

CROSSTALK



Sep / Oct 2010 **The Journal of Defense Software Engineering** Vol. 23 No. 5



GAME-CHANGING TOOLS AND PRACTICES

4 CROSSTALK Goes All Electronic: Turning Necessity into Opportunity

In this last print edition, CROSSTALK's Publisher discusses reasons for the change of format to electronic only—and the benefits of that move.
by Kasey Thompson

Game-Changing Tools and Practices

5 Static Analysis Is Not Just for Finding Bugs

This article examines how static analysis tools can provide more “human-readable” output, why they should focus on humans instead of bugs, ways to combine them in automated code review, and the pros and cons of their use in computing properties.
by Dr. Yannick Moy

9 Considering Software Supply Chain Risks

This article examines the software supply chain's complexity, common weaknesses and how to mitigate them, and how those practices could be applied to the acquisition of commercial software components.
by Dr. Robert J. Ellison and Dr. Carol Woody

13 Information Assurance Applications in Software Engineering Projects

U.S. Naval Academy computer science student projects—developing a training and personnel database with multi-level views, creating emergency notification system authentication, and two cybersecurity competitions—provide “lessons learned” in information assurance.
by Lt. Col. Thomas A. Augustine (Ret.) and Dr. Lori L. DeLooze

16 Studying Software Vulnerabilities

Injection is the single most exploited software weakness type, and the authors outline a process for building a semantic template that can study injection and other vulnerabilities—and work synergistically with other security standardization efforts.
by Dr. Robin A. Gandhi, Dr. Harvey Sij, and Yan Wu

21 The Balance of Secure Development and Secure Operations in the Software Security Equation

Effectively addressing software security requires balancing both secure development and operations, and this article shows how Common Attack Pattern Enumeration and Classification can help.
by Sean Barnum

25 Two Initiatives for Disseminating Software Assurance Knowledge

The authors examine how documenting software assurance knowledge to ensure its growth and integration into various educational settings and developing a curriculum for an MSwA degree program will improve the standing of software assurance.
by Dr. Nancy R. Mead and Dr. Dan Shoemaker

CROSSTALK

OSD (AT&L) Stephen P. Welby

NAVAIR Jeff Schwab

309 SMXG Karl Rogers

DHS Joe Jarzombek

MANAGING DIRECTOR Brent Baxter

PUBLISHER Kasey Thompson

MANAGING EDITOR Drew Brown

ASSOCIATE EDITOR Chelene Fortier-Lozancich

ARTICLE COORDINATOR Marek Steed

PHONE (801) 775-5555

E-MAIL stsc.customerservice@hill.af.mil

CROSSTALK ONLINE www.stsc.hill.af.mil/crosstalk

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the Office of the Secretary of Defense (OSD) Acquisition, Technology and Logistics (AT&L); U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). OSD (AT&L) co-sponsor: Software Engineering and System Assurance. USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: National Cybersecurity Division in the National Protection and Programs Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of CROSSTALK, providing both editorial oversight and technical review of the journal. CROSSTALK's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.



Contacting Us: Correspondence can be sent to:

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

E-mail Notification: Readers can receive notification when new editions are online by e-mailing or phoning CROSSTALK.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the author and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CROSSTALK Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.



Departments

3 From the Sponsor

29 Coming Events

30 Web Sites

31 BACKTALK



Changing the Game in Software Assurance



No developer or application manager likes to learn that their code was hacked and their applications exploited. Therefore, CROSSTALK readers who manage and write code are often the strongest advocates of “doing the right thing,” especially when it comes to software assurance (SwA). That is why the DHS is proud to sponsor this issue, primarily focused on SwA *Game-Changing Tools and Practices*.

Many SwA tools focus on automatic bug-finding, the first stage in a two-phase process where the tool finds bugs and the human then corrects them; Dr. Yannick Moy’s article, *Static Analysis Is Not Just for Finding Bugs*, argues for a larger view of SwA by looking at the computing properties of software.

With today’s global IT software supply chain, project management and software/systems engineering processes must explicitly address security risks posed by exploitable software. In *Considering Software Supply Chain Risks*, Dr. Robert J. Ellison and Dr. Carol Woody point out that a software supply chain can involve a combination of internal development, outsourced development, multiple commercial suppliers, and the use of legacy systems. The composite system inherits the risk of SwA failure at any point in such a supply chain. The authors recommend three practices: 1) mitigation of items on a CWE/SANS Institute Top 25 list linked to detailed design or coding practices, 2) mitigations associated with risk analysis, requirements, architecture, and testing, and 3) employment of a full life-cycle context for security improvement.

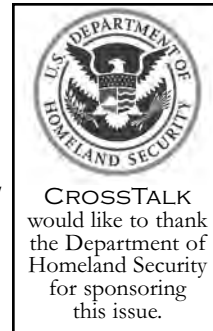
Two articles build on the SwA Automation Protocols from MITRE’s DHS-sponsored Making Security Measurable program. *Studying Software Vulnerabilities* by Dr. Robin A. Gandhi, Dr. Harvey Siy, and Yan Wu points to the potential for using these automation protocols to build tools for developers. Sean Barnum’s *The Balance of Secure Development and Secure Operations in the Software Security Equation* shows how these protocols enable development and operations staffs to better communicate and cooperate to secure applications.

Education is another essential arena for addressing the security risks posed by exploitable software. Lt. Col. Thomas A. Augustine (Ret.) and Dr. Lori L. DeLooze examine *Information Assurance Applications in Software Engineering Projects* from U.S. Naval Academy student capstone projects. Dr. Nancy R. Mead and Dr. Dan Shoemaker detail *Two Initiatives for Disseminating Software Assurance Knowledge*: Carnegie Mellon’s SwA Master’s program, providing an explicit curriculum of knowledge and skills necessary to produce a well-educated SwA professional; and the University of Detroit Mercy’s efforts to give every instructor in a computer-related discipline access to validated content and instructional materials that can be easily incorporated into existing courses.

Online readers of CROSSTALK get a bonus article: Patti Spicer’s *Gaining Software Assurance Through the Common Criteria* gives both a background of the Common Criteria and explains how its certification process provides software product assurance.

As the new Director of the National Cyber Security Division, part of my responsibility is to advance efforts like those described in this issue. I look forward to working with talented professionals like readers of CROSSTALK, who make our nation’s software and applications resilient and secure.

Bobbie Stempfley
Director, National Cyber Security Division





CROSSTALK Goes All Electronic: Turning Necessity into Opportunity

CROSSTALK is moving to an all-electronic format beginning with our November/December 2010 issue. Many reasons and discussions have shaped our decisions to make this move, with budgets, costs, and sponsorships having a major impact.

Understandably, some of our avid readers will find the change unwelcome.

The first silver lining is that we will still have a laid out version of CROSSTALK, allowing for the full editions and individual articles to be downloaded in PDF form. Secondly, the numbers suggest that it's an opportune time for changes:

- CROSSTALK's online edition averages 1.1 million visitors per month, while our hardcopy subscribers—currently less than 1 percent of our readership—has remained steady. We can now focus our mission on the methods that a majority of our readership uses.
- CROSSTALK is making every effort to be environmentally conscious. Eliminating a print version reduces our global footprint in the neighborhood of 700,000 printed pages per issue.

Readers will also notice continuing changes to our Web site. These improvements will aid all of our readers, whether they are seeking the current issue or searching our issues archive, dating back to 1994.

We understand the print version is an important tool—the passing from colleague to colleague, the earmarked reference copies on desks—it's what CROSSTALK has always been about. I'm reminded of what I saw at this year's Systems and Software Technology Conference, from Hillel Glazer's showing off of our January/February 2010 CMMI issue to Dr. Robert Cloutier's Plenary Session kudos causing a run on the May 2005 edition. Even in the electronic age, our print versions still produce a significant impact.

But we have to change. We hope future sponsorship increases, and with that a return to distributing our journal to the tens of thousands of people and businesses that like holding CROSSTALKs in their hands.

Please visit <www.stsc.hill.af.mil/crosstalk> to sign up for an electronic notification and a link to future CROSSTALK issues.

We thank you for reading CROSSTALK and ask for your continued support in this transition.

Kasey Thompson
CROSSTALK Publisher
kasey.thompson@hill.af.mil



Static Analysis Is Not Just for Finding Bugs

Dr. Yannick Moy
AdaCore

Static analysis tools are gaining popularity for safeguarding against the most common causes of errors in software. The main focus of these tools is on automatic bug-finding—the first stage in a two-phase process where the tool finds bugs and the human then corrects them. This article explains that such a goal is too narrow for critical software assurance (SwA). Instead, static analysis tools should adopt a broader perspective: computing properties of software.

Static analysis tools (see the sidebar on page 7) are very useful for finding bugs. They go far beyond the capabilities of compilers (warnings) and coding standard checkers to which they are directly related. Like compilers when they generate warnings, static analysis tools aim to detect possible run-time errors (e.g., buffer overflow) and logic errors (e.g., variables not referenced after being assigned). Like coding standard checkers, static analysis tools sometimes allow users to define their own set of patterns to flag. But static analysis tools generally perform much more sophisticated analyses than is typically found in compilers and coding standard checkers (e.g., looking at global context and keeping track of data and control flow).

The appeal of these tools is immediate, providing an almost *yes/no* answer to very hard problems (termed *undecidable* in mathematical terms). But while you're asking *Are there any bugs in this code?*, the tool is actually answering a subtly different question: *Have any bugs been detected in this code?* Thus, when a tool answers *no problems*, it means that it couldn't detect any bugs; it doesn't mean that the code has no bugs. Further, the actual question should be: *Have any shallow/common bugs been detected in this code?* As explained by a team at software integrity company Coverity: "errors found with little analysis are often better" because they are clear errors that a human reviewer will more likely understand [1].

That is the catch. Static analysis tools are not compilers whose output (object code) rarely needs to be inspected. They produce results for humans to review. At the very least, a human needs to understand the problem being reported—and also, in most cases, the reason for the report—in order to assess what, if any, correction to make.

Focusing on Human-Readable Output

Because humans ultimately label each problem reported by a static analyzer as

either a *real error* or a *false alarm* whose ratio is used to evaluate the quality of a tool, commercial tools strive to present the user with understandable warnings supported by explanations. Even trivial changes to the messages may have a large impact. In my own experience working on the static analyzer PolySpace, I was quite surprised on what seemed to be simply a cosmetic change. Messages for warnings had been reworded to reflect the associated likelihood, so that the message *out of bounds array index* associated with certain (red) and possible (orange) warnings was now turned into *Error: array index is out of bounds* and *Warning: array index may be out of bounds*.

The short message is usually accompanied by a link to a page describing the intent of the checker being exercised, and the typical errors that it finds. Some static analyzers also display more contextual information that helps the user in diagnosing the problem. For example, PREFIX, an internal tool at Microsoft, displays whether the problem occurs inside a loop or not, the depth of calls that exhibit the problematic execution, etc.

As most problems only show up in some executions reaching a particular program point, a useful piece of information is the execution path leading to these problematic executions. Static analyzers typically display such paths by coloring the lines of code defining the path (e.g., the first line of each block of code involved). The path may involve function calls, in which case the user can usually unfold the call to follow the path. Some static analyzers even display contextual explanations along the path to help follow the rationale for a given warning.

Still, as Coverity's team puts it, "explaining errors is often more difficult than finding them" [1]. This means that a balance is found in practice between explaining displayed warnings and hiding those warnings that cannot be so easily explained. As a result, real errors—which

are detected but are complex to explain—may fail to be reported: "For many years we gave up on checkers that flagged concurrency errors; while finding such errors was not too difficult, explaining them to many users was" [1].

Static Analysis for Critical SwA

Finding bugs with static analysis tools, even simple bugs that testing would catch is, of course, useful. Embedded systems expert Jack G. Ganssle advocates doing inspections before testing because inspections are 20 times cheaper than writing tests: "It is just a waste of company resources to test first" [2]. As human time is far more expensive than CPU time, the same argument shows that static analysis should be performed before inspections or testing, even for finding simple bugs.

However, the nets that static analysis tools are using to catch bugs have a large mesh, too coarse for critical SwA. One example is integer overflow: adding two large positive integers and getting a negative integer as a result. These are rather unimportant bugs for most commercial static analyzers, and are usually not even advertised on the list of vulnerabilities they look for. There is some rationale as to why integer overflow is not a high priority. At Microsoft Research, I worked with a team that augmented PREFIX with the ability to detect integer overflow bugs—and then applied it to a large Microsoft codebase comprising several million lines of C and C++ [3]. The tool returned with tens of thousands of possible integer overflows—and almost all of them were intended or benign. With special heuristics to hide most false alarms, the tool returned with many fewer warnings (still hundreds). Three days of reviewing warnings finally uncovered 15 serious bugs, most of which were related to security issues. Relying on user review to find a few serious bugs amidst a large number of warnings is not the image that commercial static analyzers are trying to achieve.

Static Analysis for Automated Code Review

Instead of advocating a fully-automated approach that considers human review as a bottleneck in the application of static analysis, some have taken the opposite view and regard static analysis as a mechanism that can expedite manual code review.

Brian Chess and Jacob West from Fortify Software devote a complete chapter in [4] to static analysis as part of the code review process. They consider warnings issued by static analysis tools as clues that a non-trivial safety or security argument has to be made by a human reviewer, based on the fact that “static analysis tools often report a problem when they become confused in the vicinity of a sensitive operation” [4]. They also insist that, whenever possible, a problem found by code review that is not reported by the tool should be the basis for a new custom rule in the static analyzer. Although many tools supply an application programming interface for defining such custom rules, it is not likely that most errors found during code review can be easily encoded into such rules (keep in mind that several organizations can create custom checkers).

Tucker Taft and Robert Dewar have gone further, explaining how to leverage static analysis tools for automated code review [5]. This requires a way to query the internal information computed by the tool instead of just the warnings it issues. They show how to conduct a code review of inputs and outputs, preconditions and postconditions, etc., based on information generated by static analyzer CodePeer. Undoubtedly, making static analysis a partner in code review presents many questions concerning the interaction between the tool and the reviewer: One must determine how much information to display, how to display it, and which queries should have displayed the information. So far, static analysis tools have largely stayed away from this issue because of the difficulties in dealing with the large amount of information available.

However, combining static analysis with code review holds the promise of each method complementing the other, since their strengths are in different areas. Tools are deterministically sound and unsound (whether by design or through errors in the tool itself or in its setup), while humans are unpredictably sound and unsound. I recently co-conducted a very small experiment to compare the results of static analysis and code review for finding bugs in a Tokeneer system [6]

whose security properties were formally verified. The results of this experiment suggest that each method catches bugs the other method misses.

Focus on Humans, Not on Bugs

Orienting static analysis towards automation-assisted code review requires shifting the focus from finding bugs to helping a human understand various issues about the code, from data-flow to exception handling to proper input validation. This does not mean abandoning warnings. On one hand, tools are very good at systematically detecting a clearly defined problem, whereas humans make errors. On the other hand, tools cannot easily deal with the specific project issues or translate informal specifications into code verification activities. Michael D. Ernst believes that “humans are remarkably resilient to partially incorrect information, and are not hindered by its presence among (a sufficient quantity of) valuable information” [7].

The idea is to automate all the things that can be automated, but no more. With enough eyes, all bugs are shallow. We cannot say the same about enough tools. The choice of what is and is not important is best left to a human to decide, provided suitable user interactions are built into the tools. The problem is that static analyzers targeted at bug-finding may not be so easy to re-architect for answering queries from a user. Many of these tools only consider sets of execution paths that do not cover all cases; therefore, they may not easily provide information on all executions.

Tools like PolySpace and Frama-C display ranges of integer variables (and pointer variables for Frama-C) on demand: When the user puts the focus on a variable in the code, the range corresponding to all the possible values of this variable (in all executions) is displayed in a tool-tip or in a side panel. PolySpace uses the same kind of interaction to display all the information it computes about possible run-time errors; it is emphasized by coloring the code using the standard convention of green for *ok*, orange for *warning*, and red for *error*.

Static Analysis for Computing Properties

An absence of run-time errors is the first property that comes to mind when talking about static analyzers. Most tools cannot compute this property, as they are designed to report only a subset of all possible errors and analyze only a subset

of all possible executions. To the best of my knowledge, only three commercial tools compute this property: the PolySpace and CodePeer tools, and the SPARK programming language. By focusing on humans rather than bugs, all three have found ways to solve the *false alarm* problem: PolySpace colors the code and lets users query individual program points for possible run-time errors; CodePeer partitions warnings into three buckets (high, medium, low) with low warnings only presented on user request; and SPARK imposes enough restrictions (checked by static analysis) that the false alarm rate is low (e.g., there can be no read of an uninitialized variable). All of these tools also allow recording manual analysis of warnings for reuse when the code is reanalyzed after being modified.

Absence of run-time errors is not the only property of interest in critical SwA. In fact, it is rather the least interesting property (things behave as they are written), except that it must hold in order for the program to respect any other property, and it could ideally be verified from source code only without any user guidance. Absence of run-time errors is sometimes framed as program correctness, which tends to boost its importance.

