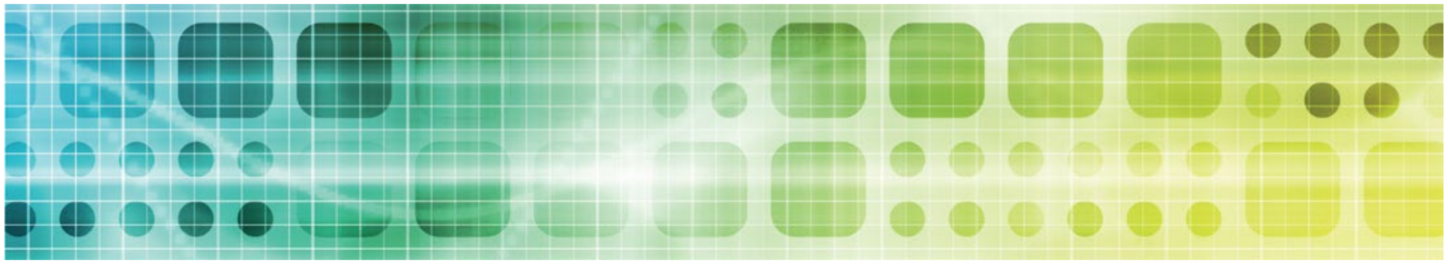
# Building Code for Medical Device Software Security

*Tom Haigh and Carl Landwehr*

# TABLE OF CONTENTS

## Staff

Kathleen Clark-Fisher, Manager, New Initiative Development
Jennie Zhu-Mai, Designer



# Introduction

Physical buildings are a well-established metaphor for software systems.[*] From the time of Hammurabi, building codes have provided a basis for assuring that buildings stand up to the demands of their environments. In modern times, building practices frequently arise from interacting communities of architects, builders, and suppliers rather than originating with a national authority. Once the practices are organized into a code, local governments may adopt it to place the force of law behind it for the common good.

The elements presented here aim to start builders of software for medical devices down the road toward a building code for software security that will reduce the vulnerability of their systems to malicious attacks, just as codes for physical buildings help their designers and builders create structures that resist threats from fire, wind, water and, in some cases, malicious attacks.

These elements and the structure of the code that organizes them were created by a group of 40 volunteers with a wide range of backgrounds, including cybersecurity, programming languages, software engineering, medical device development, medical device standards, and medical device regulation. Following several weeks of online collaboration, the group convened for two intense workshop days in New Orleans in November, 2014, under the sponsorship of the IEEE Cybersecurity Initiative and the National Science Foundation.[**] Carl Landwehr of George Washington University and Tom Haigh of Adventium Labs (ret.) organized and led the workshop.

During the workshop, the participants proposed and evaluated a set of elements for an

* 2015 Turing Award Leslie Lamport's recent article, "Who Builds a House Without Drawing Blueprints?" (CACM, April, 2015, pp. 38-41) provides a recent example. One of the authors of this report (Landwehr) explores the metaphor in more detail in "A Building Code for Building Code," Proc. ACSAC 2013.

** NSF Grants NSF CNS 14-52113 and NSF CNS 13-30491.

initial building code. Some of these elements are mature in that their application delivers clear benefits, there are well-understood methods for applying them, and there are accepted ways to measure the degree to which they have been applied. Other elements are more speculative, requiring further study to establish their effectiveness, methods of application, or measure of the quality of application. Participants also proposed research topics that could address the deficiencies in these proposed elements.

Following the workshop, the authors organized the mature elements into the "Draft Building Code for Medical Device Software Security" presented in this report. They also refined the research topics, which are listed in Appendix A to this report.

It is of course impossible to develop a complete code in a two-day workshop. The intent of this initial code is to provide a basis that developers can use to rule out the most commonly exploited classes of software vulnerabilities. To accomplish this, the participants chose to focus on code elements intended to avoid/detect/remove specific types of vulnerabilities introduced in the implementation phase, because implementation errors continue to be the major source of exploits. This draft code should be considered a starting point for a more complete code. While some elements of the draft code presented here address the design and test phases, there is a clear need for further effort to expand those aspects of the code.

