CrossTalk

May / June 2016 **The Journal of Defense Software Engineering** Vol. 29 No. 3

INTEGRATION AND INTEROPERABILITY

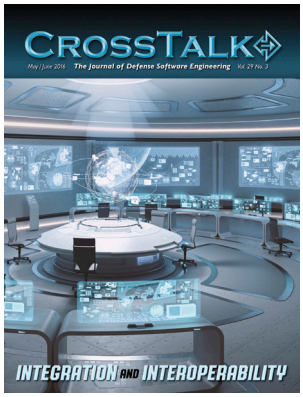
TABLE OF CONTENTS

Departments

3 From the Sponsor

38 Upcoming Events

39 BackTalk



Cover Design by Kent Bingham

Integration and Interoperability

4 **Continuous Integration in the Cloud: Improving Cost, Schedule and Technical Performance**
Program Managers need to continue to seek ways to improve cost, schedules and technical performance.
By Elfriede Dustin and Kevin Caldwell

8 **The State of Security Vulnerabilities in SCADA Human Machine Interface (HMI) Components**
Inherent design flaws and vulnerabilities that allow attackers to take control of SCADA systems.
by Aditya K. Sood

13 **DevOps Advantages for Testing: Increasing Quality through Continuous Delivery**
DevOps and continuous delivery can improve software quality and reduce risk by offering opportunities for testing and some non-obvious benefits to the software development cycle.
by Gene Gotimer and Thomas Stiehm

19 **They Know Your Weaknesses – Do You?: Reintroducing Common Weakness Enumeration**
Knowing what makes your software systems vulnerable to attacks is critical, as software vulnerabilities hurt security, reliability, and availability of the system as a whole.
by Yan Wu, Yaacov Yesha, and Irena Bojanova

25 **An Alternate Approach to Avionic Software: KISS**
Driven by customer perceptions of cost, there is a recurring drive in the avionics community to provide overarching software frameworks.
by Gerry Tyra

29 **Joint Radio Manager Enhances Service Interoperability**
With the maturity of tactical networking waveforms comes the need to consolidate the planning and management of these waveforms into a joint management system.
by Dean Nathans, Dan Preissman, and Alan Gebele

33 **Enterprise Systems Integration using Collapsing Design Structure Matrices**
Using Collapsing Design Structure Matrices (C-DSMs) to identify and develop cost-effective systems integration plans.
by John M Colombi, Michael P. Kretser, Jeff Ogden, and Paul Hartman

CrossTalk

NAVAIR Jeff Schwalb
DHS Peter Fonash
309 SMXG Karl Rogers
76 SMXG Mike Jennings

Publisher Justin T. Hill
Article Coordinator Heather Giacalone
Managing Director David Erickson
Technical Program Lead Thayne M. Hill
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Senior Art Director Kevin Kiernan
Art Director Mary Harper

Phone 801-777-9828
E-mail Crosstalk.Articles@hill.af.mil
CrossTalk Online www.crosstalkonline.org

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsors: Ogden-ALC 309 SMXG and Tinker-ALC 76 SMXG. DHS co-sponsor: Office of Cybersecurity and Communications 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.

Subscriptions: Visit <www.crosstalkonline.org/subscribe> to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

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.crosstalkonline.org/submission-guidelines>. 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 authors 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: CrossTalk 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:
For questions or concerns about crosstalkonline.org web content or functionality contact the CrossTalk webmaster at 801-417-3000 or webmaster@luminpublishing.com.

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

CrossTalk is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing <luminpublishing.com>. ISSN 2160-1577 (print); ISSN 2160-1593 (online)

CrossTalk would like to thank **NAVAIR** for sponsoring this issue.

So why is this issue of CrossTalk dedicated to the topic of Integration and Interoperability? First we must understand that more and more of the systems fielded by the U.S. Department of Defense (DoD) are more and more by design becoming Systems of Systems (SoSs). This includes weapons systems, command and control systems, large-scale information management systems, just to name a few.

A SoS is different from a single system. It is actually a set of components that when separated are still regarded as systems themselves. This means that each of these individual systems remain operational after the SoS they are associated with is disassembled. Further, each of these individual systems is independently managed. This means they can and do operate as individual entities and this continues regardless of the SoS of which they are a component.

With the idea of a SoS in mind we realize there must be processes and properties defined that allow these individual systems to operate together and exchange information. To this end and of equal importance we must look at Integration and Interoperability.

Starting with Integration we have the processes for creating a larger and more complex entity by combining or adding individual parts. It is a key step during development during which subsystems and other software components are combined. This produces a larger system in which many systems are combined to produce a SoS.

Next is Interoperability as a property of a system or SoS. It refers to the ability to exchange information among many system elements. For SoSs, the needed information exchange is in support of end-to-end SoS capabilities.

The integration process produces an integrated system, meaning that the system's elements must work together to achieve required system functions. These elements working together are then defined as interoperable.

Integration and interoperability are often used somewhat interchangeably, since the purpose of system integration is to achieve a needed degree of information exchange among system components.

Much of the above discussion was taken from the introduction of an SEI Technical Report by Carol A. Sledge, Ph.D.—Reports from the Field on System of Systems Interoperability Challenges and Promising Approaches [CMU/SEI-2010-TR-013].

In this issue of CrossTalk appear several informative articles regarding Integration and Interoperability in various System of Systems.

Jeff Schwalb
NAVAIR Process Resource Team

Continuous Integration in the Cloud

Improving Cost, Schedule and Technical Performance

Elfriede Dustin, Innovative Defense Technologies (IDT)
Kevin Caldwell, Innovative Defense Technologies (IDT)

Abstract. Program Managers need to continue to seek ways to improve cost, schedules and technical performance. This article provides a summary of industry best practices we have applied successfully that enable program managers to implement processes and practices that can result in the improved cost, schedule and technical performance that the DoD is continuously seeking.

Introduction:

For nearly 70 years, the Department of Defense (DoD) has engaged in a constant process of acquisition self-assessment, striving to continuously improve the way it acquires weapons systems. Generally, the concern is that acquisition costs are too high and the process is too slow, involving too many stove-pipes. According to the Performance of the Defense Acquisition System 2015 Annual Report, some positive change has taken place with various contracts and initiatives; however, program managers are still encouraged to actively seek ways "to save money and to set targets for doing so, not just to stay within their budgets [1]."

This article provides a summary of best practices we have implemented that are gaining momentum in the industry. These practices have resulted in the types of improved cost, schedule and technical performance that the DoD is seeking. Some of these systems engineering best practices include: virtualization, continuous integration, automated testing using Automated Test and ReTest (ATRT), and hosting continuous integration solutions in the cloud. These best practices have been applied to over 80 programs at IDT with significant, measurable results. The outcomes summarized below are taken from two "approved for public release" case studies from IDT's work with NAVAIR [2] and NAVSEA [3].

NAVAIR

- Increased testing efficiency by greater than 75%. The result is significantly less time and manpower are required to conduct testing.
- The number of requirements, permutations and configurations being tested has increased along with consistency of the testing. In addition, test teams have been able to identify software defects earlier in the schedule.
- Automated test cases are being shared and reused across the responsible contractor and government teams (e.g. removing stove-pipes). Besides the efficiency of reusing test cases, the time and scope of incorporating automation is also being accelerated.

NAVSEA

- Increased testing efficiency for those critical system and software requirements where automation was applied. The result is significantly less time and manpower being required to verify the associated requirements and system performance.
- Improved collaboration among test teams. The application of ATRT facilitated efficient sharing of analysis cases between the various AEGIS test entities. As a result, each test team gained the ability to conduct more thorough analysis at each testing stage.
- Improved software quality and reduced risk. Automation has increased requirements coverage and expanded the data able to be evaluated to assess the system performance. Additionally, sharing of analysis methods between test teams has enhanced defect resolution.

Acquisition program managers face the challenge of not only grasping all practical business concerns, but also of understanding and managing a diverse range of topics, including: risk identification and mitigation, selection and integration of commercial off-the-shelf (COTS) components, process capability, program management, architecture, survivability, interoperability, source selection, continuous integration, software development tasks, verification and validation, and contract monitoring.

The use of a comprehensive suite of management capabilities is designed to orchestrate and optimize complex software engineering oversight, Continuous Integration (CI), and human-centric acquisition processes across the value chain. Next, we will provide an overview of a few of the technologies and best practices that in our experience can increase efficiencies and reduce the work load an acquisition program manager faces.

Virtualization and Continuous Integration (CI):

Virtualization and Continuous Integration are two of the biggest time and cost savers we have implemented for our customers. We've discussed virtualization and CI in detail in our article "Efficiencies of Virtualization in Test and Evaluation" [4] in the July 2013 edition of Crosstalk. CI is one of the best industry-adopted software engineering practices in which any change to the code or environment is tested and reported on as soon as feasible. In most cases this involves nightly software builds and nightly automated test runs to allow for quick look reporting on any newly introduced issues. Virtualized development and test environments play a major role in this CI practice. In "eating our own dog-food," we've continuously expanded on these best practices. For example, we have implemented an increasingly efficient automated CI solution as a pluggable framework of CI applications which includes an automated process and the capability of being hosted in the cloud. We call this solution/methodology the CI-Cloud. Additionally, CI-Cloud orchestrates a tool-independent environment, and tools such as Jenkins, SVN and GIT version control systems are hosted and seamlessly integrated with project scheduling and management tools such as Redmine and requirements management tools. The features are described in further detail below.

CI-Process Modeling:

IDT has automated the modeling of the CI process, termed CI-Process Management (CPM), to provide a bridge between the customers receiving a delivery and developers and engineers implementing and testing a solution. Our customers can now receive continuous development status via access to the CI-Cloud, software with the automated test cases, and virtualized hardware (versus having to purchase their own hardware). This CI Process, built into the CI-Cloud, offers the following advantages:

- The CI Process enables users to model Continuous Integration and Application Delivery goals via a flow chart which describes the steps needed and the order required to achieve that goal.
- The CI Process improves the visibility, monitoring and agility of software delivery logic, resulting in higher-level and domain-specific representations that can be understood by DoD customers and DoD contractors.
- Corporate and domain-specific CI-processes can be plugged into a modifiable palette, making the CI process more easily understood.
- This CI Process Management is not an isolated process engine. Complex CI logic can be modeled as a combination of CI processes with conversion and migration rules between existing CI environments and the CI-Cloud.

(See Figure 1: "Process Modeling" for more details)

With this automated process modeling we are achieving a goal of 80% reduction in manual interactions and faster issue resolution by allowing DoD agencies to design, integrate, deploy, execute, monitor and optimize their critical software engineering acquisition processes and operations. This process will:

- Automatically prioritize and route work and tasks to stakeholders
- Guide users, contractors, developers, and program managers through decisions
- Standardize resolutions across geographies
- Leverage existing CI and Program Management systems and data
- Monitor for business events and initiate action
- Provide real-time visibility and process control

CI Pluggable Framework/Application Store:

This CI-cloud consists of a pluggable framework that allows for adding/removing CI applications and capability with ease.

- All CI-Cloud applications are portable and self-contained.
- Archives can be deployed via the CI 'appstore'.
- Upgrade, downgrade, stop, start, deploy, undeploy, as easy as clicking a button

This pluggable framework allows customers to choose their development environment with specifically preferred tools. For example, users can choose between a Java based/Eclipse development environment and a C/C#/Visual Studio/Team Foundation Server development environment. This framework comes with the build server of choice, along with the source control and automated testing solution. For example, it automatically provides access to Jenkins, SVN, and ATRT.

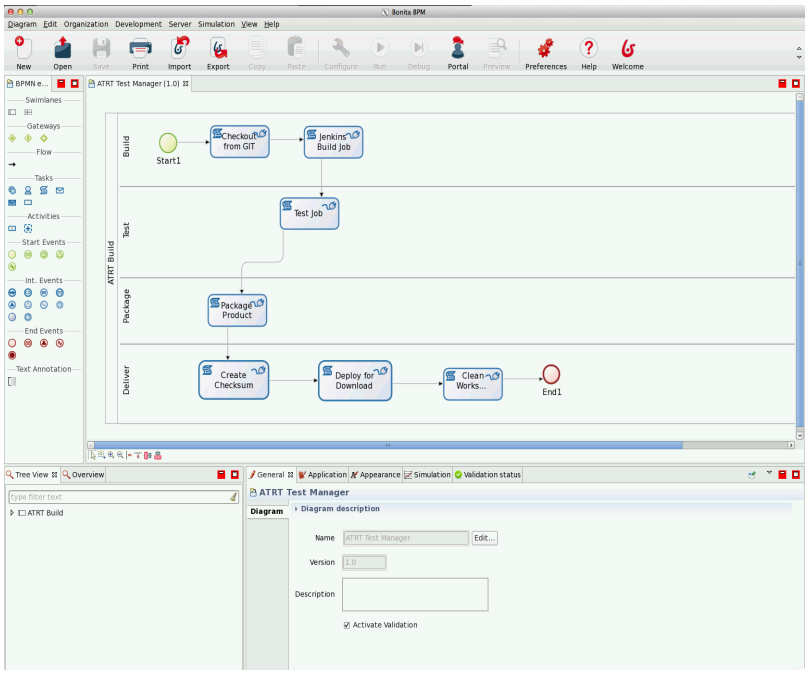


Figure 1: "Process Modeling"

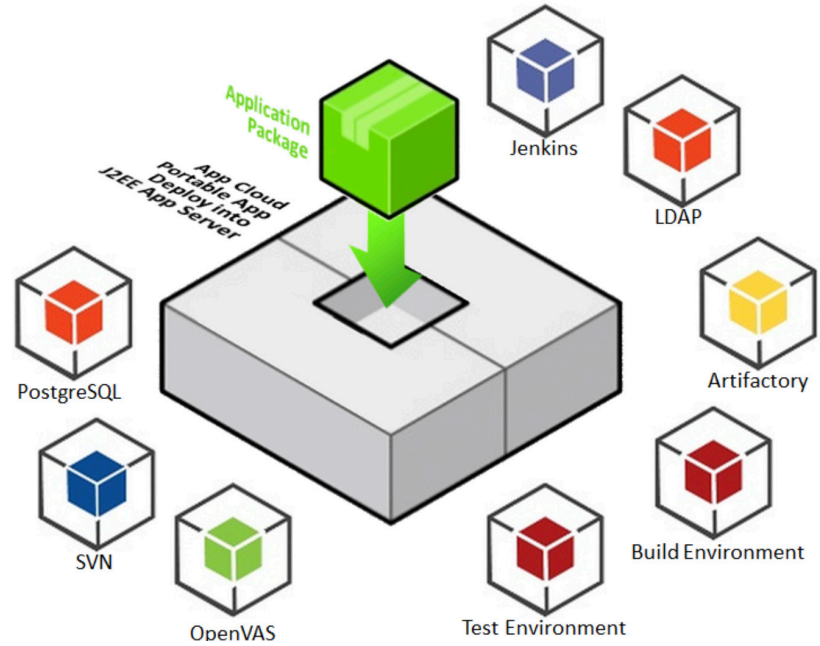


Figure 2: CI pluggable application store

CI in the Cloud:

Much has been written about the need for and benefits of cloud computing, such as quicker and cheaper delivery and reduced hardware costs. "Tech historians will look at October 22, 2015 as a watershed," according to New York Times reporter Quentin Hardy. He goes on to say, "Cloud computing is no longer on the way, just a contender, or even a competitor to traditional enterprise technology companies. Instead, it is here, full force, and all the signs are that it is about to get a lot bigger, fast [5]."

Figure 3: "Before CI-Cloud and After"

Our customer's goal of being able to access CI in the cloud was various. A few of the goals were:

1. To allow for better coordination. Now, their Sprint backlogs and schedules can be accessed and modified in CI-Cloud via Redmine by Government and contractors alike;
2. To grant access to both developers and testers. Now, both groups can use CI-Cloud for development, testing, building, nightly automated tests, and results reporting;
3. To save money on hardware;
4. To increase visibility and insights into development / test progress; and
5. To move from manual disk / software delivery to an automated pushdown download and install.
   (For more detail on this last goal, see Figure 3: "Before CI-Cloud and After".)

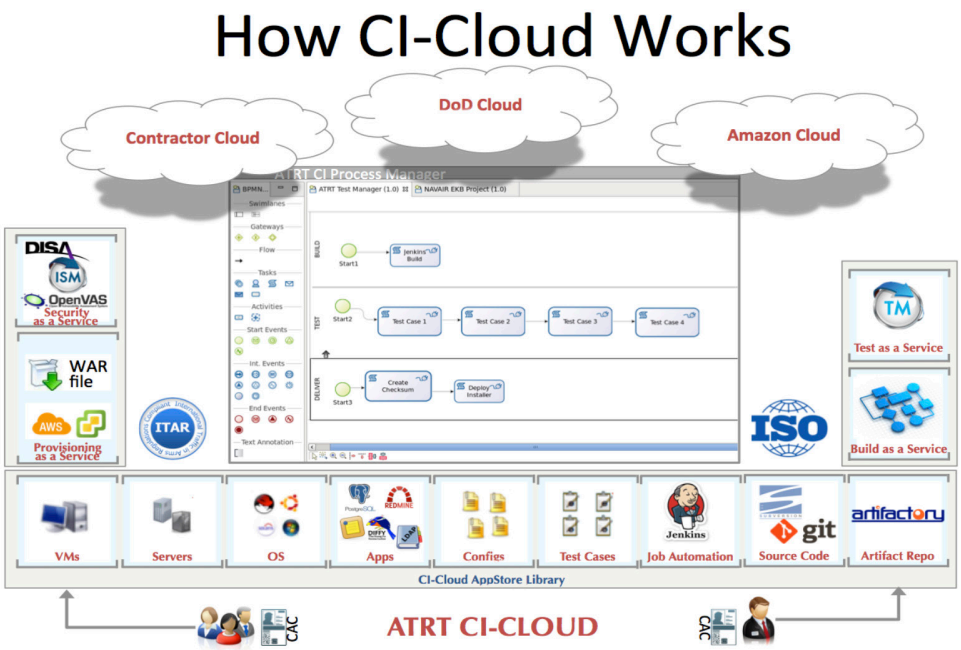


Figure 4: "How CI Cloud works"

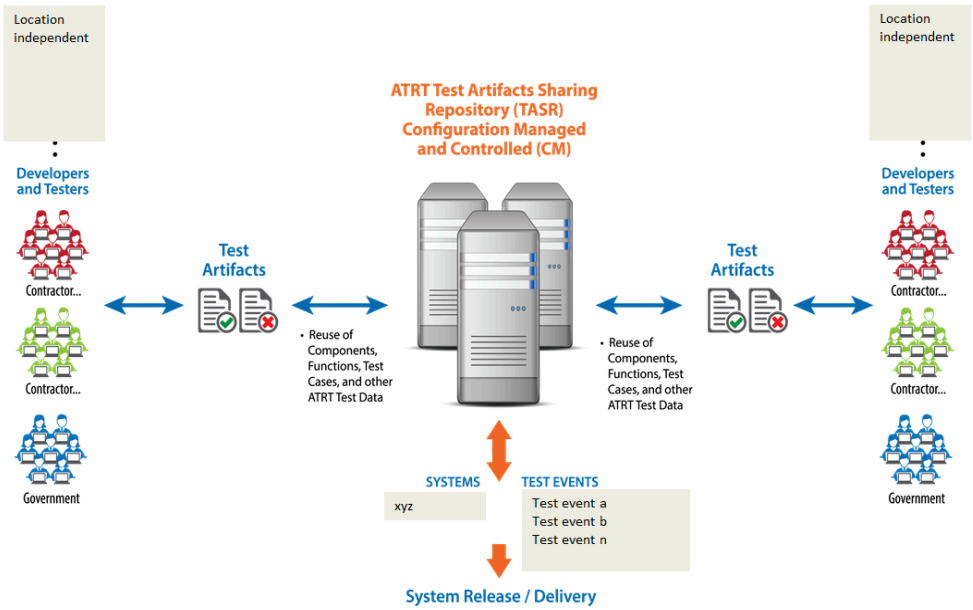


Figure 5: "Reducing Test Silos"

For CI-Cloud we chose to use Amazon Web Services (AWS) as our cloud service provider (though the CI-solution is cloud or environment agnostic).