In a recent position paper [8], software engineering pioneer David Lorge Parnas warns that this abstract notion of correctness makes no sense in practice: “Correctness is not the issue.” Indeed, correctness is always relative to a given specification and every non-trivial specification is wrong, whether it is formal or informal. The usual *wrongness* is being incomplete. This is especially true for formal specifications, because no existing formal language can express all the properties we expect from a correctly operating system, in particular for embedded software that interacts with the outside world. As an example, a correct compiler is one that must satisfy a number of requirements, including the issuing of useful error messages. No formal language can express this specification. Instead of correctness proofs, Parnas urges static analysis tool writers to focus on property calculation, which is the norm in other engineering fields.

We are interested in two types of properties:

- Functional properties like values, relations, preconditions, postconditions, and dependencies.
 - Non-functional properties like coverage, memory footprint, worst-case execution time (WCET), and profiling.
- Most static analyzers are already capable

of generating functional information because they internally compute program invariants that are predicates describing some constraints respected by the program (e.g., the fact that variable X is positive at some point, or more complex relations between variables like linear inequalities and Boolean combinations of such inequalities that hold at some point). Preconditions and postconditions are special kinds of invariants that are particularly interesting, because they make function interfaces explicit.

The first problem is that a static analyzer may compute a large number of such invariants, most of which are not of interest to the user. As already mentioned, one solution is to let the user indicate which invariants are of interest. Some tools already display the ranges of values taken by variables when a user selects such a variable in the program. Ideally, we would like to provide an arbitrary expression, say $X + Y$, and ask the static analyzer for all invariants at a specific program point that mentions this expression.

A second problem is that many static analyzers do not exactly compute invariants, either because they analyze only one path (or set of paths) at a time, or because they perform unsound simplifications during their analysis. In the former case, the predicate that characterizes the path (or the set of paths) analyzed is usually not easily readable, so simply outputting invariants of the form *predicate-for-the-path* implies *invariant-for-the-path* is unlikely to be useful. Instead, we can imagine that the path (or the set of paths) is displayed by highlighting appropriate lines in the source code (as is already done for warnings)—and that only the invariant part is displayed. Even in the case where the static analyzer performs unsound simplifications (possibly missing a real error), giving access to the internal invariants may help users understand the simplifications performed by the tool. When looking for integer overflow bugs in a large codebase at Microsoft, I found it very useful to have access to the models computed by PREFIX for each function. These models gave the invariants at function exit (postconditions) computed by the tool for a set of paths described by invariants at function entry (preconditions). This was critical to quickly discard warnings caused by an incorrect model computed by the tool, which made it possible to concentrate on actual errors.

Some static analyzers also compute non-functional properties (i.e., properties that are not related to the correctness of the program's computations). Many static analyzers warn about dead code, which is

What Is a Static Analysis Tool?

A static analysis tool (or static analyzer) has three major characteristics:

- Its input is the source code for a program in a programming language.
- It analyzes the program's structure without executing the program.
- As its primary function, the tool outputs information that is relevant to humans developing or maintaining the program.

This general definition includes tools such as coding standard checkers, *bug finders*, and test case generators.

Many static analysis tools attempt to detect problematic constructs. Ideally, such a tool should identify all constructs in a given program—and only those constructs encountering the problem during execution. Unfortunately, mathematical computability theory shows that it is impossible to produce such a tool for analyzing arbitrary programs in any nontrivial programming language. So, in practice, a tool will suffer from either one or both of these deficiencies:

- Failure to detect a problem, yielding what is (perhaps confusingly) known as a *false negative*.
- Mistakenly flagging a correct construct as a problem, yielding a false alarm (known in the literature as a *false positive*).

A tool that does not generate false negatives is said to be *sound*. A tool's precision is a measure of its ability to avoid generating false positives. Soundness and precision are tradeoffs, so the challenge for a tool provider is to strike an appropriate balance.

the same property as code coverage, only seen from the other direction. Although general coverage seems hard to attain by static analysis, unit coverage that considers the coverage of a function's constructs for all possible calling contexts (and thus all values of inputs) is much more feasible. Again, mapping the results of the analysis onto the source code provides the best user interaction here. Generating tests whose execution shows a line of code is also a constructive way to compute the property that a line of code is not dead.

Expanding on this idea, we can imagine giving a predicate at a program point, say $X < Y$, and asking the static analysis tool to produce a counterexample. This is a very efficient way to make progress when the tool does not generate an invariant which, according to the user, should hold. Without such interactions, the user is usually left wondering if the tool was not clever enough to prove the property—or if it holds at all. Additionally, seeing the actual counterexample (instead of only knowing there is one) greatly facilitates understanding of the problem. What is important here is the user interaction, which allows very quick feedback on a question that the user finds interesting.

Specialized static analysis tools already provide information such as memory footprints and WCET. For example, Airbus is using these tools to help certify their programs at the highest levels of the DO-178 avionics safety standard [9]. However, not much work has been done with these tools to provide a rich user interaction at the function level.

New ways of interacting with static

analysis tools are desirable and possible. As a very simple example, some integrated development environments (IDEs) can display the shortest path in the call graph between two functions when a user asks whether one can be called from the other. Other IDEs highlight entities based on syntactic categories, triggered when the user puts the cursor on an entity. Those are the kinds of useful interactions that static analyzers should aim for.

Conclusion

The current emphasis on static analysis will not necessarily provide the tools that are needed for critical SwA, which is based on human assessment of *fitness-for-purpose*. Useful tools are those that compute human-readable properties of the software, providing reviewers with much deeper information than is currently available. The Agile Manifesto [10] correctly recognizes that individuals and interactions should be our main focus for creating useful processes and tools.

One static analysis vendor goes as far as to admit: “No one wants to be on the hot seat when a critical vulnerability is exploited in the field or when a coding mistake causes product recalls, brand damage, or revenue losses.” I do not think that static analysis provides the kind of insurance suggested in [11]; like other systems assurance, critical SwA is not principally a matter of tools, but a matter of “leadership, independence, people, and simplicity” [12].

Static analysis for code review is certainly a very promising venue for critical SwA. Looking even further, static analysis

Software Defense Application

The defense industry—as evidenced by projects such as Software Assurance Metrics and Tool Evaluation—is paying significant attention to static analysis tools. This article helps DoD decision-makers and developers assess and select static analysis tools that meet their safety and security requirements.

used during development (e.g., for code review preparation) can help a programmer understand complex behaviors and detect subtle mistakes—like a “buddy” does in pair programming. In other words, static analysis for humans.◆

Acknowledgements

Many colleagues at AdaCore provided very valuable comments on an initial version of this article, in particular Bob Duff and Ben Brosgol.

References

1. Bessey, Al, et al. “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World.” *Communications of the ACM* 53.2. Feb. 2010 <<http://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later>>.
2. Ganssle, Jack G. “A Guide to Code Inspections.” Vers. 2.1. Feb. 2010 <www.ganssle.com/inspections.pdf>.
3. Moy, Yannick, Nikolaj Bjorner, and Dave Sielaff. “Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis.” *Microsoft Research*. 11 May 2009 <<http://research.microsoft.com/apps/pubs/?id=80722>>.
4. Chess, Brian, and Jacob West. *Secure Programming with Static Analysis*. Chapter 3, “Static Analysis as Part of the Code Review Process.” Upper Saddle River, NJ: Addison-Wesley, 2007 <http://media.techtarget.com/searchSoftwareQuality/downloads/Secure_Programming_CH03Chess.pdf>.
5. Taft, S. Tucker, and Robert B.K. Dewar. “Making static analysis a part of code review.” *Embedded Computing Design*. 16 June 2009 <<http://embedded-computing.com/making-static-analysis-part-code-review>>.
6. Moy, Yannick, and Angela Wallenburg. *Tokeneer: Beyond Formal Program Verification*. Proc. of the Embedded Real

Time Software and Systems Conference. Toulouse, France. 21 June 2010 <www.open-do.org/wp-content/uploads/2010/05/erts2010.pdf>.

7. Ernst, Michael D. *Static and Dynamic Analysis: Synergy and Duality*. Proc. of the Workshop on Dynamic Analysis. Portland, OR. 9 May 2003 <www.cs.washington.edu/homes/mernst/pubs/staticdynamic-woda2003.pdf>.
8. Parnas, David Lorge. “Really Rethinking Formal Methods.” *IEEE Computer* 43.1 (Jan. 2010).
9. Souyris, Jean, et al. *Formal Verification of Avionics Software Products*. Proc. of the 16th Annual Symposium on Formal Methods. Eindhoven, The Netherlands. 2-6 Nov. 2009.
10. Beck, Kent, et al. “Manifesto for Agile Software Development.” Feb. 2001 <www.agilemanifesto.org>.
11. Fisher, Gwyn. “When, Why and How to Leverage Source Code Analysis.” White Paper. 2007 <www.klocwork.com/resources/white-paper/static-analysis-when-why-how>.
12. Haddon-Cave, Charles. *The Nimrod Review: An Independent Review into the Broader Issues Surrounding the Loss of the RAF Nimrod MR2 Aircraft XV230 in Afghanistan in 2006: Report*. London: TSO. 28 Oct. 2009 <<http://ethics.tamu.edu/guest/XV230/1025%5B1%5D.pdf>>.



Homeland Security

The Department of Homeland Security, Office of Cybersecurity and Communications, is seeking dynamic individuals to fill several positions in the areas of software assurance, information technology, network engineering, telecommunications, electrical engineering, program management and analysis, budget and finance, research and development, and public affairs. These positions are located in the Washington, DC metropolitan area.

To learn more about the DHS Office of Cybersecurity and Communications and to find out how to apply for a vacant position, please go to USAJOBS at www.usajobs.gov or visit us at www.DHS.GOV; follow the link Find Career Opportunities, and then select Cybersecurity under Featured Mission Areas.

About the Author



Yannick Moy, Ph.D., is a senior software engineer at AdaCore, where he works on software source code analyzers CodePeer and SPARK, mostly to detect bugs or verify safety/security properties. Moy previously worked on source analyzers for PolySpace (now The MathWorks), INRIA Research Labs, Orange Labs, and Microsoft Research. He holds degrees in computer science: a doctorate from Université de Paris-Sud, a master's from Stanford, and a bachelor's from the Ecole Polytechnique. Moy is also a Siebel Scholar.

AdaCore
46 Rue d'Amsterdam
Paris, France 75009
Phone: +33.1.4970.6716
E-mail: moy@adacore.com

Considering Software Supply Chain Risks[®]

Dr. Robert J. Ellison and Dr. Carol Woody
SEI

As outsourcing and commercial product use increase, supply chain risk becomes a growing concern for software acquisitions. Hardware supply chain risks include manufacturing and delivery disruptions and the substitution of counterfeit or substandard components. Software supply chain risks, usually during development, include third-party product tampering or the introduction of exploitable software defects. This article identifies several current practices that can be incorporated in an acquisition to reduce those risks.

Commercial software is not defect-free. There are any common defects such as improper input validation, as defined by the Common Weakness Enumeration (CWE), The MITRE Corporation's list of software weakness types [1]. These weaknesses can be readily exploited by unauthorized parties to alter the security properties and functionality of software for malicious intent. MITRE, in collaboration with the SANS Institute, publishes a yearly list of the Top 25 Most Dangerous Programming Errors [2]. Such defects can be accidentally or intentionally inserted into software, and subsequent acquirers and users have limited ways of finding and correcting these defects to avoid exploitation.

A report by application security company Veracode [3] draws on the analysis of billions of lines of code and thousands of applications that they have analyzed. Their overall finding is that most software is very insecure. Regardless of software origin, 58 percent of all applications submitted for verification did not achieve an acceptable security score for its assurance level upon first submission to Veracode for testing. Table 1 has the results (by source) of software tested against the 2009 CWE/SANS Institute Top 25 list [4]; it shows the percentage of submitted software that passed the security test on the first trial. As 60 to 70 percent of the tested software failed against easily remedied weaknesses, one of Veracode's findings was the lack of developer education and motivation on secure coding.

Software Supply Chain Complexity

There has been extensive analysis of supply chains for delivery of physical material, an analysis based on data collection over decades of practice. The lack of an equivalent base of practice and data collection for software has severely limited the analysis and response to software supply chain risks.

Most supply chains are not a single link between an acquirer and a supplier. A more complex supply chain (such as that shown in Figure 1 on the next page) can involve a combination of internal development, outsourced development, multiple commercial suppliers, and legacy system usage. The composite system inherits the risk of a software assurance (SwA) failure at any point in such a supply chain. The acquirer and the primary supplier have limited visibility of the capabilities of deeply-nested sub-suppliers. Supply chain risks can be reduced but not eliminated. Once software is deployed, residual supply chain risk identification and mitigation become a continuing responsibility for the acquiring organization.

Software supply chain risk considerations must continue in sustainment. An assessment done as part of the initial acquisition for a commercial component is valid only at that time. A commercial software component can easily be deployed for five years or longer. During that period, the following can happen:

- New attack techniques and software weaknesses appear.
- Changes in acquirer usage activate product features with weaknesses that have not been considered in earlier assessments.
- A sequence of product upgrades that add features or change design can invalidate a risk assessment.
- Changes occur in the risk factors used in initial vendor and product assessment (e.g., corporate merger, subcontractors, corporate policies and staff training, or in the corporate software development process).
- Product criticality increases with new or expanded usage.

Mitigating Common Software Weaknesses in the Supply Chain

Addressing the appearance of common software weaknesses introduced in a supply chain requires knowing where to look and what to look for. Discussions of system security often include firewalls,

authentication issues (such as password strength), or authorization mechanisms (such as role-based access controls). Application security has often been ignored, in part because of the faulty assumption that firewalls and other perimeter defenses can protect the functional code. The problem is further compounded as application developers without specific security training are typically unaware of the ways their software, while meeting functional requirements, could be compromised. Security software—such as a firewall or a password management component—is usually subject to an independent security assessment that considers the development history as well as the design and operational context. There is no equivalent effort applied to the security of application components.

The pervasiveness of easily remedied weaknesses (as observed by Veracode) provides a simple attack vector that is easily exploited. A first step should be the elimination of the most pervasive common weaknesses, particularly from acquired application software.

There is currently insufficient practice data to identify best practices that could be required of suppliers, but our observation of current practice suggests activities that can improve confidence in a software supply chain [5].

Security for application software is getting increased commercial attention. In 2006, Microsoft established their Security Development Lifecycle (SDL), which served as a starting point for other efforts [6]. Today, more than 25 large-scale application software security initiatives are

Table 1: *CWE/SANS Top 25 Compliance*

Software Source	Acceptable
Outsourced	6%
Open Source	39%
Internally Developed	30%
Commercial	38%

© Copyright 2010 by Carnegie Mellon University.

under way in organizations as diverse as multinational banks, independent software vendors, the U.S. Air Force, and embedded systems manufacturers. The Software Assurance Forum for Excellence in Code, an industry-led non-profit organization that focuses on the advancement of effective SwA methods, published a report on secure software development [7]. In 2009, the first version of the Building Security In (BSI) Maturity Model [8] (BSIMM) was published¹. The Software Assurance Processes and Practices Working Group² has released several relevant documents, including [9], which is linked to the Capability Maturity Model Integration for Development. In addition, the Open Web Applications Security Project has developed a Software Assurance Maturity Model for software security [10]. Finally, the BSI website at <https://buildsecurity.in.us-cert.gov> contains a growing set of reference materials on software security practices.

The emerging collection of secure development techniques arose from addressing specific software weaknesses. The following section considers three classes of software weaknesses as a way to explain the criticality of software design and coding mistakes.

Common Weaknesses in Applications

Three common weaknesses—cross-site scripting (XSS), SQL injection, and cross-site request forgery (CSRF)—appear in the top four of the 2010 CWE/SANS list. Topping the list is XSS, which can compromise a user’s computer when they view a page on what they consider to be a trusted site. Next is SQL injection, an attacker technique that can compromise applica-

tions that query databases (e.g., where credit card data has been illegally downloaded). Ranked fourth is CSRF, where an attacker can masquerade as a trusted user of a web server only to upload malicious data to that server.

XSS

Web traffic consists of a mixture of data and script in HTML. With XSS, the attackers objective is to have users retrieve a Web page from your server that contains malicious code, say in JavaScript that the attacker wrote. The user trusts your server, and their browser will execute the malicious code as if it came from you. This vulnerability is a design error that allows the attacker to get their input into your server.

SQL Injection

Weaknesses are often associated with malformed input. The vulnerability risk is high when an application incorporates user input into a service request. Assume we have an application that displays an employee name and salary after a user enters an employee ID. If a user enters 48983, then a database query is created to retrieve all entries that satisfy the relation $ID = 48983$. An attacker’s objective is to see if the input routine will accept values that might provide additional information. The classic SQL injection example would be equivalent to the input of 48983 or $(1 = 1)$. If this input is accepted, then the query returns all entries where the $ID = 48983$ or where $1 = 1$. As the latter is always true, all employee records are returned.

CSRF

A CSRF is sort of the reverse of an XSS.

An attacker compromises a user so that the attacker can masquerade as that user, accessing their Web site and making requests. A CSRF that inserts data—combined with XSS to distribute that data—can lead to extensive and devastating consequences (e.g., XSS worms that spread throughout very large Web sites in a matter of minutes).