**Tom Haigh**
Adventium Labs (ret.)

**Carl Landwehr**
George Washington University

## Purpose

This building code provides a basis for reducing the risk that software used to operate medical devices is vulnerable to malicious attacks. Such attacks might impede or alter a device's function, leak sensitive data, or otherwise cause the device to depart from its specified behavior. Some codes begin as standards and only become codes when they are adopted legally and have the force of law behind them. Others, such as the International Building Code (see http://publicecodes.cyberregs.com/icod/ibc/), act as model codes that can be tailored for different environments and adopted legally by different jurisdictions. The code presented here is intended as the beginning of a model code for software security.

The aim in specifying this code is not to assure that future medical devices can resist every imaginable attack, but rather to establish a consensus among experts in medical devices, cybersecurity, and computer science on a reasonable model code for the industry to apply. Metaphorically, the aim is to specify the needed properties of the bricks used to build the structure, not its architecture. The reason for focusing on the "bricks" at this time is that the majority of vulnerabilities actually exploited in cyberattacks are errors in implementation rather than design. By focusing on the desired implementation properties, this code aims to ensure that these bricks are indeed sound in that they are free of most vulnerabilities currently exploited.

Proper architecture and design are, of course, critical for the safety, usability, maintainability, and effectiveness of these systems, and responsible developers need to apply sound methodologies to every construction phase, from requirements through final testing and delivery. The IEEE Cybersecurity Initiative is addressing architecture and design concerns in other efforts.[***]

[***] "Avoiding the Top 10 Software Security Design Flaws," IEEE Center for Secure Design, IEEE Cybersecurity initiative, cybersecurity.ieee.org

## Scope and Applicability

The code applies to software that operates or executes within the context of a broad range of medical devices. It was not developed with other software domains (for example, software controlling automobiles, or software in large-scale IT systems) in mind, although those domains might well benefit from a similar code. The present code focuses on the creation of software that is difficult to subvert through malicious inputs or errors in cryptographic functions. It does not address other security functions—for example, authentication, authorization, and auditing (although security logging is included). There are many different types of medical (and more broadly, healthcare) computing devices, including implantable and wearable, as well as hospital bedside and large diagnostic equipment. These device types have different computing capacities and energy sources. A mature building code might well allow or require variations according to the particular device type covered and its capabilities and operating environment, but such differences, particularly in terms of analysis of the software developed for the device, would need to be justified. These considerations are beyond the scope of the code presented here.

## Definitions

The term *medical device* is used loosely in this document. It might refer to an implant, a wearable device (sometimes called a health management device), a bedside device in a hospitable, a large-scale diagnostic device such as an MRI system, or even an electronic health record system. These device types cover a broad range of technologies, complexities, and environments. The practical adoption of a code such as the one proposed here should include a more precise specification of the device characteristics to which it is intended to apply. It might be that the code should be tailored to different devices and to environments where the device is used. These contingencies could not be addressed in this preliminary version.

## Procedures

If a developers' consortium, a standards body, or a regulatory agency were to adopt this code, it would be necessary to specify a number of procedures. For example:

- How will a software component be evaluated to determine if it meets the code? What about a system comprising many components?
- Who will decide if the submitted component satisfies the code? Companies producing or using software might self-certify that their software satisfies the code, but such assertions should be subject to impartial review by an outside group.
- How will the code be modified over time? The group administering the code should be able to update it through a voting or consensus process as conditions warrant.
- How will legacy devices be handled?

Specifying these procedures is beyond the scope of this document. Nevertheless, the procedures for evaluating whether a specified piece of software satisfies many of the code elements provided here is intended to be within the state of the art, and often within the state of the practice.

# Elements of the Code, by Category

The elements in the code are organized into 10 lettered categories, A through I and X, which are intended to be reasonably comprehensive. Several placeholder categories that relate more to design than implementation are left empty.

For each code element, four subsections are provided:

- *Description.* What is the meaning and purpose of this element?
- *Vulnerabilities addressed.* What vulnerability types will be reduced if this element is implemented properly?
- *Developer resources required.* What resources will the individual or organization developing the software/device require to satisfy this element?
- *Evaluator resources required.* What is required for a third party to assess whether the device satisfies this element?