This CI solution can be hosted in the cloud and it allows program managers to:

- Manage the entire software engineering lifecycle – from design to optimization
- Ascertain continuous process improvement using closed-loop control
- Reduce time to introduce new CI and acquisition processes
- Improve stakeholder productivity
- Cut operational costs by up to 40 percent by automating and standardizing CI and reducing hardware cost
- Improve mission delivery and drive CI and software acquisition process agility
- Extend the ROI of existing CI and Program Management technology investments
- Ensure continuous compliance with internal best practices and regulatory standards
- Increase competitive advantage and DoD agency satisfaction

(For more detail, see Figure 4: "How CI Cloud works".)

CI-Cloud and Automated Test and ReTest (ATRT):

Our article "Efficiencies of Virtualization in Test and Evaluation" [6] in the July 2013 edition of Crosstalk also provides detailed examples of automated software testing in a virtualized test environment which include: 1.) Automatic provisioning of a virtualized automated test environment; 2.) Automatic provisioning of the entire automated testing lifecycle for any type of SUT; and 3.) Continuous Integration using virtualized environments. By implementing those solutions we have been able to remove

the stovepipes of testing, allowing vendors and government facilities located on opposite coasts to share their automated test procedures, reduce the repetition of work by reusing tests, and minimize silos.

(See Figure 5: "Reducing Test Silos" for more details.)

Cloud Security and Authority To Operate (ATO):

Many security measures will be shared or inherited due to CI-Cloud building systems on top of the AWS Cloud infrastructure. CI-Cloud will provide security for its software components, and Amazon AWS GovCloud will provide security for its infrastructure. CI-Cloud is able to leverage security controls from AWS's security, meaning that CI-Cloud will not have to provide those controls for its components since Amazon AWS GovCloud is already providing them.

CI-Cloud assumes responsibility for, and management of, the guest operating system (including updates and security patches), other associated application software, and the configuration of the AWS-provided security group firewall.

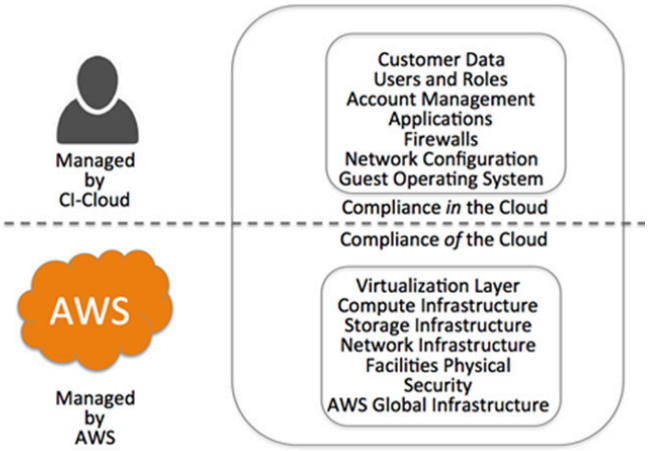


Figure 6: Amazon vs CI-Cloud security

As illustrated in Figure 6: "Amazon vs CI-Cloud security," Amazon AWS GovCloud will provide security controls from the virtualization layer down to the physical security of the facilities in which the service operates.

For more on CI-Cloud and related ATOs, stay tuned for a follow-on article that discusses security in the AWS cloud in detail.

Implementing the CI solution in the cloud is just one step towards automating the many facets of the acquisition program. The acquisition program is subject to numerous influences, both internal and external to the program. Some influences such as budget constraints, schedule constraints, and performance requirements are well quantified and easily understood. Other influences, such as stakeholder agreements, requirements stability, and contractor capability, are more difficult to assess and less obvious. These influences, or program drivers, are sources of risk to an acquisition program. For these challenges, methodologies that identify and mitigate some of these risks are available. Much more can be done to streamline and automate the acquisition process. In this article we presented a subset of some of the proven best practices that have led to saving the government money and improving efficiencies.

REFERENCES

1. http://www.acq.osd.mil/fo/docs/Performance-of-Defense-Acquisition-System-2015.pdf
2. http://idtus.com/blog/case-study-implementing-automated-testing-for-us-naval-air-systems-command-navair/
3. http://idtus.com/blog/idt-case-study-application-of-automation-to-a-navy-weapon-system/
4. http://static1.1.sqspcdn.com/static/f/702523/23063510/1373252378767/201307-Dustin.pdf?token=1Fs8vD3BCjsHswYj4vPyeZR65D0%3D
5. http://bits.blogs.nytimes.com/2015/10/23/the-cloud-is-here-separating-disrupters-from-disrupted/?_r=0
6. http://static1.1.sqspcdn.com/static/f/702523/23063510/1373252378767/201307-Dustin.pdf?token=1Fs8vD3BCjsHswYj4vPyeZR65D0%3D

ABOUT THE AUTHORS

Elfriede Dustin is Director of Solutions at Innovative Defense Technologies (IDT) where she works on developing new ideas and discovering new approaches to software engineering challenges. Software development is still an art and that makes automated software testing and software engineering a special challenge. IDT (www.idtus.com) strives to meet that challenge by producing edge of technology solutions, starting with requirements through the entire secure software testing lifecycle to defect closure. Elfriede has a B.A. in Computer Science and over 20 years of IT experience, implementing effective software engineering processes and testing strategies, both on government and commercial programs. Together with IDT CEO Bernie Gauf and IDT FSO and Sys Admin Guru Thom Garrett, Elfriede wrote her latest book "Implementing Automated Software Testing." Elfriede's goal is to continue to help further the software engineering/automated software testing advances.

Book list: amazon.com/author/elfriededustin

Kevin Caldwell is a leading security scientist working at Innovative Defense Technologies (IDT). He has over 20 years of experience leading the design, development, and delivery of innovative security and IT solutions for the DoD and commercial entities. His government experience includes hands-on lead roles in security engineering, development, and the production of information systems and cloud technologies for NAVAIR, NAVSEA, Internal Revenue Service, Department of Labor/OSHA, FBI, FCC, and SPAWAR. In 2015, Kevin designed and developed a portable application hypervisor solution, based on top of the Amazon Web Services (AWS) Cloud, for Continuous Integration (CI) activities for NAVAIR and other DoD agencies. As part of these activities, he developed and secured AWS for a CI Cloud for JMPS/NAVAIR and he prepared ATOs based on DoD RMF (8510.01) for Cloud Services. Additionally, he designed, developed, and managed a secure, cloud-based data platform and CI environment solution for the U.S. Navy which was based on AWS infrastructure, but portable to other clouds. In 2013, he developed the concepts, architecture, and capabilities for the SPAWAR/NMCI/NGEN Information Security Manager (ISM), a custom Puppet infrastructure with visual controls, necessary to support the U.S. Navy's complex *Security Vulnerability Management and Automated Remediation of Vulnerabilities* across the entire enterprise of over 500,000 endpoints.

The State of Security Vulnerabilities in SCADA Human Machine Interface (HMI) Components

Aditya K. Sood, Elastica

Abstract. Supervisory Control and Data Acquisition (SCADA) systems are becoming the primary target of attackers to launch cyber attacks against critical infrastructure. The attackers are exploiting vulnerabilities in different components of the SCADA systems to gain access so that critical systems can be abused or exploited for malicious purposes. In this paper, the state of web Human Machine Interfaces (HMIs) security is evaluated, including the inherent design flaws and vulnerabilities that allow attackers to take control of SCADA systems.

Introduction

Critical infrastructure [1] such as military defense systems, industrial systems, utilities and refineries, etc. are facing serious threats from attackers all around the world. Critical infrastructure is becoming a more frequent target in cyber attacks because of the impact these systems have on nations and organizations if these systems are exploited successfully. For example, compromising a wind turbine SCADA system can have disastrous impact on the community around. Turbines are used for generating electricity and pumping water to be used in the vicinity by the civilians and even by the organizations. If these are shut down, the day-to-day functioning of homes and businesses are disturbed on a large scale.

SCADA systems have previously encountered advanced threats such as Stuxnet [2] and Havex [3]. Stuxnet was designed by nation states to target Iranian nuclear power utilities and disrupt them accordingly to destroy the country's critical infrastructure. Havex was another advanced threat distributed through malicious updates sent by control manufactures to gain information about SCADA systems and execute unauthorized commands for nefarious operations.

Critical infrastructure is facing threats not only from advanced malware designed by attackers, but also from intentional attacks by malicious insiders and unintentional mistakes made by an organization's employees.

It is necessary to first understand what SCADA means in the context of critical infrastructure. Primarily, critical infrastructure (oil, gas, electricity, hydraulics, etc.) requires industry-specific equipment for operational purposes. This equipment is steered and administered by the computer systems, typically called either controllers or sensors. All these controllers (or sensors) are managed and controlled by dedicated management systems to form SCADA systems. By definition, SCADA systems acquire data from multiple sources in the field to perform operation analysis to control the field equipment via computer. Table 1 shows the different equipment and devices that are categorized as SCADA subsystems.

S. No	SCADA Subsystems
1	Remote Terminal Unit (RTU) Communication Infrastructure
2	Human Machine Interface
3	Instrumentation and Analytical Process Control Systems
4	Telemetry Systems
5	Data Acquisition and Application Servers
6	Programmable Logic Controllers (PLCs)
7	Historian Client for Data Acquisition
8	Network Communication Infrastructure for Intermediate Connections
9	ERP and MES Business Environment Systems
10	Industrial Cloud Computing Environment

Table 1: Potential sub-systems of SCADA Infrastructure

HMIs and SCADA are interrelated, HMIs are the control panels that can be easily managed and operated by the SCADA administrators from remote locations. The software-based HMIs restrict the use of hard-wired control panels and can be easily operated in real time when data is acquired through an application server, a computer-system designed to run support applications.

Researchers from Iowa State University presented a cyber security assessment framework [4] for SCADA systems that evaluates SCADA security vulnerabilities by taking into consideration control points, systems and scenarios. The researchers also discussed the importance of attack trees [5] in assessing security posture of SCADA systems. Researchers have also used simulation [6, 7] based approaches by designing a well-structured test bed to assess the security of SCADA systems by generating abstract models of various components. Another interesting study on reducing vulnerabilities in SCADA systems used optimization [8] techniques to restrict exploitation in SCADA systems. This research is an outcome of manual analysis of code and penetration testing techniques in controlled manner to decipher vulnerabilities in SCADA web HMIs.

In this paper, we discuss vulnerabilities that exist in SCADA web-based HMIs including thin clients that use Java, Flash, or ActiveX as underlying technologies. This article is an outcome of real-time research conducted to understand the state of SCADA web HMIs security by analyzing inherent software design flaws and security vulnerabilities that exist in globally recognized SCADA products. The vulnerabilities [9, 10, 11,12, 13, 14] disclosed during the course of this research have been reported to ICS-CERT so they can be patched quickly.

Threat Model: Involved Actors

We can divide threats in critical infrastructure environments into three categories:

Employees can make mistakes due to a lack of awareness about existing threats and social engineering tricks used by attackers, continuous use of unsanctioned applications (websites) and online services, unrestricted sharing and clicking of shared links on the social media websites, inserting Universal Serial Bus (USB) devices directly into main systems are just some of the primary factors that lead to compromise of SCADA systems through unintentional errors made by the users.

Malicious insiders, disgruntled employees that want to harm the organizations they work for, are one of the major issues every organization faces today. These people have authorized access to critical areas of infrastructure including Intellectual Property (IP) documents, financial information, etc. Physical access to servers allow malicious insiders to perform

rogue operations from inside the organization. Critical infrastructure is not immune from these risks. Malicious insiders can easily plug-in USB devices containing malware directly into the mainframe computers to infect SCADA systems. Stealing critical documents is one of biggest risks malicious insiders pose to the organizations. Theft can have disastrous impacts on the business if documents are shared publicly or with competitors.

Remote attackers also create issues for organizations that use SCADA systems. When SCADA systems were designed, the notion of advanced security was not considered. As a result, SCADA systems are riddled with vulnerabilities. In fact, every single component of SCADA system possesses inherent vulnerabilities. Remote attackers exploit these vulnerabilities to gain access to SCADA systems through direct or indirect channel exploitation. Direct channel exploitation includes techniques that remote attackers adapt to exploit vulnerabilities directly in the SCADA components to gain access. Indirect channel exploitation refers to tactics that remote attackers follow to target users that use SCADA systems, which include social engineering attacks, drive-by download attacks and a host of others. For both methods, the majority of vulnerabilities exploited are either in SCADA components or end-user client software such as browsers, third-party plugins, etc. An attacker can simply reverse engineer the vulnerable component of SCADA system to extract hard coded credentials to gain access to the system. In addition, remote attackers can also convince the users to visit malicious domain serving malware through well-structured socially engineered phishing emails.

Vulnerabilities in Web HMI Components

In this section, we will discuss high risk and most frequently noticed design flaws and vulnerabilities in SCADA web HMIs. Due to responsible disclosure guidelines, vendor names will not be disclosed; rather, the issues presented are part of a general discussion to highlight the software security problems in SCADA HMI components. The security issues are discussed in detail below:

Hardcoded Credentials

This security issue is often found in the SCADA web HMIs that use Java, Flash and other clients for communicating with the primary server to exchange data at regular time intervals. To avoid complex implementation of authentication schemes, SCADA developers embed the authentication credentials or private keys directly into the web client to streamline the process. The issue is that thin clients downloaded through browsers on end-user machines can be reversed easily by remote attackers. For example, decompiling Java and Flash clients can reveal all the information about inherent classes and ActionScript codes respectively. Attackers can easily interpret the code and extract the credentials, and as a result can directly access the SCADA device through the web. In general, hardcoding any critical information in the client software itself is a poor security design. Figure 1 shows hardcoded credentials are passed as configuration parameters in the Java client in one of the web HMI used for managing Programmable Logic Controllers (PLCs), controllers used in electric utilities.

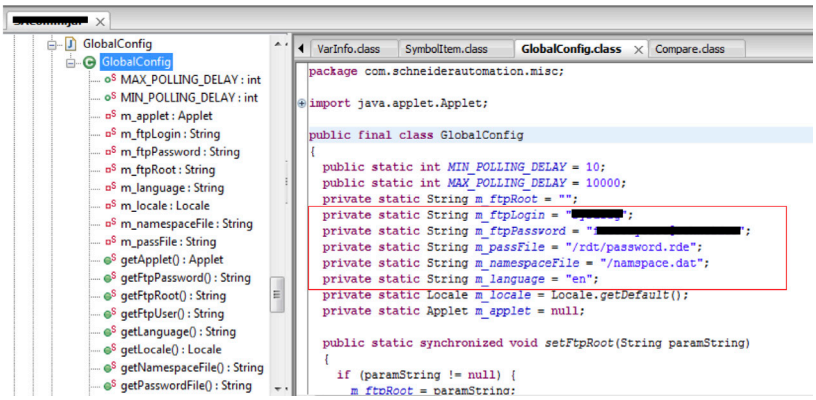


Figure 1: Hardcoded Credentials in Java Client used for Web HMI

Figure 2: HTML File Included from Remote Website in one PLC Web HMI

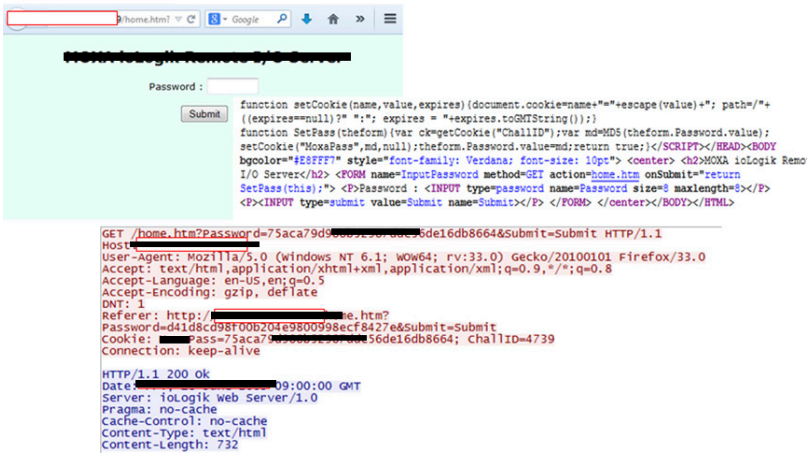


Figure 3: Password in MD5 Transmitted over non-HTTPS Channel in SCADA Web HMI used for Industrial Ethernet

File Inclusion Flaws

File inclusion vulnerabilities refer to the inability of web HMIs to restrict execution of payloads embedded in the files from remote locations. In general, web HMIs can process code from files hosted on the third-party domains. Listing 1 shows how attacker can exploit the "URL" HTTP parameter to construct exploit URL that can execute code if shared with the target. In this case, the web HMI does not validate and verify the input passed to the URL and executes the file content in the active session managed by the end-user.

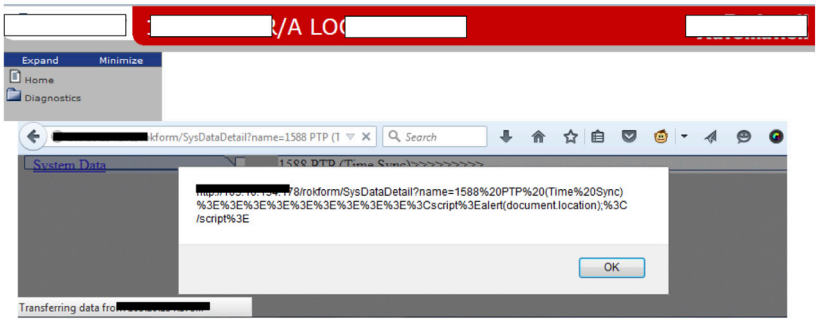


Figure 4: Cross-site Scripting in SCADA Web HMI used for Industrial Automation

```
POST /files HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:39.0) Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://[redacted]/files
Cookie:
siemens_ad_session=B4WsczI6cEDwMwueKxxJZf7GusjDoX/TEAAAAPiZE5314QWT1kMUFedwxtOqYWRtaW5pc3RyYXRvcioyMDY
4NjMSNjIqMSo=, path=/
Connection: keep-alive
Content-Type: multipart/form-data; boundary=----------------------------22883300799445
Content-Length: 203

----------------------------22883300799445
Content-Disposition: form-data; name="file"; filename="test.txt"
Content-Type: text/plain

CSFU - Upload !
----------------------------22883300799445--
```

```
HTTP/1.1 200 OK
Content-Type:text/html
Content-Length: 553

<HTML>
  <HEAD>
    <TITLE>
      File Operation Complete
    </TITLE>
    <META HTTP-EQUIV="content-type" content="text/html; charset=UTF-8" />
oder UTF-8 -->
    </HEAD>
  <LINK REL="stylesheet" TYPE="text/css" HREF="/CSS/MiniWeb.css">
  </LINK>
    <BODY>
      <SCRIPT>
      window.location.href='/files';
      </SCRIPT>
    </BODY>
</HTML>
```

Figure 5: CSFU Request Issued for Uploading File in Web HMI used for Industrial Automation

```
[redacted]/user/scripts/login_par.js

// JavaScript Document
// 11 D[redacted]

var txtChkUser                    = [redacted];  // Nome utente Login
var txtChkPassword                = [redacted]          // Password Login
```

Figure 6: Authentication Credentials Disclosure in web HMI used in Mechanical Automation Devices

File inclusion vulnerabilities can be local or remote and is not restricted to PHP processing engines. If the web HMI is using a JavaScript wrapper for file inclusion, it is still treated as file inclusion because the attacker can still include files from remote location to execute code on the client side. Figure 2 shows an example of including HTML file from remote location in one web HMI used for PLCs.

Legitimate URL:
http://<web_hmi_ip_address>/process.php?url=google.com

Exploit URL:
http://<web_hmi_ip_address>/process.php?url=http://www.malicious_code.com/exploit.php

Listing 1: Remote File Inclusion Pseudo Code

Weak Cryptographic Mechanisms

Research has revealed a number of web HMIs do not deploy strong cryptographic mechanisms to secure the communication channel between client and the server. This produces three related vulnerabilities:

Web HMIs are running over HTTP and passwords are simply hashed while transmitted at network level. This makes the web HMIs vulnerable to "password replay" attacks in which attackers sitting in the network can easily sniff the network communication with the SCADA systems and extract the hashes to replay them to obtain access to the web HMI. Password hashing over HTTP alters the structure of passwords but does not prevent against authentication attacks through replay. Even the password retrieval is easy in vulnerable SCADA systems because the hashes are generated without salts which makes them almost equal to plain text data.