Emerging Secure Development Practices

Two types of analysis—one focused on understanding and controlling the software attack surface and the other focused on understanding potential threats (threat modeling)—are good examples of SwA practices that can be incorporated early in the development life cycle and that help make supply chain security risk management more tractable. A software attack surface is a way of characterizing potential attack vectors for compromising application code. Threat modeling characterizes which aspects of the attack surface are most at risk for exploitation. These concepts are useful during development, deployment, and system operation. They help guide what information must be gathered and how it can be best used to help prioritize and mitigate (if not eliminate) supply chain security risks.

Attack Surface Analysis

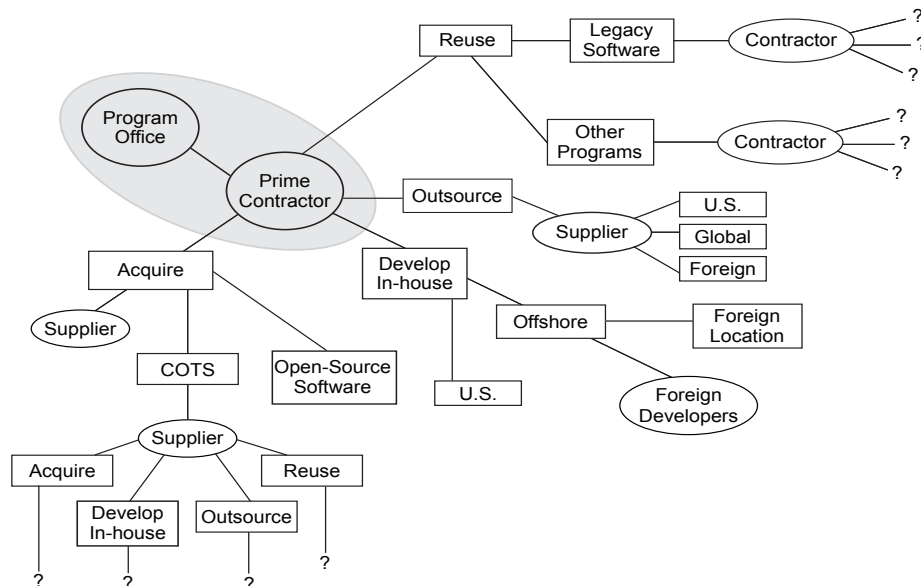
An approach to managing the scope of the software security analysis arose from pragmatic considerations. SDL developer Michael Howard observed that attacks on Windows systems typically exploited a short list of features such as open ports, services running with total access control, dynamically generated Web pages, and weak access controls [11]. Instead of counting bugs in the code or the number of vulnerability reports, Howard proposed to measure the attack opportunities, a weighted sum of the exploitable features.

An attack-surface metric is used to compare multiple versions or configurations of a single system. It cannot be used to compare different systems.

Howard’s intuitive description of an attack surface led to a more formal definition (in [12]), with the following dimensions:

- **Targets.** Data resources or processes desired by an attacker; for example, a process could be a Web browser, Web server, firewall, mail client, database server, etc.
- **Enablers.** The other processes and data resources used by an attacker, such as Web services, a mail client, or

Figure 1: Software Supply Chain



having JavaScript or ActiveX enabled. Mechanisms such as JavaScript or ActiveX give the attacker a way to execute their own code.

- **Channels and Protocols** (Inputs and Outputs). These are used by an attacker to obtain control over targets.
- **Access Rights.** Control is subject to constraints imposed by access rights. An attack surface analysis reduces supply chain security risk in several ways:
 - A system with more targets, more enablers, more channels, or more generous access rights provides more opportunities to the attacker. An acquisition process designed to mitigate supply chain security risks should include requirements for a reduced and documented attack surface.
 - The use of product features influences the attack surface for that acquirer. The attack surface can define the opportunities for attacks when usage changes.
 - It helps to focus attention on the code that is of greatest concern for security risk. If the code is well partitioned so that features are isolated, reducing the attack surface can also reduce the code that has to be evaluated for threats and weaknesses.
 - For each element of a documented attack surface, known weaknesses and attack patterns can be used to mitigate risks.
 - The attack surface supports deployment, as it helps identify attack opportunities that could require additional mitigation.

Threat Modeling

Threat modeling is a part of Microsoft's SDL [6, 13], but it is a general purpose activity that can easily be incorporated into any development life cycle. Identified as one of 10 low-cost suggestions that improve enterprise security [14], threat modeling:

- Provides a business justification for security by mapping threats to business assets.
- Enables a thoughtful conversation around risk and trade-offs during software development in an objective, quantifiable way.
- Encourages a logical thought process in determining an application's security model.
- Lets architects and developers work together to understand threats at design time and build security in, instead of hoping that the quality assurance team can discover those threats later in the life cycle.

Software Defense Application

Supply chain risks are dangerous for software acquired and utilized by the defense industry. This article examines significant supply chain risks, such as the inadvertent introduction of exploitable software defects during development and third-party product tampering. This article squarely puts responsibility on the acquirer for avoiding supply chain problems, and provides several techniques that will assist defense industry software acquirers in focusing their risk mitigation. These methods will improve software quality, in turn reducing expenses—especially in regards to exploitation recovery and system patching.

- Helps business analysts understand and create traceable security requirements.

The approach used in threat modeling is applicable to other risk assessment methodologies. Data flows or usage scenarios are identified along with critical business assets. A detailed walkthrough of a data flow considers the deployed configuration and expected usage, identifies external dependencies (such as required services), analyzes the interfaces to other components (inputs and outputs), and documents security assumptions and trust boundaries (such as the security control points). The usage scenarios can support business justifications and link threats to the criticality of business assets. Such a walkthrough can consider adversary motivations (such as the criticality of the data being handled), in addition to the technical risks.

Fuzz Testing

Increased attention on secure application software components has influenced security testing practices. All of the organizations contributing to the BSIMM do penetration testing, but there is increasing use of fuzz testing. Fuzz testing creates malformed data and observes application behavior when such data is consumed. An unexpected application failure, due to malformed input, is a reliability bug and possibly a security bug. Fuzz testing has been used effectively by attackers to find weaknesses. For example, in 2009, a fuzz-testing tool generated XML-formatted data that revealed an exploitable defect in widely used XML libraries. At Microsoft, about 20 to 25 percent of security bugs in code—not subject to secure coding practices—are found via fuzz testing [6].

Using Secure Development Practices in the Software Supply Chain

Let's see how our examples of secure development practices could be applied to the acquisitions of commercial software components. Inputs to that analysis include organization-specific information and available data on vendors and products. The key questions are: *Has the developer con-*

sidered how the software could be exploited? and Has behavior under unexpected or adverse conditions been analyzed? The evidence to answer those questions can be drawn from coding practices, static code analysis, common weaknesses analysis, attack patterns analysis, threat/vulnerability analysis, software security testing, and dynamic testing.

Techniques such as attack surface analysis, threat modeling, and fuzz testing could play multiple roles in commercial software acquisition.

Assume a commercial component is part of a larger contracted system development acquisition. In this instance, the commercial components are selected by the primary contractor. Supply chain analysis could include examining:

- The attack opportunities the component exposes in terms of features and implementation (component developer, prime contractor, or independently developed).
- The identification and mitigation of risks by the component developer (e.g., supplier fuzz testing, supplier threat modeling [or the equivalent], independent assessment, contractor fuzz testing, and acquirer fuzz testing as part of acceptance and continued for product upgrades during sustainment).
- The criticality of risks for the planned usage (contractor threat modeling as a basis for discussions with the acquirer).
- Risk mitigations (acquirer trade-off decisions with respect to functionality, costs, and acceptable risks based on contractor threat modeling).

Also note that development artifacts should include documented supply chain and threat-modeling analysis provided by the contractor to the acquirer.

Acquirer Responsibilities

Supply chain risks continue during sustainment. A documented attack surface and threat-modeling analysis—provided by a vendor—would influence the acquirer's future responses to changes in usage, threats, or supporting technologies, and should be incorporated into contracting efforts done during sustainment.

While part of the responsibility for

supply chain assurance can be outsourced to a prime contractor, the supply chain risks for individual systems have to be aggregated. For all deployed systems, the responsibility for the aggregation of supply chain risks falls to the acquirer. Software acquisition has grown from the delivery of standalone systems to the provisioning of technical capabilities integrated within a larger system-of-systems (SoS) context. This integration extends the criticality of supply chain risk analysis. Software security defects in any of the products or services are a potential supply chain security risk to all SoS participants. A set of one-off approaches for individual system supply chain assurance creates a nearly impossible task for an SoS.

Summary

A software supply chain objective should be to incorporate the identification and mitigation of likely design, coding, and technology-specific weaknesses into the development life cycle. This article provides an analysis of three practices that support that objective. Mitigations of items on a CWE/SANS Top 25 list are usually linked to detailed design or coding practices, but mitigations are also associated with risk analysis, requirements, architecture, and testing. This article—and sources like the BSI Web site—provide a foundation for establishing a full life-cycle context for security improvement. ♦

References

1. The MITRE Corporation. *Common Weakness Enumeration*. 17 May 2010 <<http://cwe.mitre.org>>.
2. The MITRE Corporation. “2010 CWE/SANS Top 25 Most Dangerous Programming Errors.” *Common Weakness Enumeration*. 5 Apr. 2010 <<http://cwe.mitre.org/top25>>.
3. Veracode, Inc. *State of Software Security Report*. Vol. 1. 1 Mar. 2010 <www.veracode.com/reports/index.html>.
4. The MITRE Corporation. “2009 CWE/SANS Top 25 Most Dangerous Programming Errors.” *Common Weakness Enumeration* 29 Oct. 2009 <http://cwe.mitre.org/top25/archive/2009/2009_cwe_sans_top25.html>.
5. Ellison, Robert, et al. *Evaluating and Mitigating Software Supply Chain Security Risks*. SEI, Carnegie Mellon University. Technical Note CMU/SEI-2010-TN-016. May 2010 <www.sei.cmu.edu/reports/10tn016.pdf>.
6. Howard, Michael, and Steve Lipner. *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006.
7. Bitz, Gunter, et al. *Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today*. 8 Oct. 2008 <www.safecode.org/publications/SAFECode_Dev_Practices1008.pdf>.
8. McGraw, Gary, Brian Chess, and Sammy Migues. *The Building Security In Maturity Model – BSIMM2*. 2010 <www.bsi-mm.com>.
9. DHS. *Build Security In – Software Assurance*. “Process Reference Model for Assurance Mapping To CMMI-DEV V1.2.” 23 June 2008 <http://buildsecurityin.us-cert.gov/swa/downloads/PRM_for_Assurance_to_CMMI.pdf>.
10. The Open Web Application Security Project. “Software Assurance Maturity Model.” 5 May 2009 <www.owasp.org/index.php/Category:Software_Assurance_Maturity_Model>.
11. Howard, Michael. “Fending Off Future Attacks by Reducing Attack Surface.” *Microsoft Developer Network*. 4 Feb. 2003 <<http://msdn.microsoft.com/en-us/library/ms972812.aspx>>.
12. Howard, Michael, Jon Pincus, and Jeannette M. Wing. *Measuring Relative Attack Surfaces*. 2003 <www.cs.cmu.edu/~wing/publications/Howard-Wing03.pdf>.
13. Swiderski, Frank, and Window Snyder. *Threat Modeling*. Redmond, WA: Microsoft Press, 2004.
14. McGovern, James, and Gunnar Peterson. “10 Quick, Dirty, and Cheap Things to Improve Enterprise Security.” *Security & Privacy* 8.2 (Mar.-Apr. 2010): 83-85.

Notes

1. BSIMM was created from a survey of nine organizations with active software security initiatives considered to be the most advanced. The nine organizations were drawn from three sectors: financial services (4), independent software vendors (3), and technology firms (2). Those companies among the nine who agreed to be identified include Adobe, The Depository Trust & Clearing Corporation, EMC, Google, Microsoft, Qualcomm, and Wells Fargo.
2. The group operates under the sponsorship of the DHS’s National Cyber Security Division. See <<https://buildsecurityin.us-cert.gov/swa/procwg.html>>.

About the Authors



Robert J. Ellison, Ph.D., is a member of the Survivable Systems Engineering Team within the Community Emergency Response Team Program at the SEI, and has served in a number of technical and management roles. Ellison regularly participates in the evaluation of software architectures and contributes from the perspective of security and reliability measures. His research draws on that experience to integrate security issues into the overall architecture design process. Ellison is currently exploring reasoning frameworks development to help architects select and refine design tactics to mitigate the impact of a class of cyber-attacks.

SEI
Carnegie Mellon University
4500 Fifth AVE
Pittsburgh, PA 15213-3890
Phone: (412) 268-7705
Fax: (412) 268-5758
E-mail: ellison@sei.cmu.edu



Carol Woody, Ph.D., is a senior member of the technical staff at the SEI. She leads the acquisition and development practices and metrics team, addressing research in four critical areas for security in software: security requirements, cyber assurance, the software supply chain, and measurement. She is experienced in all aspects of software and systems planning, acquisition, design, development, and implementation in large complex organizations. Woody is a senior member of the Association for Computing Machinery and the IEEE.

SEI
Carnegie Mellon University
4500 Fifth AVE
Pittsburgh, PA 5213-3890
Phone: (412) 268-9137
Fax: (412) 268-5758
E-mail: cwoody@cert.org

Information Assurance Applications in Software Engineering Projects

Lt. Col. Thomas A. Augustine (Ret.) and Dr. Lori L. DeLooze
United States Naval Academy

Four recent capstone projects by students in the U.S. Naval Academy's (USNA) Department of Computer Science offer some interesting insights into methodologies for information assurance (IA). This article looks at the tasks and challenges of each project and consolidates the experiences into lessons learned for designing and implementing software or systems that incorporate the IA concepts of confidentiality, data integrity, authentication, and system availability [1].

One USNA requirement (for computer science or IT undergraduates) is a capstone project. Students—in groups of three or four on a project of their choosing—must find a customer, define requirements, and meet key milestone dates in providing a software or system artifact. Projects require about 150 hours per person and must be completed and fully documented within the 15-week semester.

Over the past two years, there has been increased student motivation to choose IA-related projects. Like software or systems engineering projects in other fields, students found it especially challenging to define customer requirements and meet expectations and milestones. Faculty use these challenges as learning opportunities by allowing students to make their own project decisions, even if poor decision-making leads to a mid-project failure, because these failures will teach the students much more than a perfectly executed plan. Students found that taking on projects in the IA field of study created additional challenges in subject matter knowledge, system design, and implementation.

Team 1: Training Database with Multi-Level Views

The first project required students to develop a training and personnel database that provides proper authentication at an undetermined number of organizational or data visibility levels and minimizes repetition of data entry by using data normalization. This group found the requirements-gathering process to be fairly straightforward, as the customer understood the concept of a database with multiple levels of security. These requirements included allowing designated individuals to input and view multiple training courses as well as providing status reports to higher-level managers.

This team designed the software with an initial authentication scheme and then created a secure session that verified credentials before displaying data. *Read*, *write*, and *modify* rules were given based on data-

base views, combining multiple tables in various views. The database administrator programmed these views, which had the capability of providing granular permissions. Originally, the team planned to hard-code permissions as *read* and *modify* for all levels higher than the supervisor, meeting the customer requirements. However, after initial design review, they realized that these requirements may later change; therefore, the team redesigned access control by giving the database administrator the ability to set visibility by level or by allowing the overriding of permissions. This additional flexibility added the capability to produce a report by giving full permissions by person, group, and supervisory levels, as well as highlighting all overridden permissions.

Team 2: Emergency Notification System

The next project¹ required an undetermined number of individuals and organizations to receive emergency notification of events based on input from city, state, nationwide, or worldwide sensors. Some events only need to be seen by the local emergency services while others required visibility for governors, the military, or federal officials. Some events are simply logged, while higher-priority events may require confirmation from the appropriate source and the ability to forward data to higher levels for additional action.

This team also had a challenge with the design and implementation of permission and authentication methods. Unlike the first group, however, these students did not initially incorporate authentication at the design phase. While they understood the requirements to have multiple levels of reporting based on the users' position and authentication, the team decided to put this off until the implementation phase. Because specifying the design is arguably the most difficult step in the software engineering process, many students simply want to get started with the implementation phase. These students started imple-

menting code based on a poorly elaborated design. They split up work, each building separate Web pages for a type of emergency reporting required. This resulted in disparate pages that looked and operated differently and had no means of accepting dynamic changes. In addition, the team realized that they designed authentication by page or view rather than providing a consolidated, centralized means of authentication and a data permission schema.

Though reluctant to scrap six weeks of work, the team ultimately chose to begin from scratch and start again at the requirements phase. At this point, they developed formal interview questions for the customer and wrote a concise statement that encompassed all requirements. They then took each sentence or phrase and turned it into a well-defined requirement placed in a requirements implementation and testing matrix. From this matrix, the students created a design that incorporated every requirement. These requirements specified authentication and visibility schemas for each view. After further analysis, the team was able to design a method for centralized authentication and visibility with a small change to the database schema.

The team was able to re-accomplish requirements analysis and design in only a week, and was able to implement the backend database in another week. The team admitted that they had their doubts about whether they could finish the project on time, but were surprised to see the ease of implementing and testing well-defined requirements and design.

Team 3: Cybersecurity Competition Framework

This project—stemmed from a Polytechnic Institute of New York University competition—had students from numerous schools downloading various cybersecurity and digital forensics exercises that were timed and graded for accuracy and completeness.