## Elements intended to avoid/detect/remove specific types of vulnerabilities at the implementation stage (A)

This section opens with several items related to programming language selection, use, and analysis and then proceeds to other topics.

Several building code elements aim to reduce vulnerabilities by controlling the selection and use of programming language. Languages such as C and C++ are widely used because they provide programmers with flexibility, perceived efficiency, and compatibility with large bodies of legacy software. However, these languages are also conducive to mistakes that are difficult to find and that provide attackers with vulnerabilities to exploit. Of particular interest is the loosely defined property of memory safety, which is defined informally[2] as the prevention of memory access errors of the following types:

- buffer overflow,
- null pointer dereference,
- use after free,
- uninitialized memory use, and
- illegal free (of an already freed pointer, or a not malloced pointer).

For a broad discussion of these issues, the reader is referred to other work.[3] This code contains a more limited set of elements deemed most important for medical device software.

### Use of memory-safe languages

- *Description.* Some programming languages are designed to prevent many of the memory access errors listed previously. Selecting such a language effectively rules out large classes of these vulnerabilities.

- *Vulnerabilities addressed.* Addresses memory safety vulnerabilities including buffer overflow, null pointer dereference, use after free, uninitialized memory use, and illegal free.
- *Developer resources required.* Requires programmers trained in the selected language, compilers, and runtime libraries for the language.
- *Evaluator resources required.* Requires the ability to recompile the source code using a compiler for the memory-safe language (to confirm that the object modules have been produced as claimed).

### Language subsetting

- *Description.* To reduce the possibility that known exploitable language constructs will occur in programs, the developer restricts implementers to use only a subset of language features or constructs, avoiding those known to be risky or ambiguous. Use of a restricted subset of a language might also improve performance of static analysis tools on the software. Subsets of several languages, including C (MISRA C, see http://www.misra-c.com), Ada (SPARK Ada, see http://www.spark-2014.org/about), are available.
- *Vulnerabilities addressed.* Addresses memory access and other weaknesses resulting from the use of the proscribed constructs.
- *Developer resources required.* Requires programmers trained in subset use, as well as code scanners to enforce subset constraints.
- *Evaluator resources required.* Requires access to source code and scanning tool to confirm that programs abide by subset constraints.

### Use of secure coding standards

- *Description.* To reduce the possibility of exploitable vulnerabilities in languages susceptible to memory access errors, but without restricting programmers to a language subset, adherence to standard usages of the language structures should be required. Using the standard can reduce the possibility of memory access and other exploitable errors substantially. Secure coding standards are available for C, C++ and Java.
- *Vulnerabilities addressed.* Addresses memory access and some other types of implementation errors.
- *Developer resources required.* Requires programmers trained in coding standard use, and software to check programs produced for conformance to the standard.
- *Evaluator resources required.* Requires source code and automated checker for conformance to standards. If conformance cannot be mechanically checked, manual auditing might be required.

### Automated memory safety error mitigation and compiler-enforced buffer overflow elimination

- *Description.* For software written in non-memory-safe languages (for example, C/C++), use compiler transforms that enforce memory safety (for example, SAFECode,[4] WIT,[5] Baggy Bounds Checking,[6] and SoftBound[7]). Develop policy on what to do when a runtime error is detected (for example, reset device).
- *Vulnerabilities addressed.* Addresses memory access errors.
- *Developer resources required.* Requires access to software checking tools and source code.
- *Evaluator resources required.* Requires the ability to rerun tools used by the developer on the source/binary; confirming that an appropriate compiler has compiled all the software with the instrumentation enabled.

### Automated thread safety analysis

- *Description.* The developer annotates multithreaded code to declare desired thread safety properties. Tool (compiler option) assures that the policies are enforced.
- *Vulnerabilities addressed.* Addresses race conditions and deadlocks.
- *Developer resources required.* Requires programmers capable of developing correct annotations, and access to a compiler capable of processing them.
- *Evaluator resources required.* Requires the ability to determine that appropriate annotations have been made (manual) and that all the software was processed with the appropriate compiler and options (or recompile it) and the ability to review the specified safety policies.

