IEEE CYBER SECURITY

Building Code for Power System Software Security

Carl E. Landwehr, *Cyber Security and Privacy Research Institute (CSPRI), George Washington University*
Alfonso Valdes, *Information Trust Institute (ITI), University of Illinois*

IEEE IEEE computer society

READ ON!

Public Access Encouraged

Staff

Brian Kirk, Manager, New Initiative Development
Jennie Zhu-Mai, Designer

TABLE OF CONTENTS

Introduction

Both the attractiveness of power systems as targets of cyberattack and their vulnerability to remote attack via digital networks are evident from recent world events. While policy makers seek means to deter such attacks politically, the most effective way to reduce their attractiveness as targets is to reduce their vulnerability to such attacks. This can be done; these are engineered systems built to satisfy specifications. The results of the workshop presented here aim to reduce the vulnerability of future power systems to remote attacks that exploit vulnerabilities in the code – software or firmware – that controls their operation. The approach taken is to develop a consensus "building code" for building the software that controls these systems. Such a building code can provide a basis for customers to specify the security required of power system software components, for vendors to produce them, and for third parties to evaluate important aspects of their security properties. The availability and use of such a code can enable the marketplace to reward producers of systems with stronger security properties.

BACK TO TABLE OF CONTENTS

Power System Context

This effort is motivated by the power system environment, including supply, demand, transmission, distribution, generation, smart grids, and microgrids, including residential use. The systems in this environment have requirements for both local and remote access as well as local and remote control. This access will be via networks that support digital communications. Some may be isolated, but some will be Internet-connected. To maintain the reliability and safety of these structures, cybersecurity is an issue of increasing concern in the power system environment as a whole.

In the realm of physical structures, building codes can incorporate a very broad range of requirements, from architectural and design requirements that apply to large public structures or neighborhoods to requirements on type and strength of materials to be used in construction. But a building code is not a design manual. It is a guideline that provides minimum expectations and recommended practices so that a building that conforms to the code should at least be safe and sound. Similarly, a software building code for power systems, while it cannot guarantee overall system security or reliability, will improve the security posture of the software and systems being developed in this industry. Software must, as always, meet organizational and operational requirements, mitigate threats, and minimize flaws.

BACK TO TABLE OF CONTENTS

The Need for a Secure Software Development Process

It continues to be the case that most successful cyber intrusions exploit vulnerabilities that were accidentally introduced into the software at the implementation stage, *i.e.*, when programmers convert specifications to code. For this reason, this draft building code focuses most strongly on techniques for preventing the introduction of such implementation flaws or for finding and correcting them. However, the consensus of this workshop was that there is a fundamental need for a secure software development process to be put in place to organize the production of software for power systems. Participants proposed that two flows of requirements must be conducted in parallel as part of this process:

System requirements → device requirements → software

Security policy → security requirements → secure implementation

Security Policy's Central Role

In this context, security policy becomes part of the system requirements, and system security must be seen as not only preventing unintended things from happening but also ensuring that the system does perform its intended functions. Security policy in this light becomes the statement of what it means for the system to provide service that is dependable and secure in the sense of [4], in which a protection mechanism (e.g., a circuit breaker) is dependable to the extent that it operates at appropriate times and is secure to the extent that it doesn't operate at other times. In this lexicon, a system is considered reliable to the extent that it is both dependable and secure.*

A system is secure only with respect to its stated security policy (and insecure only when and if those policy statements are violated). The specific security controls included in a system (e.g., authentication, access control, information flow control, cryptographic controls) are chosen in order to implement and enforce the policy.

Overall system design will determine whether software, hardware, or people operating the system are responsible for assuring that particular aspects of an overall security policy are correctly enforced. This document primarily addresses those aspects of security policies that are to be assured by software.

The building code can assist in the selection of proper controls to achieve the system's security policy as part of the software development process, just as codes for physical buildings assist the architect, developer, and builder in determining the safe width for stairways and fire exits. The essential first step in developing secure software is the security policy; the

* Other technical communities define these terms differently, e.g. [5]

BACK TO TABLE OF CONTENTS

remainder of this building code is intended as a guideline to assist in the selection of controls and implementation of the controls necessary to enforce the policy.

Minimization of Function

Among cybersecurity professionals, it is often said that complexity is the enemy of security [6,7,8]. Nevertheless, the economics of chip production and software production have led to the prevalence of computing hardware with broad capabilities and software that frequently includes many features and options bundled together. Features include a chip or application that the purchaser does not even know are present have often been exploited to penetrate a system.

Although not originally proposed as an element for the building code (and hence not included explicitly in the draft report), the principle of disabling unneeded / unused functions was part of the workshop consensus. Different functions of a device might be disabled according to the application in which it is to be used; the building code would apply to the software developed for the device regardless of its application. Note that if the software implementing a disabled function is not removed, care must be taken to assure that it cannot be activated through the exploitation of flaws elsewhere in the system.