USNA students felt that they could

improve the competition by designing a better interface for serving and grading the cyber challenges. As such, they set out to create a Web-based software framework that served various scenarios and received team responses for an IA competition. Requirements included authentication and proper visibility of scenarios for an undetermined number of teams, competition referees, scenarios, and team responses. Additionally, since they were creating a framework for a hacking competition, they had to design a system that maintained the integrity and availability of the data despite possible hijack attempts from less scrupulous teams.

In consultation with the faculty, students chose to both build the framework that was to serve the cyber challenges and create individual challenges as a proof of concept for their serving framework. As such, they assigned two team members to build the framework, while two others independently built challenges. The requirements called for a broad range of possible cyber challenges including: digital forensics of disk images, analysis of network traffic, analysis of software code vulnerabilities, and the identification and mitigation of operating system and applications security configurations.

Like the others, this team started by gaining detailed requirements for which protection against common software vulnerabilities was key. They quickly realized that system security had to be built into the design process. With this additional requirement, they realized the design would be the most difficult aspect of their project and allocated additional time for this milestone. Before designing in security, the team—both to protect their infrastructure and to develop challenges for the competition framework—had to understand how hackers use vulnerabilities to get into systems. Each student chose to specialize in specific network, operating system, application, and database security techniques. To better understand these techniques, students reviewed previous coursework, examined DoD and National Security Agency (NSA) security guides², and interviewed network security administrators. They found that the most difficult portion of securing an application against hackers is not the actual implementation of a specific configuration or fix, but in thinking like the hacker and predicting how people will use potential vulnerabilities to disrupt operations.

Though the team found implementation of the database and associated views to be relatively trivial, they found the documentation process to be challenging.

Documentation had to include reasons for their design decisions and security settings, so that future maintainers could add features but still capitalize on the security features built into the framework.

Students noted that there was great value in understanding and implementing the security guides. While their IA course had many hands-on experiences, it was only through the course of this project that they realized the complexity of securing applications.

Team 4: Cyber Defense Exercise

In this project, students were required to design and implement a complete network based on an intricate set of constraints. After implementation, students had to operate and defend this network against NSA experts posing as attackers. Called the Cyber Defense Exercise [2], the competition is modified annually to increase the cybersecurity skills required of student participants. Several years ago, the focus was on active defense, while the more recent exercises have focused on the trade-offs that need to be made between limited resources, operations of a network, security, time, and expertise required.

Students were provided with a 40-page directive spelling out the rules of the competition along with listing the network services that must be provided during a week-long exercise (and the points to be deducted if these services were either not operational or had security compromises). Essentially, students were given a very detailed requirements document with total freedom to produce any design. Though students were asked to turn in their designs, referees only verified that they met budgetary constraints. Cross-referencing requirements to design was a task left totally to each competing student group.

Though students were not required to gather requirements from an external customer, they did have to interpret the directive and design a complete network given the assumptions, constraints, and requirements. Students were challenged with creating a design that could provide users with a number of services, such as e-mail, chat, Web, databases, file servers, and mission-specific applications. This design had to remain operational while withstanding attacks from NSA network experts posing as hackers.

Student team members had taken both networking and IA courses, yet there was still a great deal of knowledge needed for the secure design of an operational network. Students augmented their knowledge

of secure design with NSA security guides, Defense Information Systems Agency security checklists³, and various security-specific books and references. Despite these numerous references, students were still challenged with consolidating this information and meeting the required constraints. Perhaps the students' greatest challenge was verifying the security of their design and implementation, which was done creating a test environment that mimicked the actions of the attackers. The team used security testing tools like the Metasploit Framework (which provides pre-packaged exploits) to test if the system is vulnerable to attack. Students also used other sites like <www.milw0rm.com> to test their system against additional potential exploits; however, the use of these more advanced techniques required great experience and training.

In addition to the testing of security, students had challenges in ensuring that the tightened security did not impact network operations. The students found this to be a great challenge. This balance between continued operations versus increased security involves business case and risk analysis, a skill that generally requires expertise in both network security and the mission area supported by the system.

Lessons Learned

Through these student projects, we can learn a number of security-related lessons about gaining requirements, as well as designing and implementing IA-focused systems or software.

Design Authentication and a Data Permissions Schema Early

Nearly all application or system development requires authentication methods. Students found that the best results were achieved by planning for both position-level and personal-level authentication for data visibility during the design phase. Even when requirements only call for simple authentication, customers tend to ask for a layered authentication by data type, organizational position, or data view. Rework tends to be extensive and time-consuming. Planning for authentication in the design phase of systems will likely save time and resources in the long run.

Use Security Guides

Students found that despite more than 100 classroom hours spent learning about networks and IA techniques, additional application-specific information is required when designing and developing IA-focused applications or systems. The NSA and the Defense Information Systems

Agency produce security guides for various operating systems and applications. These guides have been produced and tested by numerous experts and can complement the developers' knowledge to meet design specifications for applications requiring a cybersecurity focus.

Test Applications Against Known Security Frameworks

Relatively few software or system developers have the skills required to test system designs and implementations against well-known attacks using exploits in systems availability, data confidentiality, and integrity. Rather than recreating exploits that may require a greater effort than actually securing the system or application, system testers are encouraged to use existing security testing frameworks. While these testing frameworks can demonstrate potential holes in security, they should be used in concert with secure programming techniques, design, as well as documented and tested security techniques.

Plan for Regression Testing

Students working on these projects noted the need for an updated, descriptive test plan and follow-on regression software testing. This is true in any software application, but tends to be highlighted in security-focused applications. Security enhancements are rarely made in a single place. Instead, these changes are made in the operating system, database framework, application, and various configuration files. Students found that a single change in any one of these areas forced regression errors that were very difficult to detect without a well-formulated and implemented test plan. Students learned that this testing needs to be done as changes are made, or it becomes necessary to back out entire blocks of changes to find the root cause of bugs.

Manage Security Expectations

Customers generally understand those requirements and expected outcomes that are directly related to their subject matter expertise. Through these projects, students noted that customers expect an application to be secure, but do not understand the resource costs or operations tradeoffs required to make this a reality. Students noted the need to manage customer security expectations in the requirements phase and later in the design phase. Students believed that the best way to manage security requirements and associated customer expectations was to provide a security, operations, and resource matrix that cross-references security trade-offs.

Software Defense Application

Through the lessons learned by USNA students, this article is a refresher for defense software developers on why it is important to design authentication and data permissions early, follow security guides, use existing techniques for security testing, do regression testing as changes are made, ensure that customers understand security costs and tradeoffs, recognize the unintended impact of users on security, and thoroughly document all elements of security architecture.

Understand Security Impact on Operations

Even after managing customers' security expectations and implementing security (expected to exceed requirements), students found that users can have the greatest unintended impact on security. In testing these applications with actual users (as they would use them), students found that users will bypass security, in turn impairing operations or user-expected procedures. These user-caused workarounds can reduce security effectiveness and give the application owner a false sense of security. Students learned that for security controls to remain effective, designers must understand existing user processes and procedures—and then design security architectures around these or build them in a user-friendly alternative.

Document Reasons for Security Architecture

Like many software professionals, students found project documentation among the most challenging processes. As a learning tool, students were required to make changes to projects based on documentation that either they created or (in some cases) was created by other student teams. Though effective documentation is always challenging, students found this difficulty was magnified when trying to modify security architectures. Ultimately, they noted that understanding the security architecture

documentation is not enough to effectively make changes to security without impacting operations or functionality. Instead, students found it easier to effectively manage security changes when they had documentation that also explained the reason for decisions, limitations in technology, the state of the intended operating system's security, and operational or process tradeoffs associated with security decisions. ♦

References

1. Maconachy, W. Victor, et al. *A Model for Information Assurance: An Integrated Approach*. Proc. of the 2001 Workshop on Information Assurance and Security. West Point, N.Y. 5-6 June 2001.
2. Augustine, Thomas, and Ronald C. Dodge, Jr. *Cyber Defense Exercise: Meeting Learning Objectives thru Competition*. Proc. of the 10th Colloquium for Information Systems Security Education. Adelphi, MD: 61-67.

Notes

1. This project was inspired by the 2007 film "Live Free or Die Hard," where terrorists took control of emergency services, the electric grid, and city-wide traffic signals.
2. For access to these guides, visit: <www.nsa.gov/ia/guidance/security_configuration_guides>.
3. See <<http://iase.disa.mil/stigs/checklist>>.

About the Authors



Lt. Col. Thomas A. Augustine, D.Sc., is retired from the United States Air Force after a career in communications and information. His most recent assignment was as an assistant professor of computer science at the USNA, specializing in networks and IA.

E-mail: thomas.augustine@hotmail.com



Lori L. DeLooze, Ph.D., retired from the United States Navy as a career information professional. She is currently an assistant professor of computer science at the USNA, specializing in software engineering and IA.

**572 Holloway RD, Stop 9F
Annapolis, MD 21402
Phone: (410) 293-6820
E-mail: delooze@usna.edu**

Studying Software Vulnerabilities

Dr. Robin A. Gandhi, Dr. Harvey Siy, and Yan Wu
The University of Nebraska at Omaha

There have been several research efforts to enumerate and categorize software weaknesses that lead to vulnerabilities. To consolidate these efforts, the Common Weakness Enumeration (CWE) is a community-developed dictionary of software weakness types and their relationships. Yet, using the CWE to study and prevent vulnerabilities in specific software projects is difficult. This article presents a novel approach for using the CWE to organize and integrate the vulnerability information recorded in large project repositories.

Many vulnerabilities in today's software products are rehashes of past vulnerabilities. Developers are often unaware of past problems or they are unable to keep track of vulnerabilities that others have reported and solved. Interestingly, this is not because of a scarcity of information. In fact, a plethora of information about past vulnerabilities is available to developers. Most software development projects dedicate some effort to documenting, tracking, and studying reported vulnerabilities. This information is recorded in project repositories, such as change logs in source code version control systems, bug tracking system entries, and mailing list communication threads. As these repositories were created for different purposes, it is not straightforward enough to extract useful vulnerability-related information. In large projects, these repositories store vast amounts of data, oftentimes burying the relevant information. Therefore, efforts to summarize lessons learned from past vulnerabilities in a software project are essentially non-existent. In the face of growing software complexity, it is even more critical to improve the mental model of the software developer to sense the possibility of vulnerability.

The CWE standardization effort provides a unified and measureable set of software weaknesses for use in software assurance activities [1]. CWE is a community-driven and continuously evolving taxonomy of software weaknesses. According to [1], the CWE vision is twofold, enabling:

- A more effective discussion, description, selection, and use of software security tools and services that can find weaknesses in source code and operational systems.
- A better understanding and management of software weaknesses related to architecture and design.

However, the CWE is often compared to a kitchen sink, as it aggregates weakness categories from many different vulnerabil-

ity taxonomies, software technologies and products, and categorization perspectives. While the CWE is comprehensive, using its highly tangled web of weakness categories is a daunting task for stakeholders in the software development life cycle (SDLC).

The unique characteristics of a weakness—its preceding design or programmer errors, resources/locations that the weakness occurs in, and the consequences that follow the weakness (such as unauthorized information disclosure, modification, or destruction)—are either expressed together within a single CWE category or spread across multiple categories. Such complexity makes it difficult to trace the information expressed in the CWE to the information about a discovered vulnerability in multiple project-specific sources (such as a log of code changes, source code differences, developer mailing list discussions around bugs, bug-tracking databases, vulnerability databases, and public media releases). Therefore, to facilitate CWE use in the study of vulnerabilities, we have developed easy-to-understand templates for each conceptually distinct weakness type. This template can then be readily applied to aggregate and study project-specific vulnerability information from source code repositories.

Each template is a collection of concepts related to a single weakness type. The concepts are identified by extracting and distilling information from all relevant CWE categories for a particular weakness type. Since the concepts in the templates provide meaning to the usage of certain words and sentences that describe vulnerability information, we call them *semantic templates*.

While the CWE is a collection of abstract categories, the Common Vulnerability Enumeration (CVE) is an ever-growing compilation of actual known information security vulnerabilities and exposures, as reported by software development organizations, coordination centers, developers, and individuals at

large. CVE assigns a common standard identifier for each discovered vulnerability to enable data exchange between security products and provide a baseline for evaluating coverage of tools and services [2].

In this article, we outline the process of building a semantic template to study the injection software weakness type. In recent times, injection is the single most exploited weakness type. It occurs upon failure to adequately filter user-controlled input data for syntax that can have unintended consequences on the program execution. As stated in CWE-74, a distinguishing characteristic of the injection weakness is that “the execution of the process may be altered by sending code in through legitimate data channels, using no other mechanism” [3]. For example, consider a Web application that accepts user input to dynamically construct a Web page that is instantly accessible to other users. Web blogs, guest books, user comments, and discussion pages typically provide such functionality. If the user input that gets included in the dynamic construction of a Web page is not appropriately sanitized for HTML and other executable syntax (e.g., JavaScript), then active user-chosen Web content (such as redirection to malicious Web pages) can be injected into the Web application and later served to other clients that load the tainted Web page in their browsers. This instantiation of the injection weakness is most commonly referred to as cross-site scripting (XSS). As observed in CWE-79 [4], the structure of the dynamically generated Web page is altered by sending code (HTML and JavaScript) in through legitimate user input channels to the Web application.

We also discuss the application of the injection semantic template to study artifacts related to a confirmed XSS vulnerability (CVE-2007-5000, see [5]) in the Apache HTTP Server project. For the interested reader, we have previously elaborated on the buffer overflow semantic template in [6].

Building a Semantic Template

When it comes to security vulnerabilities, we face an interesting paradox. On one end, we are inundated with discovered vulnerability information from its detection to its fix. On the other end, there is most often a lack of security knowhow among stakeholders in the SDLC. We realize that during software development, especially in the implementation stage, the details a programmer has to remember to avoid security vulnerabilities can be enormous. The mere existence of long checklists and guides (such as the CWE) is not enough. To deal with enormous details, the use of long checklists needs to be facilitated by simple cognitive guides or templates. Therefore, to effectively and quickly study the large amounts of information associated with vulnerabilities, we ask the following four fundamental questions:

1. What are the software faults? In other words, what are the concrete manifestations of errors in the software program and design related to omission (lack of security function), commis-

sion (incomplete security function), or operational (improper usage) categories that can precede the weakness?

2. What are the defining characteristics of the weakness?
3. What are the resources and locations where the weakness commonly occurs?
4. What are the consequences? In other words, what are the failure conditions violating the security properties that can be preceded by the weakness?

Answers to these questions are highly tangled in current CWE documentation. For each major class of weakness (such as injection), a large number of CWE categories can be identified to find answers to these questions. As a result, a significant amount of work is needed to identify the trail of CWE categories such that the chain of events that lead to a vulnerability can be reconstructed. To facilitate such analysis, the creation of a semantic template can be viewed as a systematic process of untangling the CWE categories and their descriptions into different bins that

correspond to the four questions. We first describe the preparation and collection phase of building a semantic template.

Preparation and Collection Phase

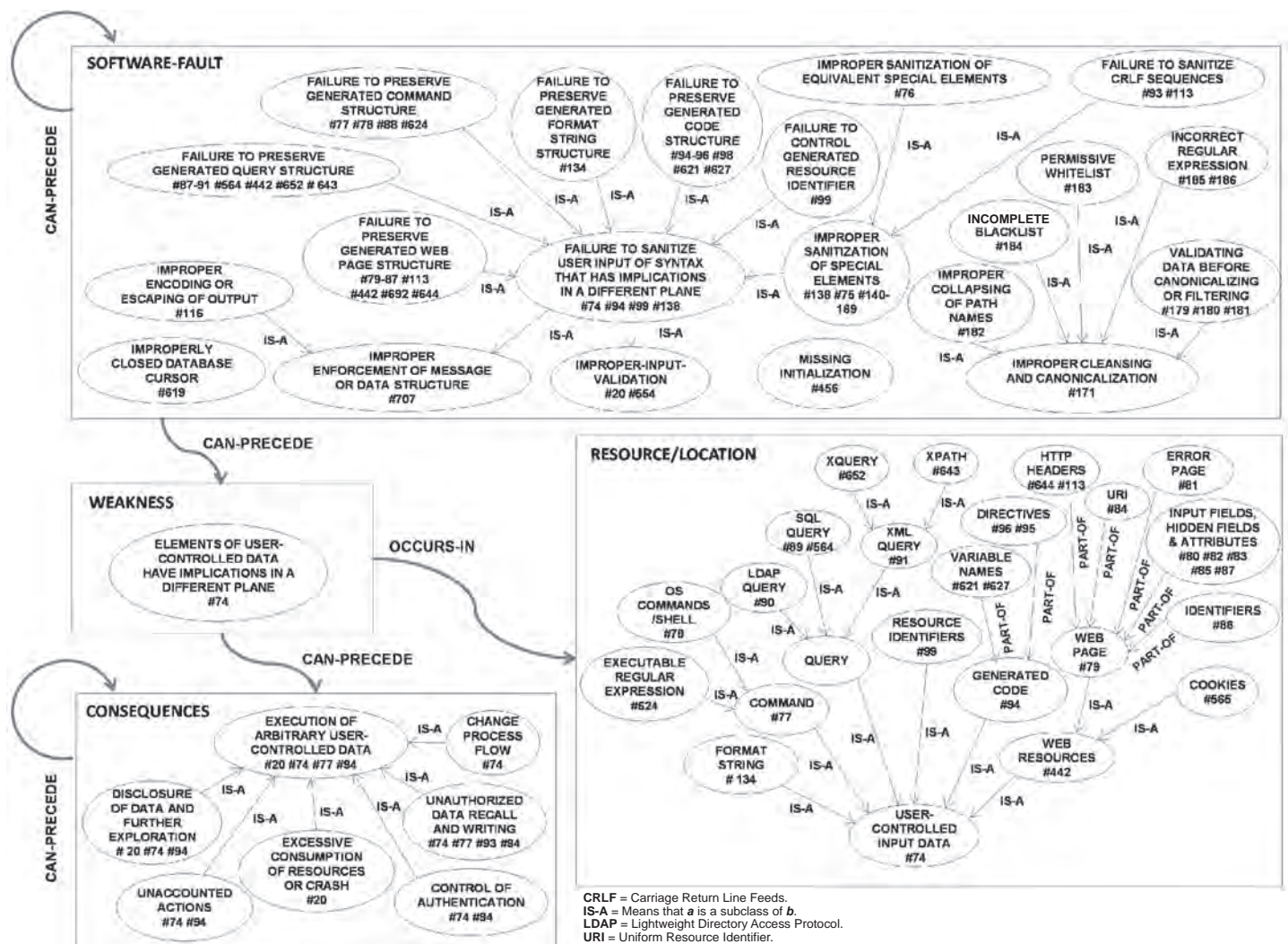
Selection of Content

Since the CWE is continuously evolving, it is important to note that our template is based on Version 1.6 [7]. The CWE uses views to integrate multiple categorizations of weaknesses that share several CWE categories. We use the two most prominent views of the CWE: the development view (CWE-699) of CWE categories, suited for practitioners in the SDLC, and the research view (CWE-1000), suited for research purposes (as it has a deep and abstract hierarchical structure).