### Automated analysis of programs (source/binary) for critical properties

- *Description.* Critical properties desired of a binary (or source) program are 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.
- *Vulnerabilities addressed.* Addresses any vulnerability countered by the specified properties.
- *Developer resources required.* Requires developers capable of specifying the properties desired of the implementation in the language accepted by the verification tools involved, or access to experts with this ability.
- *Evaluator resources required.* Requires the ability to generate and review output of verification tools applied to the programs analyzed.

### Modified condition decision coverage

- *Description.* This is a criterion for test coverage that has been successfully applied to life-critical avionics software for many years and is part of standards for automotive, rail, and process control systems. It requires a specification of system behavior and testing against that specification to achieve the following coverage:

  - Each entry and exit point is invoked at least once.
  - Each decision has taken each possible outcome at least once.
  - Each condition in a decision takes on every possible outcome at least once.
  - Each condition is shown to independently affect the outcome of the decision.
  - This test coverage criterion subsumes statement and branch coverage, requires $k + 1$ tests for $k$ conditions, and ensures $t$-way combination coverage of at least $(1 + t)/2t$.

- *Vulnerabilities addressed.* This is a general tool for assuring implemented software performs as designed. It is not targeted at detecting specific vulnerabilities but has proven effective for assuring safety in many life-critical systems.
- *Developer resources required.* Requires system specification at a level of detail sufficient to validate test results. The coverage criterion demands extensive testing of the software and might not be feasible for large code bases; it is appropriate for life-critical medical software.
- *Evaluator resources required.* Requires resources to review test results (some automation should be possible to see that test and specification match) and assure fielded software is the tested software.

### Operational use case identification and removal of unused functions

- *Description.* Use cases for the device are specified and software components required by each use case are identified. Software not required by any use case is considered for removal from the system to eliminate the possibility of attacks exploiting software unneeded for system function. Rather than a detailed, line-by-line code level analysis, this element can be applied most effectively at a relatively high level of abstraction to be sure that unused libraries, function collections, and applications are eliminated.
- *Vulnerabilities addressed.* Addresses software vulnerabilities located in unused components.
- *Developer resources required.* Requires identification of a comprehensive set of use cases (sometimes difficult in practice) and ability to track each use case back to software required for it.
- *Evaluator resources required.* Requires manual review of software components present against the specified use cases.

## Elements intended to assure proper use of cryptography (B)

### Accredited cryptographic algorithms and implementation

- *Description.* Cryptographic algorithms that resist serious analysis are notoriously difficult to invent and to program correctly.[8] Organizations such as the National Institute of Standards and Technology hold open competitions to create cryptographic algorithms. While different environments place different requirements (for example, differing amounts of energy and computational power to devote to cryptography and different time horizons for storing secrets), 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.
- *Vulnerabilities addressed.* Addresses weaknesses in cryptographic algorithms and implementations.
- *Developer resources required.* Requires access to expertly vetted cryptographic algorithms and implementations.
- *Evaluator resources required.* Requires the ability to audit software for use with vetted cryptography or to automatically verify implemented cryptography against a vetted specification.

### Secure random numbers

- *Description.* Generating random numbers for use in initializing pseudorandom number generators and cryptographic algorithms, using them correctly, and avoiding reusing them are challenging problems. Mistakes can nullify even well-designed cryptographic mechanisms. As advised in other work,[8] developers should adopt established approaches that field experts have vetted rather than attempting novel solutions.
- *Vulnerabilities addressed.* Addresses ineffective cryptographic mechanisms.
- *Developer resources required.* Requires access to vetted procedures for random number generation; might be platform-dependent.
- *Evaluator resources required.* Requires manual review of design and code.

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

### Digitally signed firmware and provenance (supply chain)

- *Description.* Developer and integrator affix a digital signature to software/firmware installed in a device. In case of subsequent device malfunction or compromise, the signature of the software present at the time of failure can be recomputed and compared with the signature of the distributed version to detect tampering.
- *Vulnerabilities addressed.* Addresses software provenance, helping establish accountability for fielded software. This element does not aim to eliminate vulnerabilities in the software/firmware.
- *Developer resources required.* Developer (or third party) needs a signing key, to protect that key, and to compute and store digital signatures for the protected components.
- *Evaluator resources required.* Evaluator needs to assure the integrity of signing mechanisms and operational mechanisms for signature verification.