Challenges

Expanding the scope of the software building code from a focus on elimination of implementation errors to include system security policies and secure software development processes is a significant step. While a single organization may be able to implement and control secure software development procedures for software it develops internally, it is difficult to find a product today that doesn't incorporate software developed by others, including software with roots in the community of open source developers. Assuring that all of the software in a system was developed in accordance with a particular secure software development process will be a significant challenge for most companies. (The requirement for a software "bill of materials" in the draft code will at least allow the sources of software to be identified.)

In general, there are three ways to gain confidence that a piece of software will function as specified. First, one may have confidence in the people who built the software, for example, if they have produced similar software in the past and it has performed well. Seeking this kind of confidence might lead one to establish certification processes for individuals and for identifying what software was produced by certified individuals. Second, one may have confidence in the process or methodology used to build and test the software. This approach leads to the secure software development process requirement

BACK TO TABLE OF CONTENTS

embraced by the workshop consensus, and might lead to certification of software development processes and identifying software that was produced in accordance with a particular process. A third way to assure that software will behave as specified is by examining the software itself, the output of the software development process. This third kind of assurance is the strongest, in the sense that it reasons about the actual code that will operate the system, but it is difficult (often impossible) to achieve simply by testing the code, because the state spaces involved are far too large for exhaustive testing. Techniques for mathematical verification of software can provide this kind of assurance. This approach might call for the certification of the tools and processes used in the verification. The size of software to which techniques have been successfully applied continues to grow, but remains a limiting factor. A successful approach to the development of secure power system software may well involve all three of these kinds of assurance for the foreseeable future.

BACK TO TABLE OF CONTENTS

Purpose

This code is intended to provide a basis for reducing the risk that power system software is vulnerable to malicious attacks that might impede system operation or compromise the integrity or confidentiality of data used or generated by the system. The aim in specifying a model code is not to assure that future systems are invulnerable to any anticipated attack but to record a consensus among experts from industry, academia, and government laboratories that represents a baseline set of requirements for the security of software and firmware in power systems. To act in the same way as building codes for physical structures, such a code will need to evolve over time and hence will need to find an appropriate home in a body with a continuing existence and continuing participation by relevant groups. Procedures will need to be established for defining terms precisely, for proposing and adopting changes, for establishing conformance to the code, and so on. The workshop participants offer this baseline code in hope that it will eventually lead, either through the establishment of a more formal building code structure or through adoption in some other form by relevant bodies, to a safer and stronger cyberinfrastrcture for power systems generally.

BACK TO TABLE OF CONTENTS

Elements Recommended for Inclusion, by Category

In creating the categorization below, the aim is to be comprehensive. Consequently, there are some categories for which no proposed elements were identified or agreed upon by the participants These empty categories are retained to highlight unmet needs.

For each element of the code, the following subsections are provided:

- Description: What is the meaning and purpose of this element?
- Vulnerabilities addressed: What kinds of vulnerabilities will be reduced or eliminated if this element is implemented properly?
- Developer resources required: What resources will the individual or organization developing the software/device require in order to satisfy this element?
- Evaluator resources required: What is required for a third party to assess whether the device satisfies this element?
- References

BACK TO TABLE OF CONTENTS

Elements intended to avoid/detect/remove specific types of vulnerabilities at the implementation stage

Secure software development process with assurance against subversion along with evidence of conformance

- *Description:* Vendors must develop security-critical software within the framework of an established methodology for secure software development. No specific methodology is required, but relevant examples include Microsoft's Secure Development Lifecycle (SDL) and the coding practices developed by SAFECODE. Evidence that the delivered software was developed within the chosen methodology must be available for review. Any third-party software incorporated into security-critical functions must be shown to provide equivalent assurance against accidental incorporation of vulnerabilities.
- *Vulnerabilities addressed:* Methodologies of the required type aim to reduce or eliminate a wide range of software vulnerabilities including memory safety errors, integer overflows, SQL injection, etc.
- *Developer resources required:* Developer must be able to select and implement a given methodology, develop software in accordance with it, and also develop the evidence to demonstrate conformance.
- *Evaluator resources required:* Evaluator must be able to review the delivered software and the conformance evidence and assess compliance.
- *References:* For information on Microsoft's Security Development Lifecycle, see https://www.microsoft.com/en-us/sdl/. Information on the industry-wide SAFECODE intiative, is available at https://www.safecode.org.