Extraction of Relevant Weaknesses

The next step is to identify the CWE category that identifies the weakness of interest at the most abstract level. For the *Injection* weakness, CWE-74 is such a category [3]. Referred to as the root category,

Figure 1: Injection Semantic Template



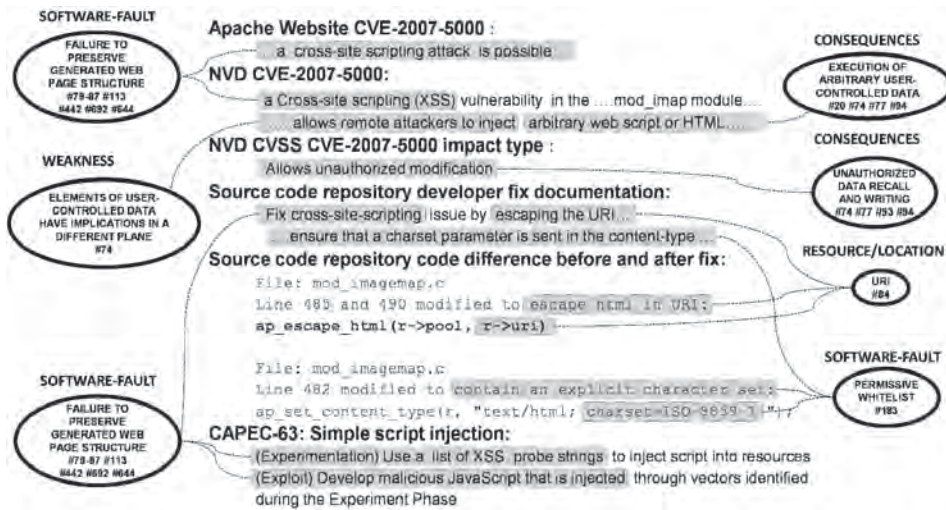


Figure 2: Annotation of Information Pieces for Vulnerability CVE-2007-5000 with Concepts of the Injection Semantic Template

we start here and adopt four strategies to gather weaknesses related to it in the CWE development and research views:

1. Navigate hierarchical relationships of the root category (*Parent* and *Child Of*).
2. Navigate non-taxonomical relationships such as *Can Precede*, *Can Follow*, *Peer-of* in the CWE hyperlinked document [7].
3. Keyword search on the CWE document [7] for weaknesses that have the injection weakness described in their primary or extended description. Keyword search is followed by exploration of *Parent*, *Sibling*, and *Child* categories of the discovered CWE category, for relevance to the root category.
4. Visualization [8] of the root category and its related weaknesses identified by automatically parsing the CWE specification available in XML [1].

While applying each strategy, use of heuristics and some degree of judgment is required on part of the subject matter expert to include a CWE category into the pool of relevant weaknesses. Details about the CWE categories—discovered by applying our strategies to gather weakness related to the root category CWE-74—can be found at <<http://faculty.ist.unomaha.edu/rgandhi/st/injectioncwe.pdf>>. Table 1 gives some summary statistics roughly describing the scale of the

work involved. It speaks volumes about the complexity of the *mental model* that developers need to be aware of to understand the consequences of their coding and design decisions, such that injection weakness can be avoided. Although hyperlinked, navigating the CWE documentation and various graphical representations is tedious and non-intuitive. While different CWE views help to accommodate multiple perspectives, it adds an additional layer of complexity.

Template Structuring Phase Separation of Tangled CWE Descriptions

In this phase, the descriptions of the set of CWE categories from the previous phase are carefully analyzed for their correspondence to either a *Software Fault* that leads to injection; defining characteristic of the injection *Weakness*; *Resource/Location* where injection weaknesses occur; or *Consequences* that follow from a injection weakness. After these parts have been separated and placed in appropriate bins, well-formed and succinct concepts for the injection semantic template are identified in each bin. For example, by analyzing the descriptions for CWE-74 in [1], the following concepts (shown in quotes) can be systematically identified for each of the

semantic template conceptual units (shown in bold).

- **Software Fault:** “Failure to sanitize user input of syntax that has implications in a different plane.”
- **Weakness:** “Elements of user-controlled data have implications in a different plane.”
- **Resource/Location:** “User controlled input data.”
- **Consequences:** “Execution of arbitrary user-controlled data,” “Disclosure of data and further exploration,” “Unaccounted actions,” “Control of authentication,” “Unauthorized data recall and writing,” and “Change process flow.”

While some of these concepts overlap with the CWE-79, this category identifies the following unique and more specific concepts:

- **Software Fault:** “Failure to preserve generated Web page structure,” derived from CWE-79, is a more specific software flaw than a “Failure to sanitize user input of syntax that has implications in a different plane,” which is derived from CWE-74.
- **Resource/Location:** “Web page” (output that is served to other users), which is a “User controlled input data” that is addressed in CWE-74.

Filtering Concepts and Introducing Abstractions

The CWE categories are class, base, or variant weakness, with class being the most general. Class weaknesses are described in a very abstract fashion, typically independent of any specific language or technology. Base weakness is also described in an abstract fashion, but with sufficient details to infer specific methods for detection and prevention of the weakness. On the other hand, variant weaknesses are described at a very low level of detail, typically limited to a specific language or technology.

With the original intent of the semantic template to make weakness more understandable, we derive the primary concepts for software faults and weakness characteristics from the more general class and base CWE categories—while preserving traceability to the CWE categories (with more specific variants) using their identifiers. This design decision was taken primarily to avoid missing the forest for the trees. We expect it to be easier for developers to remember a more generic model of the weakness rather than a detailed one. However, in the case of the Resource/Location conceptual unit, it is not uncommon to extract concepts in the

Table 1: Measures Related to the Collection of Injection Related CWEs Measures

Measures	Value
Total number of CWEs relevant to injection	46
Total number of relationships among CWEs relevant to injection	37
Average number of relationships (inward and outward) per CWE	1.6
Highest depth of the hierarchy among CWEs relevant to injection	4 (including root)
Total number of pages in the CWE document relevant to injection	83

template from variant weaknesses. For the Consequences conceptual unit, we have discovered that the concepts extracted from consequences listed for class and base CWE categories provide comprehensive coverage of consequences identified from more specific-variant CWE categories.

Template Structuring and Representation

In this sub-task, the identified concepts for the template are structured and related to each other based on the relationships between their corresponding CWE categories. From this effort, a highly structured collection of interdependent concepts emerge (as shown in Figure 1 on page 17). Each concept in the semantic template of Figure 1 includes numbers that identify relevant CWE categories. The semantic template reduces duplication of content across related CWE categories while putting them in the context of each other.

Template Refinement and Tailoring

The template can be easily used to study vulnerability information gathered from multiple sources or reconstruct a successful software exploit. Related to both CWEs and CVEs, the Common Attack Pattern Enumeration and Classification (CAPEC) [9] provides a standard way to capture and communicate the manner in which software weaknesses can be exploited. They are stepwise operationalizations of attacks against software systems. By mapping specific vulnerabilities (CVEs) and attack patterns (CAPECs) to the semantic template, it is further refined and checked for obvious omissions. In the following section, we describe such mapping in the context of the XSS vulnerability from CVE-2007-5000. We also expect the semantic templates to be tailored for a specific project, product, or organization.

Using the Semantic Template to Study Vulnerabilities

We use the injection semantic template to study the vulnerability information available from multiple project specific sources for the reported XSS vulnerability CVE-2007-5000 in the Apache HTTP server. These sources include the CVE vulnerability descriptions; media reports about the vulnerability on the Apache HTTP server project public Web site; change history in the open source code repository; source code versions (before and after the fix); and related CAPECs as test cases. The semantic template allows us to anno-

Software Defense Application

As the government and defense sector adopts standards for tracking and detecting specific vulnerabilities, there is an urgent need for developers to build software artifacts to avoid weaknesses that cause vulnerabilities in the first place. Semantic templates have multiple usage scenarios in software assurance, such as to study past vulnerabilities in source code repositories, suggest test cases for a identified software resource, elicit requirements for avoiding weakness, and provide intuitive explanation-based guidance to developers when conditions that lead to weaknesses are detected.

tate the natural language vulnerability descriptions in order to understand and reconstruct the way the injection weaknesses occur. The semantic template also allows extrapolating or identifying missing information (if any).

The semantic template provides intuitive visualization capabilities for the collected vulnerability information. In Figure 2, the vulnerability artifacts related to CVE-2007-5000 are filled into the template. A larger visualization can be found at <http://faculty.ist.uomaha.edu/rgandhi/st/injectioncve.pdf>. Figure 2 provides an integrated view that shows how developers can effectively reason about why the vulnerability occurred; brainstorm possible attack vectors (CAPECs); and discuss the adequacy of performed fixes. Stakeholders in the SDLC can consume technical details with relative ease and guided explanation.

We expect that over a collection of CVE vulnerabilities in a particular project, their mappings to specific weakness categories will reveal recurring error patterns and provide project-specific measures for identifying the most prominent CWE weaknesses for which developers need awareness and training.

Synergy with Other Security Standardization Efforts

The semantic template provides a unified view of software weaknesses (CWE), actual vulnerabilities (CVE), and relevant attack patterns (CAPEC) that can be used to develop and prioritize risk-based test cases for the most exploited software flaws. Many source code static analysis tool reports now provide explicit mappings from their error reports to CWE and CVE identifiers. However, exploring a CWE category and its related weaknesses (with currently available textual and limited visualization formats) poses a significant burden to the tool users. To this end, the concepts in the semantic template maintain explicit traceability to CWE identifiers and hence can be used to provide an intuitive, visual, and layered explanation to the tool user in the context of the discovered flaw. The tool user can also

examine the fix information from past vulnerabilities to determine the course of action to take. In addition, mapping of attack patterns (CAPECs) to software faults in the semantic template provides concrete scenarios to test and justify the fix adequacy.

With the availability of the Malware Attribute Enumeration and Characterization [10] standardization effort and its mappings to the CWE, we expect to use the semantic template to study what software flaws most often contribute to successful malware behaviors and CAPECs. For example, the flaws that precede the injection weakness would most likely contribute to the success of malware behavior for delivering a malicious payload.

Currently, the process of encoding the known vulnerabilities and attack patterns into the template is manually performed. While manual population of templates is scalable for recording of new vulnerabilities as they are detected, relating past vulnerabilities with the templates requires automation. An empirical study with the Apache repository will be conducted to assess the accuracy of this automated process.

As part of our future work, we also expect to build associations of the semantic template with the Knowledge Discovery Metamodel (KDM) [11]. The KDM defines an ontology for software assets and their relationships; this could be leveraged to describe the software faults and resources in the semantic template using a language-independent semantic representation. In turn, the semantic templates could provide abstractions and visualizations to enhance the explanation of KDM-based software mining results.

Conclusion

The CVE grows by roughly 15 to 20 vulnerabilities every day. Each discovered vulnerability produces several information pieces extending from its discovery to its fix. With over 600 entries and more than 20 different views, the CWE provides a significant body of knowledge for classifying and categorizing software weaknesses. However, it is a difficult task

to use the CWE for conducting a systematic study of observed vulnerabilities.

This article describes a process to systematically study software vulnerabilities using several software assurance community standards. A semantic template enables us to systematically assimilate the information pieces related to a vulnerability. This integrated information allows fundamental questions to be answered:

- How do software flaws lead to a vulnerability?
- What are the consequences of exploiting the vulnerability?
- How were they exploited?
- What resources were involved?
- How were they fixed?
- Are the applied fixes sufficient?
- What project specific measures can be produced for the CWE weakness categories that the vulnerability is related to?
- How do the discovered vulnerability and its fix revise our confidence in the software system?
- What other weaknesses still remain?
- What steps should be taken to prevent the vulnerabilities in general?
- Can tools be optimized to look for the discovered patterns?

Answering these questions is essential for an organization to measure the effectiveness of its secure software develop-

ment activities and justify the corresponding assurance given to customers. ♦

Acknowledgement

This research is funded in part by DoD/Air Force Office of Scientific Research, National Science Foundation Award Number FA9550-07-1-0499, under the title “High Assurance Software.”

References

1. The MITRE Corporation. *CWE—Common Weakness Enumeration*. 10 Apr. 2010 <<http://cwe.mitre.org/>>.
2. The MITRE Corporation. *CVE—Common Vulnerabilities and Exposures*. 10 Apr. 2010 <www.cve.mitre.org>.
3. The MITRE Corporation. *CWE—Common Weakness Enumeration*. “CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component.” 5 Apr. 2010 <<http://cwe.mitre.org/data/definitions/74.html>>.
4. The MITRE Corporation. *CWE—Common Weakness Enumeration*. “CWE-79: Improper Neutralization of Input During Web Page Generation.” 5 Apr. 2010 <<http://cwe.mitre.org/data/definitions/79.html>>.
5. The MITRE Corporation. *CVE—Common Vulnerabilities and Exposures*.

“CVE-2007-5000 (under review).” 9 Sept. 2007 <<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5000>>.

6. Wu, Yan, Robin A. Gandhi, and Harvey Siy. *Using Semantic Templates to Study Vulnerabilities Recorded in Large Software Repositories*. Proc. of the 6th International Workshop on Software Engineering for Secure Systems (SESS ’10) at the 32nd International Conference on Software Engineering (ICSE 2010), South Africa, Cape Town. 2010.
7. Martin, Robert A. *CWE Version 1.6*. The MITRE Corporation. 29 Oct. 2009 <http://cwe.mitre.org/data/published/cwe_v1.6.pdf>.
8. Siy, Harvey. “Injection-Related CWEs – Graph-Viz Visualization.” <www.cs.unomaha.edu/~hsiy/research/zgrviev/injectionCWEs.html>.
9. The MITRE Corporation. *CAPEC—Common Attack Pattern Enumeration and Classification*. 18 May 2010 <<http://capec.mitre.org>>.
10. The MITRE Corporation. *MAEC – Malware Attribute Enumeration and Characterization*. 10 Apr. 2010 <<http://maec.mitre.org>>.
11. “KDM 1.1.” *Object Management Group*. 10 Apr. 2010 <www.omg.org/spec/KDM/1.1>.

About the Authors



Robin A. Gandhi, Ph.D., is an assistant professor of information assurance in the College of Information Science and Technology at the University of Nebraska, Omaha

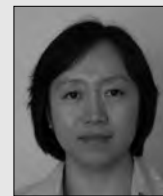
(UNO). He received his doctorate from The University of North Carolina at Charlotte. The goal of Gandhi’s research is to develop theories and tools for designing dependable software systems that address both quality and assurance needs. Gandhi is a member of the DHS’s Software Assurance Workforce Education and Training Working Group.

**Nebraska University Center for Information Assurance
College of Information Science and Technology (IS&T)
6001 Dodge ST
PKI 177 A
Omaha, NE 68182-0500
Phone: (402) 554-3363
E-mail: rgandhi@unomaha.edu**



Harvey Siy, Ph.D., is an assistant professor in the Department of Computer Science at the UNO. He received his doctorate in computer science from the University of Maryland at College Park. He conducts empirical research in software engineering to understand and improve technologies that support the development and evolution of reliable software-intensive systems. Siy has previously held positions at Lucent Technologies and its research division, Bell Laboratories.