### Software/firmware update validation

- *Description.* The aim is to enable valid updates to operational software while minimizing the possibility that the update mechanisms can be subverted to install fraudulent updates. The vendor applies an encrypted checksum on the updated software and then validates the checksum, via a trusted path, at the time the update is applied.
- *Vulnerabilities addressed.* Addresses installation of fraudulent software updates and loss of accountability to the system producer.
- *Developer resources required.* Developer (or third party) needs a signing key, to protect that key, and to compute and store digital signatures for the updates it produces.
- *Evaluator resources required.* Evaluator needs to assure the integrity of signing and operational mechanisms for signature verification.

### Whitelisting

- *Description.* The aim is to avoid execution of untrustworthy, possibly malicious, applications. Prior to execution of application software, the software is checked against a list of authorized applications (the whitelist). Entering new applications in the whitelist is a privileged operation, not under operator control.
- *Vulnerabilities addressed.* Addresses execution of unvetted application software. The mechanism does nothing to remove vulnerabilities from applications; it only assures that the application to be executed is included on the whitelist.
- *Developer resources required.* Requires that

the design and implementation include a whitelisting mechanism and attendant software to permit privileged users to update the whitelist needs.

- *Evaluator resources required.* Requires manual review of whitelisting mechanism specification and implementation (if the mechanisms are specified formally, automated assistance is possible).

## Elements intended to impede attacker analysis or exploitation but not necessarily remove flaws (D)

### Nonexecutable data pages

- *Description.* Storage is divided into code segments that might be read or executed but not written and into data segments that might be read or written but not executed. Temporary storage (stacks, heaps, and global variables) is assigned to data segments and so cannot be used by attackers to execute instructions.
- *Vulnerabilities addressed.* This element does not eliminate vulnerabilities in software; it does make it more difficult for the attacker to exploit them. It does not prevent attackers from using return-oriented programming attacks.
- *Developer resources required.* Developer needs to organize code to take advantage of this structure and its supporting hardware mechanisms. Note that just-in-time compilation and other mechanisms designed to develop and install code during operation will pose problems.
- *Evaluator resources required.* Evaluator needs to review the use of mechanisms for assigning code and data to storage segments.

### Full recognition of inputs before processing

- *Description.* A component that accepts an input without checking its validity presents a path that an attacker can probe. In general, designers should consider the input language grammar and use the most restrictive grammar consistent with required component functions. Designers should then be sure that inputs are checked for conformance to that grammar before processing those inputs.
- *Vulnerabilities addressed.* Addresses exploitation of input-handling code by maliciously crafted inputs.
- *Developer resources required.* Requires specification of input language, program source code, and software framework for generating recognizer for input language.
- *Evaluator resources required.* Requires audit of software and its data language definitions for adherence to the design principle. Audit must identify the code that checks and handles inputs immediately upon receipt, and evaluate whether the checking code is complete as a recognizer for a given definition of

valid and expected data, and isolated from other functionality. With appropriate constraints on specification and implementation languages and procedures, automation might assist the review.

### Least operating system privilege

- *Description.* The least-privilege principle calls for the operating system to grant programs/processes only those privileges required for them to carry out their specified functions.[9] Programs should be designed so that the number of privileges needed and the amount of time those privileges are needed is minimized. Programs requiring root or administrator privileges should use fine-grained operating system level privileges when available. For those systems that allow enabling/disabling of privileges (for example, effective UIDs, effective and maximum privilege/capability sets, and so on), privileges should be enabled only for those system calls needing them. Privileges should be removed when no longer needed.
- *Vulnerabilities addressed.* Addresses exploitation of over-privileged processes.
- *Developer resources required.* Designers must keep the principle in mind as they organize system components. Implementers must abide by the constrained design and avoid granting privileges in the implementation not called for in the design.
- *Evaluator resources required.* Automated static analysis can reveal whether privileges are enabled only where specified. Manual analysis is required to determine if the design adheres to the principle.