MD5 is used as a hashing algorithm in variety of flavors. MD5 is prone to collision attacks [15] and has already been deprecated. A number of the default SSL certificates shipped with HMI components use MD5 for hashing.

Vulnerable SCADA web HMIs transmit passwords over network in simple base64 encoded format over non-HTTPS channels. It is very easy to decode the base64 format.

Figure 3 shows MD5 of the password being transmitted over non-HTTPS channel to one of the web SCADA HMI

Injection flaws

Injection flaws exist in SCADA web HMIs because the HMI software fails to verify and validate the input provided by the end-user. Since the web HMI has not scrutinized the input provided by the client, it treats the code as legitimate and executes it directly in the application via a browser. Injection flaws include Cross-site Scripting (XSS), SQL Injection, and many others. The XSS vulnerabilities [16] are more prevalent than other injection methods. XSS vulnerabilities allow attackers to execute JavaScripts from third-party domains, either to steal critical information or download malware onto the end-user system by exploiting the injection flaw in web HMIs. Figure 4 highlights a XSS vulnerability discovered in one of the SCADA web HMI component used in Industrial automation systems.

Session Riding: No Request Tokens

Cross-site Request Forgery (CSRF) [17] vulnerabilities are also prevalent in SCADA web HMIs. Web HMIs are unable to determine and verify the requests originated from end-user clients because web HMIs are not designed to implement security tokens with every HTTP request initiated by the client software (browser). It shows web HMIs are not designed securely to prevent CSRF attacks since they do not verify the origin of the requests. If web HMIs implement tokens in HTTP requests, it becomes easy for the HMI component to check for the tokens before processing the request. With this design, attackers can target end-users who are running active sessions with the web HMIs.

To exploit CSRF vulnerability, the attacker has to craft a HTTP request and force the end-users to visit a web page that issues the same HTTP request without the user having any

knowledge of it. Since the user has an active session with the web HMI, the requests get accepted and associated commands get executed on the web HMIs. If the web HMI does not ask for old password in the "change password" component of the application and does not implement any tokens, the attacker can embed the exploit in the webpage and convince the user to visit that webpage through link sharing. When the user clicks the link, the browser sends a request to the web HMI and the password is changed to the attacker's choice. After that, an attacker can control the web HMI. Additionally, different insecure web HMI functionalities can be targeted by attackers to execute unauthorized commands by tricking users. Cross-site File Uploading (CSFU) is another variant of CSRF in which attackers can upload files to web HMIs in unauthorized manner. Figure 5 shows an example of CSFU when tokens are used by web HMIs.

Inherent Design Flaws

Insecure design issues also create security problems because certain parts of the software are implemented in an obsolete fashion that indirectly impact the security of the web HMIs. Research revealed a number of insecure design issues:

- Insecure ActiveX controls is one of major problems common to web HMIs. ActiveX is based on Microsoft proprietary technologies like the Component Object Model (COM) and Object Linking Environment (OLE) to download content from remote locations on the Internet. Earlier web HMIs use ActiveX with the Internet Explorer (IE) browser to perform data download operations from SCADA controllers; a number of systems still use it. A number of vulnerabilities [18, 19] have been disclosed by security researchers that show how ActiveX can be exploited in web HMIs.
- A number of of Java clients used by web HMIs do not support HTTPS, which means the client software is never able to establish HTTPS channel of communication to encrypt the communication.
- Sensitive credentials are transmitted using HTTP GET requests, which is considered a very weak authentication design because HTTP GET requests are easily cached by intermediate proxies and all the data is present in the server log files. This allows privacy and information leakage.
- Use of obsolete libraries while developing SCADA web HMIs mean inherent vulnerabilities are carried forward into the final product. Sometimes, vendors take a long time to update libraries, which again highlights the issue of reducing the time to deploy patches.
- Information disclosure through unrestricted resources is another observed insecure design issue. It is possible to access certain subcomponents of the web HMIs without any authentication which reveals a plethora of information about the device and how it is performing.
- A number of web HMIs disclose unwanted information in the client side code such as JavaScripts and caches which potentially results in information leakage and reveal internal details about the web HMIs. Figure 6 shows authentication credentials are disclosed in the JavaScript file on the client side.

In this section, we have discussed and highlighted the most frequent seen vulnerabilities in web HMIs.

Countermeasures

To combat threats against SCADA systems, several factors should be taken into consideration to restrict or minimize the impact of threats. Several recommendations are outlined below:

- Organizations should know about the nature of existing threats and associated impacts. Threat intelligence sharing among public and private organizations, including government agencies, is vital. Having evidence-based knowledge about threats upfront helps organizations make necessary changes and required updates in existing security solutions to avoid threats by simply harnessing the power of exchanged information as a part of threat intelligence. This strategy help organizations restrict the impact of threats, thereby avoiding business and brand damages. Using shared intelligence, organizations will spend less time in self-generating Indicators of Compromise (IOCs) and related content and make the shared information directly applicable and actionable in the organizational environment.
- SCADA vendors have to play significant roles in making SCADA systems more secure. When these systems were designed, there was no concept of advanced security. Thus, SCADA systems have become veritable vulnerability goldmines. SCADA vendors should do the following
  - Vendors need to opt into the process of Secure Development Lifecycle (SDL) to design upcoming SCADA products with robust security. Firmware web HMIs should be designed by taking threat models into consideration. This helps vendors understand how the SCADA system components can be exploited by attackers. Vulnerabilities disclosed earlier can be used as baselines to make SCADA components free from vulnerabilities.
  - Vendors must reduce the time window for patching vulnerabilities. It should not take five to six months for vendors to eradicate vulnerabilities. Companies are becoming proactive in handling security vulnerabilities, but more efforts are required to reduce the window of exposure.
  - SCADA vendors should implement bug-bounties, which uses online researchers to hunt vulnerabilities in products and disclose them under responsible guidelines. Researchers will get paid for finding the vulnerabilities. This initiative can be fruitful because SCADA software or components can be tested by a number of researchers with different threat models.
  - SCADA vendors also need to work in collaboration with government agencies such as Industrial Control Systems (ICS) Computer Emergency Response Team (CERT) to set up strong channels of communication to address reported vulnerabilities. A number of SCADA vendors are already working in collaboration with ICS-CERT, but many are not. This should be globally acceptable process for all SCADA vendors to ensure security of SCADA environments.
- Traditional security solutions such as Intrusion Prevention Systems (IPS), Intrusion Detection Systems (IDS), firewalls, SIEMs, etc. are already deployed as security anchors in the majority of SCADA organizations. However, with the advanced tactics used by attackers and rising threats from malicious insiders, it has become necessary to also look for potential anomalies in users' behaviors. Traditional security solutions are not equipped to provide this intelligence. Next generation security

solutions that use data mining and machine learning approaches to provide more context around the incident are required. A solution that has the capability to highlight the anomalies in SCADA systems users' behaviors can be very fruitful. Malicious communication can be marked as anomalous because malicious activities show deviation from normal profiling. That helps organizations to determine possible root causes so that actions can be taken upfront to secure against those risks.

Security training and insidious penetration testing exercises, which includes techniques and tactics to be followed by employees while encountering certain types of threat have become the de facto methods of imparting education to employees. Security training explains how attackers exploit users' understanding of certain software and trigger social engineering attacks. Penetration testing conducted in a controlled manner by security organizations reveals the different tactics used by attackers. This helps employees understand how attacks are carried and how they can prevent themselves from being exploited. Organizations are conducting active phishing attacks as a part of penetration testing exercises to determine the level of understanding their employees have and how many would fall for an actual attack. This proactive step is essential to minimize the risks associated with malicious spear phishing attacks. Employees managing SCADA systems should undergo rigorous training to combat threats upfront. Having proper security training will make employees think twice before inserting unknown USBs or personal USBs into SCADA mainframe computers.

Conclusion

This article discussed the existing state of SCADA HMI security, the common types of vulnerabilities, and how attackers exploit them. SCADA vendors are encountering myriad security issues in current SCADA systems and proactively patching issues on a regular basis. SCADA HMIs are not well designed and secured. As a result, the attackers can easily subvert the integrity of SCADA systems by simply exploiting design flaws and inherent vulnerabilities to gain access. SCADA vendors need to incorporate SDL and several security processes to make sure SCADA software is free from insecure code. It should be understood clearly that security is a process and should be followed during every design and release of the code (firmware). Vendors also need to work on reducing the time to patch reported vulnerabilities. Government agencies such as ICS-CERT are playing a significant role in securing SCADA infrastructure across globe by working as an intermediary between SCADA vendors and researchers, but more support and research is needed from the security community so that every SCADA component can be made secure, robust and non-exploitable.

Acknowledgement:

I would like to thank DEFCON reviewers for selecting the research to be presented at DEFCON 23. I would also like to thank Industrial Control Systems (ICS) Computer Emergency Response Team (CERT) for working with vendors to provide support in patching the vulnerabilities highlighted in this research.

ABOUT THE AUTHOR

Dr. Aditya K. Sood is a cyber-security expert whose research focuses on malware automation and analysis, application security, secure software design, and cybercrime. At present he works as Director of Security and Cloud Threat Labs, Elastica, Blue Coat Systems. His work has been featured in several media outlets including the Associated Press, Fox News, etc. He has been an active speaker at industry conferences and presented at BlackHat, DEFCON, and many others. Dr. Sood obtained his Ph.D. from Michigan State University in Computer Sciences. He is also the author of the "Targeted Cyber Attacks" book.

REFERENCES

1. DHS, Critical Infrastructure Sectors, http://www.dhs.gov/critical-infrastructure-sectors
2. A. Greenberg, What Stuxnet's Exposure As An American Weapon Means For Cyberwar, Forbes, http://www.forbes.com/sites/andygreenberg/2012/06/01/what-stuxnets-exposure-as-an-american-weapon-means-for-cyberwar/
3. M. Asante, America's Critical Infrastructure Is Vulnerable To Cyber Attacks, Forbes, http://www.forbes.com/sites/realspin/2014/11/11/americas-critical-infrastructure-is-vulnerable-to-cyber-attacks/
4. T. Chee-Wooi, L. Chen-Ching, and G. Manimaran, "Vulnerability Assessment of Cybersecurity for SCADA Systems," in Power Systems, IEEE Transactions on , vol.23, no.4, pp.1836-1846, Nov. 2008, doi: 10.1109/TPWRS.2008.2002298
5. T. Chee-Wooi, L. Chen-Ching, and G. Manimaran, "Vulnerability Assessment of Cybersecurity for SCADA Systems Using Attack Trees," in Power Engineering Society General Meeting, 2007. IEEE , vol., no., pp.1-8, 24-28 June 2007, doi: 10.1109/PES.2007.385876
6. A. Shahzad, N. Xiong, M. Irfan, M. Lee, S. Hussain, and B. Khaltar, "A SCADA intermediate simulation platform to enhance the system security," in Advanced Communication Technology (ICACT), 2015 17th International Conference on , vol., no., pp.368-373, 1-3 July 2015
7. C. Wang, L. Fang, and Y. Dai, "A Simulation Environment for SCADA Security Analysis and Assessment," in Measuring Technology and Mechatronics Automation (ICMTMA), 2010 International Conference on , vol.1, no., pp.342-347, 13-14 March 2010
8. K. Sung-Hwan, E. Jung-Ho, and C. Tai-Myoung, "A study on optimization of security function for reducing vulnerabilities in SCADA," in Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2012 International Conference on , vol., no., pp.65-69, 26-28 June 2012, doi: 10.1109/CyberSec.2012.6246099
9. ICS-CERT, Schneider Electric Modicon PLC Vulnerabilities, https://ics-cert.us-cert.gov/advisories/ICSA-15-246-02
10. ICS-CERT, Clorius Controls A/S ISC SCADA Insecure Java Client Web Authentication, https://ics-cert.us-cert.gov/advisories/ICSA-15-013-02
11. ICS-CERT, Schneider Electric SCADA Expert ClearSCADA Vulnerabilities (Update A), https://ics-cert.us-cert.gov/advisories/ICSA-14-259-01A
12. ICS-CERT, Prisma Web Vulnerabilities, https://ics-cert.us-cert.gov/alerts/ICS-ALERT-15-224-03
13. ICS-CERT, Rockwell Automation 1766-L32 Series Vulnerability (Update A), https://ics-cert.us-cert.gov/alerts/ICS-ALERT-15-225-02A
14. ICS-CERT, Rockwell Automation 1769-L18ER and A LOGIX5318ER Vulnerability (Update A), https://ics-cert.us-cert.gov/alerts/ICS-ALERT-15-225-01A
15. E. Thompson, MD5 collisions and the impact on computer forensics. Digit. Investig. 2, 1 (February 2005)
16. D. Bates, A. Barth, and C. Jackson, Regular expressions considered harmful in client-side XSS filters, In Proceedings of the 19th international conference on World wide web(WWW '10). ACM, New York, NY, USA
17. R. Pelizzi and R. Sekar, A server- and browser-transparent CSRF defense for web 2.0 applications. In Proceedings of the 27th Annual Computer Security Applications Conference(ACSAC '11). ACM, New York, NY, USA, 257-266.
18. ICS-CERT, Mitsubishi Electric Automation MC-WorX Suite Unsecure ActiveX Control, https://ics-cert.us-cert.gov/advisories/ICSA-14-051-02
19. ICS-CERT, WellinTech KingView ActiveX Vulnerabilities, https://ics-cert.us-cert.gov/alerts/ICS-ALERT-13-256-01

DevOps Advantages for Testing

Increasing Quality through Continuous Delivery

Gene Gotimer, Coveros
Thomas Stiehm, Coveros

Abstract. DevOps and continuous delivery can improve software quality and reduce risk by offering opportunities for testing and some non-obvious benefits to the software development cycle. By taking advantage of cloud computing and automated deployment, throughput can be improved while increasing the amount of testing and ensuring high quality. This article points out some of these opportunities and offers suggestions for making the most of them.

DevOps

DevOps is a software development culture that stresses collaboration and integration between software developers, operations personnel, and everyone involved in the design, creation, development, and delivery of software. It is based on the same principles that were identified in the Agile Manifesto [1], but while many agile methodologies focus on development only, DevOps extends the culture to the entire software development lifecycle.

Central to DevOps is continuous delivery: delivering software often, possibly multiple times each day, using a delivery pipeline through testing stages that build confidence that the software is a viable candidate for deployment. Continuous delivery (CD) is heavily dependent on automation: automated builds, testing, and deployments. In fact, the reliance on automated deployment is so key that DevOps and CD are often erroneously considered synonymous with automated deployment.

Having a successful delivery pipeline means more than just adding automation. To be effective, tests of all types must be incorporated throughout the process in order to ensure problems aren't slipping through. Those tests include quality checks, functional testing, security tests, performance assessments, and any other type of testing you require before releasing your software.

The delivery pipeline also opens up opportunities to add more testing. Static analysis tools can review code style and test for simple security errors. Automated deployments allow automated functional testing, security tests of the software system as deployed, and performance testing on production-like servers. Continuity of operations (COOP) plans can be tested every time that the infrastructure changes, not just annually in front of the auditors.

With this additional testing, CD can produce software that has fewer defects, can be deployed more reliably, and can be delivered far more confidently than traditional methodologies. Escaped defect rates drop, teams experience lower stress, and delivery is driven by business need. The benefits aren't just slight improvements. In fact, a 2015 report on DevOps from Puppet Labs found that teams using DevOps experience "60 times fewer failures and recover from failures 168 times faster than their lower-performing peers. They also deploy 30 times more frequently with 200 times shorter lead times [2]."

The choices of tools and frameworks for all of this automation has grown dramatically in recent years, with options available for almost any operating system, any programming language, open source or commercial, hosted or as-a-service. Active communities surround many of these tools, making it easy to find help to start using them and to resolve issues.

Continuous Integration

Building a CD process starts with building a Continuous Integration (CI) process. In CI developers frequently integrate other developer's code changes, often multiple times a day. The integrated code is committed to source control then automatically built and unit tested. Developers get into the rhythm of a rapid "edit-compile-test" feedback loop. Integration errors are discovered quickly, usually within minutes or hours of the integration being performed, while the changes are fresh on the developer's minds.

A CI engine, such as Jenkins [3], is often used to schedule and fire off automated builds, tests, and other tasks every time code is committed. The automated build for each commit makes it virtually impossible for compilation errors and source code integration errors to escape unnoticed. Following the build with unit tests means the developers can have confidence the code works the way they intended, and it reduces the chance that changes had unintended side effects.

Important: Continuous integration is crucial in providing a rapid feedback loop to catch integration issues and unintended side effects.

The choice of CI engine is usually driven by the ecosystem you are working in. Common choices include Jenkins for Linux environments and Team Foundation Server [4] for Windows environments.

Code Coverage

CI can also tie-in code coverage tools that measure the amount of code that is executed when the unit tests are run.

Code coverage can be a good guide as to how well the code is unit tested, which in turn tells you how easy it should be to reorganize the code and to change the inner workings without changing the external behavior, a process known as refactoring [5]. Refactoring is an important part of many agile development methodologies, such as extreme programming (XP) [6] and test-driven development (TDD) [7].

In TDD, a test is written to define the desired behavior of a unit of code, which could be a method or a class. The test will naturally fail, since the code that implements the behavior is not yet written. Next, the code is implemented until the test passes. Then, the code is refactored by changing it in small, deliberate steps, rerunning the tests after each change to make sure that the external behavior is unchanged. Another test is written to further define the behavior, and the "test-implement-refactor" cycle repeats.

By definition, code behavior does not change during refactoring. If inputs or outputs must change, that is not refactoring. In those cases, the tests will necessarily change as well. They must be maintained along with, and in the same way as, other source code.

Without sufficient code coverage you cannot be sure that behavior is unchanged. A change in the untested code may have an unintended effect elsewhere. Having enough unit testing and code coverage means you are free to do fearless refactoring: you can change the design and implementation of the software without worrying something will break inadvertently. As the software evolves and you learn more about how the software should have been written you can go back and make changes rather than living with early decisions. In turn, you can move faster at the beginning by "doing the simplest thing that could possibly work [8]" rather than agonizing over every decision to (impossibly) make sure it will address all future needs, known and unknown.

Important: Unit testing and code coverage is about more than just testing. It also enables fearless refactoring and the ability to revisit design and implementation decisions as you learn more.

Code coverage tools are usually programming language-dependent. JaCoCo [9] is an excellent open-source choice for Java, Coverage.py [10] for Python, and NCover [11] is a popular commercial tool for .NET. Every popular programming language today is likely to have several code coverage tool options.

Mutation Testing

Code coverage can't tell the whole story. It only counts how many lines (or methods, or classes, etc.) are executed when the unit tests run, not whether that code is tested well, or at all.

Mutation testing [12] is a process by which existing code is modified in specific ways (e.g., reversing a conditional test from equals to not equals, or flipping a true value to false) and then the unit tests are run again. If the changed code, or mutation, does not cause a test to fail, then it survives. That means the unit tests did not properly test the condition. Even though code coverage may have indicated a method was completely covered, it might not have been completely tested.

Mutation testing generally runs many times slower than unit tests. But if it can be done automatically then the cost of running the mutation tests is only time it takes to review the results. Successful mutation testing leads to higher confidence in unit tests, which leads to even more fearless refactoring.

Suggestion: Use mutation testing tools to determine how effective your unit tests are at detecting code problems.

Tools for mutation testing are available for various programming languages and unit test frameworks. Two mature tools are PIT Mutation Testing [13] for Java and Ninja Turtles [14] for .NET. Humbug [15] is a popular choice for PHP and many options exist for Python [16].

Static Analysis

Static analysis tools are easy to use via the CI engine. These tools handle many of the common tasks of code review, looking at coding style issues such as variable and method-naming conventions. They can also identify duplicate code blocks, possible coding issues (e.g., declared but unused variables), and confusing coding practices (e.g., too many nested if-then-else statements). Having these mundane items reviewed automatically can make manual code reviews much more useful since they can focus on design issues and implementation choices. Since the automated reviews are objective, the coding style can be agreed upon and simply enforced by software.

Important: Static analysis can allow manual code reviews to concentrate on important design and implementation issues, rather than enforcing stylistic coding standards.