Static and dynamic code analysis (throughout development cycle)

- *Description:* Apply static and dynamic code analysis techniques to expose (and remediate as appropriate) software vulnerabilities. For developers, it is likely to be most effective to apply these tools regularly to software as it is developed, so that errors are found, and can be fixed, as soon as possible. The tools can be applied after the software is developed (including to software provided by third parties) and can still provide valuable information about the presence (or absence) of classes of errors; however it is generally acknowledged that it is significantly more costly to remediate errors found later in the development process.
- *Vulnerabilities addressed:* Memory safety (buffer overflows, use-after-free errors, null pointer dereference errors, etc.)
- *Developer resources required:* Access to relevant program analysis tools and programmers trained to use them effectively.
- *Evaluator resources required:* Access to the

BACK TO TABLE OF CONTENTS

software and analysis tools in order to replicate (or not) results supplied by the vendor.
- *References:* See NIST Software Assurance Metric and Tool Evaluation (SAMATE) reports, available at: https://samate.nist.gov/index.php/SAMATE_Publications.html. For a list of source code security analyzers, see https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html.

Use of memory-safe/type-safe languages

- *Description:* Memory-safe languages can eliminate or substantially reduce the likelihood of many classes of coding errors that have often led to exploitable vulnerabilities. These include buffer overflows, null pointer dereferences, use-after-free errors, and references to uninitialized memory. *Rust* and *Go* are relatively recent memory-safe languages; others include *F#*, *C#*, *Python,* and *Haskell*. Developers who select other common languages (e.g., *C*, *C++*) that don't provide memory safety need to provide evidence that their implementations avoid these problems.
- *Vulnerabilities addressed*: Memory safety errors.
- *Developer resources required:* Access to compilers and tools for memory safe languages and programmers trained in them.
- *Evaluator resources required:* Ability to assure that the programming language was in fact used to create the software (e.g., source code and a compiler).
- *References:* See results reported for probability of security errors in programming contest submissions reported in A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build It, Break It, Fix It: Contesting Secure Development," *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2016; https://arxiv.org/abs/1606.01881.

System and component fuzz-testing

- *Description*: Conventional testing generally aims to compare the results of a software implementation against its specification by exercising the functions included in the design in both normal and limit cases, so the test inputs are often designed to check particular cases and are not random. Fuzz testing essentially submits random inputs to a software component or system to see if unexpected behavior can be elicited and possibly exploited to subvert the behavior of the component or system. Participants agreed that fuzz-testing at both the component and system level should be a requirement of the building code, since attackers are quite likely to use it to seek paths into the system.
- *Vulnerabilities addressed:* Like other testing methodologies, fuzz-testing cannot guarantee the absence of vulnerabilities, but its use can

BACK TO TABLE OF CONTENTS

reveal a broad range of vulnerabilities include memory safety problems, race conditions, and many others. If these vulnerabilities can be found and remediated prior to deployment, they will be unavailable for exploitation by attackers.

- *Developer resources required:* Personnel who understand fuzz testing, the intricate details of the interfaces implemented, and have the tools available to conduct it. Like any testing regime, requires a specification of system behavior against which the tested behavior can be compared. Fuzz testing is random and cannot be exhaustive, and it provides more assurance as more tests are run. Consequently, an assurance regime that depends heavily on fuzz testing will demand significant computing resources.
- *Evaluator resources required:* The ability to review fuzz testing output and to judge its comprehensiveness.
- *References*: The original paper on fuzz testing: B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM 33*, Dec. 1990. Many tools are available for fuzz testing; some depart from the completely random model and incorporate coverage metrics or target boundary and limit cases. Microsoft has published guides on "how much" fuzzing is appropriate as well as on types of fuzzing to be applied.

Stress Testing

- *Description*: The aim of stress testing is to explore the behavior of a component or system when it is operated with relatively limited resources – e.g., memory, CPU, or network communications bandwidth may be limited in relation for a high required demand for service. These conditions can occur in normal operation if there is high demand, but they may also be artificially induced by an attacker mounting, for example, a denial of service attack on the system. A properly designed system should show graceful degradation in the face of stress testing and should recover normal operation smoothly as the stress is removed. Participants agreed that stress testing at both the component and system level should be a requirement of the building code.
- *Vulnerabilities addressed*: Like other testing methodologies, stress testing cannot assure flaws or design weaknesses are absent, it can only reveal only reveal those that the tests exercise. Stress testing may reveal a variety of implementation failures that occur when design parameters (e.g., maximum table sizes or queue lengths) are reached. Stress testing should also reveal failures in recovery mechanisms.
- *Developer resources required*: Requires personnel who understand stress testing and have the tools available to conduct it.

BACK TO TABLE OF CONTENTS

Requires a specification of expected system behavior under high-stress conditions and expected recovery modes when stress is removed.

- *Evaluator resources required:* Requires the ability to review stress testing results and to judge the comprehensiveness of the tests.
- *References:* Textbook on performance testing generally: H.H. Liu, *Software Performance and Scalability: A Quantitative Approach*, John Wiley & Sons, Inc., 2009.

