAC has no concept of integrity when placing a particular binary into a domain and granting it the permissions associated with that domain, so whitelist anti-malware should be layered on to prevent modified binaries from loading. Neither whitelisting nor MAC can detect a compromised kernel, so rootkit prevention technology should be added as another layer to mitigate these vulnerabilities.

By layering these technologies, we found that a secure system that is compatible with an embedded environment can be provided. This solution provides the best level of integrity of the software running on the embedded system, and the reach of attacks can be minimized. Also, the layered solution provides ample warning of attempted infections so additional measures can be taken outside of the embedded system. If an attack is detected against an embedded system, the network firewall can be hardened to stop that attack in particular and then forensic data can be gathered on the attack and the affected systems can be patched and updated.

VIII. Long-Term Administration

Long-term administration of an embedded system running blacklist anti-malware requires frequent signature updates. Regular updates must be pushed to the embedded system and regular scans must be made to ensure infections do not already exist that were not previously known about. Device burden is also a large problem. As previously mentioned, regular system scans must be performed. These scans create a large, irregular burden to disk I/O and to the CPU. Recent measurements show that up to 95 percent of the CPU processing power can be consumed during a scan.

Long-term administration of a system using whitelist antivirus depends on the environment. In a corporate environment, where software is updated very regularly, the administration of whitelist anti-malware would be time- and cost-prohibitive. However, in an embedded system, administration is minimized and only needs to be done when the embedded system itself is updated, which is generally not often. Additionally, the maintenance of the whitelist generally would fall to the firmware provider, therefore decreasing the required maintenance further. An embedded system running whitelist anti-malware should require no intervention between firmware updates for a device owner.

Another consideration in long-term administration is burden to the system. The whitelist anti-malware system we tested saw a 15 percent increase to system boot time but only had a 0.5 percent increase in the time to complete a task during runtime. The reason for a large impact to boot time but a smaller impact to general running time is that the whitelist anti-malware we tested uses cryptographic signatures to verify integrity but also caches integrity lookups. This means that at first boot, the system must run cryptographic analysis on every executable, but after an initial check has been done, the CPU burden decreases. In our tests, SELinux showed only a 0.5 percent increased burden overall. SELinux has no cryptographic security, so it adds little burden. The rootkit prevention software we tested, a variation of the program described in [2], showed an overall 5 percent increased runtime burden. All told, this provided a system with about 20 percent increased boot time and 6 percent increased runtime burden. Because all of these times are constant, they are easy to account for in an embedded system in contrast to blacklist anti-malware, which has inconsistent burden on a system.

IX. Configuration Management

Another interesting benefit of whitelist anti-malware is that it can be used to protect system configuration. Any embedded system will have configuration files that are not modified in the field. Things such as boot order and the disk to be mounted are set up in the firmware and never modified by the end user. By modifying the system binaries that use those configuration files and having them request integrity scans of their configuration files, the configuration integrity can be guaranteed. We call this voluntary scanning. Because the individual executables have had their integrity verified, it can be guaranteed that they will voluntarily scan their configuration. Then, all that must be done is to create signatures for the various unchanging configuration files on the system, and the same whitelist anti-malware that protects the system executables and libraries can be extended to protect configuration and scripts.

Fig. 2 shows the general flow of whitelist integrity verification and voluntary scanning. An executable is loaded into memory from disk at the request of another process or user. The whitelist anti-malware automatically scans the executable to verify its integrity before it is allowed to be placed into executable memory. Once the application has loaded, it then attempts to load its configuration files. Because it has been modified to include voluntary scanning by the whitelist anti-malware system, it first requests that the whitelist system scan the configuration file to verify its integrity. If the configuration file has integrity, the executable is notified and continues to load its configuration.

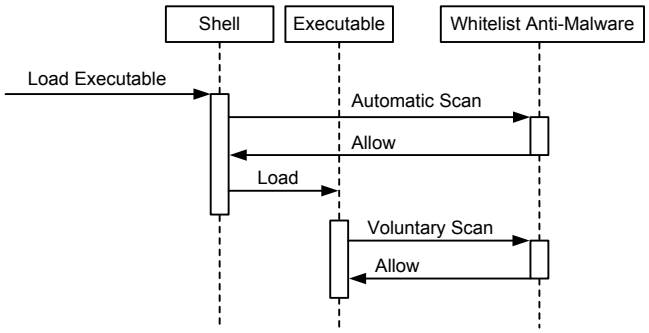

Fig. 2. Whitelist Integrity Verification and Voluntary Scanning

X. Compliance Considerations for NERC CIP Versions 3 and 5

NERC CIP from its inception recognized that devices need to have malware protections in place. The specific technology is not mandated, but the need for it is and the policies and procedures to keep the technology updated are mandated. NERC CIP Version 3 is specific to the device and states that every critical cyber asset (CCA) needs malware protection, and if the device cannot provide this, a technical feasibility exception (TFE) must be submitted. This was a huge generator for TFEs because many of these devices were embedded, so the end user could not install malware protection software.