Static analysis tools can also identify some serious problems. Race conditions, where parallel code execution can lead to deadlocks or unintended behavior, can be difficult to identify via testing or manual code review, but they can often be detected via static analysis. SQL and other injection vulnerabilities can also be identified, as can resource leaks (e.g., file handle opened but not closed) and memory corruption (e.g., use after free, dangling pointers).

Since static analysis tools can be fast and can easily run automatically as part of the edit-compile-test cycle, they can be used as a first line of defense against coding errors that can lead to serious security and quality issues.

Important: Static analysis tools can provide early detection of some serious code issues as part of the rapid CI feedback cycle.

Every popular programming language has a selection of static analysis tools -- many of them open source. But even easier than choosing one or more and integrating them with your build process or CI engine is installing the excellent open-source tool known as SonarQube [17]. It integrates various analyses for multiple programming languages and displays the combined results in an easy-to-use quality dashboard that tracks trends, identifies problem areas, and can even fail the build when results are beyond project-defined thresholds.

Delivery Pipeline

The delivery pipeline describes the process of taking a code change from a developer and getting it delivered to the customer or deployed into production. CD generally evolves by extending the CI process and adding automated deployment and testing. The delivery pipeline is optimized to remove as many manual delays and steps as practical. The decision to deploy or deliver software becomes a business decision rather than being driven by technical constraints.

The delivery pipeline is often described as a series of triggers: actions such as code being checked into the source control system, that initiate one or more rounds of tests, known as quality gates. If the quality gate is passed, that triggers more processes, which lead to more quality gates. If a quality gate is not passed, the build is not a viable candidate for production, and no further testing is done. The problems that were discovered are fixed and the delivery pipeline begins again.

The delivery pipeline should be arranged so the earliest tests are the quickest and easiest to run and give the fastest feedback. Subsequent quality gates lead to higher confidence that the code is a viable candidate and they indicate more

expensive tests (in regards to time, effort, or cost) are justified. Manual tests migrate towards the end of the pipeline, leaving computers to do as much work as possible before humans have to get involved. Computers are significantly cheaper than people and humans often work slower than computers. They get sidetracked, go to meetings, and don't work around the clock.

The CI process is often the first stage of the delivery pipeline, being the fastest feedback cycle. Often the CI process is blocking: a developer will wait until the quality gate is passed before continuing. Quality gates later in the pipeline are non-blocking: work continues while the quality checks are underway.

While it can be tempting to arrange the delivery pipeline in phases (e.g., unit testing, then functional tests, then acceptance tests, then load and performance tests, then security tests), this leaves the process susceptible to allowing serious problems to progress far down the pipeline, leading to wasted time testing a non-viable candidate for release and extending the time between making a change and identifying any problems. Instead, quality gates should be arranged so each one does enough testing to give confidence the next set of tests is worth doing.

For example, after some functional tests, a quick performance test might be valuable to make sure a change hasn't rendered the software significantly slower. Next, a short security check could be done to make sure some easily detectable security issue hasn't been introduced. Then a full set of regression tests could be run. Later, you could run more security tests along with load and performance testing. Each quality gate has just enough testing to give us confidence the next set of tests is worth doing.

Suggestion: Do just enough of each type of testing early in the pipeline to determine if further testing is justified.

Negative Testing

The first tests written are almost always sunny-day scenarios: does the software do what it was intended to do? We should also make sure there are functions that the software doesn't do: rainy-day scenarios. For example, one user shouldn't be able to look at another user's private data. Bad input data should result in an error message. A consumer should not be able to buy an item if they do not pay. A web-user should not be able to access protected content without logging in. Whenever you identify sunny-day tests, you should also identify related rainy-day tests.

Identifying these conditions while features are being developed will lead to more tests, which will help build more confidence that new features aren't inadvertently introducing security holes. The tests will form a body of regression tests that document how the software is intended to work and not to work. As the code gets more complex, you will be able to fearlessly refactor knowing that you are not introducing unintended side effects.

Important: Sunny-day testing is important, but rainy-day testing can be just as important for regression and security. You need to test both to be confident the code is working correctly.

Automated deployment

Some types of testing aren't valuable until the code is compiled, deployed, and run in a production-like environment. Security scans might depend on the web server configuration. Load and performance tests might need production-sized systems. If deployment is time consuming, error prone, or even just frustrating, it won't be done frequently. That means you won't have as many opportunities to test deployed code.

While an easy, quick, reliable manual install makes it easier to deploy more often, having an automated process can make deployments almost free, especially when deployments can be triggered automatically by passing quality gates. That lets the delivery pipeline progress without human interaction. When there are fewer barriers to deploying, the team will realize there are more chances to exercise the deployment process. When combined with the flexibility of cloud computing resources, deployments will become a regular course of action rather than a step taken only late in the development cycle.

Important: Automated deployments will be used more often than simple manual deployments. They will be tested more often and the delivery pipeline will find more uses for them.

Configuration management tools that perform automated deployments are a class of tool that has garnered a lot of attention in recent years, and many excellent tools, frameworks, and platforms are readily available, both commercially and open source. Puppet [18], Chef [19], and Ansible [20] lead the pack with open-source products that can be coupled with commercial enterprise management systems. Active ecosystems have evolved around each of them with plenty of community support.

Using automated deployments more often gives you more chances to validate that your deployment process works. You can't afford to hope that it works because it runs; you have to verify that it successfully deployed and configured your system or systems using an automated verification process. It has to be quick, so you can afford to run it on each deployment. It should test the deployment, not the application functionality, so focus on the interfaces between systems (e.g., IP addresses and firewalls), configuration properties (e.g., database connection settings), and basic signs of life (e.g., is the application responding). Repeatedly deploying to different environments and then verifying the deployment works gives you higher confidence it will work when deploying to production, which is the deployment that really counts.

Suggestion: Each deployment should be followed with an automated deployment verification suite. Make the deployment verification reusable, so the same checks and tests can be used after each deployment, no matter which environment.

Deployment verification checks can usually be automated using the same tool you use for functional and regression testing. If that tool is too heavyweight or can't be easily integrated into the pipeline, consider a lightweight functional testing framework like Selenium [21] and/or one of the xUnit test frameworks [22], such as JUnit [23] for Java or nUnit [24] for .NET.

Exploratory Testing

Manual exploratory testing is not made obsolete by adopting automated testing. Manual testing becomes more important since automated tests will cover the easy things, leaving the more obscure problems undiscovered. Testers will need increasing amounts of creativity and insight to detect these issues, traits almost impossible to build into automation. The very term exploratory testing highlights the undefined nature of the testing. Automated tests will never adapt to find issues they aren't testing for. This is known as the paradox of automation. "The more efficient the automated system, the more crucial the human contribution [25]."

The delivery pipeline does not have to be an unstopping conveyor belt of releases. Human testers cannot cope with a constant stream of new releases. They cannot deal with the software changing mid-test or even mid-test cycle. Even when they find one problem, there is value in continuing their tests to see if the same problem exists in related functions, or looking for unrelated issues in other parts of the code. There is a balance to make sure time isn't invested testing a non-viable candidate from production and restarting a test suite to fix every little problem individually.

Waiting for human testers to be ready to start a new test cycle slows down the rest of the pipeline. In order to incorporate their testing and not constantly interrupt their test cycles as new versions of the software are made available, consider on-demand deployments, where the pipeline does not deploy to the exploratory testing environment until the testers choose it to be deployed. Or perhaps the software is deployed automatically to a new dynamic environment each time it is packaged, and the testers move on to the most recent (or most important, or most promising) environment. In this way, there is always an environment available for the testers to use without pulling the rug out from under them during their test cycle, thereby buffering the bottleneck [26].

While you want to reduce the time testers spend testing a build that is not viable, you also don't want to start so late as to be a constraint for other activities. Consider running the exploratory testing in parallel with other automated and non-automated tasks, minimizing the wait by placing it at the end of the cycle rather than the start. Think about time boxing (defining and enforcing a fixed duration) the test cycle.

Suggestion: Deployments for manual testing must be coordinated so testers can have a stable environment. Consider on-demand deployments, and make sure the pipeline is only waiting at the end of manual testing, not the beginning.

Parallel Testing

Just as with the exploratory testing, other long-running tests should be run in parallel to make progress while waiting for longer tests to complete. Taking advantage of automated deployments, multiple environments can be built so some tests can be done at the same time using different resources. This can mean doing multiple types of tests at one time, or breaking one type of tests into smaller chunks that can be handled in parallel.

Often four one-day-long tasks are preferable to one four-day-long task because the shorter tasks give additional opportunities for feedback. The fourth day might not need to be needed if there is a show-stopper identified on day three. In parallel, those tests might be run in two parallel tracks, taking a total of two days only. Or perhaps a two-day stress test can be undertaken in parallel with a two-to-three day security scan, to reduce the effect of the bottleneck.

Suggestion: Long-running testing should be done in parallel as much as practical, so that you don't have to wait days or weeks for individual test phases to be completed in sequence.

Infrastructure

Development teams need infrastructure to get their work done. Source code repositories, CI engines, test servers, certificate authorities, firewalls, and issue tracking systems are all examples of tools that might be required, but they are often not deliverables for the project.

Infrastructure doesn't stay static. Systems need to be moved or replicated. They get resized. Applications, tools, and operating systems get upgraded. Hardware goes bad. And other projects need to use the same or similar infrastructure. Setting up your infrastructure is never a one-time occurrence. Even though this infrastructure is internal-facing, it quickly becomes mission critical to the development team.

Treat it like you do production code. Automate the deployment so that redeploying is as easy as pushing out a new version of the software you are writing. Use the same automated deployment tools since you already have experience and tools to support them.

Suggestion: Use your familiarity with the automated deployment tools to automate your infrastructure deployments as well. Treat automated deployment code and infrastructure as mission critical.

Case Study – Forge.mil

DISA's Forge.mil supports collaborative development for the DoD. It is built using commercial off-the-shelf software coupled with open-source tools and custom integration code, written in a variety of programming languages (e.g., Java, PHP, Python, Perl, Puppet). The team used agile techniques from the beginning in order to maximize throughput for the small team doing the integration and development work. The project also served as an exemplar project to demonstrate and document how agile techniques could be used within DoD projects.

An early focus on continuous integration led the team to identify several bottlenecks in the delivery process. Functional testing was manual, slow, and hard to do comprehensively. Development, test, and integration environments were all configured differently from each other and different than production. Deployments were manual, long, complicated, and unreliable. Security patches were often applied directly into production with limited testing, almost always in response to information assurance vulnerability alerts (IAVAs). A team of about two dozen developers, testers, integrators, managers, and others were delivering software to production once every six months. A software release was a big, scary event, carefully planned and scheduled

weeks in advance by the entire team. Problems were identified in the days after each release (often by end users), carefully triaged, with hot fixes deployed or workarounds documented.

The team focused on removing some of these bottlenecks, concentrating on improved functional and regression testing. After discovering the book Continuous Delivery by Jez Humble and Dave Farley [27], they began using Puppet scripts for configuration management which greatly improved the reliability of production deployments. Consistent, production-like deployments in other environments could be performed on-demand in minutes, many times a week. Proactive security testing and vulnerability patching became convenient and did not disrupt other development and testing activities. The bottlenecks the team had identified earlier were eliminated or greatly reduced, one-by-one.

Over time, the team size decreased to less than a dozen people. Software was confidently deployed to production every two weeks with neither drama nor concern. Full regression tests, performance tests, and security tests were regular occurrences multiple times a week. Security patches were incorporated into the normal release cycle, often being fully tested and deployed to production before the IAVAs were even issued. Reports of issues after releases (aka escaped defects) disappeared almost completely. Software releases were driven by business needs and the project management office, not by technical limitations and risks identified by the developers, testers, and integrators.

More details are available in Continuous Delivery in a Legacy Shop - One Step at a Time [28], originally presented at DevOps Conference East 2015 in Orlando, Florida.

Conclusion

The journey towards a continuous delivery practice relies heavily on quality tests to show if the software is (or is not) a viable candidate for production. But along with the increased reliance on testing, there are many opportunities for performing additional tests and additional types of tests to help build confidence in the software. By taking advantage of the automated tests and automated deployments, the quality of the software can be evaluated and verified more often and more completely. By arranging the least expensive tests (in terms of time, resources, and/or effort) first, a rapid feedback loop creates openings to fix issues sooner and focus more expensive testing efforts on software that you have more confidence in. By having a better understanding of the software quality, the business can make more informed decisions about releasing the software, which is ultimately one of the primary goals of DevOps.

FURTHER READING

1. Humble, Jez, and David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River, NJ: Addison-Wesley, 2011. Print.
2. Kim, Gene, Kevin Behr, and George Spafford. The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win. Portland, Oregon: IT Revolution, 2013. Print.
3. Duvall, Paul M., Steve Matyas, and Andrew Glover. Continuous Integration: Improving Software Quality and Reducing Risk. Upper Saddle River, NJ: Addison-Wesley, 2007. Print.
4. Fowler, Martin, and Kent Beck. Refactoring: Improving the Design of Existing Code. Reading, MA: Addison-Wesley, 1999. Print.

ABOUT THE AUTHORS

Gene Gotimer is a proven senior software architect with many years of experience in web-based enterprise application design, most recently using Java. He is skilled in agile software development as well as legacy development methodologies, with extensive experience establishing and using development ecosystems including: continuous integration, continuous delivery, DevOps, secure software development, source code control, build management, release management, issue tracking, project planning & tracking, and a variety of software assurance tools and supporting processes.

Tom Stiehm has been developing applications and managing software development teams for twenty years. As CTO of Coveros, he is responsible for the oversight of all technical projects and integrating new technologies and application security practices into software development projects. Most recently, Thomas has been focusing on how to incorporate DevOps best practices into distributed agile development projects using cloud-based solutions and how to achieve a balance between team productivity and cost while mitigating project risks. Previously, as a managing architect at Digital Focus, Thomas was involved in agile development and found that agile is the only development methodology that makes the business reality of constant change central to the development process.

REFERENCES

1. Beck, Kent, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. "Principles behind the Agile Manifesto." Agile Manifesto. N.p., 2001. Web. 10 Dec. 2015. <http://agilemanifesto.org/principles.html/>.
2. Puppet Labs 2015 State of DevOps Report. Rep. PwC US, 22 July 2015. Web. 10 Dec. 2015. <https://puppetlabs.com/2015-devops-report>
3. "Welcome to Jenkins CI!" Jenkins CI. CloudBees, n.d. Web. 17 Dec. 2015. <https://jenkins-ci.org/>.
4. "Team Foundation Server." Team Foundation Server. Microsoft, n.d. Web. 19 Jan. 2016. <https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx>.
5. Fowler, Martin, and Kent Beck. Refactoring: Improving the Design of Existing Code. 1st ed. Reading, MA: Addison-Wesley, 1999. Print.
6. "Extreme Programming." Wikipedia. Wikimedia Foundation, n.d. Web. 21 Jan. 2016. <https://en.wikipedia.org/wiki/Extreme_programming>.
7. "Test-driven Development." Wikipedia. Wikimedia Foundation, n.d. Web. 21 Jan. 2016. <https://en.wikipedia.org/wiki/Test-driven_development>.
8. "Do The Simplest Thing That Could Possibly Work." Do The Simplest Thing That Could Possibly Work. Cunningham & Cunningham, Inc., n.d. Web. 12 Dec. 2015. <http://c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork>
9. "JaCoCo Java Code Coverage Library." EclEmma. N.p., n.d. Web. 19 Jan. 2016. <http://eclemma.org/jacoco/>.
10. "Coverage." Python Package Index. N.p., n.d. Web. 19 Jan. 2016. <https://pypi.python.org/pypi/coverage>.
11. "NCover | .NET Code Coverage for .NET Developers." NCover. Gnoso, n.d. Web. 19 Jan. 2016. <http://www.ncover.com/>.
12. "Mutation testing." Wikipedia. Wikimedia Foundation, n.d. Web. 12 Dec. 2015. <https://en.wikipedia.org/wiki/Mutation_testing>
13. "Real World Mutation Testing." PIT Mutation Testing. N.p., n.d. Web. 19 Jan. 2016. <http://pitest.org/>.
14. "NinjaTurtles - Mutation Testing for .NET (C#, VB.NET)." NinjaTurtles. N.p., n.d. Web. 10 June 2015. <http://www.mutation-testing.net/>.
15. "Humbug." GitHub. N.p., n.d. Web. 19 Jan. 2016. <https://github.com/padraic/humbug>.
16. "Index of Packages Matching 'mutationtesting'." Python Package Index. N.p., n.d. Web. 19 Jan. 2016. <https://pypi.python.org/pypi?%3Aaction=search&term=mutation%2Btesting&submit=search>.
17. "Put Your Technical Debt under Control." SonarQube™. SonarSource, n.d. Web. 20 Jan. 2016. <http://www.sonarqube.org/>.
18. "Open Source Puppet." Puppet Labs. Puppet Labs, n.d. Web. 20 Jan. 2016. <https://puppetlabs.com/puppet/puppet-open-source>.
19. "Chef." Chef. Chef Software, Inc., n.d. Web. 20 Jan. 2016. <https://www.chef.io/>.
20. "Ansible Is Simple IT Automation." Ansible. Ansible, Inc., n.d. Web. 20 Jan. 2016. <http://www.ansible.com/>.
21. "Selenium - Web Browser Automation." SeleniumHQ. N.p., n.d. Web. 20 Jan. 2016. <http://www.seleniumhq.org/>.
22. "XUnit." Wikipedia. Wikimedia Foundation, n.d. Web. 20 Jan. 2016. <https://en.wikipedia.org/wiki/XUnit>.
23. "JUnit." JUnit. N.p., n.d. Web. 20 Jan. 2016. <http://junit.org/>.
24. "NUnit." NUnit. N.p., n.d. Web. 20 Jan. 2016. <http://www.nunit.org/>.
25. "Automation." Wikipedia. Wikimedia Foundation, n.d. Web. 17 Dec. 2015. <https://en.wikipedia.org/wiki/Automation#Paradox_of_Automation>
26. Goldratt, Eliyahu M. "Theory of Constraints." Wikipedia. Wikimedia Foundation, n.d. Web. 17 Dec. 2015. <https://en.wikipedia.org/wiki/Theory_of_constraints>
27. Humble, Jez, and David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River, NJ: Addison-Wesley, 2011. Print.
28. Gotimer, Gene. "Continuous Delivery in a Legacy Shop - One Step at a Time." SlideShare. Coveros, Inc., 12 Nov. 2015. Web. 20 Jan. 2016. <http://www.slideshare.net/ggotimer/continuous-delivery-in-a-legacy-shop-one-step-at-a-time>.

They Know Your Weaknesses—Do You?: Reintroducing Common Weakness Enumeration

Yan Wu, Bowling Green State University
Yaacov Yesha, University of Maryland University College
Irena Bojanova, University of Maryland Baltimore County

Abstract. Knowing what makes your software systems vulnerable to attacks is critical, as software vulnerabilities hurt security, reliability, and availability of the system as a whole. The Common Weakness Enumeration (CWE), a community effort that provides the foundation for such knowledge, is not sufficient, accurate and precise enough to serve as the common language measuring stick and provide a common baseline for developers and security practitioners. In this article, we introduce the relevant body of knowledge that consolidates CWE, including the Semantic Template and Software Fault Pattern efforts, and how static analysis tools add value through CWEs. We also provide future directions, present our vision on CWE formalization, and discuss the value of CWE for not only software assurance community, but also for Computer Science.

1. Introduction to Common Weakness Enumeration (CWE)

Software weaknesses could be exploited to compromise a system's security. This is especially critical for systems such as the Department of Defense (DoD) systems, in which the amount of software is very large. Software assurance countermeasures should be applied to address anticipated attacks against a system. Such attacks are enabled by software vulnerabilities, and those countermeasures reduce those vulnerabilities or remove them [12].

Common Weakness Enumeration (CWE) [1] is a collection of software weakness descriptions that offers a way to identify and eliminate vulnerabilities in computer systems. CWE is also used to evaluate the tools and services developed for finding weaknesses in software. CWE is community-developed and maintained by MITRE Corporation [1].

A preliminary classification of vulnerabilities, attacks, and related concepts was developed by MITRE's CVE [2] team. That effort began in 2005., CWE was developed as a list of software weaknesses that is more suitable for software security assessment [14].