### Antitampering of hardcoded secrets/keys/data within medical device software

- *Description.* Employ appropriate software/hardware protections against malicious observation/modification of medical device secrets by the device possessor. Solutions relying solely on software (white box cryptography) and solutions that exploit widely available hardware (trusted platform modules with supporting software) are available.
- *Vulnerabilities addressed.* Addresses unauthorized access or deliberate modification of application generated and/or managed data by a malicious device owner. In particular, this restricts side channel attacks.
- *Developer resources required.* Requires access to appropriate software/hardware packages and expertise to apply them correctly.
- *Evaluator resources required.* Requires manual review of application of the selected mechanisms; potentially requires red-team testing to evaluate overall effectiveness.

## Elements intended to enable detection/attribution of attack (E)

Security event logging

- *Description.* Provide a tamper-resistant audit trail for security-related events, such as software installation, user authentication, and so on).
- *Vulnerabilities addressed.* Addresses accountability by providing an after-the-fact trail for forensic analysis.
- *Developer resources required.* Requires identification of security related event types (for example, authentications, privilege level changes, and software updates), and implementation of tamper resistant, append-only security event logs.
- *Evaluator resources required.* Requires manual review of identified security related event types and of design and implementation of logging mechanisms and security event generation mechanisms.

## Elements intended to assist in safe degradation of function during an attack (F)

None proposed. This is a design consideration.

## Elements intended to assist in restoration of function after attack (G)

None proposed. This is a design consideration.

## Elements intended to support maintenance of operational software without loss of integrity (H)

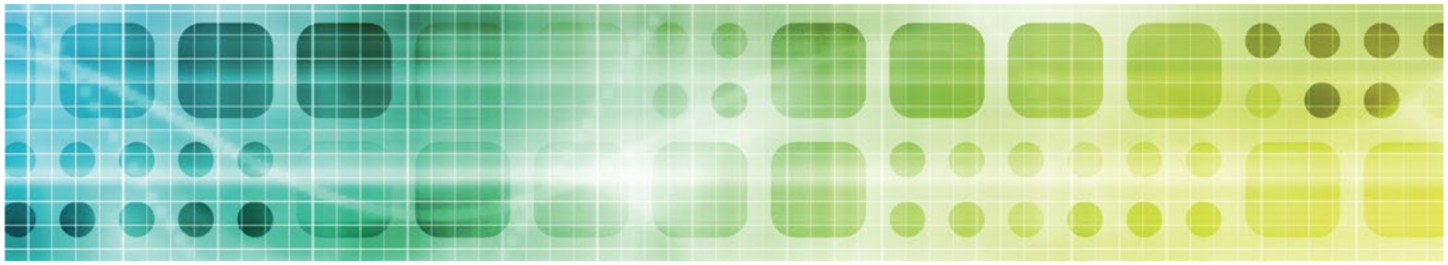
This is a design consideration. However, it is related to software/firmware update validation under the previous "Software/firmware update validation" element.

## Elements intended to support privacy requirements (I)

None proposed. This is a design consideration.

## Desired characteristics of the building code, for example, standard names use, building code maintenance over time, and scope (X)

Use within the code itself of standard names for types of attacks/attack patterns and vulnerabilities. There are no proposed standards at this time.



# Conclusion

The draft building code presented here must be viewed as the beginning, not the end, of an effort to create a foundation for building medical devices that are free of the most commonly exploited vulnerability types. For this work to have real effect, it must be carried forward by those with responsibilities for building and evaluating medical devices and for creating the framework of standards surrounding their development and use.

## Acknowledgments

The draft building code and research agenda provided in this report are the product of a two-day invitational workshop convened in New Orleans, 19–21 November 2014. Forty people from a wide range of backgrounds including medical device development, standards, regulation, cybersecurity research, programming languages, and software engineering participated in the workshop. See Appendix B for the list of participants.

Support for the workshop was provided by the IEEE Cybersecurity Initiative (see http://cybersecurity.ieee.org) and the National Science Foundation's Secure and Trustworthy Cyberspace program, both directly (NSF CNS 14-52113, Creating a Building Code for Medical Device Software Security) and through the NSF's Trustworthy Health and Wellness project (NSF CNS 13-30491, Trustworthy Health and Wellness).