NERC CIP Version 5 also requires malware protection and the procedures to keep it updated, but applies it to the system instead of the individual devices. This allows more freedom in the type of technology to select, and network-based technology can cover clients that do not have the capabilities to run malware protection.

For embedded devices, it is up to the manufacturer to provide the solution, either in the device or the system solution recommendations. When the whitelist malware protections discussed in this paper are implemented, the compliance to NERC CIP is accomplished. A small number of update procedures are required, keeping the operational costs low.

XI. Conclusion

Control systems are a very important area of focus for cybersecurity, and current malware protection technologies are not ideal in that environment. However, with the proper application of various anti-malware techniques such as whitelisting and deny-by-default, a control system can be reliably secured. This paper shows that control systems are an ideal candidate for cybersecurity.

Future research is still needed into rootkit protection to find ways to secure the kernel further and constrain exploits to the smallest area possible.

Control system owners reduce the total cost of ownership and improve cybersecurity by selecting technology from manufacturers investing in anti-malware solutions with whitelist architectures.

XII. References

[1] McAfee Labs, "McAfee Labs Threats Report," August 2014. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2014.pdf.

[2] J. Reeves, A. Ramaswamy, M. Locasto, S. Bratus, and S. Smith, "Intrusion Detection for Resource-Constrained Embedded Control Systems in the Power Grid," *International Journal of Critical Infrastructure Protection,* Volume 5, Issue 2, July 2012, pp. 74–83.

XIII. Biographies

Josh Powers received his B.A. from Washington State University in 2008. He joined Schweitzer Engineering Laboratories, Inc. in 2010 as a software engineer and has focused on power system network security research. He has broad experience in the field of software engineering, but has focused largely on security and networking. He spent most of his time during college and shortly after working in public sector information technology, focusing on network security. Josh is currently working on his M.S. in computer science.

Rhett Smith is the development manager for the security solutions group at Schweitzer Engineering Laboratories, Inc. (SEL). In 2000, he received his B.S. degree in electronics engineering technology, graduating with honors. Before joining SEL, he was an application engineer with AKM Semiconductor. Rhett is a Certified Information Systems Security Professional (CISSP).

Implementing a vertically hardened ICS/SCADA control stack: from kernel to application runtime

Sergey Bratus

January 21, 2016

Architectural summary

The primary principle of our study was to implement *vertical integration* of security measures: a set of features distributed across the systems components and supporting each other to implement a security policy operating from the kernel boot time to throughout the control application's runtime.

In particular, we combined the intra-process ABI-based memory protection technology, *ELFbac*, the input-validation assurance methodology of *LangSec* (both originally developed by us), with the proven *Grsecurity/PaX* Linux kernel hardening.

Each of these technologies can be applied independently, but they work best in concert. ELFbac enforces programmer intent with respect to data flows between the application's intra-memory code and data units on the ABI level, whereas Grsecurity/PaX's UDEREF feature enforces such intent between the kernel and the userspace. In turn, the LangSec methodology of application design ensures that the security-critical units that handle input validation can be properly separated to take the best advantage of ELFbac policies to mitigate potential exploits targeting these units via crafted inputs.

In order to showcase the composition of these approaches, we built a Linux-based filtering proxy for the DNP3 protocol. This proxy applies *exhaustive inspection* of the DNP3 frames before passing them between an outstation and a master controller; by this we mean full inspection of all syntactic elements of the protocol, such as headers and objects, and all relationships between them specified by the protocol documentation than can be expressed in the form of a grammar. This methodology brings the absolute majority of the DNP3 frame validity checks forward, into the input-checking unit we call the *recognizer*; this unit is then isolated by an ELFbac policy from the rest of the processing.

The choice of the filtering proxy as the test application for the stack policies was made with the view towards generality: our proxy code is meant to be extensible to other applications that act on DNP3, such as rewriting proxies and adapters to other protocols. Such extensions would replace the processing part but keep the input-validating, parsing part of our code.