Fault-injection testing

- *Description:* Fault injection testing aims to evaluate component and system behavior when faults occur. This testing approach therefore focuses on exercising fault- and error-handling code within the system that may be rarely invoked in operation. Faults may be injected at compile time by modifying the source code or at run time by modifying system data or protocol messages flowing over a network. Specifications must address the expected response to induced failures so that test results can be evaluated. Participants agreed that this type of testing should be applied to high-fidelity representations of operational power systems but should definitely not be conducted on live operational systems.
- *Vulnerabilities addressed:* Vulnerabilities likely to be revealed through fault-injection testing are those found in error-handling and recovery routines.
- *Developer resources required:* Personnel conversant with fault injection testing and tools to assist in conducting tests and evaluating results.
- *Evaluator resources required:* The ability to evaluate fault-injection test results and to assess their comprehensiveness.
- *References:* M.-C. Hsueh, T.K.Tsai, R. Iyer. "Fault Injection Techniques and Tools," *Computer*, Apr. 1997, p. 75 ff.

BACK TO TABLE OF CONTENTS

Elements intended to assure proper use of cryptography

Accredited cryptographic algorithms and implementations

- *Description*: Cryptographic algorithms that resist serious cryptanalysis are notoriously difficult to invent and to program correctly. While different environments make different demands on cryptography (for example, differing amounts of energy and time to devote to cryptographic operations and different time horizons for protecting keys), developers should seek algorithms that have received some external, open certification rather than attempt to develop their own. If for some reason suitable algorithms are not available and invention is required (this should be a last resort), developers should take care to get expert review prior to adopting and implementing their own crypto- algorithms. Weaknesses in cryptography often come in the implementation of the algorithm, key management, and surrounding protocols. Externally developed and certified implementations should be sought; custom implementations of cryptographic components require careful vetting by experts. In power system environments, cryptography may more often be called upon to assure the integrity of commands from operators and data from sensors rather than to protect their secrecy. Proper selection and implementations of algorithms for these requirements, proper use of cryptographic software packages, and proper management of keys will be essential to assuring that the requirements are met in practice.
- *Vulnerabilities addressed: addresses* Weaknesses in cryptographic algorithms, implementations, and use.
- *Developer resources required*: The ability to understand the cryptographic requirements of the system, select appropriate algorithms and implementations, and to use the selected packages correctly.
- *Evaluator resources required:* The ability to review and evaluate the system requirements and the developers design, selections, and implementations.
- *References:* "Use Cryptography Correctly," in IEEE Cybersecurity Initiative: Avoiding the Top Ten Software Security Design Flaws, p. 19; https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf.

Secure random numbers

- *Description*: Generating random numbers for use in initializing pseudorandom number generators and cryptographic algorithms, using them correctly, and avoiding reusing

BACK TO TABLE OF CONTENTS

them are challenging problems. Mistakes can nullify even well-designed and implemented cryptographic mechanisms. As advised in other work, developers should adopt established approaches that experts have vetted rather than attempting novel solutions. Even established approaches for random number generation need to be subjected to appropriate testing to assure their effectiveness.

- *Vulnerabilities addressed*: Susceptibility to cryptanalytic attacks on integrity and confidentiality that exploit poor selection of keys and other numbers intended to be random.
- *Developer resources required*: Access to vetted procedures for random number generation; these may be platform-dependent. Requires testing the procedures and documenting the results.
- *Evaluator resources required*: Ability to review and evaluate developer's design and implementation of random number generation and use, as well as reviewing test results.
- *References*: "Use Cryptography Correctly," in IEEE Cybersecurity Initiative: Avoiding the Top Ten Software Security Design Flaws, p. 19; https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf.

Elements intended to assure software/firmware provenance and integrity, but not to remove code flaws

Software Bill of Materials

- *Description:* Originally posed as "layered, traceable assurance and verification," the participants felt that it was too difficult to formulate as a checkable building code element, but agreed that a bill of materialsspecifying what software (including version or release number) is included in a system and the source of all of the software components in the systemis both critical and checkable.
- *Vulnerabilities addressed:* This element does not prevent vulnerabilities but permits identifying whether vulnerabilities discovered in software components are included in the system and hence may require patching/remediation. In this way it can be a critical tool for system defenders, but also for attackers, if they have access to it.
- *Developer resources required:* Ability to determine and specify where each line of code in the delivered product originated
- *Evaluator resources required:* Ability to map provided bill of materials against delivered software components.
- *References:* "H.R.5793 - 113th Congress (2013-2014): Cyber Supply Chain Management and Transparency Act of

BACK TO TABLE OF CONTENTS

2014 - Congress.gov - Library of Congress". Available at https://www.congress.gov/bill/113th-congress/house-bill/5793.