Many people collaborated to create this workshop and produce the results. The chair and vice-chair, authors of this document, particularly want to thank the IEEE Cybersecurity Initiative and the NSF for providing the funds to support participant travel and expenses. Kathleen Clark-Fisher, program director for the

IEEE Cybersecurity Initiative, provided logistical support during the meeting, and Theresa McNeill of IEEE was instrumental in making the meeting arrangements. Katelyn Anders of the Cyber Security and Policy Research Institute at George Washington University assisted with participant travel reimbursement. We are grateful to the members of the workshop steering committee for their encouragement and advice, and particularly Pat Baird and Ken Hoyme who provided helpful contacts and assisted in recruiting participants. Most of all, we thank the workshop participants for their contributions both in advance of the workshop to the website used to develop and coordinate the draft building code and for their active collaboration during the meeting.

## References

1. C.E. Landwehr, "A Building Code for Building Code: Putting What We Know Works to Work," *Proc. 29th Ann. Computer Security Applications Conf.* (ACSAC), 2013, pp.139–147; http://www.landwehr.org/2013-12-cl-acsac-essay-bc.pdf.
2. L. Szekeres et al., "SoK: Eternal War in Memory," *Proc. 2013 IEEE Symp. Security and Privacy*, 2013, pp. 48–62; http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6547101.
3. *Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use*, ISO/IEC TR 24772:2013, ISO/IEC, 2013.
4. M. Belk et al., "Fundamental Practices for Secure Software Development," 2nd ed., A Guide to the Most Effective Secure Development Practices in Use Today, 8 Feb. 2011; http://www.safecode.org/safecode-updates-guidance-on-secure-development-practices.
5. P. Akritidis et al., "Preventing Memory Error Exploits with WIT," *Proc. 2008 IEEE Symp. Security and Privacy*, 2008, pp. pp. 263–277.
6. P. Akritidis et al., "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors," *Proc. 18th Usenix Security Symp.*, 2009, pp. 51–100.
7. S. Nagarakatte et al., "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," *Proc. 2009 ACM Sigplan Conf. Programming Language Design and Implementation* (PLDI), 2009.
8. IEEE Center for Secure Design, *Avoiding the Top Ten Security Flaws*, 2014; http://cybersecurity.ieee.org/images/files/images/pdf/CybersecurityInitiative-online.pdf.
9. J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, no. 9, Sept. 1975, pp. 1278–1308.

# Appendix A. Research Agenda for Medical Device Software Security

Several of the elements proposed for the building code require further research before they are included in the code. Some of these would be valuable but there are not yet practical methods for applying them. Others are specified at too high a level of abstraction for practical evaluation without substantial human effort and interpretation. Still others require additional research to show their effectiveness. These elements are listed and described here.

## Assurance cases using the Object Management Group's Structured Assurance Case Metamodel based tooling

The Structured Assurance Case Metamodel (SACM) provides a general method of stating and analyzing security claims and for exchanging these claims among vendors and providers. This, or a similar, technique could have great value as medical devices become more complex and more highly connected. However, there is not yet a convincing body of evidence that the current SACM approach will prove effective for medical devices. Research must be performed to apply SACM and related tools to medical devices and then to measure the effectiveness of the tools and techniques. Questions to answer include:

- Which sorts of assurance properties for medical devices can be established more easily using the tools and techniques, and what sort of cost reduction can be achieved?
- How much do the tools and techniques reduce ambiguity and confusion when exchanging security claims among multiple parties?
- How much safer and more secure are devices when developers and analysts use the tools and techniques?
- How much faster can faults be fixed and the device be recertified by using the tools and techniques?

## Minimization of computational power exposed to inputs

While it seems intuitive that there are benefits to adopting a formal, language theoretical approach to analyzing and limiting the computational power associated with device inputs, there is no strong evidence confirming this intuition. Moreover, there are not yet techniques and tools that developers can use to limit inputs or that evaluators can use to assess the gap size between what computational power the inputs must provide and the power they actually provide.

### Protection of critical state data

An attacker who gains access to critical state data can wreak havoc in many ways including data collection or modification, modification of program execution, or even seizure of complete control of the device. Although there are several forms of protection that might be applicable (encryption, code obfuscation, and oblivious computing), there is no compelling evidence that any of these techniques would actually work in the context of medical devices.

### Risky module identification

Software engineering research has produced techniques to identify error-prone software modules based on problem reports and software development records. It is possible that these or similar techniques will be useful in assuring classes of security vulnerabilities absent in medical device software, but there is currently no evidence to support this hypothesis.