Working with a prototype that nevertheless implemented over 70-80% of the underlying complex protocol specification in a functional, testable way gave us a complex enough structure to test our security policies. The summary of lessons learned follows.

Recommendations for commercial development

The ELFbac model is recommended for the industry developers who seek to leverage both the existing Linux ecosystem and the best-of-breed Linux kernel self-protection of Grsecurity/PaX, while remaining compatible with the standard C/C++ ABI and its build chain. ELFbac is complementary to the mandatory access controls (MAC) such SELinux, which operate at the granularity level of an entire process, and can be combined with an ELFbac intra-process policy without any additional costs.

More specifically, SELinux policies apply to system calls issued by a process entirely based on the process' SELinux identity label, regardless of the order of issue or of the particular code executing in the process at

the time of the system call. For example, an allowed system call can be issued either by the main executable or any of its loaded libraries, at any stage of the process' timeline; it is all the same to SELinux.[1]

In short, SELinux and other MAC schemes do not concern themselves with the order of system calls not with the order of memory accesses that happen inside a process. For control processes that naturally contain distinct phases of operation, this level of control is clearly not expressing the programmer's intended operation of the process.

ELFbac is the only policy of its kind that allows the developer to enforce the *intra-process* access control between the structural units of a program at its runtime—such as between libraries loaded into the process and their sensitive data to which these libraries are intended to have exclusive access. Further, ELFbac's intra-process policies isolate sensitive code units from accidentally operating on untrusted, un-validated, potentially maliciously crafted data, and the sensitive data units from being accessed or operated upon code units not intended to do so.

Thus we recommend the ELFbac protections for any front-end ICS/SCADA systems facing untrusted data. Although an ELFbac policy requires knowledge of programmer intent of the program's units, such as the sequence in which these units are expected to execute and the kinds of data they are supposed to execute on, these intents are typically easily observable at the development time and well-understood by the programmers.

An explicit enforcement mechanism for these intents will both help the programmer catch errors and help communicate these actual intents to Unix runtime, where they are currently almost entirely ignored. This ELFbac achieves at the cost of at most few lines per compilation and scoping units of a program what modern programming languages strove to achieve via new language semantics means: an enforcement of existing and clear intent at runtime—while, unlike most of them, maintaining compatibility with the core C/C++ ABI, binary OS utilities, and the build chain tools.

Lessons learned

Modifying programs to support ELFbac policies. Our DNP3 proxy code was developed with a view towards being deployed under an ELFbac policy, but was functionally completed and unit-tested separately from the kernel work. This was a deliberate choice, because we wanted to apply the policy to an existing, functional application and thus observe the integration effects and challenges.

The resulting integration effort required, besides including one-line ELFbac unit policy annotations per unit, several small modifications to the application's memory allocation code. In particular, the original code performed all of its allocation from the common heap, including the raw input buffers.

Since ELFbac policies specifically enforce such relations as intended exclusive access to raw input buffers only by the parser—the rest of the program should only consume validated, well-typed objects created by the parser—we changed the allocation of the raw input buffers to a separate static arena labeled for ELFBac. The change was minimal, and applied to the proxy "driver"[2] code only (since the underlying *Hammer* parser construction kit we used works with externally provided buffers).

We also noted that the Hammer kit and therefore the parsers derived from it can also work with externally provided allocators, which, unlike many other such tools, allows separation of the parser's own scratch space from the rest of the program's memory objects involved in processing, affording another degree of protection from the parser's potential weaknesses.

In all, the amount of code changes to the application to enforce intended separation of its input-validating front end from its proxy back-end was small and trivial, and did not apply to either the parsing or processing logic, but rather to the overall "driver" code.[3]

[1] For this reason, SELinux permission policies are colloquially described as a "bag of permissions", per process.

[2] I.e., code unit containing its *main()* function.

[3] We also derived a non-trivial observation from refining this policy to isolating not just the raw buffers from the rest of the program to the parser's exclusive access, but also isolating the parser's own "scratch space" from the validated objects that it prepared. Namely, validating parsers that must backtrack (a necessity for protocols with context-sensitive syntax elements such as DNP3) need an additional deep-copy step to gather the parsed elements—and only those elements—into a "ready zone" that alone is exposed to the rest of the program. This happens because a backtracking parser cannot know when allocating an element whether that element will be taken up in the final parse or discarded during a backtracking step. This is an argument for simpler parsers with unambiguous grammars that do not need to backtrack.