Digitally signed software and firmware with update validation

- *Description:* Both firmware and software that implement critical functions should be digitally signed, and the private signing keys must be carefully managed. The developer must either identify and distinguish critical vs. non-critical functions, or else the signatures must apply to all software and firmware. Files containing critical system configuration data will also benefit from these controls.
- *Vulnerabilities addressed:* This element does not prevent or eliminate vulnerabilities in software or firmware but aids in addressing software provenance (se Software Bill of Materials as well) and accountability in case of failures or attacks. Reduces vulnerability to spoofed updates or rollbacks.
- *Developer resources required:* Infrastructure to generate, distribute, update and protect signing keys; ability to integrate signing and validation functions in delivered system.
- *Evaluator resources required:* Evaluator needs to assure the integrity of signing mechanisms and operational mechanisms for signature verification.
- *References:* W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A Secure and Reliable Bootstrap Architecture," *Proc. 1997 IEEE Symp. on Security and Privacy,* IEEE, 1997.
- D. K. Nilsson, L. Sun, and T. Nakajima, "A Framework for Self-Verification of Firmware Updates over the Air in Vehicle ECUs," *Proc. 2008 IEEE Globecom Workshops,* IEEE, 2008.
- A. Cui, M. Costello, and S. J. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," *Proc. 20th Network and Distributed Systems Symp.* (NDSS) 2013, Internet Society, San Diego, CA, Feb. 2013.

Elements intended to impede attacker analysis or exploitation but not necessarily remove flaws

Specification of system information flows with effective enforcement

- *Description:* While the confidentiality of information in power systems is a concern, the integrity and flow of information, particularly control information sent to and received from cyberphysical systems, is usually the most critical concern. The developer must specify the flow of critical information through software and hardware components and make use of software and hardware mechanisms, including mandatory access controls (MAC),

BACK TO TABLE OF CONTENTS

rings of protection, privilege mechanisms, capability mechanisms, one-way flow devices, etc., as available and appropriate. This broad requirement concerns both system security policy and system architecture.

- *Vulnerabilities addressed:* Enforcement of information flow constraints does not necessarily eliminate implementation errors that could be exploited by maliciously crafted inputs, but it can limit the effects to the domains "downstream" from the exploitable flaw.
- *Developer resources required*: Ability to understand and architect information flows within the system and to employ available mechanisms to enforce them.
- *Evaluator resources required*: Ability to understand and assess both system function and developer's information flow specification and implementation.
- *References*: See French ANSSI http://www.ssi.gouv.fr/uploads/2014/01/Managing_Cybe_for_ICS_EN.pdf, esp. Appendix B, GP02, and US DHS https://ics-cert.us-cert.gov/sites/default/files/documents/Seven%20Steps%20to%20Effectively%20Defend%20Industrial%20Control%20Systems_S508C.pdf, item 4, for examples of related guidance.

Input Validation: All input accepted by control software must be well-defined (via a grammar or equivalent means), as syntactically simple as possible (regular or context-free syntax preferred), and fully validated before use.

- *Description:* Demonstrating the effectiveness of input validation, i.e., demonstrating that invalid inputs can be identified and are in fact rejected, was agreed to belong in the code. Simplification of inputs, which can reduce the difficulty of validation, was considered desirable but did not gain consensus as a code requirement.
- *Vulnerabilities addressed:* Exploitation of input-handling code by maliciously crafted input. Accepting invalid inputs can lead to unpredictable system behavior. Input validation can protect against buffer overflows and related memory safety errors.
- *Developer resources required:* Requires that for each possible system input, the range of acceptable inputs be unambiguously specified and that the implementation assure inputs are validated as specified.
- *Evaluator resources required:* Ability to review both the input specification and the code responsible for validating inputs.
- *References:* L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security Applications of Formal Language Theory," *IEEE Systems Journal*, vol. 7, no. 3, Sept. 2013; http://langsec.org/papers/langsec-tr.pdf.

BACK TO TABLE OF CONTENTS

Appropriate component separation / isolation

- *Description:* Providing isolation between components so that malfunction or penetration of one component cannot affect those isolated from it is a fundamental software and security engineering technique. In power systems, it is appropriately used to separate non-critical functions from critical ones, which implies that the critical functions have been explicitly identified. Mechanisms for achieving the separation can include hardware support for isolating machine domains (e.g., privilege modes, rings, segmentation, capabilities). Software sandboxing mechanisms can also be effective but may require additional evidence to assure their strength.
- *Vulnerabilities addressed:* This element does not remove specific classes of vulnerabilities but prevents or raises the difficulty for an attacker who exploits a vulnerability in one component to leverage that exploitation in other components.
- *Developer resources required:* The ability to distinguish more critical from less critical functions/components; ability to organize security architecture to exploit underlying security isolation mechanisms (processes, sandboxes, virtual machines); ability to map the design to the underlying separation mechanisms correctly.
- *Evaluator resources required:* Access to relevant design and implementation documents from developer and ability to interpret and evaluate them correctly.
- *References:* C. Greamo and A. Ghosh, "Sandboxing and Virtualization: Modern Tools for Combating Malware," *IEEE Security & Privacy,* vol. 9, no. 2, Mar./Apr. 2011, pp. 79–82.