**Department of Computer Science
College of IS&T
6001 Dodge ST
PKI 281 B
Omaha, NE 68182-0500
Phone: (402) 554-2834
E-mail: hsiy@unomaha.edu**



Yan Wu is currently pursuing her doctorate in information technology at the UNO, and is expecting to receive her degree in Spring 2011. The goal of her research

is to conduct empirical study on analyzing software engineering knowledge in order to support the development and maintenance of reliable software-intensive systems.

**Department of Computer Science
College of IS&T
6001 Dodge ST
Omaha, NE 68182-0500
E-mail: ywu@unomaha.edu**

The Balance of Secure Development and Secure Operations in the Software Security Equation

Sean Barnum
The MITRE Corporation

Software security is about reducing the risk that software poses to those who use it or are affected by it. This requires thought and action more than simply at the point of development or use. It requires a more holistic approach, balancing secure development and secure operations. The bad news is that these two capable domains typically do not interact much or understand each other. The good news is that there are active ongoing efforts focused on addressing this gap.

Effectively addressing software security requires adequately balancing the secure development and the secure operations domains (see the mechanisms listed in Table 1).

The objective of security in development is to prevent security issues in the software causing vulnerability. In the best case, this means preventing such security issues from ever entering the software to begin with. This best-case approach is driven by activities such as effective security training, security policy definition, security requirements specification and review, secure architecture and design, and architectural risk analysis. In the worst case, this means at least preventing such security issues from ever being fielded into live systems. This later life-cycle approach is typically driven by activities such as secure code analysis, security testing, and penetration testing.

The objective of security in operations is to prevent security issues in deployed systems by securing their infrastructure, configuration, and use. So, the ultimate goal would be to have all operating software totally free from vulnerability and fully secure. Given the complexities involved in today's software and the ever-changing threat landscape, the reality is that no software can ever be presumed as *fully secure* and will typically be under ongoing and consistent attack. Beyond the initial security engineering of software operational deployment, the bulk of secure software operations is about continuous situational awareness and incident response. Recognizing real-world practicalities, it is focused on answering the foundational, ongoing secure operations questions:

- Are we being attacked? (Were we attacked?)
- How are we being attacked?
- What is the objective of the attack?
- What is our exposure?
- Who is attacking us?
- What should we do to protect against these attacks in the future?

The commonality between the secure development and secure operations domains is the central role of understanding how adversaries attack software. While both domains have a need to understand how software is attacked, the specific needs of each domain differ in level of abstraction and in purpose—but in a synergistic fashion. The secure development domain needs to understand the attacker's perspective in abstract terms in order to improve security across a wide range of contexts, rather than individual instances. The secure operations domain needs to understand the attacker's specific variations of behavior in gory detail in order to recognize it, understand it, estimate its effect, and plan its mitigation. Due to the reciprocal balance between the top-down perspective of secure development and the bottom-up perspective of secure operations, there is an opportunity for each domain to address its own requirements in such a way that also provides value to the other's primary focus (see Figure 1, next page).

Given the differing requirements between the two domains (to characterize attacks and potentially exchange this information), a flexible mechanism is required to capture, describe, and share knowledge about common patterns of attack. One such mechanism is the attack

pattern object as specified and leveraged by the Common Attack Pattern Enumeration and Classification (CAPEC), as outlined at <http://capec.mitre.org>. CAPEC is a publicly available catalog of attack patterns along with a comprehensive schema and classification taxonomy intended to form a standard mechanism for identifying, collecting, refining, and sharing attack patterns among the software community. Established in 2000, the attack pattern concept represents a description of common attack approaches abstracted from a set of known real-world exploits. While this source of raw data comes primarily from the secure operations domain, attack patterns today are primarily a construct used by the secure development community to aid software developers in improving the assurance profile of their software.

In this role, attack patterns offer the secure development community unique value in several areas such as:

- Representing abuse cases (how an attacker would intentionally abuse a software system) during requirements elicitation, specification, and review.
- Mapping identified threats to the software's modeled attack surface as part of threat modeling activities during architecture and design.
- Guiding and prioritizing secure code analysis during implementation. This

Table 1: *Mechanisms for Secure Development and Operations*

Mechanisms of Secure Development	Mechanisms of Secure Operations
<ul style="list-style-type: none"> • Effective Security Training • Security Policy • Security Requirements • Secure Architecture and Design • Secure Coding • Security Testing • Penetration Testing • Risk Management 	<ul style="list-style-type: none"> • Secure Configurations • Firewalls • Proxies • Intrusion Detection Systems • Intrusion Prevention Systems • Real-Time Data Monitoring • Operational Monitoring and Control • Incident Response • Forensics • Anti-Tamper Mechanisms

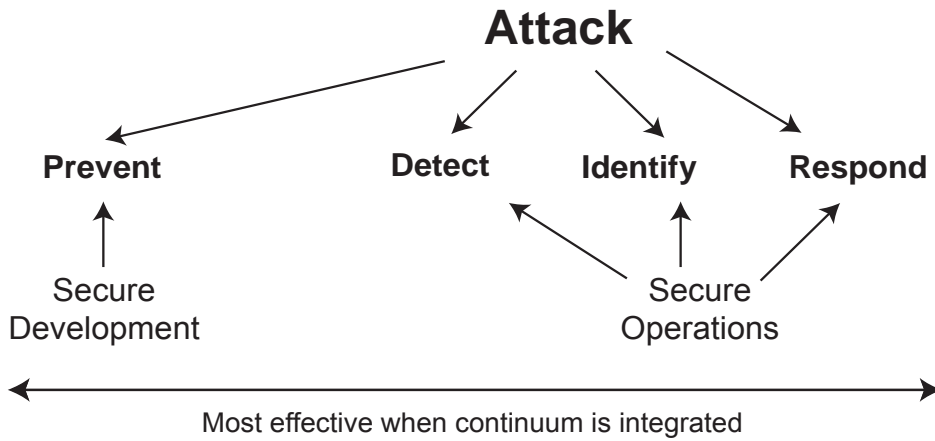


Figure 1: How Secure Development and Operations Can Work Together

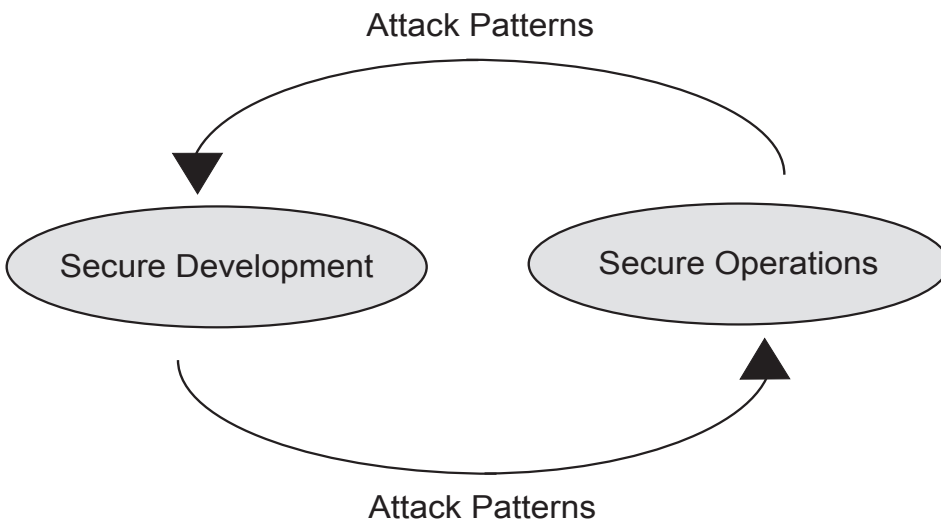


Figure 2: Attack Patterns Bridge Secure Development and Operations

Question	Role of Attack Patterns
Are we being attacked? (Were we attacked?)	Attack patterns offer structured descriptions of common attacker behaviors to help interpret observed operational data and determine its innocent or malicious intent.
How are we being attacked?	Attack patterns offer detailed structured descriptions of common attacker behavior to help interpret observed operational data and determine exactly what sort of attack is occurring.
What is the objective of the attack?	Elements of attack patterns outlining attacker motivation and potential attack effects can be leveraged to help map observed attack behaviors to potential attacker intent.
What is our exposure?	The structure detail and weakness mapping of attack patterns can provide guidance in where to look and what to look for when certain attack pattern behaviors are observed.
Who is attacking us?	Attack pattern threat characterization and detailed attack execution flow can provide a framework for organizing real-world attack data to assist in attribution.
What should we do to prevent against attacks in the future?	Attack patterns offer prescriptive guidance on solutions and mitigation approaches that can be effective in improving the resistance tolerance and/or resilience to instances of a given pattern of attack.

Table 2: Attack Patterns Help Answer Questions Regarding Secure Operations

includes identifying specific high-risk areas requiring greater analysis rigor as well as the most relevant weaknesses to look for.

- Identifying, specifying, and prioritizing security test cases.
- Serving as attack templates for penetration testing and objective persona descriptors for red team penetration testing.

The future potential for CAPEC attack patterns lies beyond their evolving and continued use within the secure development community. The secure operations community can utilize CAPEC to assist in situational awareness of deployed systems under attack and aid in response and mitigation. Several characteristics of attack patterns make them relevant for the secure operations community:

- Attack patterns provide high-level rather than simply low-level detailed patterns of attacks against software.
- Much of secure operations is about analyzing low-level activity for patterns and composing them into higher levels of abstraction to detect, identify, and respond to attacks.
- Software assurance attack patterns provide a top-down, high-level context for both the method and the intent of attacks.
- Efforts are currently under way to formalize the CAPEC attack pattern schema in order to provide adequate detail of attacks for aligning and integrating their context with bottom-up incident analysis characterizations.

Attack patterns offer a unique and practical bridge between the two domains, as shown in Figure 2.

Using attack patterns makes it possible for the secure development domain to leverage significant value from secure operations knowledge, enabling them to:

- Understand the real-world frequency and success of various types of attacks.
- Identify and prioritize relevant attack patterns.
- Identify and prioritize the most critical weaknesses to avoid.
- Identify new patterns and variations of attack.

Through the use of attack patterns, it is also possible for the secure operations domain to leverage significant value from secure development knowledge. This enables those in the secure operations domain to provide appropriate context to help answer the foundational secure operations questions (see Table 2).

One of the maturation paths currently under way for CAPEC involves integrat-

ing and refining lower-level attack attributes and characteristics to better support automatable integration of both domains. So far, this effort has been focused on enhancing attack pattern descriptions with greater levels of attack execution flow detail and on the addition of two new constructs: **Target_Attack_Surfaces** and **Observables**.

The **Target_Attack_Surfaces** construct is intended to give a structured characterization of the relevant portions of the targeted software that an attack is

attempting to exploit. This sort of detail can be valuable within an operational context, assisting in attack detection, identification, and characterization through mapping of observed effects on target software assets and resources. The current draft schema (see Figure 3) focuses on characterizing functional services, protocols, command structures, etc. Future schema revisions should extend this conceptual construct to address a broader set of attack surface characteristics.

The **Observables** construct is intend-

ed to capture and characterize events or properties that are observable in the operational domain. These observable events or properties can be used to adorn the appropriate portions of the attack patterns in order to tie the logical pattern constructs to real-world evidence of their occurrence or presence. This construct has the potential for being the most important bridge between the two domains, as it enables the alignment of the low-level aggregate mapping of observables that occurs in the operations

Figure 3: CAPEC - High-Level Attack Surface Draft Schema (Figures 3 and 4 were created with Altova XMLSpy)

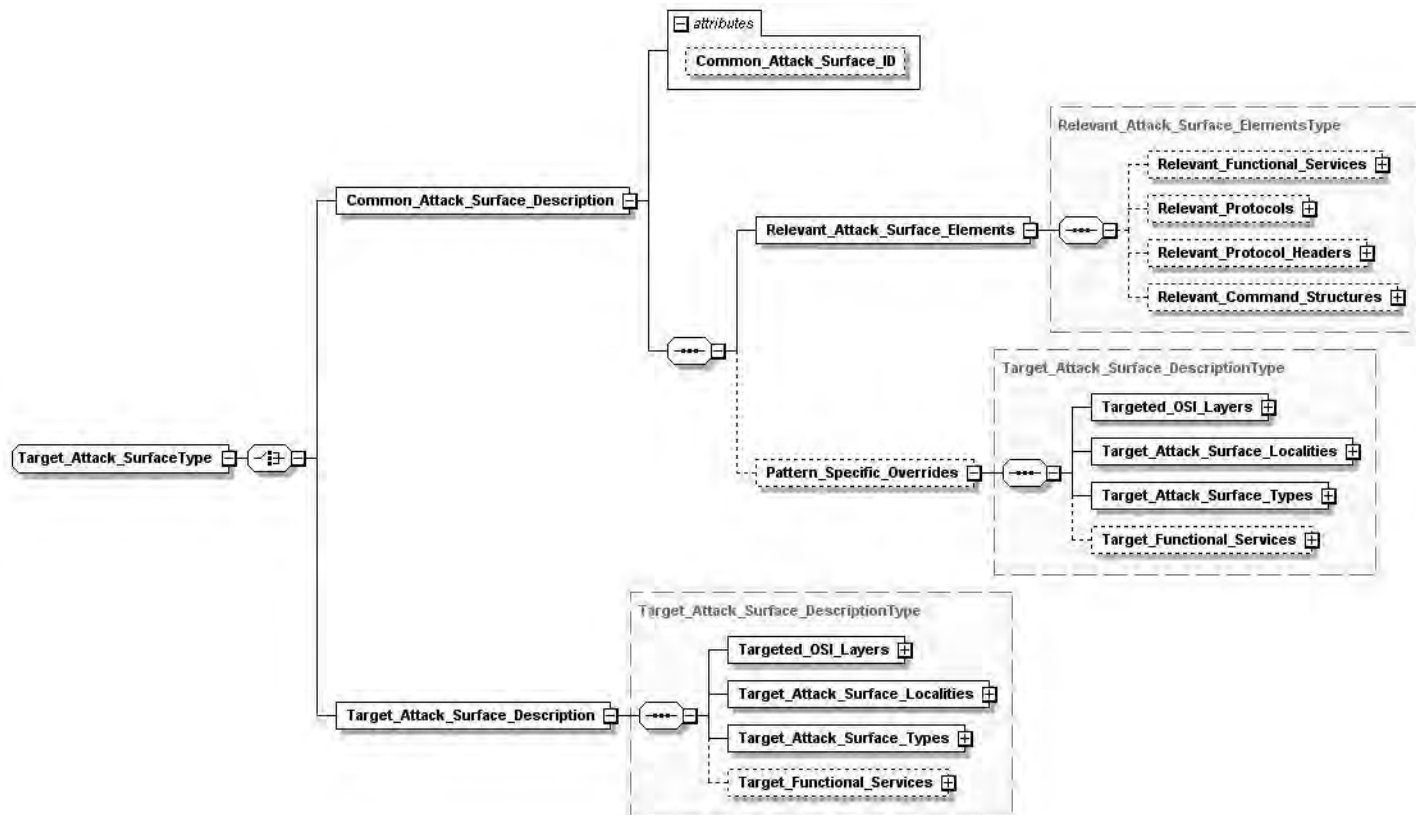
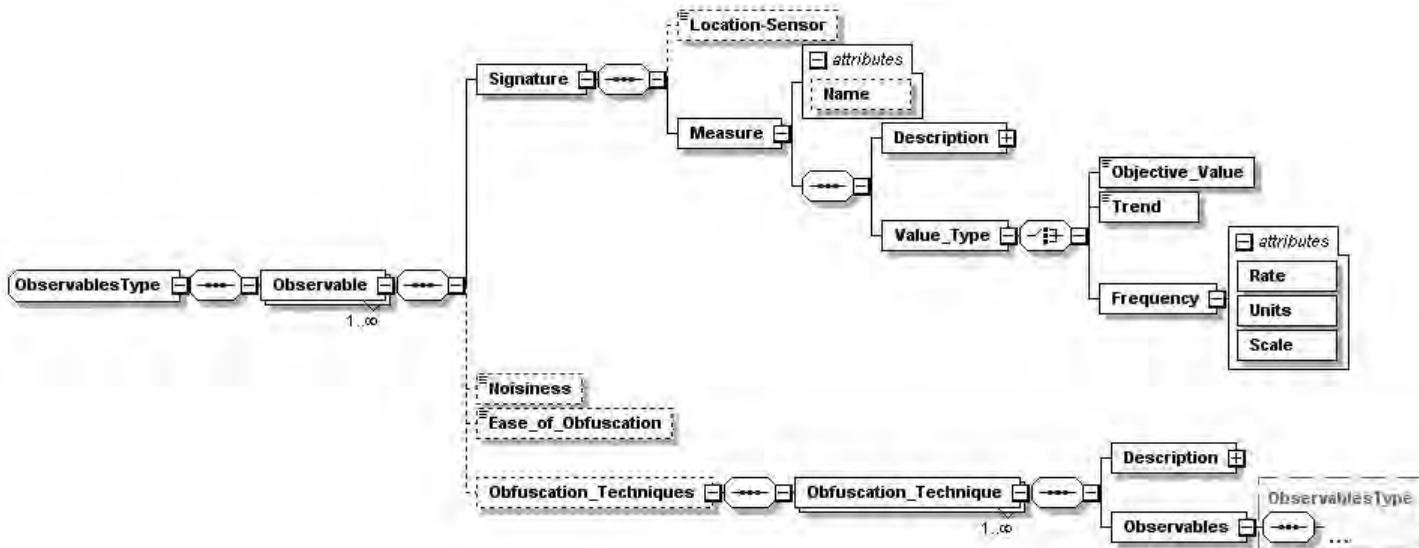


Figure 4: CAPEC - Observables Draft Schema



Software Defense Application

The DoD—along with its supporting defense industry—has identified cybersecurity as one of its top priorities today and going forward. The software security portion of this battle is currently being fought on two fronts, secure development and secure operations, with little coordination between the two. This article discusses the objectives and activities unique to each of these areas as well as some of their shared commonality in the relevance of understanding the attacker’s perspective. Most importantly, it introduces attack patterns as a resource that characterizes this commonality and offers a practical and actionable bridge for coordination and collaboration between the secure development and secure operations communities. An understanding of attack patterns and their relevance to a unified approach to software security should be requisite knowledge for all those working in DoD software.

domain to the higher-level abstractions of attacker methodology, motivation, and capability that exist in the development domain. By capturing them in a structured fashion, the intent is to enable future potential for detailed automatable mapping and analysis heuristics.