Synchronous vs asynchronous I/O: the trade-offs. Synchronous Unix-style I/O is the easiest to read and write and thus the easiest to adopt. It also fills the need of protective traffic proxying in many circumstances. However, we also want adoption of our framework by those who need performant servers serving many connections at once—and this means asynchronous I/O, even though its overall code structure is less transparent.

The lesson we learned after considering a number of designs was that the parser should be *decoupled from the I/O model* (blocking on non-blocking event loops), as well as from the thread/process models. More specifically, it must support non-blocking, chunk-wise processing. This is the design that finally implemented.

For parsers that serve as front-ends to non-trivial semantic actions on the parsed objects—such as adaptors to different protocols, or rewriting DNP3 normalizers—the most natural parser design is to queue semantic action, including output operations. In particular, our interface for both message pretty-printing and rewriting uses queued write callbacks, which are called when a recognition verdict is rendered by the parser code.

This decouples our parser from the specific I/O and concurrency models at a negligible complexity cost.

ELFbac's additional guarantee: control over pointer dereferencing. Dereferencing pointers in the wrong context is a known security concern. One additional power of ELFbac is that, even though it does not control the passing of pointers to system calls and program units, it will trap attempts by wrong code units to dereference these pointers if they point to an ELFbac-labeled data section (if the dereference means an access against the policy).

For example, the proxy's "driver" code that coordinates the parser never dereferences pointers to the buffers it gets back; it passes them to system calls unchanged.

Since ELFbac also allows its policies to reduce the system calls made by each code unit[4], an ELFbac policy can and should reduce the parts capable of making syscalls. Such parts should act a *bridge* to the OS, decoupled from the main computation and possessing least memory access privileges.

Our application is constructed as just such a bridge.

Filtering vs transforming traffic. Our additional lesson is that although a per-message pass/fail decision structure of the code for a proxy might seem simple, it is not actually so in practice. In fact, a structure suited to transforming the messages is actually a simpler even for pure filtering purposes and a more sound choice to implement. In a nutshell, an abstract parser API should support protocol normalization (ours does).

This is because rendering a pass-fail decision would require the parser to be aware of the network session and context, especially when the desired behavior is to drop only some messages selectively, not the entire connection. This is precisely what happens in a DNP3 TCP session shared by several endpoints.

Handling simultaneous connections, multiple endpoints on a single stream reminiscent of another stack on top of TCP is in fact typical for ICS/SCADA, and is naturally accommodated by our design.

Validation and resource control. Small payloads in DNP3 (and other ICS protocols) can result in construction of large objects, consuming resources on the receiving endpoint. Thus a resilient application must resist resource consumption attacks.

From testing our application both via fuzzing and know-bad payload scenarios, we concluded that building the parser as a modular structure allows flexibility w.r.t. semantic object representation (which may be resource-intensive). Creation of these representations must be explicitly a part of the semantic actions, separate from the recognizer.

This encourages moving all possible checks that logically precede the creation of an object into the recognizer—which is where it really belongs, since no objects should be constructed (or acted upon) until the incoming data that describes them is validated.

[4]and even by the process state, of which several different states may correspond to the same code unit, differently reached.

Applying the LangSec methodology to DNP3: a deeper understanding of the protocol. LangSec is a mission-assurance methodology for software that must handle inputs safely. In a nutshell, LangSec posits that the design of the input-handling software must start with analyzing the grammar of the protocol, gathering all syntactic validity requirements under this grammar, and writing the parsing/validating code to resemble this grammar as closely as possible. The latter is best achieved by employing the *parser combinator* approach, which has been implemented for a variety of production languages, including C++, Java, Python, etc.

LangSec further posits that more syntactically complex protocols are liable to lead to buggy and exploitable implementations. When the nature of validity checks for protocol messages is unclear, the likelihood of the implementation actually failing to check what the subsequent code assumes it had checked increases—indeed, becomes very hard to avoid and consistently eliminate, as the example of SSL/TLS implementations shows across the board.

Therefore, LangSec recommends syntactically simpler formats that stay within the regular or context-free classes of languages.[5] On the contrary, syntactic elements of a protocol that introduce context-sensitivity must be avoided and handled with utmost care—and, if possible, filtered out from the language. A LangSec analysis of a protocol tends to point out these elements as pitfalls; a large set of famous vulnerabilities upholds this analysis.

We applied this analysis to DNP3, coming up with the grammar for its frames based on the protocol specification, and extending this grammar with the specification's requirements wherever these requirements touched on mutual co-occurrence or relationships between elements. In doing so, we found a number of problem spots, which, not unexpectedly, corresponded to many vulnerabilities found in DNP3 parsers by previous fuzzing efforts (see below).