Authentication and access control (human – device and device – device)

- *Description:* Authentication of human operators to machines is critical to providing accountability for operator-initiated actions and a basis for implementing role based (or other) access controls. As automation and attack sophistication increase, it will become more important for the machine to authenticate itself to the operator as well (i.e., so that the operator can be sure she is communicating with the intended machine and that its configuration is accurately portrayed). A complicating factor may be the need for emergency access by human operators. In addition, devices will require mutual authentication, for similar reasons. Some current standards for substation operation already impose authentication requirements. The consensus was that the code should require

BACK TO TABLE OF CONTENTS

two-factor authentication of operators, but at the same time should provide for audited "break-glass" emergency access for critical functions. Device-device authentication was seen as important, but requiring further research prior to imposing a building code requirement.

- *Vulnerabilities addressed:* This element does not generally detect or remove vulnerabilities in software or hardware, but it provides accountability for actions taken and provides the basis for authorizing system access. Authenticated communications can enable detection of traffic inserted by unauthorized third parties.
- *Developer resources required:* Ability to design and incorporate appropriate authentication mechanisms, including two-factor authentication.
- *Evaluator resources required:* Ability to evaluate authentication mechanism design and implementation.
- *References:* IEEE Std. 1686-2007 for Intelligent Electronic Devices. IEEE Std. 1815 (DNP3) also describes a machine-to-machine authentication process.

Elements intended to enable detection/attribution of attack

Security event logging

- *Description:* Provide a tamper-resistant audit trail for security-related events, such as software installation, user authentication, and attempted intrusion. The audit trail must not be overwritten by a flood of events; and there shall be a provision for offline storage. It was noted that certain kinds of power fluctuations might themselves be indicators of security--relevant events, but such fluctuations are expected to be captured by the power monitoring systems and hence did not require inclusion in this element.
- *Vulnerabilities addressed:* This element does not prevent or detect vulnerabilities, it aims to provide a record that would permit reconstruction and understanding of adverse activities after the fact and may assist in restoring the system to a valid state. If software monitoring a log can detect a malfunction or attack based on the logged actions, it may be able to initiate recovery actions or inhibit further damage.
- *Developer resources required:* Identification of security related event types (for example, authentications, privilege level changes, and software updates) including intrusion attempts, and implementation

BACK TO TABLE OF CONTENTS

of tamper-resistant, append-only security event logs.

- *Evaluator resources required:* Manual review of identified security related event types and of design and implementation of logging mechanisms and security event generation mechanisms.
- *References:* IEC 61850, IEEE 1815 (DNP3) already call for related functions.

Elements intended to assist in safe degradation of function during an attack

No elements proposed specific to this category, but see "back-out" functionality element.

Elements intended to assist in restoration of function after attack

Inherent "back-out" functionality // trustworthy recovery

- *Description:* Provide mechanisms that support restoration to secure functional state after a successful attack has been detected. Providing this capability can affect the system design broadly.
- *Vulnerabilities addressed:* This element does not prevent or eliminate vulnerabilities but aims to restore system function after a vulnerability has been exploited.
- *Developer resources required:* Requires the developer anticipate potentially successful attack modes and provide recovery mechanisms (e.g. backups inaccessible to attackers) that can be invoked when system degradation is detected.
- *Evaluator resources required:* Ability to assess adequacy of developer's design and recovery mechanisms.
- *References:* P. Gallagher, *A Guide to Understanding Trusted Recovery in Trusted Systems*, NCSC-TG-022, U.S. National Computer Security Center, Dec. 1991; https://fas.org/irp/nsa/rainbow/tg022.htm.

Elements intended to support maintenance of operational software without loss of integrity

No elements proposed specific to this category. However, it is related to software/firmware update validation under the previous element "Digitally signed software and firmware with update validation"

BACK TO TABLE OF CONTENTS

Conclusion

How might this report be used?

This report serves as an example of how a building code might be developed for software with security responsibilities in a particular domain. In itself, it records the consensus of a group of experienced industry, academic, and government laboratory individuals who are concerned with the security of future power systems. If it is to be used more widely, it needs to be circulated, read, considered, revised, amplified and perhaps eventually adopted by relevant organizations in the industry. It can also serve as a basis for industry and government standards groups considering how to proceed to help make the cybersecurity properties of future power systems an asset rather than a liability.

Acknowledgments

The authors thank all of the participants for their contributions to the workshop, which included considerable work in advance of the meeting itself. The willingness of all of the participants to travel to UIUC (including some from Europe and Australia), to share their views and to engage in spirited discussion made the workshop both productive and pleasurable. This report aims to capture the consensus of those present at the meeting. The authors are grateful to the group leaders and keynote speakers, who had the opportunity to review draft versions of the report, and whose comments have improved it. Responsibility for the final report, and any errors in it, remains with the authors.