1.1 History of CWE

There have been several community efforts to leverage the existing large number of diverse real-world vulnerabilities. For example, an important step towards creating the needed collection of software weakness types was the establishment of the CVE (Common Vulnerabilities and Exposures) list [2] in 1999 by MITRE. Another important step from MITRE was creating the Preliminary List Of Vulnerability Examples for Researchers (PLOVER) in 2005. PLOVER includes more than 1,500 CVE names, and 290 types of software weaknesses. The organization of those vulnerabilities is based on the types of weaknesses among 290 types that cause each vulnerability [1].

The consolidation and evolution process of CWE [1] occurred during earlier efforts to classify vulnerabilities by answering three basic questions:

1. How did the vulnerability enter the system?
2. When did the vulnerability enter the system?
3. Where does the vulnerability appear? Or - Where is the vulnerability now?

Over a period of time, other revisions and ways to classify vulnerabilities were introduced. Until more recently, vulnerability categorizations have been developed as enumerations of weaknesses.

The CWE vision is to consolidate these efforts, and it is often compared to a "Kitchen Sink", although in a good way, as it aggregates many different taxonomies, software technologies and products, and categorization perspectives. While it provides a comprehensive record of software weaknesses, it can be a daunting task for developers to untangle the complex web of interdependencies that exist among software weaknesses captured in the CWE.

Figure 1 presents the CWE efforts context and community.

1.2 CWE Concepts

Common Weakness Enumeration (CWE) [1] is a collection of descriptions of software weakness types stored as .xml, .xsd and .pdf documents. There are four major types of CWE-IDs: 1) Category, 2) Compound Element, 3) View, and 4) Weakness. The weaknesses covered by CWE have weakness IDs. Category and Compound Element are aggregations of weaknesses. Category aggregates types of weaknesses, and Compound Element aggregates a group of several events that together can result in a successful attack. View IDs are "assigned to predefined perspectives with which one might look at the weaknesses in CWE." [1]

Information provided for CWEs includes:

- CWE Identifier Number/Name of the weakness type
- Description of the type
- Alternate terms for the weakness
- Description of the behavior of the weakness
- Description of the exploit of the weakness
- Likelihood of exploit for the weakness
- Description of the consequences of the exploit
- Potential mitigations
- Node relationship information
- Source taxonomies
- Code samples for the languages/architectures
- CVE Identifier numbers of vulnerabilities for which that type of weakness exists
- References [1].

2. CWE Related Practices

Around CWE, there is a list of relevant body of knowledge such as Common Weakness Scoring System (CWSS), Common Vulnerabilities and Exposures (CVE), and Common Attack Pattern Enumeration and Classification (CAPEC). They are utilized by many institutions, including DoD, to identify and mitigate the most dangerous types of vulnerabilities in the software [12]

2.1 Use of CWE

CWE was established for those who create software, analyze software for security flaws, and provide tools and services for finding and defending against security flaws in software [1]. The CWE Compatibility and Effectiveness Program is based on six requirements: 1) "CWE Searchable," 2) "CWE Output," 3) "Mapping Accuracy," 4) "CWE Documentation," 5) "CWE Coverage," and 6) "CWE Test Results."

Meeting the first four requirements is needed for a product or a service to be designated as "CWE Compatible," and meeting all six requirements is needed for a product or service to be designated as "CWE Effective." [1] Static analysis tools are also encouraged to map their reports to corresponding CWEs so that the results from different tools could have a standard baseline to be matched and compared.

2.2 Common Weakness Scoring System (CWSS)

The Common Weakness Scoring System (CWSS) [3] is included in CWE project. Numerically scoring software weaknesses is important, as both software developers and software consumers need to compare weaknesses in order to prioritize among various activities related to avoiding and eliminating them. CWSS enables such scoring by methods such as: Targeted, Generalized, Context-adjusted, and aggregated. CWSS 0.8 is based on the Targeted scoring method. This method is applicable to a particular package. The CWSS 0.8 scoring formula includes eighteen factors, which are divided into three groups: The Base Finding Group, the Attack Surface Group, and the Environmental Group.

2.3 Common Vulnerabilities and Exposures (CVE)

CVE is a dictionary of security vulnerabilities. It was established in 1999 in response to lack of standardization of names of vulnerabilities: different repositories could refer to the same vulnerability by a different name, resulting in difficulty in comparing software security tools.

CVE provides standard identifiers for security vulnerabilities [2], and help in finding information about a vulnerability, including ways of, and available products for, eliminating the

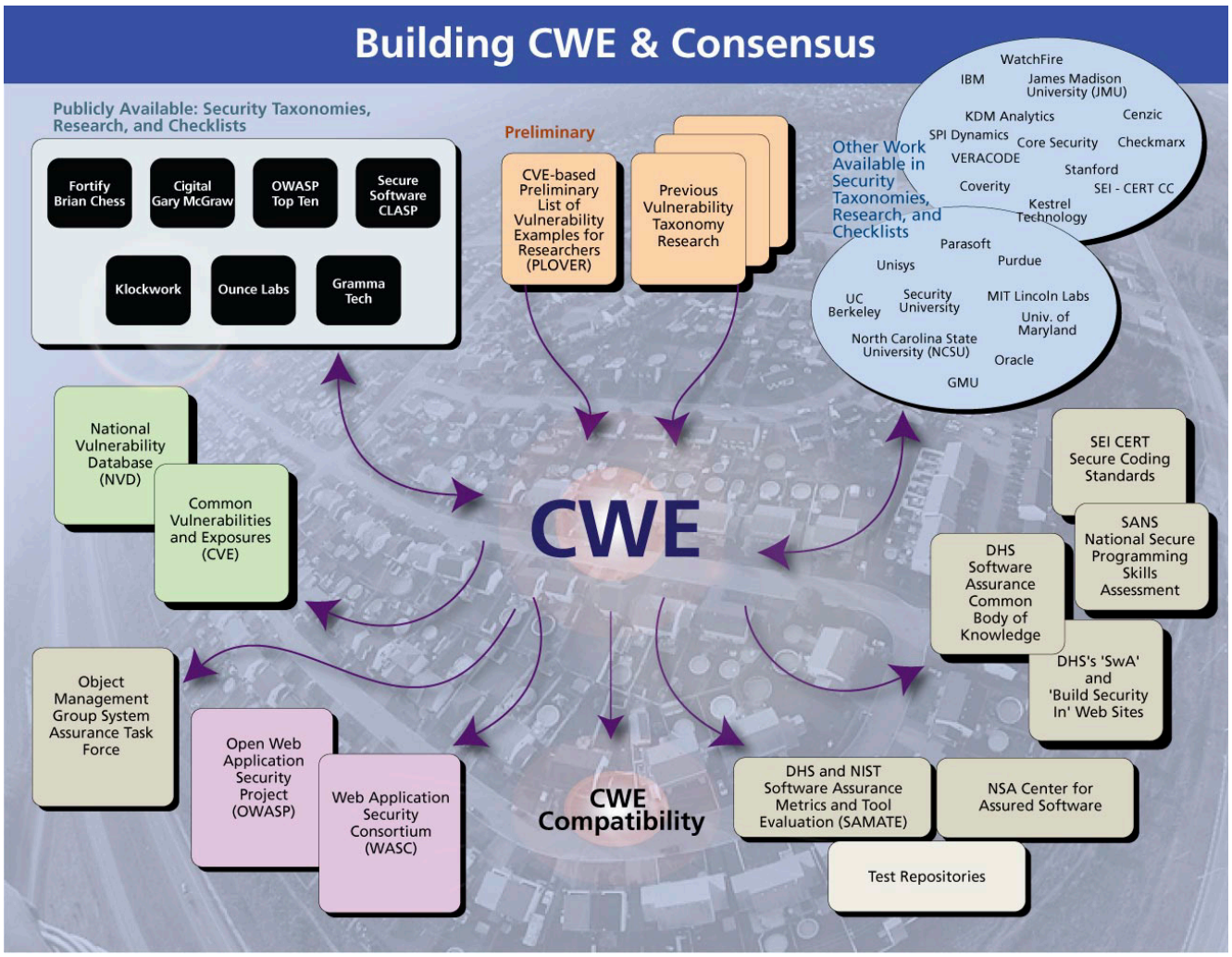


Figure 1. CWE Efforts Context and Community [http://cwe.mitre.org [1]

vulnerability. It can also help in determining whether particular tools are adequate for detecting attacks that are based on particular vulnerabilities [2].

After discovering a potential security vulnerability, a CVE Numbering Authority (CNA) can assign to it a CVE identifier [2]. Then the CVE Editor posts the information on the CVE List. The Primary CNA is MITRE Corporation. Other CNAs are software vendors, (for example, Apple Inc. and Adobe Systems Incorporated), third-party coordinators, (for example, CERT/CC), or researchers (for example, Core Security Technologies). The CVE Editor is MITRE Corporation.

2.4 Common Attack Pattern Enumeration and Classification (CAPEC)

Common Attack Pattern Enumeration and Classification (CAPEC) [4] was released in 2007. It includes descriptions of attack patterns. Information provided by CAPEC is needed in the process of finding vulnerabilities in software. In order to protect against attacks, knowledge of attack patterns is valuable, in addition to knowledge of software weaknesses that can be exploited by such attacks.

3. CWE in Practice

This section describes how the static analysis tools use CWEs to tag their tool reports and why it can add value to their products.

CWE contains a fairly comprehensive collection of application architecture, design, code, and deployment errors along with mitigation advice and examples of vulnerable and correct code segments. It also describes the SANS top 25 most dangerous software errors, that often "allow attackers to completely take over the software, steal data, or prevent the software from working at all." [1]

Because of its usefulness, CWE is already recognized and adopted by many organizations. For example, 40 organizations with 71 products and services already participated in the CWE Compatibility and Effectiveness Program (http://cwe.mitre.org/compatible/organizations.html). CWE has been adopted by NIST's National Vulnerability Database (NVD) (http://nvd.nist.gov) with mappings between CVEs and CWEs, and the Open Web Application Security Project (OWASP) – Top Ten Project (https://www.owasp.org/index.php/owasp_top_ten_project). Also, as part of the NIST SAMATE project, warnings from different tools that refer to the same weakness are being matched to corresponding CWE IDs to facilitate tools evaluation [9].

State-of-the-art static analysis tools today are able to find significant types of software security weaknesses. Many tools that support CWE are accompanied by public listings of the CWEs, and they are effective at finding and tag their vulnerability reports with corresponding CWE IDs. However, some mappings are not very precise, as CWE is organized into a hierarchy and some weakness types are refinements of other weakness types; also a single vulnerability may be the result of a chain of weaknesses or the composite effect of several weaknesses. The reality is that no single tool can detect all weaknesses and multiple tools should be used for complete coverage and better they all support CWE identification to facilitate the communication among them.

Customers also ask for the mappings of found weaknesses to the CWE IDs, as this provides common grounds for evaluating tools' performance and weaknesses' coverage. Therefore, even Static Analysis Tools that claim to be responsible for only limited number of weakness types [1] should not underestimate the importance of CWE and the mappings to CWE IDs.

4. Improving CWE

This section describes existing efforts, which include Semantic Template and Software Fault Pattern, to improve the readability and usability of CWEs.

CWE is a collection of weaknesses with a highly tangled structure at various levels of abstraction, mixed contents of attack, behavior, feature, flaws, and all by natural language representations. It means that using its relatively unstructured weakness categories is a daunting task for stakeholders in the software development community. To help utilize the valuable contents of CWE, efforts have been made by both academia and industry to improve the readability and usability of the CWE.

Wu et. al. [5] reorganized categories of CWEs into Semantic Templates to help developers and researchers construct a more clear mental model and improve the understanding of weaknesses. To facilitate the CWE use in the study of vulnerabilities, easy-to-understand templates for each conceptually distinct weakness type have been developed. The templates can then be readily applied to aggregate and study project-specific vulnerability information from source code repositories.

Another approach to improve the CWE is Software Fault Patterns (SFPs) [8]. SFPs decompose CWEs by fine granularity patterns with white-box definitions, then compose them into original CWEs with invariant core and variation points. With the purpose of being integrated into a standards-based tool analysis approach, SFPs focus more on the source code faults and the features that can facilitate automation. Such automation can potentially be very valuable for software assurance activities described in [12], because CWE has an important role in those activities [12].

4.1 Semantic Templates

A Semantic Template is a human and machine understandable representation that contains the following four elements [5]:

1. Software faults that lead to a weakness
2. Resources that a weakness affects
3. Weakness characteristics
4. Consequences/failures resulting from the weakness.

The required information pieces are either expressed together within a single CWE entry or spread across multiple entries. Such complexity makes it difficult to trace the information expressed in the CWE to the information about a discovered vulnerability from multiple sources. Therefore, to facilitate CWE use in the study of vulnerabilities, easy-to-understand templates for each conceptually distinct weakness type have been developed. These templates can then be readily applied to study project-specific vulnerability information from project repositories. For example, figure 2 shows the Semantic Template for Buffer Overflow, which is an aggregation of information collected from 42 CWEs. In this Buffer Overflow

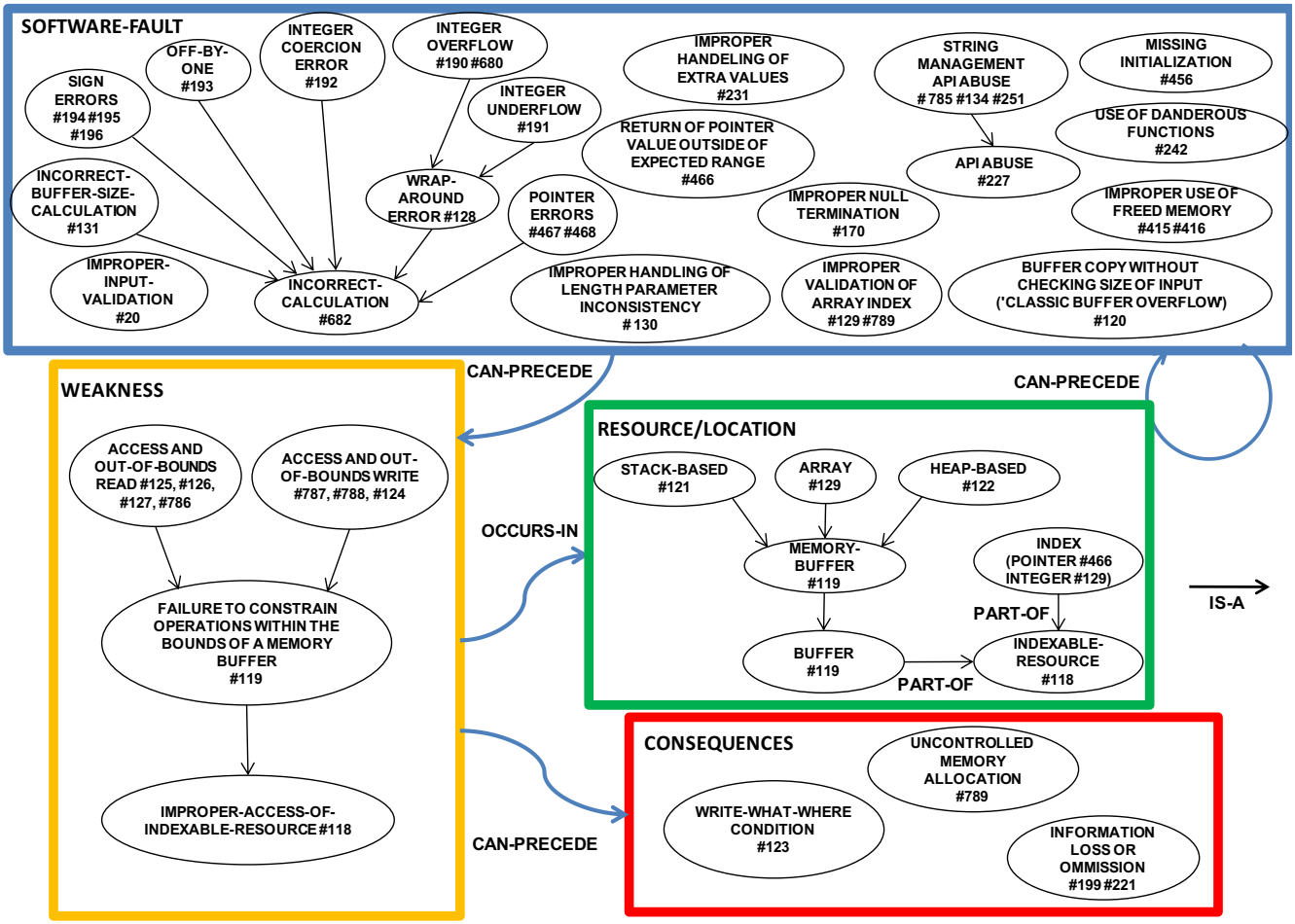


Figure 2. Buffer Overflow Semantic Template

Semantic Template, the four groups of relevant information were carefully collected and synthesized with "is-a" relationship inside of each group and "can-precede", "occurs-in" between the groups so that the lifecycle of a weakness from the starting point (software fault) to the end (consequences) is clearly presented.

The Semantic Templates also can provide intuitive visualization capabilities for the collected vulnerability information such as the CVE vulnerability descriptions, change history in the open source code repository, source code versions (before and after the fix), and related CAPECs [6]. Semantic Templates were shown to be helpful to programmers in constructing mental models of software vulnerabilities by an experiment described in [7]. In this experiment, 30 Computer Science students from a senior-level undergraduate Software Engineering course were selected to study six sets of vulnerability-related material with or without Semantic Templates in a pre-post randomized two-group design. The experimental results revealed that the group with the aid of Semantic Templates could analyze vulnerabilities with shorter time and higher recall on CWE identification accuracy.

4.2 Software Fault Patterns

Software Fault Patterns (SFPs) was developed by KDM Analytics Inc. By identifying and developing white box definitions for SFPs as a formalization process, they could be integrated into a standards-based tool analysis approach, benefiting both real-time embedded and enterprise software assurance systems. Those identified SFPs will be common to more than one CWE and can be used to further define CWEs [8].

The SFP is targeted at preventing cyber-attacks by collecting and managing knowledge about exploitable weaknesses and building more comprehensive prevention, detection and mitigation solutions. With the knowledge extracted from CWE taxonomy, three transformations were executed to extract common patterns and white-box knowledge, redefine existing weaknesses as specializations of the common patterns, then invariant core and variation points are identified to redefine each SFP to further represent weakness specializations [8].

KDM Analytics defines an SFP as a common pattern with one or more associated pattern rules (conditions), representing a family of faulty computations. The SFP structure is organized by the primary SFP definition which refers to the entire secondary cluster and is arranged into invariant core and variation points [8]. SFPs can map to multiple CWEs in such a way that each CWE in the family can be defined as a specialization of the SFP with its specific variations on the identified parameters. To date, 21 primary clusters, which include totally 62 secondary clusters, and 36 unique SFPs

have been identified. 632 CWEs have been categorized while only 310 of them are identified as discernible CWEs. Identified SFP definitions could lead to the development of more accurate testing tools and also improve developer education and training. They also provide benefits for a possible future formalization, since for each CWE, only the variation extension to a formalized SFP is required.

As the proof of recognition of the SFP research work, CWE-888: Software Fault Pattern (SFP) Clusters was incorporated by MITRE as a view into the CWE dictionary.

Both Semantic Templates and SFPs are designed to help understand and automate the vulnerability study. While Semantic Templates emphasize mental model construction from the human perspective, with the explanation of the four main elements of a vulnerability's lifecycle, while SFP's approach focuses on the "foot-holds", which are places in the code that present the necessary conditions for vulnerabilities, with the emphasis on the computation side to aid the test cases generator's work.

5. Future Directions on Improving CWE

This section provides future directions and our vision on CWE formalization.

CWE is a unique community effort and already has been proved to be extremely useful. For example, the NIST SAMATE project has utilized CWE during the past four Static Analysis Tool Expositions (SATE), whose goal is to advance research in static analysis tools that look for security defects in source code [9]. CWE is "a unifying language of discourse and a measuring stick for comparing tools and services" [10]. It is used in a wide variety of domains by developers and testers to look for known weaknesses in the code, design, and architecture of their software products; by consumers to make informed decisions when selecting software security tools and services; by researchers to develop new approaches and tools for software testing; and by professors to teach software developers how to avoid known weaknesses on architecture, design, and code level, in order to avoid security problems on applications, systems, and networks.