Our DNP3 parser was built according to this grammar, utilizing the Hammer parser construction kit, to which we added several combinators needed for DNP3 specifically.

We plan to publish a series on articles on the syntactic pitfalls of the DNP3 specification—and, therefore, of its implementations. We also identified a simpler subset of DNP3 that has much more easily parseable syntax, to be described in a separate publication; we recommend using this safer subset for future protocol development.

Leveraging Grsecurity/PaX, a state-of-the-art kernel hardening technology. Grsecurity/PaX is the industry's leading kernel hardening technology. Meant for general-purpose systems, the Grsecurity/PaX patch is sufficiently more complex than it needs to be for control systems that do not need, e.g., to support just-in-time compilation (JIT) at runtime and can place other restrictions on the applications as a matter of policy.

A version of Grsecurity/PaX offering a cohesive subset of its protective features has been ported by the Grsecurity/PaX team for the ARM platform of our project.

We integrated our ELFbac kernel mechanism with it without conflict, thus leveraging the strongest practical kernel self-protection mitigations in industry into our prototype hardened stack.

Challenge: Separating allocation of address space from the allocation of memory is a useful design pattern. Finally, the ELFbac approach favors arena-based memory allocations rather using the same genetic heap and *malloc()* for all allocations.

This should not, in theory, present a problem on Unix systems, since physical memory pages are only committed on a page fault; but in reality the Linux kernel does not favor preallocations of large blocks, as these run afoul of the *ulimit* and other memory limiting checks for processes, fixed by internal parameters—so preallocation should be tested and these limits adjusted from stock kernel defaults accordingly.

Robustness evaluation

We subjected the proxy to fuzz-testing by both the *Aegis* specialized DNP3 fuzzer developed by Adam Crain and Chris Sistrunk—with which they demonstrated that most commercial DNP3 implementations

[5] Such as those that can be fully parsed by an anchored regular expression or a pushdown automaton, respectively; JSON is an example of the latter.

were vulnerable to attacks via crafted inputs—and with the *American Fuzzy Lop* fuzzer from Google's Micha Zalewski, a state-of-the-art fuzzer that learns from source code.

Despite vigorous testing, our application did not reveal any bugs beyond the classic resource-exhaustion attack that we subsequently fixed. Although we hoped for exactly this result due to our parser construction methodology, we note that few commercial implementations approached it in Crain's and Sistrunk's testing—and those that did, unlike ours, implemented small and restrictive subsets of the DNP3 protocol.

Validation methodologies

While fuzz-testing described above provides an empirical evaluation of the system's overall robustness, the individual components must have their respective unit-testing and validation methodologies that can be applied to them in isolation. The following describes how our designs provide for it.

ELFbac policies. An ELFbac policy is essentially meant to contravene any accesses by code units to data units not allowed by the policy. Since both code and data units are, in fact, ABI units of the executable, the relative positions of either the reference or the referent within their respective units do not matter, so long as these are placed within their boundaries.

Consequently, the easiest and most comprehensive way to test an ELFbac policy's efficacy is to insert memory references to disallowed sections into its code units, and assert the resulting memory traps.

Representing the policy's allowed accesses as a bipartite labeled graph between the code sections and the data sections of an executable, the exhaustive test is easily derived as the complement of that graph. For each specific edge of this complement graph, a unit test for the policy's intended rejection of it is easily constructed by placing a violating access instruction at the top of the particular code section.

LangSec parsers. The LangSec methodology for constructing input-validating parsers lends itself to exhaustive unit-testing by design. Namely, under this methodology, parsers for the whole protocol messages are constructed from the parsers for their simpler parts, and so on, down to the simplest elements such as integer and string fields. Consequently, for every protocol unit from the primitive types up, a definitive function that validates these elements and these elements alone exits, and can be tested independently of all others.

In other words, the structural units of the parser correspond to the structural units of the grammar correspond to natural unit tests for the functions implementing their parsing and validation.

Conclusion

We demonstrated that ABI-based intra-memory protection policies that do not place a considerable burden on the programmers so long as they follow reasonable design practices are feasible for control applications receiving and filtering the DNP3 protocol. The resulting implementation was robust and withstood vigorous fuzz-testing with state-of-the-art tools.

We also demonstrated that our approach composes with the state-of-the-art Linux kernel hardening technology, Grsecurity/PaX. A cohesive set of protection features has been adopted and reviewed for our project's ARM platform by the Grsecurity team.