The current **Observables** draft schema (see Figure 4 on the previous page) adorns the **Attack Step**, **Attack Step_Technique**, **Attack Step Outcome**, and **Attack Step Security Control** elements of the attack pattern schema. It focuses on characterizing specific observable measures, their value, their sensor context, and how accurate or easy to obfuscate they are. Future schema revisions should flesh out

the construct to cover other relevant dimensions. Changes will be based on input and collaboration from the operations community and other aligned knowledge standardization efforts needing this construct (e.g., Common Event Enumeration [CEE] and Malware Attribute Enumeration and Characterization [MAEC]).

People interested in learning more about CAPEC, CEE, MAEC, and other related knowledge standardization efforts can gain better insight and join in the community collaboration efforts by going to the Making Security Measureable Web site <<http://msm.mitre.org>> and the Software Assurance Community Re-

sources and Information Clearinghouse <<https://buildsecurityin.uscert.gov/swa>>.

Summary

Effective software security requires a balanced approach between secure development and secure operations. The commonality between these two domains is the central role of understanding how adversaries attack software. CAPEC attack patterns offer a mechanism for structured characterization of common attacks that enable a useful exchange of information relevant to both domains, also aligning low-level observations to high-level contexts for mutual benefit.

CAPEC is currently a resource leveraged primarily by the secure development community, but there is an opportunity and a strong need for increased collaboration from the secure operations community. It will help shape and refine CAPEC to more effectively serve both communities, potentially acting as an integrating bridge to eventually yield a more holistic software security capability.

We encourage readers within both communities to become actively involved and lend their knowledge and voices to our unifying efforts. ♦

DEPARTMENT OF DEFENSE SYSTEMS ENGINEERING



Delivering Innovation, Agility, and Speed



Department of Defense *Systems Engineering* applies best engineering practices to


- Support the current fight; manage risk with discipline
- Grow engineering capabilities to address emerging challenges
- Champion systems engineering as a tool to improve acquisition quality
- Develop future technical leaders across the acquisition enterprise

The Department of Defense seeks experienced engineers dedicated to delivering technical acquisition excellence for the warfighter.

See www.usajobs.gov

Director of Systems Engineering • Office of the Director, Defense Research and Engineering
3040 Defense Pentagon • Washington, DC 20301-3040 • <http://www.acq.osd.mil/se>

About the Author



Sean Barnum is a software assurance principal at The MITRE Corporation, serving as a thought leader and senior advisor on software assurance and cybersecurity projects. He has more than 20 years of experience in the software industry in the areas of development, software quality assurance, quality management, process architecture and improvement, knowledge management, and security. He is involved in numerous knowledge standards-defining efforts, including Common Weakness Enumeration, CAPEC, and other elements of software assurance programs for the DHS, DoD, and the National Institute of Standards and Technology. He is co-author of the book “Software Security Engineering: A Guide for Project Managers.”

Phone: (703) 473-8262
E-mail: sbarnum@mitre.org

Two Initiatives for Disseminating Software Assurance Knowledge

Dr. Nancy R. Mead
SEI

Dr. Dan Shoemaker
University of Detroit Mercy

Education in software assurance (SwA) is an essential element in the effort to produce secure code. This article describes two efforts that support national cybersecurity education goals: development of SwA learning artifacts that can be integrated into conventional learning environments and establishment of a reference curriculum for a master's degree program, known as the MSwA.

There is general recognition that software engineering practice can best be improved through education. In fact, the establishment of a National Cyberspace Security Awareness and Training Program was among the three highest priorities in [1], which describes the program's purpose as to "improve cybersecurity knowledge, and understanding of the issues" and to produce a "sufficient number of adequately trained ... personnel to create and manage secure systems." The cornerstone of the initiative was the mandate to ensure "adequate training and education to support cybersecurity needs" [1].

The aim of these initiatives was to guarantee that SwA practices would be embedded in the day-to-day actions of the overall workforce [2]. The problem with SwA is that there was no single point of reference to "guide the development and integration of education and training content relevant to software assurance" [3]. That led the DHS to publish a 387-page Common Body of Knowledge (CBK), which specifies a comprehensive set of recommended practices for secure SwA. These range from "heavyweight design methods" to "model contract language for vendors" [3]. The problem is that none of these recommendations have made their way into common use.

The traditional means of disseminating any kind of new knowledge into the society at-large is through formally constituted education, training, and awareness programs [2]. Back in 2003, the National Strategy recognized that fact in Action/Recommendation 3-6, which states that research and development efforts should be conducted in the general area of secure SwA in order to coordinate "the development and dissemination of best practices for cybersecurity." [1].

The obvious question eight years later is, "How close are we to achieving that goal?" The two projects discussed in this article are designed to promote more secure software teaching in higher education. Together, they represent the first attempts to ensure that the principles and practices of secure SwA knowledge are

embedded in mainstream higher education processes.

The problem with SwA knowledge is that it is crosscutting rather than disciplinary. In essence, the knowledge base for SwA is located in a range of traditional studies [4]. That includes dissimilar CBK areas such as software engineering, systems engineering, information systems security engineering, safety, security, testing, information assurance, law, and pro-

"The two projects ... are designed to promote more secure software teaching ... they represent the first attempts to ensure that the principles and practices of secure SwA knowledge are embedded in mainstream higher education processes."

ject management [4]. As a result, secure SwA content might appear in many different places and be taught in many different ways in conventional education settings.

It is clearly unacceptable to approach the teaching and learning process in such a disjointed way. Therefore, the way educators promulgate secure SwA knowledge has to be coordinated. In order to coordinate the teaching and learning process, a formal effort has to be made to integrate "software assurance content ... into the body of knowledge of each contributing discipline" [5]. There are two practical barriers to achieving that outcome:

1. It is not clear what specific knowledge

and skills should be taught in each area.

2. There are no validated methods for delivering that knowledge once it has been identified.

Logically, the first step in integrating new knowledge into conventional learning environments is to identify, relate, and catalogue what is presently out there.

Project I – Documenting Knowledge

The goal of one project—funded by the DoD and conducted at the University of Detroit Mercy (UDM)—is attempting to identify and document any knowledge, from any source, that could be related to SwA. That knowledge came from all traditional computing disciplines, such as computer science, software engineering, and information systems. Nevertheless (besides the strictly technical areas), the project also incorporated the conventional areas of information security as well as relevant knowledge from the behavioral and social sciences. The knowledge was obtained from all accessible public and private sector sources.

The resulting knowledge base is contained in the DoD's National Software Assurance Repository (NSAR). The NSAR encompasses and relates all commonly accepted practices, principles, methodologies, and tools for SwA. It is managed by an automated online knowledge management system with an underlying knowledge management system roughly based on the CBK; however, to ensure the validity of the CBK framework, the mind map was fine-tuned and validated through conducting a classic Delphi study using a panel of 11 nationally known secure SwA experts.

The knowledge base contains as many life-cycle methodologies and tools for assuring software as could be identified. It also itemizes all related supporting principles and concepts that are aimed at increasing the assurance and security of internally developed and sustained software. That also includes products and ser-

vices purchased from outside vendors. The knowledge base is evolutionary and inclusive: As the literature of the field expands or new sources are identified, that material will be catalogued and added to the current knowledge base.

Pedagogy Development

The actual purpose of the UDM project was not simply to gather knowledge; it was also to ensure the teaching of secure software topics in all appropriate education, training, and awareness settings. In support of that goal, the project has packaged the contents of its knowledge base into discrete learning modules. These modules are designed to facilitate the efficient SwA knowledge transfer to all relevant teaching and learning settings. As a result, the modules can be incorporated into a wide range of teaching and learning environments. They are appropriate for graduate, undergraduate, community college, and even high school education, as well as in training and awareness applications.

The modules are intended to be separate, standalone learning artifacts capable of conveying all of the requisite knowledge for a given topic. At a minimum, each module can be delivered in a con-

ventional classroom. However, the modules embody supporting material that also allows delivery in a range of asynchronous and Web-enabled learning environments. That flexibility facilitates the efficient transfer of new workforce skills and practices to all types of settings.

Each module conveys a logical element of SwA practice. The entire collection of these modules is mapped to the body of knowledge contained in the knowledge base, which is structured on the most commonly accepted model for secure SwA practice (the CBK). This mapping provides precise guidance about the places where the newly developed instructional content fits within the commonly accepted understanding of the correct elements of practical SwA work.

Each of the actual teaching modules incorporates a set of conventional learning artifacts that are easily recognizable to traditional educators. Every module includes:

1. A table of learning specifications.
2. Presentation slides for each concept contained in the module.
3. An evaluation process.
4. Any relevant Web-enabled supporting material.

5. A model delivery system.

There is also an accompanying pedagogical methodology for each individual learning module. In other words, every module incorporates a validated set of teaching tools, with each tool being optimized to ensure the maximum knowledge transfer for all potential teaching settings.

Mapping for Broad-Scale Integration

In order to ensure integration into conventional higher education curricula, the UDM project has formally mapped all of its secure SwA courseware modules to the standard set of computing topics specified for three of the five computer disciplines in the Computing Curricula 2005 standard (CC2005) [6]. This standard is a joint authorization of the Association for Computer Machinery (ACM), IEEE, and Association for Information Systems. Since these are the three associations that are responsible for developing and overseeing computing curricula worldwide, the CC2005 can be considered to be exhaustively authoritative.

The elements of secure SwA practice were mapped from the CBK to the generally accepted curricular recommendations (as itemized in CC2005). The aim of the mapping process was to identify where specifications for secure practice contained in the CBK fit within the recommendations for curricular content in each of the disciplines of computer science, software engineering, and information systems.

The mapping itself was a keyword-based process, utilizing the terms from the curricular requirements contained in Tables 3.1 and 3.2 of CC2005 as the search criterion. Where instances of that term were found in the CBK, anecdotal analysis was employed to determine the intent of the term with respect to the discussion of secure SwA. Those intents were noted, aggregated, and then categorized into highly specific concepts for secure SwA that had to be communicated along with the teaching of each of the conventional CC2005 curricula elements. The detailed mapping of concepts to recommendations was used to tailor the integration of the associated secure SwA teaching module for supporting or facilitating the specific intent of that term.

The project provides a detailed specification of where each learning module best fits within CC2005's curriculum. It also provides a justification for why the module was placed where it was in that particular curriculum. The justification is based on the mapping between the module and the recommended topics for a

**CIVILIAN TALENT IS MISSION-CRITICAL.
LET'S GET TO WORK.**

NAVAIR
CIVILIAN
CHOICE IS YOURS.

Discover more about Naval Air Systems Command today.
Go to www.navair.navy.mil

Equal Opportunity Employer | U.S. Citizenship Required

Work for Naval Air Systems Command (NAVAIR) and you'll support our Sailors and Marines by delivering the technologies they need to complete their mission and return home safely. NAVAIR procures, develops, tests and supports Naval aircraft, weapons, and related systems. It's a brain trust comprised of scientists, engineers and business professionals working on the cutting edge of technology.

You don't have to join the military to protect our nation. Become a vital part of NAVAIR, and you'll have a career with endless opportunities. As a civilian employee you'll enjoy more freedom than you thought possible.

standard computer science, software engineering, and information systems curriculum. For instance, the project provides specific recommendations for the precise place in an information systems curriculum where new secure SwA content could be added to current testing topics. The justification is necessary to help individual curriculum designers understand where the learning module should be placed in their curricula. The justification also facilitates the integration and acceptance of that module within the traditional higher education and training communities.

Project 2 – MSwA Curriculum

The second education initiative to support the National Strategy focused primarily on development of a reference curriculum for an MSwA. The SEI is leading this ongoing education effort in support of the DHS's National Cyber Security Division. This is a particularly important focus because much of the body of knowledge in secure SwA is based on a foundation of software engineering principles and practices. This project specifies a set of topics and all of the attendant prerequisite knowledge and requirements needed to ensure a properly educated SwA professional. It differs from the prior project in that it identifies just the key knowledge elements required to produce a well-educated practitioner—and structures those elements into a comprehensive curriculum.

The curriculum development team includes technical staff from the SEI and faculty from a number of universities, including international representation. The reference curriculum includes guidelines that were used to develop the curriculum, prerequisites and proposed outcomes, curriculum architecture, a curriculum body of knowledge, implementation guidelines, and a glossary of terms. A number of existing artifacts (including the CBK), the recent graduate software engineering curriculum guidelines [7], and the older SEI reports on graduate software engineering education [8, 9] are inputs to the project. The team also referenced [10] as needed, as software engineering knowledge is fundamental to SwA. The project was inspired in part by the DHS Build Security In (BSI) Web site <<https://buildsecurityin.us-cert.gov>>, which contains articles providing practical advice on SwA to practitioners. It is this practitioner focus that is central to the curriculum development effort. Another important resource for the team (also inspired by the BSI Web site) is [11], which was used along with the previously noted resources

to identify the SwA practices to include in the curriculum.

In order to stay grounded, an invited review team for the curriculum was also involved in the process. In addition, some key industry managers and practitioners generously agreed to be surveyed in order to help validate our understanding of the desired outcomes. The curriculum also includes a detailed list of knowledge units and corresponding Bloom's taxonomy levels [12].

Establishment of a new degree program is a very ambitious undertaking. As a consequence, the team expects that some universities will elect to establish

“Our understanding of the knowledge that is needed to ensure capable SwA is beginning to be shaped by these two projects. In that respect—and particularly given the critical importance of secure software to the national interest—they are working together to advance that process.”

tracks or specializations in SwA within existing graduate disciplines (e.g., Master's-level programs in Software Engineering [MSwE]), rather than establishing a whole new degree. Therefore, guidance is provided on how to implement a track or specialization, and sample course syllabi are also provided. Team members at Monmouth University and Embry-Riddle Aeronautical University developed candidate implementation strategies for incorporating curriculum elements at their universities.

In addition to the MSwA reference curriculum, this project also produced a set of sample outlines for SwA courses that could be offered at the undergraduate level [13]. These courses might form an area of concentration within a computer science or software engineering under-

graduate degree program for any prospective adopter.

Curriculum Transition Plans

There are a number of transition activities that accompany this curriculum work, as a curriculum is only the first step in effecting change in education. The team has started to work with the IEEE Computer Society towards professional recognition, including a seminar at the March 2010 Conference on Software Engineering Education and Training to raise awareness¹. The curriculum has been presented at the 2010 Curriculum Development in Security and Information Assurance workshop, at a June 2010 DHS Software Assurance Working Group meeting, and also in the Information Assurance Capacity Building Program². Finally, the team will also form a group to work with and provide assistance to universities who wish to offer SwA graduate courses. The team has also started tailoring the curriculum into course offerings that would fit at the community college level.

Looking beyond these near-term activities, the team plans to develop more extensive faculty development workshops, course materials, and course offerings in this area. They also hope to work towards SwA certification along the same lines as IEEE's Computer Software Development Professional. There is an opportunity for distance education in this area, and eventually they may look at high school educational opportunities. The team feels that SwA education is essential at all levels, in order to ensure that software and software-intensive systems are developed with assurance in mind.

Conclusions

Our understanding of the knowledge that is needed to ensure capable SwA is beginning to be shaped by these two projects. In that respect—and particularly given the critical importance of secure software to the national interest—they are working together to advance that process. Both projects are beginning to establish the foundation for moving into the mainstream of education, training, and awareness a field that has historically not been either well understood nor well recognized.

The maturity of SwA education will have advanced when:

- MSwA programs—and SwA specializations within MSwE programs—are widely available.
- The SwA materials database is commonly used in course development.
- SwA offerings are standard elements

of undergraduate computer science and software engineering degree programs.

- The SwA body of knowledge has been codified.

In the case of the MSwA curriculum project, these master's programs and tracks provide an explicit curriculum of knowledge and skills necessary to produce a well-educated SwA professional. Ultimately, the curriculum will be supported by the needed course materials and course offerings. In the case of the UDM project, every instructor in a computer-related discipline now has access to validated content and instructional materials that can be easily incorporated into existing courses.