CWE is meant to be "a formal" list of software weakness types [1]. However, the CWE descriptions are currently in natural language and sometimes not accurate or precise by using phrases such as "correctly perform," "intended command," "intended boundary." For example, the description summary of CWE-119 in http://cwe.mitre.org/data/definitions/119.html includes the term "intended boundary", which is too vague. It does not indicate that it is the boundary given by the formal semantics.

> CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer "The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer."

While, to mitigate the vagueness of the definition as much as possible, our tentative definition of CWE-119 is: The software can access through a buffer a memory location not allocated to that buffer [11].

Therefore, the next logical step is to formalize CWE definitions, as formal approaches are less ambiguous and offer high level of accuracy. Our vision for CWE formalization and creating a system of accurate, precise definitions of CWEs, although a high-bar, is as follows:

- Revamp CWE entries towards Software Fault Patterns
- Review for accuracy existing CWE description summaries and white-box descriptions
- Analyze descriptions meaning and remove ambiguities
- Precisely define CWE entries with required accuracy
- Decide on a formal specification language
- Formalize CWE definitions
- Determine approach for validating CWE definitions
- Determine approaches for automated generation of tools for validation and verification towards particular weaknesses.

It is challenging to identify known weaknesses as well as newly discovered weaknesses, but it is challenging also to describe them in a succinct and unambiguous manner. Formalization should come in place and help further "shape and mature the code security assessment industry and dramatically accelerate the use and utility of automation-based assessment." [1]

Semantic Templates builds on CWE, and introduces a novel reorganization of CWE. One example for a potential use of Semantic Templates is for automatic change analysis. Patches provided by contributors to open source software may introduce vulnerabilities. Semantic Templates may help in organizing knowledge about known vulnerabilities in a way that will help patch contributors to detect vulnerabilities [5].

Once formalized the CWE definitions could be easily expressed through formal description techniques (FDT) and used as an input for generation of testing codes. This would facilitate automatic generation of more precise CWE-compatible software analysis and profiling tools for discovery of vulnerabilities or prioritizing vulnerabilities in terms of threats and impacts. Especially valuable would be the application for generation of dynamic analysis tools, which are better at discovering run-time vulnerabilities that cannot be captured with static-code analysis techniques – for example, buffer overflow lends itself to such dynamic analysis.

6.Conclusion

CWE provides common terminology for software developers, security experts, researchers, and customers to discuss software vulnerability in design, systems architecture, and source code. Software is central to computer science and as one of the purposes of CWE is to help avoid and eliminate software flaws in various stages of software production, CWE is of value not only to the software assurance community, but to computer science as a whole.

Improving quality of software development to reduce instances of weaknesses takes work from language designers, compiler writers, educators, assurance tool developers, researchers, vulnerability trackers, software engineers, and many more. If people in these roles disagree about what constitutes a particular weakness, or even whether it is a weakness at all, communication would be difficult at best. Therefore, broadly accepted definitions should be developed to allow diverse groups to work effectively together. It is important the definitions to be unambiguous and

complete to allow professional in the field to understand precisely what different software assurance tools, services, technologies, or methods can detect, mitigate, or prevent. Pure formalization of CWE would allow automatic generation of software components and tools to test for weaknesses that lead to exploitable vulnerabilities in software, create wrappers to filter out attacks that exploit them, or even rewrite the code to eliminate them.

Once precisely defined, CWEs could be formally described using a specification language such as Alloy (http://alloy.mit.edu/alloy). At its core, Alloy has a simple but expressive logic based on the notion of relations. Its syntax is designed to make it easy to build models incrementally and it has a rich sub-type facility for factoring out common features and a uniform and powerful syntax for navigation expressions.

To provoke further thinking and discussions throughout the Software Assurance community and beyond, we pose the following questions:

- What other formal methods can be used to help formalize CWEs with required accuracy and precision and at the same time allow for further extensions?
- To what granularity should CWEs be formalized? Finer granularity means more flexibility (especially when new weaknesses are identified, the extracted commonalities can reduce the re-invent work) but more effort to create them; Coarser granularity indicates the easy-to-use weakness items while we need to re-invent the wheel every time.
- How can the formalized CWEs be used and in which domains? For education and training? To prevent vulnerabilities? To integrate into software IDEs, test tools, and tools that generate test tools? To integrate in application security and development security technical implementation guides such as that of DOD [13].
- How can an automatic system be constructed to record newly identified vulnerabilities and classify them by CWEs? With better formalization and finer granularity of CWE definitions (which also means limited dictionary for weaknesses, better taxonomy of vulnerabilities), text mining could be the potential technique to mapping CVEs to CWEs at least semi-automatically.

ABOUT THE AUTHORS

Yan Wu is currently working as an assistant professor at Computer Science Department of Bowling Green State University, and she previously was a guest researcher in SAMATE team at NIST. She received her Ph.D. degree in InformationTechnology in 2011 from the University of Nebraska at Omaha. The main 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.
E-mail: yanwu@bgsu.edu

Yaacov Yesha is a Professor at the Department of Computer Science and Electrical Engineering at the University of Maryland, Baltimore County. He received his Ph.D. in Computer Science in 1979 from the Weizmann Institute of Science. He has received substantial research funding from government and industry. He was a program committee member of several conferences and a Chair of two workshops at IBM CASCON 2007.
E-mail: yayesha@cs.umbc.edu

Irena Bojanova is a professor and program director of Information and Technology Systems at UMUC. She is the founding chair of the IEEE CS Cloud Computing STC, a general chair of the IT Professional Conference <http://tinyurl.com/itproconf> , and coeditor of Encyclopedia of Cloud <http://tinyurl.com/EncyclopediaCC> Computing (Wiley, to appear in 2014). She is also an associate editor in chief of IT <http://www.computer.org/itpro> Professional and an associate editor of IEEE Transactions on Cloud Computing <http://www.computer.org/portal/web/tcc> . You can read her cloud computing blog at <http://www.computer.org/ portal/web/Irena-Bojanova>.
E-mail: irena.bojanova@umuc.edu

REFERENCES

1. MITRE. "CWE Common Weakness Enumeration." http://cwe.mitre.org
2. MITRE. "CVE Common Vulnerabilities and Exposure." http://cve.mitre.org
3. MITRE. "CWE Common Weakness Enumeration Common Weakness Scoring System (CWSS) CWSS Version 0.8." June 2011. Project Coordinator: Bob Martin, Document Editor: Steve Christey. http://cwe.mitre.org/cwss
4. MITRE. "Common Attack Pattern Enumeration and Classification (CAPEC) TM A Community Knowledge Resource for Building Secure Software." http://makingsecuritymeasurable.mitre.org/docs/capec-intro-handout.pdf
5. Y. Wu, R. A. Gandhi, and H. Siy. "Using semantic templates to study vulnerabilities recorded in large software repositories." 2010 ICSE Workshop on Software Engineering for Secure Systems, SESS '10, pages 22-28, New York, NY, USA, 2010. ACM.
6. R. Gandhi, H. Siy, Y. Wu. "Studying Software Vulnerabilities." CrossTalk, The Journal of Defense Software Engineering, September/October 2010.
7. Y. Wu, H. Siy, R. Gandhi. "Empirical Results on the Study of Software Vulnerabilities (NIER Track)." 33rd International Conference on Software Engineering (ICSE 2011), Honolulu, Hawaii. May 2011.
8. B.A. Calloni, D. Campara, and N. Mansourov. (2011). Embedded Information Systems Technology Support (EISTS) ---Task Order 0006: Vulnerability Path Analysis and Demonstration (VPAD), Volume 2 - White Box Definitions of Software Fault Patterns.
9. NIST. "Special Publication 500-297 Report on the Static Analysis Tool Exposition (SATE) IV." http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-297.pdf
10. R. Martin, S. Barnum, S. Christey. "Being Explicit about Security Weaknesses" http://cwe.mitre.org/documents/being-explicit/BlackHat_BeingExplicit_WP.pdf
11. Paul E. Black, Yan Wu, Yaacov Yesha, Irena Bojanova, in preparation.
12. Deputy Assistant Secretary of Defense for Systems Engineering (DASD(SE)) and Department of Defense Chief Information Officer (DoD CIO). Software Assurance Countermeasures in Program Protection Planning. Washington, D.C. 2014. www.acq.osd.mil/se/docs/SwA-CM-in-PPP.pdf
13. DISA for DOD, Application Security and Development Security Technical Implementation Guide (STIG), Version 3, Release 8. 25 July2014, http://iase.disa.mil/stigs/app-security/app-security/Pages/index.aspx
14. MITRE: CWE Common Weakness Enumeration, Frequently Asked Questions (FAQ), http://cwe.mitre.org/about/faq.html#A.8

An Alternate Approach to Avionic Software KISS

Gerry Tyra, Lockheed Martin Aeronautics

Abstract. Driven by customer perceptions of cost, there is a recurring drive in the avionics community to provide overarching software frameworks. It is believed that such a framework will simplify the software development process and promote reuse, hence reducing costs. While such goals may be laudable, this paper presents the argument that one-size-fits-all solutions will not fare well in most real world developments. A minimalist application environment is presented as an alternative. Like the tradeoffs between complex instruction set computers (CISC) and reduced instruction set computers (RISC), software has the option of complex frameworks vs. Keep It Simple...

Introduction

Avionic software development costs too much. Or, at least, it is *perceived* to cost too much. In an ideal world, a statement of work (SOW) is drafted, it is competitively bid, and the winning bidder, who is deemed to be technically competent, completes the work within the schedule and budget.

Unfortunately, we don't live in an ideal world. Talk to any group that has been around long enough to have earned their gray hairs, and you hear the same horror stories. The programs that over run have some combination of the usual suspects: The SOW was vague or had holes in it. The contracting authority didn't have funding to match the scope of the SOW, but didn't reduce the scope. The estimators were too optimistic. The proposal team underbid to win. Various aspects were unrecognized, hence were not factored into the bid at all. The contracting office slipped in additional requirements that were never negotiated. System engineering regurgitated the high level requirements without a high level design, leaving the independent product teams to go their own ways. The design phase was shorted to start producing code. Design decisions were made that range from "It seemed like a good idea at the time", to plain boneheaded. Schedule compression caused coding time to push into what should have been integration time. The integration facilities are unstable, out of configuration management and overbooked. All this before you get to flight test, which has its own issues.

Such is human nature and, so, schedules slip and budgets are overrun.

But, how are programs to overcome the tendency to blow budgets?

The bureaucratic solution is to add more process, structure and oversight. Uniformity will reduce cost and encourage reuse, while constraining "disruptive" activities. So was Ada, and other more recent standards efforts (e.g. FACE, UCI...), born.

Along the way, there has also been the effort to use Commercial-Off-The-Shelf (COTS) solutions. Some of these are still unproven academic exercises. Others are proven commercial products, but designed for specific environments, such as server farms.

When your main tool is a hammer, everything starts to look like a nail. The observant reader will notice that these solutions do little to address the root cause problems.

The alternative that is presented here is to provide a minimalist set of interfaces and a maximum degree of flexibility in implementation. Thereby simplifying design and integration, and providing a code base less subject to error. While this approach does not overcome the root causes, it does try to constrain them.

Requirements and Fallacies:

When your old and wise Computer Science professor tried to teach you "best practices," he was looking at the entire "ecology" of computer applications. But, avionics software is in its own niche. It is not that the general rules don't apply, rather, there is a difference in priorities that drives the software engineer to a different optimum solution. Here are a few most frequently abused realities of real-time.

SWaP

Size, weight and power; on an aircraft, these three are king. Ignore them and your program will fail. While you could build a flying server farm on a C-5 or an AN-124, a blade server attached to a hand launched quadcopter is not going to fly. Every kilogram of computer that you put on board is one less kilogram of fuel or payload available. Every additional watt has to be generated, using fuel, and then the resulting heat has to be dissipated. The bigger the system, the harder it is to fit inside the airframe, which usually implies that it will be harder to get at to maintain.

And, as a reminder to the ground station developer, if your facility is to be deployed to a remote site, the same rules apply. What does it take to transport your system? How many aircraft sorties will it take to supply fuel to the generators powering your system? Or feed the technicians operating it?

Tactical Bandwidth Is More Valuable Than SWaP

In any theater of operation, secure tactical bandwidth is scarce. That one satellite that you want to bounce off of may have a higher value than your entire program. Don't assume that all of the bandwidth is yours; this isn't a local 10 Gb/sec Ethernet connection. The bandwidth that you are allocated may be tiny. Be prepared to live with what you get.

Learning Curves

The more complex the tool set is, the longer it takes to become proficient using it. For a program with a lot of people unfamiliar with the tools, either you have to delay while they learn, or proceed and risk bad design choices that can haunt a program for years.

Services Cost

There is a school of thought that advocates an independent process/service for each defined task. The services are then strung out like a string of pearls. Sometimes, this is required. Sometimes it is useful. But, the process to process communications increases system latency, sometimes to the point of operational system failure.

What is Open?

One recurring theme is to use self-defining interfaces, (e.g., CORBA, XML, UCI, FACE, et.al.) and using virtual machines to mask underlying hardware. The argument is that this will simplify updates and allow swapping components. The reality is that there is nothing "Open" about the avionics system of an aircraft. It is designed, tested and certified as a whole. There are too many examples of what happens when something is updated without testing, "because it was only a minor change." Why pay the overhead for an open system, when it has to be integrated, tested and certified in a closed system environment?

And when the time comes to go off-board, reread the comment on bandwidth above.

Standards and Standardized

Using any standard entails having to carry the baggage that goes with it. Ethernet works. It works well in many applications, and in others, not so well. But it is an ongoing, evolving standard, moving from 10Mb/sec to 100Gb/sec in three decades. On the other hand, Mil-Std-1553 has been basically stagnant over the same period.

Similarly, you can use an Android phone to call a friend with an iPhone, though you can't share apps. The underlying technology is improving constantly, even if your older device can't make use of the newer options and speeds. Then there is Link 16, again a stagnant technology.

Consider the difference between the commercial standards and the military standards. The commercial standards grow because the commercial players are investing in the technology in order to attract customers. The military standards are decreed and there is only one customer. The developers may suggest innovations, but that one customer is in control.

Some standards you want to use, some you are required to implement. One size never fits all. Just because you want to use an interface, doesn't mean that it is available, or can be made available within the scope of your program. Always do a cost/benefit analysis in the context of your application.

Moore's Law Will Not Save You

So, your software is running on the ragged edge of what your hardware will support? The newest hardware will certainly fix it, right?

No, it won't.

Software bloat will eat any hardware improvement[2]. And that assumes that you will get a tech refresh. The reality is that many systems are never updated; only replaced when they reach the end of their service life. Even systems with planned tech refreshes are subject to having those updates delayed by years. Live within your means, as they are. Don't count on a refresh that may never come to save your program.

Abstraction Does Not Help

The argument has been made, repeatedly, that if we work at a higher level of abstraction, we will see tremendous improvements in developer productivity. In some areas this may be true, but for most of avionics, it is not the case.

What isn't covered by a good library function has to be coded by hand. And drawing a lot of UML pictures to describe a function is no less labor intensive than just writing the function (though you may have a more understandable design when you are done). And that simple picture you drew in UML can generate some truly hideous code that you will have to integrate and maintain.

Writing "a = b + c" is more efficient that trying to do it in machine code or assembly language. But burying that same equation in multiple generations of derived C++ classes does not make it easier.

Reuse Is a Myth

Reuse would save time and money, if it worked. Most of the time, the old code was written for a different platform, with a different interface and different requirements. Or, worse, it was slapped together on an IR&D project or an expedient program. In which case, the code is held together with baling wire and chewing gum. You can use such code as a design starting point, but do you really want to live with the maintenance headaches of the code itself? There are two cases for reuse: established libraries, and when you are bringing an existing subsystem into your program without modification.

MLS Is a Trap

Security is important. Losing secrets is bad. Keeping the secrets is expensive. So, Multi-Level Security (MSL) is frequently proposed as a solution. In the author's opinion, it fails for two reasons.

The first is perception of the environment. In a vast networked server environment, there are well defined islands of highly classified data in a sea of less classified/unclassified data. In an airframe, pretty much everything talks to everything else and all the data is needed. Consider all of the interactions that take place among the navigation, sensor, pilot interface and weapons system in order to have a weapon released. The islands and sea have become a swamp, the ground and water are both very muddy. Maintaining functionality with separation is difficult, if not impossible.

The second issue is time. It can take years to get an MLS system certified for operational release. And every update has to be re-certified. Some interfaces, such as Electronic Warfare (EW) can be changing from one mission to the next. If the update is critical to safety of flight, the entire fleet could be grounded until the certification is completed.

It's Not What You Say, Its How You Say It

C and C++ have issues. But, Java has a couple extra prob-

lems. One problem is the foot print, referring back to SWaP. The other problem is security, how stable is that virtual machine really? How much control do you have over it for maintenance? Who coded what, where and when?

Java has its uses and is very good in some applications. But, does it belong on your aircraft?

Do. Or do not. There is no try.

When bad things happen, we don't want them to become worse. But, how often are we successful at error recovery versus just paying lip service to it?

As an example, the C++ try/catch pair imposes a cost on computation, but how often does a catch actually do something useful? A print out to the console, when there is no console, doesn't help. Such a catch does nothing to resolve the root cause. If your memory allocator has run out of memory to allocate, the catch can report this. But, there is little it can do to alter your current lack of memory. You have a fundamental problem that should have been found in integration and test. Your application is about to crash. The best that you can hope for is a log file that will be useful back in the lab. But that is for another day.

Another common requirement is to check for a valid pointer being passed into a function. If you spend the CPU cycles verifying that a pointer is valid, what have you gained? An invalid pointer will crash the application. A trapped pointer will result in anomalous behavior (usually a premature return) in the function. Why did you call that function in the first place and what were you expecting it to do? What does not doing the expected imply to the system as a whole?

Now, extend this to error trapping in general. If the trapping does something that keeps the vehicle in the air and on mission, do it. If it only masks the root error and delays an inevitable system reset, why are you doing it?[1]

Power Point Slides Do Not a Design Make

At the Preliminary Design Review (PDR), the design team explains the direction that they are going in. There should be some initial analysis, though not compete, presented to demonstrate that the proposed approach can be made to work.

At the Critical Design Review (CDR), there should be enough solid data presented, along with supporting data, that a new team should be able to come in and take over the effort.

But how many CDRs are buried in pretty Power Point presentations that have no useful data? The design group does it because they had a milestone to meet, but they weren't actually ready for it. Program Management wants to get past the milestone, so it lets the problem slide. The Contracting Office also wants to show progress so it:

1. Overlooks the lack of supporting data.
2. Lacks the internal technical expertise to recognize the problem.
3. Doesn't notice because the presentation is such a snow job.
4. The Contracting Office just rotated in new personnel, they don't have a clue yet.

Keep It Simple S... Problem Space

In the beginning, there was hardware.

If your sizing and timing estimates, with I/O requirements, point to a microcontroller, rejoice. Get a simple development system and use C, perhaps with a little assembly and the standard packages that come with the chip. Don't build a world class super-computer when a single chip will suffice.

However, if you are doing something more complex, such as the Mission Systems suite for an aircraft, you will need a more sophisticated design.

At the simplest level a computer takes input, manipulates it, and provides outputs. If the process uses discrete and/or serial I/O, there is real work to be done. Bit twiddling for I/O is labor intensive and exacting. Similarly, the implementation of algorithms has to be done, state transitions properly defined and implemented, the epitome of "No Silver Bullet[3]." These things take time and effort. The details cannot be abstracted away.

But the environment that they exist in can be simplified.

Except under exceptional requirements, it is recommended to buy, not build. A program should also consider paying to get access to the source code for the OS and BSPs. Even well designed, well supported software has been known to demonstrate obscure bugs, usually late at night and at a remote site. The young engineers might think they can build it better and faster, but the wise manager has probably seen this all before. The OS, boards and BSPs represent man-years of engineering effort and acquired experience. Few programs can afford or justify this type of expenditure to build, maintain and support these items from scratch. After all, the objective is to make the plane fly sooner, not later.

Starting with the obvious; go back to the requirements and start partitioning the problem into functional units. But, maintain logically functional blocks. Don't subdivide just for the sake of subdividing. Then identify a logical set of processes to execute those functions and map those processes to hardware, keeping reasonable performance margins.