BACK TO TABLE OF CONTENTS

Craig Preuss of the IEEE Power and Energy Society was particularly helpful in recruiting participants and assisting the organization of the workshop, although he was unable to participate in person. The IEEE Cybersecurity Initiative, and in particular Brian Kirk of the IEEE Computer Society, provided funds and organizational support that were essential to the conduct of the workshop. The U.S. Department of Energy, through its Cyber Resilient Energy Delivery Consortium (CREDC DoE Award Number DE-OE0000780) activities at the University of Illinois at Urbana-Champaign (and in particular Amy Clay Moore) provided excellent facilities and logistics support. The National Science Foundation (NSF CNS-1452113) and the Center for Security and Privacy Research at George Washington University provided additional support.

References

1. C.E. Landwehr, A Building Code for Building Code: Putting What We Know Works to Work, *Proc. 29th Annual Computer Security Applications Conference* (ACSAC), New Orleans, Dec. 2013.
2. Workshop to Develop a Building Code and Research Agenda For Medical Device Software Security: Final Report, Report GW-CSPRI-2015-01, 8 Jan. 2015; http://www.cspri.seas.gwu.edu/s/Landwehr-Building-Code-Final-Edit-Report-3-q0jj.pdf.
3. Building Code for Medical Device Software Security. (with Thomas Haigh), IEEE Computer Society, Mar. 2015; http://cybersecurity.ieee.org/images/files/images/pdf/building-code-for-medica-device-software-security.pdf.
4. North American Electric Reliability Corporation (NERC), Reliability Fundamentals of system Protection: Report to the Planning Committee, NERC System Protection and Control Subcommittee, Dec. 2010; http://www.nerc.com/comm/PC/System%20Protection%20and%20Control%20Subcommittee%20SPCS%20DL/Protection%20System%20Reliability%20Fundamentals_Approved_20101208.pdf.
5. A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, *IEEE Trans on Dependable and Secure Computing*, vol. 1, no., 1, pp. 11–33.
6. K. Zetter, "Three Minutes with Security Expert Bruce Schneier," *PC World*, 28 Sept. 2001; https://www.schneier.com/news/archives/2001/09/three_minutes_with_s.html.
7. D. Geer, "Complexity is the Enemy," *IEEE Security & Privacy Magazine*, Aug. 2008. p. 88; http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4753682.
8. J. Goldfarb, "Complexity is the Enemy of Security" *Security Week*, 11 Feb. 2015; http://www.securityweek.com/complexity-enemy-security.

BACK TO TABLE OF CONTENTS

Appendix A. Research Agenda for Power Systems Software Security

Input Simplification

Some participants proposed that inputs should be required to be simplified to improve the assurance that inputs can be mechanically validated. For example, "Protocols with complex message formats such as DNP3, IEC 61850, etc. must be restricted by their recognizer modules to subsets actually used by specific devices and valid for these devices. Non-conforming inputs should be rejected." The consensus was not to include this requirement in the initial code because protocol implementations may be licensed from third parties and hence difficult to modify. Research may be warranted into techniques for simplifying input language complexity and for wrapping existing implementations so that (potential) flaws in third party implementations cannot be exploited.

Verified OS and hardware

Some participants proposed to require the use of verified operating system and hardware platforms for critical devices. Some low power/low function devices do not include operating systems, so if such a requirement were included in a future code, the scope of its application would need to be made clear. Verfication would enable assuring that system initialization leads to a secure initial state. Critical properties desired of a binary (or source) program would need to be specified precisely. The subject program is then analyzed against a model embodying the semantics of the (hardware/software) execution environment to verify that the desired properties are present. The participants recognized there have been substantial advances in tools that can be applied to carry out formal verification of software and that some substantial software systems, including the seL4 kernel, have been verified. The technology is seen as cost-effective and is in use by chip vendors to verify hardware designs. The relative simplicity of some power system components would seem to bring them within reach of the technology. However, on balance, the participants felt that there was more research to be done before this element could be placed into the code.

Automated conformance checking

This proposed element is meant to cover mechanisms to check whether a software program

BACK TO TABLE OF CONTENTS

conforms to the building code. Tools (some more automated, like SAT solvers, and some requiring more manual assistance, like theorem provers) are, and have been for some time, available for this purpose. This element is closely related to the proposed "Verified OS and hardware" element, except that the proposed conformance is to the building code rather than to a functional specification, and a similar discussion applies.