In both projects, the boundaries and elements of the teaching and learning process for SwA education are clarified. They are initial steps in the long road to being able to assure the correctness and integrity of the nation's software with total confidence. Together, they create a direction and foundation on which the future of the profession can be built.

References

1. Clark, Richard A., and Howard A. Schmidt. *A National Strategy to Secure Cyberspace*. Washington: The Presi-

dent's Critical Infrastructure Protection Board. Feb. 2003 <www.us-cert.gov/reading_room/cyberspace_strategy.pdf>.

2. Cogburn, Derrick L. *Globalization, Knowledge, Education and Training in the Information Age, United Nations Educational, Scientific and Cultural Organization*. Director, Centre for Information Society Development in Africa and Africa Regional Director, Global Information Infrastructure Commission. 1 Dec. 2009 <www.unesco.org/webworld/infoethics_2/eng/papers/paper_23.htm>.
3. Redwine, Samuel T., Ed. *Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire and Sustain Secure Software, Version 1.1*. Washington D.C.: DHS, 2006.
4. Mead, Nancy R., Dan Shoemaker, and Jeffrey Ingalsbe. "Integrating Software Assurance Knowledge into Conventional Curricula." *CROSSTALK* Jan. 2008 <www.stsc.hill.af.mil/crosstalk/2008/01/0801MeadShoemakerIngalsbe.html>.
5. Shoemaker, Dan, et al. *A Comparison of the Software Assurance Common Body of Knowledge to Common Curricular Standards*. Proc. of the 20th Conference on

Software Engineering Education and Training. Dublin, Ireland. 3-5 July, 2007.

6. The Association for Computing Machinery, The Association for Information Systems, and The Computer Society. *Computing Curricula 2005: The Overview Report, Computing Curricula Series*. 30 Sept. 2005 <www.acm.org/education/curric_vols/CC2005-March06Final.pdf>.
7. Integrated Software & Systems Engineering Curriculum Project – Stevens Institute of Technology. *Graduate Software Engineering 2009 (GSWE2009) Curriculum Guidelines for Graduate Degree Programs in Software Engineering*. 30 Sept. 2009 <www.gswe2009.org/>.
8. Ford, Gary. *1991 SEI Report on Graduate Software Engineering Education*. SEI, Carnegie Mellon University. Technical Report CMU/SEI-91-TR-002. Apr. 1991 <www.sei.cmu.edu/reports/91tr002.pdf>.
9. Ardis, Mark, and Gary Ford. *SEI Report on Graduate Software Engineering Education*. SEI, Carnegie Mellon University. Technical Report CMU/SEI-89-TR-21. June 1989 <www.sei.cmu.edu/reports/89tr021.pdf>.
10. IEEE Computer Society. *Guide to the*

WANTED

Electrical Engineers and Computer Scientists Be on the Cutting Edge of Software Development

The Software Maintenance Group at Hill Air Force Base is recruiting civilian positions (U.S. Citizenship Required). Benefits include paid vacation, health care plans, matching retirement fund, tuition assistance and time off for fitness activities. Become part of the best and brightest!

Hill Air Force Base is located close to the Wasatch and Uinta mountains with many recreational opportunities available.

Send resumes to:
phil.coumans@hill.af.mil
or call (801) 586-5325

Visit us at:
<http://www.309SMXG.hill.af.mil>




Software Defense Application

Cybersecurity has been an area of national interest for almost a decade. Education has been noted for years—all the way up to the White House—as one of the most important elements in securing cyberspace. Yet, the DHS's Common Weakness Enumeration [14] documents 797 common defects—and the list is still growing. That is due to current software engineering practice, which has generated software defects at a relatively constant rate for the past 40 years. Those defects—according to a 2008 International Data Corporation survey (see <www.coverity.com/html/press_story65_08_04_08.html>—now cost the average U.S. corporation \$22 million dollars annually. Worse, they leave DoD systems, as well those of all government and industry, susceptible to attack. This article shows successful educational experiences in developing concepts and passing along the principles and practices of secure SwA knowledge.

Software Engineering Body of Knowledge (SWEBOK). 2004 <www.computer.org/portal/web/swebok/htmlformat>.

11. Allen, Julia, et al. *Software Security Engineering: A Guide for Project Managers*. Upper Saddle River, NJ: Addison-Wesley, 2008.
12. Bloom, Benjamin S., ed. *Taxonomy of Educational Objectives: The Classification of Educational Goals: Handbook I: Cognitive Domain*. New York: Longman, 1956.
13. Mead, Nancy R., Thomas J. Hilburn, and Richard C. Linger. *Software Assurance Curriculum Project Volume II:*

Undergraduate Course Outlines. SEI, Carnegie Mellon University. Technical Report CMU/SEI-2010-TR-019. Jan. 2010.

14. The MITRE Corporation. *Common Weakness Enumeration*. 17 May 2010 <<http://cwe.mitre.org>>.

Notes

1. The seminar will be distributed at a later time in the Virtual Training Environment format.
2. This is a faculty development program that was held this July at Carnegie Mellon University.

About the Authors



Nancy R. Mead, Ph.D., is a senior technical staff member for the CERT Program at the SEI. She is also a faculty member in the Master of Software Engineering and Master of Information Systems Management programs at Carnegie Mellon. Her research interests are information security, software requirements engineering, and software architectures. Mead has more than 150 publications and invited presentations. She is an IEEE fellow and a Distinguished Member of the ACM. Mead received her doctorate in mathematics from the Polytechnic Institute of New York and has bachelor's and master's degrees in mathematics from New York University.

SEI
4500 Fifth AVE
Pittsburgh, PA 15213-3890
E-mail: nrm@sei.cmu.edu



Dan Shoemaker, Ph.D., is the Director of the Institute for Cyber Security Studies, a National Security Agency Center of Academic Excellence, at the UDM. He has been professor and chair of computer and information systems at the UDM for 25 years, and co-authored the textbook, "Information Assurance for the Enterprise." His research interests are in the areas of secure SwA, information assurance and enterprise security architectures, and information technology governance and control. Shoemaker has both a bachelor's and doctorate degree from the University of Michigan, and master's degrees from Eastern Michigan University.

Computer and Information Systems
College of Business Administration
University of Detroit Mercy
Detroit, MI 48221
Phone: (313) 993-1202
E-mail: shoemadp@udmercy.edu

COMING EVENTS

September 27-October 1

13th Semi-Annual DHS Software Assurance Forum

Gaithersburg, MD

<https://buildsecurityin.us-cert.gov/bsi/events/1133-BSI.html>

October 31-November 3

Milcom 2010

San Jose, CA

www.milcom.org

November 7-11

18th International Symposium on the Foundations of Software Engineering

Santa Fe, NM

<http://fse18.cse.wustl.edu>

December 4-8

MICRO-43

Atlanta, GA

www.microarch.org/micro43

December 14-16

DHS Software Assurance Working Group Sessions

McLean, VA

<https://buildsecurityin.us-cert.gov/bsi/events/1135-BSI.html>

January 4-7, 2011

Hawaii International Conference on System Sciences

Koloa, Kauai, HI

www.hicss.hawaii.edu

February 28-March 4, 2011

14th Semi-Annual DHS Software Assurance Forum

McLean, VA

<https://buildsecurityin.us-cert.gov/bsi/events/1136-BSI.html>

COMING EVENTS: Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: <kasey.thompson@hill.af.mil>.

WEB SITES

Software Assurance Metrics and Tool Evaluation (SAMATE)

<http://samate.nist.gov>

After reading Dr. Yannick Moy's *Static Analysis Is Not Just for Finding Bugs*, visit this DHS National Cyber Security Division and National Institute of Standards and Technology-sponsored site, brimming with information on cutting-edge static analysis tools. The SAMATE Web site establishes a methodology for evaluating software assurance tools such as Source Code Security Analyzers, Web Application Vulnerability Scanners, and Binary Code Scanners. There is also the SAMATE Reference Dataset, a community repository of example code and other artifacts to help end-users evaluate tools and developers test their methods. Finally, you can learn more about SAMATE's Static Analysis Summits and Workshops, where the software assurance community has come together—sometimes defining the state-of-the-art in software security tools at these gatherings.

Evaluating and Mitigating Software Supply Chain Security Risks

www.sei.cmu.edu/reports/10tn016.pdf

Earlier this year, the authors of this issue's *Considering Software Supply Chain Risks* were also involved in co-writing this DoD-focused technical report. In this document, Dr. Robert J. Ellison and Dr. Carol Woody (this time with John B. Goodenough and Charles B. Weinstock) present an initial analysis of how to evaluate and mitigate the risk of these unauthorized insertions. The analysis is structured in terms of actions that should be taken in each phase of the DoD acquisition life cycle: initiation, development, configuration/deployment, operations/maintenance, and disposal.

Mids Win Cyber Defense Exercise (CDX)

www.navy.mil/search/display.asp?story_id=52902

Information Assurance Applications in Software Engineering Projects details several lessons learned for U.S. Naval Academy (USNA) student capstone projects, including participation in the National Security Agency/Central Security Service's CDX. During this spring's 10th Annual event, network specialists—whose careers are based around securing the government's most sensitive communication systems—challenged Service Academy teams, specifically testing their ability to defend computer networks through projects the students designed, built, and configured. This article discusses the 2010 event, chronicling the project and sharing interviews with members of the USNA Midshipman's winning team.

National Vulnerability Database (NVD)

<http://nvd.nist.gov>

After reading *Studying Software Vulnerabilities*, you may want to visit this government repository of standards-based vulnerability management data. Information in the NVD enables automation of vulnerability management, security measurement, and compliance. Sponsored by the DHS's Cyber Security Division, its primary resources include a Common Vulnerabilities & Exposures and Common Configuration Enumeration search engine; access to the

National Checklist Program repository; Security Content Automation Protocol-compatible tools and data feeds; an official Common Platform Enumeration Dictionary; a Common Vulnerability Scoring System; and a breakdown of Common Weakness Enumeration.

Common Attack Pattern Enumeration and Classification (CAPEC)

<http://capec.mitre.org>

As outlined in Sean Barnum's *The Balance of Secure Development and Secure Operations in the Software Security Equation*, attack patterns are a powerful mechanism to capture and communicate the attacker's perspective and their approaches to exploit software. The MITRE Corporation's CAPEC Web site is an essential resource in that fight. A DHS-sponsored software assurance initiative, CAPEC provides a publicly available catalog of attack patterns as well as a comprehensive schema and classification taxonomy. The Web site identifies relevant security requirements, misuse, and abuse cases; provides context for architectural risk analysis and guidance for security architecture; prioritizes and guides secure code review activities; provides context for risk-based and penetration testing; leverages lessons learned from security incidents; and helps identify appropriate prescriptive organization policies and standards.

The National Strategy to Secure Cyberspace

<http://georgewbush-whitehouse.archives.gov/pcipb>

In their article *Two Initiatives for Disseminating Software Assurance Knowledge*, Dr. Nancy R. Mead and Dr. Dan Shoemaker discuss the 2003 White House initiative that listed—as its third of five priorities—the need to establish a National Cyberspace Security Awareness and Training Program. At this Web site, you can read the entire strategy, including a case for action, a letter from then-President George W. Bush, as well as the sections detailing the other four priorities: a National Cyberspace Security Response System; National Cyberspace Security Threat and Vulnerability Reduction Program; securing governments' cyberspace; and national security and international cyberspace security cooperation.

Defense Research & Engineering (DDR&E)

www.acq.osd.mil/ddre

As part of the DoD, the DDR&E extends the capabilities of current warfighting systems, develops breakthrough capabilities, and develops counter-strategic scientific and engineering options. At this Web site, viewers can read reports prepared for Congress including the "Strategic Communication Science and Technology Plan," learn the latest technology news from DDR&E and other government entities, and find out about their Science, Technology, Engineering and Mathematics Education and Outreach program. The site also promotes their latest portal, DefenseSolutions.gov, which guarantees 30 days-or-less response to ideas for products, services, prototypes, and concepts that advance the military's missions.



Tools of the Trade (or “Why Access Isn’t Just the Name of a Database Program”)

This issue talks about how tools and practices change “the game.” I am not sure really what “the game” is, but I certainly understand tools and practices. And—as long as we are talking about electronic gadgets—why, I LOVE tools!

At the lowest level, a “tool” is pretty easy to define. According to my handy Merriam-Webster Dictionary (the online version), it is a “device that aids in accomplishing a task.” This definition works for me (pun intended)¹.

Do you have to be “intelligent” to use a tool? Depends on your definition of “intelligence.” Anthropologists have decided that monkeys are intelligent because they have been observed in the wild using sticks to help dig and twigs to help them capture termites for food. While some previous co-workers come to mind with this anecdote², I would like to propose an alternative definition of tool use. To me, significant tools alter my “access.” Let me elaborate.

Prior to the 1980s, computers were certainly awesome machines. They allowed users like me to automate processes that used to take me days (or weeks). My first personal computer allowed me to organize my VHS tapes, my CD collection, play games, etc. And then, I bought a modem. WOW—I was able to exchange programs and data with other users! We had bulletin boards to exchange programs, simple newsgroups allowing me to exchange thoughts ... and e-mail! I had access to others.

And then the Internet evolved. Stores put inventories online. Books and pictures and advertisements and Web sites abounded. Not only did I have access to others, I had access to information. Lots of it—but too much to easily manage.

And behold: Search engines came into being! In just a few years, we replaced saying “I’ll look that up” with “I’ll google that.” In fact, the Merriam-Webster online dictionary gives the lower-case definition of “google” not as a noun (and a name for a search engine), but as a transitive verb! So now I can google—and have access to useful information.

And, during these years when the Internet was becoming as commonplace as indoor plumbing, another driving force was shaping our use of tools: the cell phone. Remember the “good old days” when you could leave your office and were expected to be unreachable for long periods of time? You received paper messages? Then the cell phone (and its diminutive cousin, the beeper) gave people instant access to others. Can’t find them? Call. Can’t get them to answer? Leave a voice mail. Haven’t responded to voice mail yet? Txt them³!

And, I must confess that I did not stop with the cell phone: I use both an iPhone and an iPad. Never thought I would—after all, I just wanted my phone to make and receive phone calls. But once I tried it, I was hooked. I’m able to browse the Internet pretty much anywhere, anytime. Somebody wants to know the name of Judy Garland’s stunt double during the “Wizard of Oz”? No problem: Snap out the phone, hit Google Search, and

find out⁴. And my iPad is even more addicting. I have about 20 books loaded on it, and can use it anywhere. If I desperately need Internet access, it has 3G capability—so that I don’t look like a TOTAL geek⁵ with an iPad in one hand and an iPhone in the other. I now have instant access to useful information combined with instant access to others. Anytime, almost anywhere. In fact, when I am driving home to visit my folks, I frequently check my phone to see what kind of coverage area I am in. If I don’t see any bars of coverage, I compulsively check frequently until I am in range again. And wonder: Was anybody trying to reach me? We have become so dependent on our tools that we can’t stand being out of access!

I used to tell the story about a boss I had many, many years ago. The rest of us were updating to new dual-core Pentium machines, a new operating system, and we were putting in a T1 line for really high-speed Internet access.

And we got to thinking about our boss. A few of us made the suggestion that perhaps—to increase office productivity—we should give him a computer running MS-DOS 3.1, Internet access via a Hayes 300 Baud Smartmodem, and a 16-color 640x200 Color Graphics Adapter monitor. It might not have speeded up his access to us, but it would certainly increase our productivity by, oh, say 1,000 percent: His constant barrage of almost totally useless e-mails would slow down, and we could do real work.

Tools are work multipliers. They are supposed to permit you to do more in a limited time, and are supposed to make your life easier and more productive.

What tools do you use? How well do you use them? Does your use of the “tools of your trade” hinder others? Are you getting a Hayes 300 Baud Smartmodem for Christmas?

—David A. Cook, Ph.D.

Stephen F. Austin State University
<cookda@sfasu.edu>



Notes

1. Oddly enough, the very first definition I got when googling “tool” was from Urban Dictionary: “One who lacks the mental capacity to know he is being used. A fool. A cretin. Characterized by low intelligence and/or self-esteem.” I’ll save this definition for another column at a later date.
2. No names, but if you looked here, you were worried I was going to mention you, right?
3. While dictionaries do not yet define “txt” as a word, it is interesting to note that Microsoft Word did not flag it as misspelled!
4. Bobbie Koshay. Found it on my first try, using my iPhone, at <<http://thewizardoffoz.info>>. Also note that Caren Marsh-Doll also helped out with blocking and camera tests.
5. Yeah—I already know. Too late. Don’t e-mail.



Homeland Security

Software Assurance



Software is essential to enabling the nation's critical infrastructure.

To ensure the integrity of that infrastructure, the software that controls and operates it must be secure and resilient.

Software Assurance Community Resources and Information Clearinghouse provides collaboratively developed resources. Learn more about relevant programs and how you can become involved.

<https://buildsecurityin.us-cert.gov/swa/>

Security must be "built-in" and supported throughout the lifecycle. Visit <https://buildsecurityin.us-cert.gov> to learn more about the practices for developing and delivering software to provide the requisite assurance. Sign up to become a free subscriber and receive notices of updates.

The Department of Homeland Security provides the public-private collaboration framework for shifting the paradigm to software assurance.



NAV AIR



CROSSTALK thanks the above organizations for providing their support.