Remember that the farther apart two processes are, the greater the bandwidth cost to have them communicate.

Keeping It Simple

The first step towards simplicity is to provide a small, clean, stable framework capable of handling the mundane activities of the processes. The emphasis is on small, constrained and maintainable. There should be little or no middleware between an application and the OS, simply requiring a POSIX compliant OS solves many problems. A few select libraries can be used to abstract tedious activities (e.g., abstract the basic Ethernet or IEEE 1394 interfaces). This will allow the man-hours to be spent on application design, implementation and test, not fighting the middleware and OS.

All applications should be derived from the smallest number of base classes or templates. These base classes operate in a consistent manner and interface with the middleware interfaces consistently. And, consistency is a primary virtue for maintenance.

While the author has his own opinions on how to build a robust mission system, the details exceed the scope of this paper. Interested parties are invited to check out the methodology and sample code at:
http://www.planet-tyra.com/Software/index.html

Conclusion: How Simple is Keeping It Simple?

Start with a good solid design. But recognize that as work progresses, the design will shift and morph. Expect this and allow for it. If your original design was not overburdened with extraneous structures, the evolution should be mostly painless. With too complex a structure, any change is like scratching cut crystal, it is likely to crack and shatter.

This will not save a program from bad requirements or underbidding. But it will save some redesign, refactoring and late nights in the integration lab.

It is not the intent of this paper to dictate a particular approach to implementing software for avionics systems. Rather, it only hopes to show that there are alternative approaches to structuring the required software.

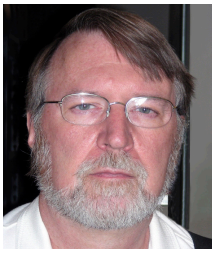
Disclaimer:

This paper presents the opinions of the author and does not represent the means or practices of Lockheed Martin.

REFERENCES

1. Don't confuse test code traps with what is needed in released flight software.
2. Blog by Brian Maccaba, "Why Software Doesn't Follow Moore's Law" http://www.forbes.com/sites/ciocentral/2014/05/19/why-software-doesnt-follow-moores-law/
3. "The Mythical Man Month: Essays on Software Engineering. Anniversary Edition" Frederick P. Brooks Jr. 1995

ABOUT THE AUTHOR



Gerard Tyra is a Sr. Staff Embedded Software Engineer with Lockheed Martin Aeronautics, Advanced Development Programs. He joined LM Aero in 2003. He trained as an Aeronautical Engineer (BS Engineering Science, Purdue University), and he served in the Navy's Civil Engineering Corps. After leaving the Navy, he migrated into real time embedded software while working for Martin Marietta. His work has covered many aspects of sensors and mission systems for submarines, armored vehicles, aircraft, missiles and spacecraft.

Joint Radio Manager Enhances Service Interoperability

Dean Nathans, G2 Software Systems
Dan Preissman, Project Manager Warfighter Information Network
Alan Gebele, Leidos

Abstract. With the maturity of tactical networking waveforms comes the need to consolidate the planning and management of these waveforms into a joint management system. This consolidated system is called the Joint Enterprise Network Manager (JENM). Soldiers can operate JENM's software application to plan and manage the next generation of lower and mid-tier radio waveforms, which include: the Soldier Radio Waveform (SRW), Wideband Networking Waveform (WNW), and the Mobile User Objective System (MUOS) on Software Defined Radios, as well as the Single Channel Ground and Airborne Radio System (SINCGARS) and Satellite Communications (SATCOM) Legacy Waveforms.

Introduction

An innovative, agile development process has given JENM the capabilities to provide multifold benefits for the warfighter. Foremost, rather than operating a unique manager for each waveform and radio type, a single network manager with a common user interface significantly reduces the equipment needed to manage a tactical network. JENMs manage the Wideband Networking Waveform (WNW), Soldier Radio Waveform (SRW), Mobile User Objective System (MUOS), as well as SINCGARS and SATCOM Legacy Waveforms on lower and mid-Tier Tactical Software Defined Radios (SDRs). JENM improves interoperability since the single type of manager provides a consistent configuration of the many parameters needed among multiple interconnected sub-networks. With an advanced Service Oriented Architecture (SOA), JENM is able to present a common user interface for management of the diverse networking waveforms and be far more user friendly than multiple managers. The Department of Defense (DoD) will save substantial costs in the development and logistics of future systems. This article explains JENM's capabilities and how it works in the hands of the warfighter, its role in advancing interoperability, the methodology for its agile software development, and current product status.

JENM System Overview

Wired and wireless networks require a network management system to configure, monitor and re-configure network devices in order for data packets to properly transit the network and respond to interruptions. A network management system configures devices, such as switches, routers, and security devices. Modern SDRs that host military networking waveforms contain routers and switches and other networking devices that must be similarly configured to a mission's specific communication requirements. Additionally, SDRs and military waveforms are configured for over the air management aspects such as time slot allocation, timing, and information assurance aspects. A wireless radio network for a military system should be able to operate without fixed infrastructure, and also have the capability to connect and interoperate with a wired network.

JENM's management capabilities are able to:

1. Design a Network
2. Load Radios and/or Load Devices
3. Monitor the Networks
4. Manage the Networks

How JENM Works

JENM's concept allows the user to develop a network design based on mission communication requirements. For each radio node in the network or sub-network, JENM develops radio/waveform configuration files including parameters that are unique to individual nodes, parameters that are common to all nodes in the network, and parameters dealing with radio services. Configuration parameters enable the radio and waveform to tune variable aspects of their operation that are changed based on mission communication requirements and operating environment.

JENM develops the configuration parameters based on user inputs and planning rules. The user provides basic network communication characteristics of a mission and JENM then uses planning rules, associated logic, and waveform configuration data to develop the plans. The network plans are translated into radio configuration files for each radio node in the networks or sub-networks. As part of the planning process, JENM interfaces with feeder data sources to obtain information such as Internet Protocol (IP) address ranges and spectrum allocations from the unit's higher level planning tools. This information is optimized into a tactical network configuration for the mission.

After developing the network plans, JENM uses them to develop the radio/waveform configuration files. The files are loaded directly to the radios or via military load devices. While there is overlap, each waveform and radio is configured differently. JENM abstracts the planning details involving complex functionality of multiple waveforms from the User, with a common user interface. This abstraction and the single JENM application results in significant planning time savings as compared to using different tools with different interfaces to plan for each waveform. Recent versions of JENM have reduced planning times by a factor of ten.

Once a network design is put into operation based on the mission's requirements, JENM has the capability to monitor and manage its networks in the field. The network manager can monitor aspects like: topology, performance and faults in the configured radios and waveforms, utilized bandwidths, and input/output data rates. As JENM monitors configured tactical networks, the user can act upon monitoring information or respond to mis-

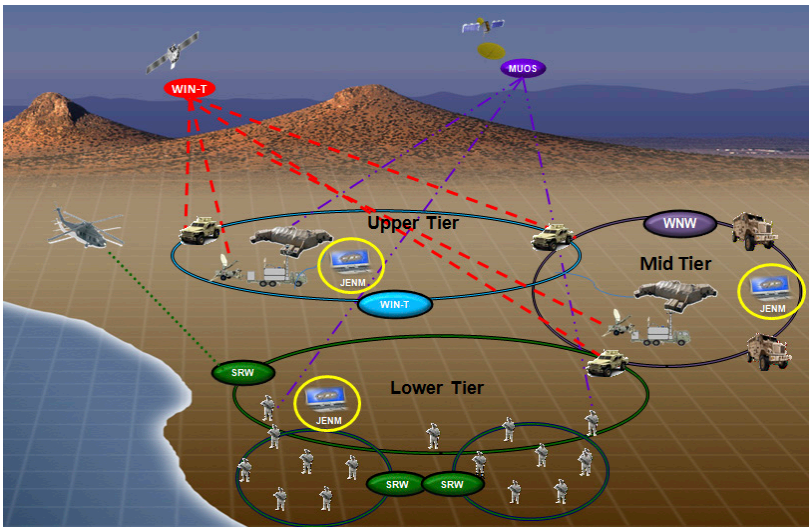


Figure 1: Typical Placement of JENMs in Army Tactical Network

sion changes using over-the-air management to reconfigure the networks. JENM will issue commands, which are sent to radios over the air to change the network configuration or information assurance aspects of network operation. This capability is known as Enterprise Over-The-Air Management (EOTAM).

JENM Improves Interoperability

There are many configuration parameters which need to be set for advanced networking waveforms. These parameters deal with waveform configuration settings at their equivalent three layers: Layers 1, 2, and 3 of the Open Systems Interconnect (OSI) Model. At Layer 1, settings involve configuration of bandwidths, modulation types, data rates and other parameters necessary for RF operation. Layer 2 settings involve wireless slot allocation, timing, and distribution. Layer 3 settings involve wireless and wired routing and packet distribution settings.

After a warfighter using JENM designs a network and sets the parameters, JENM configures the complex network of radios, consistently for all nodes in the network, sub-networks and for interface with connected upper tier backbone networks. It is essential the parameter settings for all nodes are consistent in order for there to be interoperability amongst radio nodes, as well as waveforms. JENM's User Interface properly configures the multitude of settings for all nodes while abstracting the details from the user operating in the field. JENM can do this for single service or joint scenarios enabling interoperability among networks and with higher level networks connecting to the Department of Defense Information Network (DoDIN).

Figure 1 depicts JENM's placement within an Army Tactical Network. The JENM's are strategically positioned at tactical communication control points to manage networks based on the mission and resulting network requirements. JENMs continuously communicate with each other to maintain the consistent network plans, provisioning and configuration for all nodes in all sub-networks throughout the lower and mid-tier tactical network with nodes running on multiple hardware types. Consistent network formation of different echelons and between services include a full awareness of inter-network, gateway and border requirements. The networks shown connect to high bandwidth backbones such as WIN-T resources.

JENM Architecture

JENM uses a SOA to manage the different radios with different interfaces. JENM's SOA leverages the inheritance aspects of object oriented programming to enable plug-ins that facilitate interface with a variety of radios.

Figure 2 illustrates the JENM SOA Multi-Layer Architecture. Items in blue are part of the Consumer Layer that includes user and application specific external interface software to radio equipment. Items in pink are the Service Layer which includes entry points into business logic functions and SOA standard interfaces including Representational State Transfer (REST) and external interfaces. Items in pink also comprise the Component Layer involving the plugin design patterns specific to the service and implementation of business logic. Items in green are part of the persistent Data Layer.

Designing a network is performed within the Designer and Network Development Service within the JENM application. The Designer includes an external interface, with which the warfighter inputs network formation data, and JENM then checks and validates the data. This information is used to design the network or networks and produces a network plan. JENM then develops the Network Plan and displays it to the user. The Network Provisioning Service consumes the plan, produces configuration files for the radios in the network, and loads them directly or via load devices. Each of the network services may interface with other devices in the network to request and obtain feeder data and provide configuration files.

Figure 3 is an illustration of the JENM's plugin approach. Since JENM manages multiple waveforms and radios a flexible architecture approach is needed which can be easily extensible to additional waveforms and radios. The figure shows that there is a plugin for each radio type which adds derived characteristics to the basic packager characteristics. The figure also includes a Target Packager Plugin. There is a similar plugin architecture for waveforms. In the past JENM has responded to the unique interface needs of each radio type. Going forward, JENM is working with the tactical networking radio developers to establish a set of common interfaces which are based on commercially accepted specifications such as Extensible Markup Language and Ethernet to further reduce costs for JENM and the radio programs.

JENM Agile Software Development Process

An agile software development methodology has become a critical component for the success of the JENM program. Agile development methodology rapidly responds to many customers among all services with different interfaces and radio network requirements, without the cumbersome overhead of the traditional waterfall software development processes. The JENM

program office acting as the product owner works closely with the lead developer to establish development priorities. The program office and the lead developer work together to plan development efforts into Sprints, or the creation and prioritization of Product Backlog Items (PBIs) that are responsive to the customer's needs in meeting all components of performance, schedule and cost. PBIs are planned into monthly Sprints to provide incremental capability additions. The lead developer acts as the software architect and leader of the Scrum of Scrums.

Before embracing the Agile Development Process, the JENM product office would let out large development contracts with fixed deliverables to contractors, who would, in turn, develop a system. The product office had limited ability to respond to the changing requirements of the customer base during the long contract development cycle. When the government-led Software Support Activity took over as lead developer, the product office was able to respond more quickly to customer needs without the delays inherent in large contracts.

Figure 4 illustrates JENM's Agile Software Development Process. The process begins with the establishment of the JENM Punch List. The punch list is the list of capabilities or product features needed broken down into small manageable pieces that can be accomplished within individual sprints through a list of PBIs. The punch list also includes a prioritized set of issues reported by users, integrators, and testers.

The punch list is reviewed and approved by the JENM Configuration Control Board (CCB) consisting of representatives from the JENM Program, its Software Support Activity (SSA), customer offices, and the user community. Once the monthly list of prioritized PBIs is established and approved by the CCB, a subsequent monthly planning meeting prioritizes future Sprint PBIs. The monthly list of PBIs is then executed by the government-led SSA. The SSA conducts daily scrums. The CCB approves incremental software releases with customer programmatic needs in mind. The JENM Program monitors its performance using the velocity of planned, in-progress and completed PBIs, requirements burn down, and issues burn down.

JENM Organization and Product Status

The JENM Product Management Office (PdM JENM) is responsible for development of the JENM Product. PdM JENM is a product office within the Project Manager Warfighter Information Network - Tactical (PM WIN-T) in the Army's Program Executive Office for Command Control Communications (PEO C3T). The JENM product office is collocated with the Joint Tactical Networking Center of PEO C3T in San Diego, CA, and it also has a subset of staff at PEO C3T at Aberdeen Proving Ground, Md. The lead developer for the JENM is the Network Management Reference Implementation Laboratory (NMRIL) Software Support Activity located at SPAWAR Systems Center Pacific, San Diego, CA. Contractor support to the NMRIL government staff in the development of JENM includes Booz Allen Hamilton, G2 Software Systems, Harris (formerly Exelis), Northrup Grumman Corporation, Tactical Engineering and Analysis, and additional subcontractors. Government activities

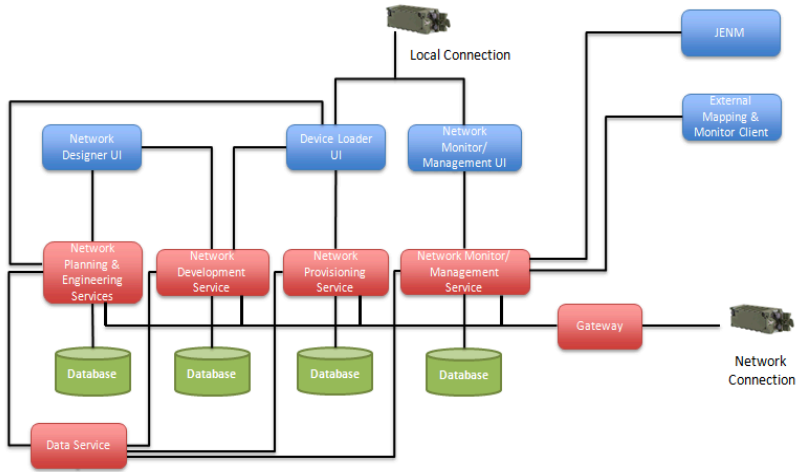


Figure 2 JENM SOA Multi-Layer Architecture

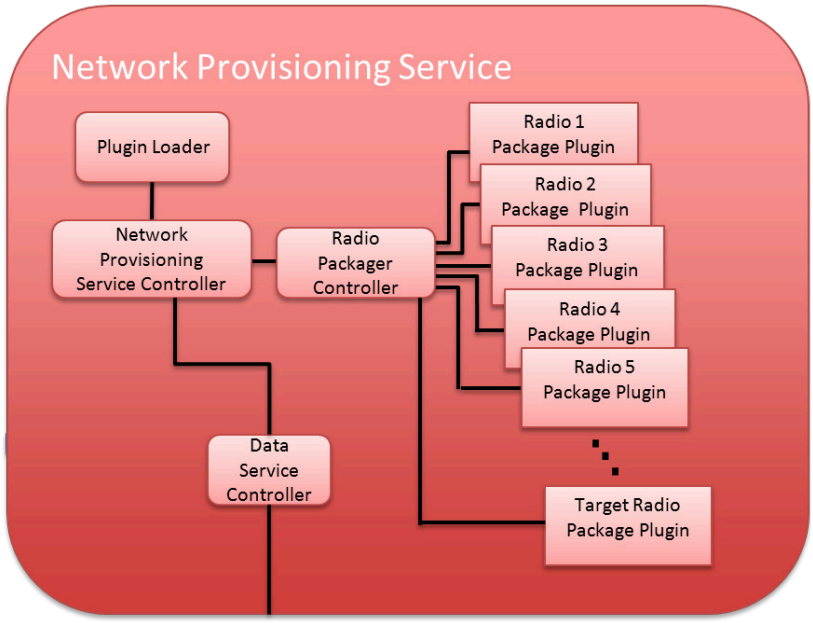


Figure 3 Example of Plugin Architecture

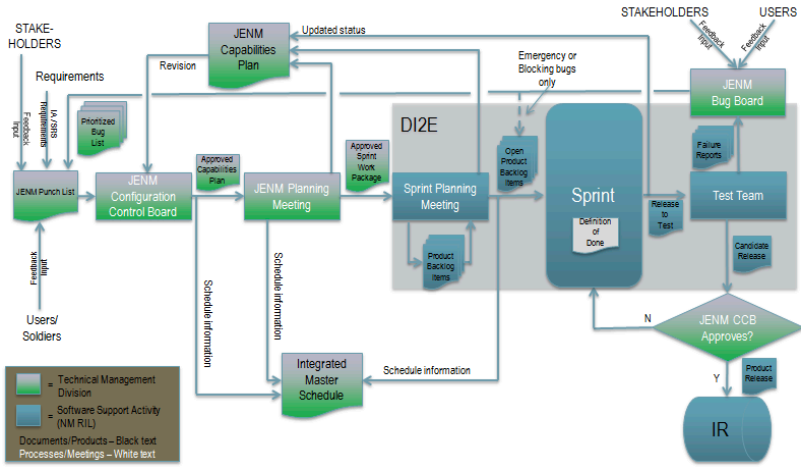


Figure 4 JENM's Agile Software Development Process

also supporting the NMRIL include the Army's Communications - Electronics Research, Development and Engineering Center, the Navy's SPAWAR Systems Center Atlantic, and others in support of radio program offices.

Version 3.3 of the JENM Software Application was released in December 2015. The application manages the WNW, SRW, MUOS, SINCGARS, and SATCOM waveforms on Joint Service SDR Programs of Record including Handheld, Manpack and Small Form Fit (HMS) Program Rifleman Radios, and Mid-Tier Networking Vehicular Radios (MNVR), and AN/PRC-117G for MUOS. A key addition in the JENM v3.3 is the ability to perform over-the-air management of the MNVR Radios. The JENM v3.3 also has many user interface enhancements which significantly reduces reliance on field service representatives. JENM has received positive user feedback based on Army Network Integration Evaluation (NIE) test events, as well as from program specific test events such as the MUOS Operational test event.

"We have developed JENM to provide a user friendly capability to configure and manage lower and mid-tier networking waveforms in a single software application," said Lt. Col. Matthew Jury, former JENM product manager within the Army's WIN-T project office. "With JENM, our warfighters are better equipped to configure networking waveforms to operate seamlessly within and across tactical networks. Working closely with networking and radio program product managers within the Army and with other services, we continue to add features to accommodate management of networking waveforms in a joint force."

Looking forward in the near term, the JENM will continue to develop new versions of software to support the evolving joint service program of record networking radios with capabilities added in step with their procurements. As the tactical networks evolve, the JENM role will be expanded to include configuration and management of selected network control, information assurance, and routing devices. Also the JENM interfaces will be expanded to interoperate with higher level managers and additional mission command devices with roles in configuring networks.