### Runtime detection of code tampering via antitamper/anticorruption mitigation techniques

Digital signatures can provide assurance that code has not been altered prior to load time, but there is still a risk of modification after the signature has been checked and the code is executing. Some hardware/software mechanisms have been devised to check software while it is running. Application of such mechanisms to medical device software is a topic for study.

### Security assurance cases using 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, they have not been used as much in the security domain. Also, the strength of any eliminative argument depends on the completeness of the set of posited counter-examples. No work has been done to identify security-related counter-examples for medical devices or for analyzing the completeness of a set of counter-examples.

### Trusted computing base

The notion of a trusted computing base (TCB) is a well-established IT security construct. Particularly for operating systems, there is a solid understanding of the functionality and components necessary for a TCB. However, little if any research has been performed to identify the functionality and components of a TCB for medical devices. It is quite likely that different device classes will require different TCBs. In principle, it should be possible for vendors and evaluators to agree on a common TCB that could be tailored for different device classes.

### Notations that expose cyber mitigations (such as insulation diagrams)

In designing physical buildings, different types of diagrams are generated. Some show the physical dimensions and composition of walls and foundations, for example. Others show plumbing, wiring, and heating/ventilation functions. Similarly a logical circuit diagram might illustrate the logical paths for signals and data in a computational component without showing the physical routing and the insulation along paths. To carry out a proper failure analysis for a circuit board, both the logical and physical diagrams are needed. There has been no effort to develop analogous diagrams in the context of software that might, for instance, reveal the potential effects of breaking through one or more security barriers. Such diagrams could help convey the true depth of a set of defenses.

### Compiler-based integer overflow protection

Techniques such as As-if Infinite Range integer models (specifically for C and C++ languages) have been developed and prototyped but have not yet been incorporated in production compilers.

# Appendix B. List of Participants

- Homa Alemzadeh, *University of Illinois, Urbana-Champaign*
- Paul Anderson, *Grammatech*
- Sergey Bratus, *Dartmouth College*
- Tom Brennan, *Proactive Risk*, *OWasp*
- Ian Bryant, *UK Trustworthy Software Initiative, University of Warwick*
- Blaine Burnham, *University of Southern California*
- Jonathan Carter, *Arxan Technologies*, *OWasp*
- John Criswell, *University of Rochester*
- Jeremy Epstein, *US National Science Foundation*
- Scott Erven, *Protiviti*
- Dale Fay, *University of Michigan Radiology*
- Anura Fernando, *Underwriters Laboratories*
- Brian Fitzgerald, *US Food and Drug Administration*, *discussion group leader*
- Ken Fuchs, *Center for Medical Interoperability*
- Christopher Gates, *Illuminati Engineering*
- Sol Greenspan, *US National Science Foundation*
- Tom Haigh, *Adventium Labs* (ret.), *vice-chair and draft author*
- Marijn Heule, *University of Texas-Austin*
- Warren Hunt, *University of Texas-Austin*
- James Jacobson, *Siemens Healthcare Diagnostics*
- Michelle Jump, *Stryker*, *discussion group leader*
- Chandu Ketcar, *Cigital*
- D. Richard Kuhn, *US National Institute of Standards and Technology*
- Carl Landwehr, *George Washington University*, *chair and draft author*
- Steve Lipner, *Microsoft*
- Dan Lyon, *Cigital*
- Robert Martin, *MITRE*
- James McDonald, *Kestrel Institute*
- Michael McNeil, *Philips*, *discussion group leader*
- Steve Myers, *Indiana University*
- Nathanael Paul, *University of South Florida*
- Eric Petersen, *Welch Allyn*
- Stephanie Preston, *Battelle*
- Robert Seacord, *Carnegie Mellon University-Software Engineering Institute (SEI)/CERT*
- Shahid Shah, *Netspective*
- Tucker Taft, *Adacore*
- Roshan Thomas, *MITRE*, *discussion group leader*
- Eugene Vasserman, *Kansas State University*
- Sam Weber, *Carnegie Mellon University-SEI/CERT*
- Chuck Weinstock, *Carnegie Mellon University-SEI*, *discussion group leader*