Formal requirements specification

At least three senses of formal requirements specification were discussed during the meeting. For those pursuing formal verification of programs, a formal (in the sense of mathematical logic) specification of the desired properties of the program is required. The difficulty of creating such a specification is an impediment to the development of verified OS and hardware, just discussed, and suggested that incorporating a building code element for a formal specification is premature at this time. The participants also discussed formal security policy models, in the sense of the Trusted Computer System Evaluation Criteria, and endorsed the idea that without such a model, particularly one addressing mandatory integrity requirements, it is essentially impossible to specify when a security violation has occurred. On the other hand, "formal" used in the sense of having a form, a structure, leads to a different interpretation of "formal requirements specification". It was noted that IEEE Standard 1686 for Intelligent Electronic Devices provides a framework for cybersecurity requirements for such devices. The consensus placed this proposed element on the research agenda.

Active defense and automated response

This proposed element aims to automate the current activities of attack detection and response. As such, it aims to reduce vulnerabilities only as mitigations to observed attacks. Some current activities such as the DARPA Cyber Grand Challenge have incentivized this approach, but the participants felt that the technology is not mature enough to include in a building code at this time.

Assurance cases with eliminative arguments

Analysts who use this technique try to increase the confidence in a security assertion by posing counter-examples and then presenting evidence that eliminates as many counter-examples as possible. When a counter-example cannot be eliminated completely, the evidence can provide bounds on the potential impact of the counter--example. While assurance cases have been used successfully in the safety domain, their development for use in the security domain is less mature. The strength of any eliminative argument depends on the completeness of the set of posited counter-examples. No work has

BACK TO TABLE OF CONTENTS

been done to identify security-related counter-examples specifically for power system devices.

References

1. [IEEE] http://cybersecurity.ieee.org/
2. [L13] C.E. Landwehr, "A Building Code for Building Code: Putting What We Know Works to Work," *Proc. 29th Annual Computer Security Applications Conference* (ACSAC), New Orleans LA, ACM, NY, pp. 139–147; http://www.landwehr.org/2013-12-cl-acsac-essay-bc.pdf.
3. [MDSSA] Workshop to Develop a Building Code and Research Agenda For Medical Device Software Security: Final Report, Report GW-CSPRI-2015-01, 8 Jan. 2015; http://www.landwehr.org/2015-01-landwehr-gw-cspri.pdf.
4. [MDSSB] C.E. Landwehr, and T. Haigh, "Building Code for Medical Device Software Security," IEEE Computer Society, Mar. 2015; http://cybersecurity.ieee.org/images/files/images/pdf/building-code-for-medica-device-software-security.pdf.
5. [MSSDL] Microsoft Security Development Lifecycle; http://www.microsoft.com/en-us/sdl/default.aspx.
6. [NIST14] National Institute of Standards and Technology. Framework for Improving Critical Infrastructure Cybersecurity, version 1.0, 12 Feb. 2014; http://www.nist.gov/cyberframework/upload/cybersecurity-framework-021214.pdf.
7. [SDL06] Michael Howard and Steve Lipner. "The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software (Developer Best Practices)."

BACK TO TABLE OF CONTENTS

Appendix B. List of Participants

Note: Organizational affiliations are shown for information only. The workshop results and report do not necessarily represent the views of these organizations.

Kaibin Bao, Karlsruhe Institute of Technology (KASTEL)
Ian Bryant, Trustworthy Software Foundation
Chris Chelmecki, Basler Electric, *Discussion Group Leader*
Art Conklin, University of Houston
Adam Crain, Automatak
Dennis Gammel, Schweitzer
Andrew Ginter, Waterfall-Security
Mark Heckman, University of San Diego
Marijn Heule, University of Texas at Austin
Carl Landwehr, George Washington University (CSPRI)
Chad Lloyd, Schneider Electric
Dario Lobozzo, Radiflow
Johan Malmström, ABB
Scott Mix, North American Electric Reliability Corporation (NERC)
Ken Modeste, UL, *Discussion Group Leader*
Tommy Morris, University of Alabama – Huntsville
David Nicol, University of Illinois at Urbana-Champaign
Rajesh Nighot, Nebulian
Michael Pyle, Schneider Electric
Edwards Reed, AESec, Inc.
Craig Rieger, Idaho National Laboratory
Benjamin Salazar, Lawrence Livermore National Laboratory
Chet Sandberg, Consulting Engineer
William Sanders, University of Illinois at Urbana-Champaign
Roger Schell, AESec, Inc., *Keynote address*
Steven Templeton, University of California, Davis
Eric Thibodeau, Gentec
Mike Thiems, Basler Electric
Alfonso Valdes, University of Illinois at Urbana-Champaign
Zhenyuan Wang, ABB, *Discussion Group Leader*
Sam Weber, New York University
Jin Wei, University of Akron
Andrew West, SUBNET Solutions, Inc., *Keynote address and Discussion Group Leader*
Chuck Weinstock, Software Engineering Institute (CMU), *Discussion Group Leader*
Reid Wightman, RevICS
Carol Woody, Software Engineering Institute (CMU)
Tim Yardley, University of Illinois at Urbana-Champaign

BACK TO TABLE OF CONTENTS