ABOUT THE AUTHORS

Dean Nathans is a Senior Engineer working for G2 Software Systems assigned to support the Joint Enterprise Network Manager (JENM) Technical Management Division in the JENM Product Management Office, within the Project Manager Warfighter Information Network -Tactical Project Office. Mr. Nathans performs systems engineering, network engineering, and interface development for the JENM Product. Mr. Nathans has over thirty years of experience with military communications and navigation systems in industry and government including positions of Senior Engineer, Chief Engineer, and Deputy Program Manager with major acquisition programs. Mr. Nathans has a Master's Degree in Electronics Engineering and a Bachelor's Degree in Electrical Engineering. He has received many awards for his service, including the Superior Service Award and Meritorious Civilian Service Award.

Dan Preissman is the Product Lead for the Joint Enterprise Network Manager (JENM) within the Project Manager Warfighter Information Network - Tactical Project Office. Mr. Preissman has served in this position since 2007 when the JENM was early in its development, and has led the JENM through to its current version 3.3 product release. Prior experience has included management and technical leadership positions involving military aircraft standards, aircraft electrical systems, and automated electrical/electronic test systems. Mr. Preissman has a Master's Degree in Engineering Management, and a Bachelor's Degree in Electrical Engineering. He also holds a Master's Of Arts Degree in National Security and Strategic Studies. Mr. Preissman has received many awards for his service including the Superior Civilian Service Award, and Bronze Order of Mercury Medal.

Alan Gebele is a Senior Software Engineering Manager working for Leidos assigned to support the Joint Enterprise Network Manager (JENM) Technical Management Division in the JENM Product Management Office, within the Project Manager Warfighter Information Network -Tactical Project Office. Mr. Gebele acts as the Deputy Team Leader for a group of software, systems engineers and project managers who manage the JENM requirements and priorities for the JENM Product in support of the JENM Product Owner. Mr. Gebele has over 30 years of telecommunications and military communications experience including the network monitoring development lead for the Joint Network Management Systems developed to plan and manage Joint Task Force level networks in support of Combatant Commanders mission requirements. Prior assignments include the software development and systems engineering roles software products developed for commercial telecommunications providers in the USA, European and African for the full life-cycle of operations. Mr. Gebele has a Master's Degree in Computer Science from Brown University, and a Bachelor's Degree in Computer Science from Purdue University.

Enterprise Systems Integration using Collapsing Design Structure Matrices

John M Colombi, Air Force Institute of Technology
Michael P. Kretser, Air Force Institute of Technology
Jeff Ogden, Air Force Institute of Technology
Paul Hartman, Air Force Institute of Technology

Abstract. Many large enterprises, such as the US Air Force (USAF) logistics community, evolve over many years creating a variety of distributed, functionally redundant, and highly interrelated information systems. This paper proposes using Collapsing Design Structure Matrices (C-DSMs) to identify and develop cost-effective systems integration plans. In addition to identifying a roadmap for system reductions, the algorithm also tracks integration and operating and maintenance (O&M) costs. An example demonstrates the technique, inspired by a recent enterprise resource planning (ERP) program.

Background

A CrossTalk article by Eric Maass addressed the "daunting task" of building a common enterprise from disparate organizations and integrating numerous enterprise-class applications [1]. Other CrossTalk articles have addressed the technical solutions of integrating enterprise systems, such as applying Service Oriented Architectures (SOA) [2]. Still, the path to developing and implementing enterprise systems integration often eludes executives as many decisions must be made without supporting tools. "Companies require a realistic route to implementation that sequences migration" as they typically do not know what the end state should look like or where to begin [3]. This problem can be exacerbated if there is not a significant enterprise governance structure. "In many companies, business-IT governance is not managed cohesively or from a holistic, firm wide perspective. Instead, decisions are made in siloed fashion within individual business functions or units, with little thought given to how those decisions might affect other parts of the company or the company as a whole" [4].

A variety of options exist to integrate legacy enterprise systems. One option is converting to a shared services model where legacy systems are attaching to a common middleware. Shared services have shown promise as legacy systems do not have to be replaced, but only upgraded. Another option is to replace the legacy systems with a modern Enterprise Resource Planning (ERP) system. An ERP is a cross-functional information system driven by an integrated suite of software modules that supports the basic internal business processes of a company. To move to a single ERP solution, an organization would save its persistence data, decommission existing legacy IT applications, install the ERP which may require changing to modern data-driven integrated business processes, and may pay to uniquely customize reports, interfaces, or data conversions [5].

While there have been some successful ERP implementations, there have been far more failures and many others demonstrated less than the expected return on investment (ROI) or delays [6],[7],[8],[9],[10]. A failed ERP implementation hurts the implementing organization in at least three ways: cost of development and implementation up to the point of failure, reinvestment costs in legacy systems to implement currently needed capabilities, and continued cost of unrealized efficiencies [11]. After years of struggle with implementing ERPs, the US Air Force commissioned the RAND Corporation to research "early planning issues associated with ERP programs" and make recommendations "how these issues may be manifested during program execution." This report was published in 2013 after the cancellation of the Expeditionary Combat Support System (ECSS). In the report [12], the authors investigate the conditions for successful ERP implementation and break these down to: the Business Case, Governance, Business Process Reengineering (BPR), Organization Change Management, and IT Acquisition. For each of these conditions they provide challenges that the USAF will have to overcome to be successful. These align with other published success factors for ERP implementations, most of which include the need for a vision or "To-Be" architecture [13], [14], [15].

Enterprise architects need tools and methods that provide a roadmap that allows them to see their current state (the "As-Is"), their desired state (the "To-Be"), and a transition plan. While there are several standard representations for depicting As-Is and To-Be architectures, supported by various architecture frameworks (Zachman, the Open Group Architecture Framework, the Department of Defense Architecture Framework, etc.), there is little literature explaining how transition planning should best be accomplished. Currently, many of the techniques employ expert opinion, consultants, and "gut" decisions by process owners leading to varying results. While this paper proposes a quantitative approach using optimization of legacy IT system relationships, it will be up to engineers and architects to communicate their methods to senior leaders and decision makers. Large scale enterprise IT improvements are often fraught with resistance to change.

Design Structure Matrices

Due to the complexity of enterprise information systems, dependency models such as design structure matrices (DSM) could be a suitable technique. DSMweb.org defines DSM as "a simple tool to perform both the analysis and the management of complex systems. It enables the user to model, visualize, and analyze the dependencies among the entities of any system and derive suggestions for the improvement or synthesis of a system." Thus, it should allow analysis of system dependences that could lead to system integration efforts.

As a matrix, the DSM captures the relationships between components of a system, or systems themselves, across the rows and columns [16] (see Figure 1). In addition, DSMs allow mathematical manipulation of the relationships, which is conducive to the construction of an automated roadmap algorithm. Traditional DSMs have not been used as a systems integration tool.

The algorithm used in this paper extends previous work by Thebeau [17]. Our algorithm implements a multiple objective optimization, based on system relationship strengths, to find which

Type	Example Dependencies
Structural	• # of Interfaces/Interface control documents (ICDs) • Interface complexity (point-to-point interface, enterprise application interface/ service encapsulation, etc) • Projected Lifespan (how long is it required) • Maintainability, Adaptability, Flexibility, Planned updates
Functional	• Shared functionality, commonality of systems
Informational	• Number of information exchanges (in/out) • Frequency of exchange (daily, real-time, monthly) • Diversity of exchanges (transaction types, batch) • Volume of data across the interface • Common data elements • Interoperability
Implementation	• Likelihood of successful integration • Performance requirements (Service level agreements)
Financial	• Cost to integrate/ modify legacy systems • Cost/ schedule to translate /import legacy data

Table 1: Taxonomy of Enterprise Information Systems DSM relationships

System	Total Cost*	O&M New Sys*	Intg Costs*	O&M % of Intg Cost	Num Legacy Sys	Avg Intg Cost Per Sys	Estimate O&M Old Sys	Total O&M Old Systems
System A	137.5	4.5	124.8	3%	17	7.34	1	17
System B	86.4	7.8	70.2	9%	18	3.90	1	18
System C	83.9	4.8	68.7	6%	22	3.12	1	22
System D	159.7	7.9	132.4	5%	11	12.04	1	11
System E	32.2	0.7	29.3	2%	1	29.30	1	1
System F	18.9	1.2	16.6	6%	1	16.60	1	1
System G	44	6	25.6	14%	6	4.27	1	6
Total / Avg	**562.60**	**4.70**	**467.60**	**6%**	**76**	**10.94**	**1**	**76**

Table 2: Masked Source Data (all costs in $Millions)

Iteration	Systems to be Integrated	Number of Systems Remaining	% Reduction Achieved	Investment/ Integration Costs for this Iteration	O&M Savings for this Iteration
1	79/92; 32/89; 72/82; 56/80; 22/76; 59/91; 10/43; 7-38; 26/46; 21/77; 18/45; 16/60;	88	12%	$439.2M	$53.5M
2	70/86; 27/84; 20/81; 48/65; 47/63; 26/62; 34/55; 3/51; 39/64; 12/24; 11/80;	77	13%	$393.7M	$50.8M
3	43/63; 20/62; 88/46; 36/37; 18/35; 28/75; 23/33; 19/50; 17/42;	67	13%	$261.3M	$47.1M
4	27/30/41; 39/47; 33/35; 7/32; 20/42; 18/55; 16/63;	58	13%	$336.3M	$50.5M
…		…	…	…	…
22	2/3;	3	25%	$28.2M	$5.2M
23	1/3;	2	33%	$33.1M	$5.0M
24	1/2;	1	50%	$33.0M	$4.4M

Cumulative Integration Costs: $3.4B	Total Cumulative Savings: $5.08B (O&M Baseline – Integration Costs)

Table 3: Example System Integration Plan

System \ System	A	B	C	D
A	1	0	0	.5
B	0	1	0	0
C	.3	.75	1	0
D	.25	0	0	1

Figure 1: Example DSM capturing the strength of relation between legacy IT systems. One or more matrices may be used to capture multiple factors. Note that systems may not be related (0) and systems relationships may not be symmetric.

rows and columns of the matrix (representing the legacy systems) should be clustered or integrated. Thebeau's clustering approach is modified to allow for integration and removal of systems, as well as allowing user constraints on target reductions. The enterprise size "collapses" over time; thus the suggested name, Collapsing Design Structure Matrices.

Design structure matrices (DSM) have been used for a variety of applications, and thus, the interactions between systems (rows and columns) model varying characteristics [18]. The interactions (values in the matrix) may capture spatial, energy, information or material relationships. The static DSM, as presented by Browning [18], serves as our representation where rows and columns represent legacy systems and the matrix entries represent the strength of the inter-system relationships. In fact, multiple matrices could capture several different types of interactions between systems [19]. For enterprise systems engineering, such relationships could include the following taxonomy in Table 1.

In addition to the relationships between systems, systems may also have individual characteristics that may need to be incorporated into the transition plan. Such factors could include Operations and Maintenance Costs (annual), priority/mission requirements, persistent data issues or system criticality. Technically, Table 1 shows why legacy systems should be integrated, as well as an indication of difficulty (cost/ resources) to integrate.

Collapsing DSM Approach

The objective of this algorithm is to minimize a penalty function through clustering related systems, then integrating (or collapsing) those systems together to produce a smaller, less complex enterprise. The process can be repeated until the enterprise has reached its desired level of reduction, or it has become a single system. Each iteration represents a time period specified by the user, as well as the desired amount of reduction and number of systems to integrate into a cluster can be tailored accordingly. If one iteration represents a single year and the systems are complex, cluster size and overall enterprise reduction should be small. The smaller, more frequent, incremental approach is the driving methodology behind this approach, as it lends to more frequent progress checks with decision makers. This approach could be conducive to DoD as an alternative to a complete ERP replacement for hundreds of legacy systems.

The algorithm searches for which systems should be put into which clusters, and how many clusters should best be required. Clusters represent a group of highly-related systems that are "most alike", and have a higher likelihood of compatibility and less risk for the purpose in integration. Systems in a cluster will be integrated and require investment for integration costs. We define a penalty function, called Total Integration Effort, that represents the effort of clustering systems, penalizing for too many inter-cluster relations (InterClusterIntg) and cluster size (IntraClusterIntg). This summation is across all rows *i* and columns *j* of the Design Structure Matrix and all proposed clusters *k,* shown in Equation 1.

Equation 1:

$$\text{Total Integration Effort} = \sum_{k}\sum_{i}\sum_{j}\left(\text{InterClusterIntg}_{kij} + \text{IntraClusterIntg}_{kij}\right)$$

The algorithm is written in MATLAB, modified from Thebeau [17] to produce an integration plan of optimal reductions. The algorithm follows as such:

1. Each system *i* is initially placed in its own cluster *k*
2. Calculate the *Total Integration Effort (TIE)*
3. Implement simulated annealing for approximating the global minimum of *TIE*.
    a. Select a random system *i*
    b. Accept bids from other clusters to integrate system *i*
    c. Probabilistically decide to cluster system *i* attempting to lower *TIE*
    d. Ensure all constraints satisfied
4. Loop back to Step 3 a set number of times or until convergence criteria met
5. Result is a set of clustered legacy systems (to integrate) that minimizes *TIE*

The algorithm allows specifying several user-defined constraints. These included target reduction, reduction margin (+/- percentage), and minimum and maximum cluster size. This allows the enterprise architect or engineer to control how aggressive the reductions will be per iteration.

Example of an Enterprise Application

The following example is based on government data and cost estimates for a large enterprise program. The information is masked, and some random values are generated to represent realistic solutions. While the actual program had many more systems to integrate, our example will use 100 systems. A DSM matrix is created that represents the pair-wise relationships and strengths between these 100 systems. The authors obtained program cost estimates on legacy IT systems. To produce our example cost estimates, data followed the distributions from Table 2 for total cost, O&M costs, integration costs and O&M cost savings.

For this illustrative example, a 10% goal for system reduction is set with a margin of +/- 2%. This would reduce 100 systems down to 88-92 systems in the first iteration (year). A range is provided to allow the algorithm to explore answers that are slightly above and below the intended target, as a "better" solution may

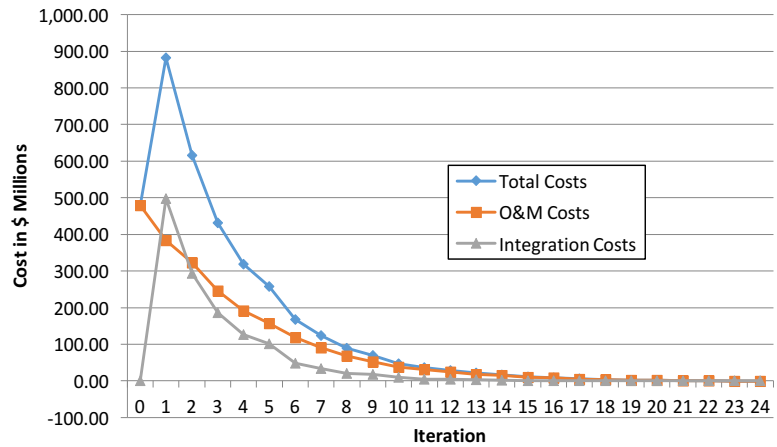


Figure 2: Integration, O&M and Total Cost Estimations for 10% target reductions

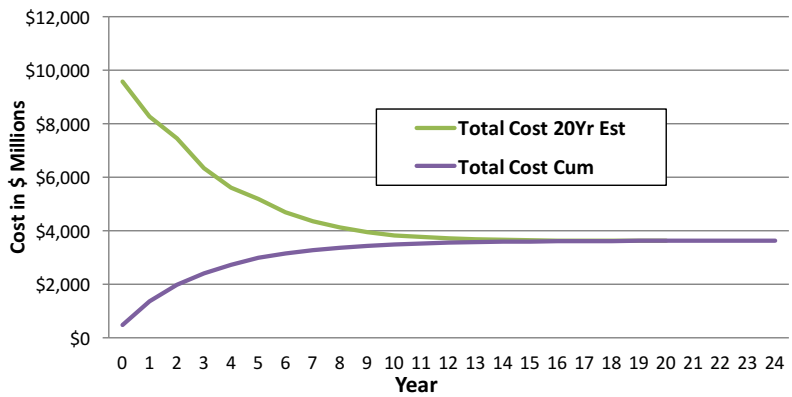


Figure 3: Total Cost Projections over 20 years, if the integration algorithm stopped (top estimated curve and bottom (cumulative) curve.

be available. The other significant constraint is the maximum number of systems allowed per cluster.

The algorithm calculates a series of acceptable solutions and returns the "best" solution that minimized the Total Coordination Cost penalty function. A list of clustered systems is identified for integration. The user can then rerun the algorithm repeatedly until the desired total reduction is achieved, which could be a single system. For this example, the final solution presents is a 24-step system integration plan that held to the constraint of a 10% percent reduction per iteration. Table 3 shows the first few and last few integrations, the size of the enterprise (systems remaining), and cost information.

The total integration costs for this solution was $1.35 Billion over 24 integration periods (years) with an average cost of $56.4M per year. The original enterprise annual O&M costs were $479M per year, while the final system solution's O&M costs are estimated at $1.8M annually. In Table 3, the baseline at time zero represents the current enterprise. At iteration 1, integration costs are spent and O&M costs are removed resulting in the total cost of the solution, if the decision maker decides not to integrate further. However, as the iterations continue, a downward trend is realized showing that after iteration 3, the total cost drops below the baseline O&M cost. After three iterations, this enterprise has begun to see the financial benefit of integrating from 100 to 67 systems. This chart illustrates the benefit of the approach as planners can choose to stop integrating at any point, and do not have to "buy-in" to a full ERP replacement. This method of collapsing the enterprise supports the technical decision as which systems should be integrated based on one or more of the assessed factors from Table 1. Often, it will be short-term investment and integration costs that drives the final decision making.

In Figure 3, a 20-year projection is calculated to determine what the estimated cost of the integrations would be over 20 years. This figure starts with the baseline of the current enterprise and provides the total cumulative cost should the enterprise chose not to integrate. Thus, this would be 20 years with the current O&M estimates. At Year 1, the cost is calculated using 19 years of O&M savings and subtracting the first year integration costs. At Year 2, the cost is calculated by the first year of O&M costs (for all 100 systems), plus 18 years of the new O&M costs (for 88 systems), minus the investment from the first two years of integration costs. This trend continues for 20 years and produces the descending cost projection curve (if integration stops) while the increasing curve represents the estimated cumulative costs. Eventually around Year 14, the two curves merge as the incremental savings (by integrating the last few systems) is marginal.

Discussion and Conclusion

We propose the use of Design Structure Matrices (DSMs) which allow the visualization, modeling and analysis of system relationships. For large complex enterprises, understanding the IT system relationships seems appropriate for use in enterprise integration planning. The relations may be structural, functional, informational or financial. The modifications to traditional DSMs allow the size of the enterprise to collapse as systems are integrated during each iteration of the algorithm. An example case demonstrates utility for practitioners. Practitioners are able to select the type, or types, of system relationships, then manipulate the constraints as to how much reduction should be applied each year. The government sector may find utility in this method as change in government systems is much slower and the size of the enterprises is generally much larger. Government enterprise evolution can be stymied by politics, policies, laws, culture, resistance to change and organizational command hierarchies. Though these factors will still be present, this Collapsing-DSM approach attempts to quantify and optimize integration steps using unbiased system relationships, while still tracking the integration cost investments and estimated O&M costs and savings.

Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense or the U.S. Government.

ABOUT THE AUTHORS

John M. Colombi is an Associate Professor of Systems Engineering at the Air Force Institute of Technology (AFIT), where he serves as Program Chair. His research interests include systems and enterprise architecture, complex adaptive systems, acquisition, modeling and simulation and human systems integration. Dr. Colombi retired with 21 years of Air Force experience as a developmental engineer with assignments in C2 systems integration, systems engineering, biometrics, info security and comm networking.

Paul Hartman is Director, Center for Operational Analysis (COA). The COA is a premier research facility within the Air Force Institute of Technology, Graduate School of Engineering and Management, which directly supports Department of Defense (DoD) strategic objectives. Dr. Hartman has over 28 years of demonstrated expertise serving in a wide variety of Program Management, Supply Chain Management, Maintenance Management and Logistics Policy positions.

Michael P. Kretser received his B.S. in Computer Science from Limestone College, Gaffney, SC in 2005, the M.S. in Logistics Management from the Air Force Institute of Technology (AFIT), Dayton, OH in 2008, and his Ph.D. from AFIT in 2015. He has 20 years of Air Force experience in aircraft maintenance, logistics, teaching and leadership. His research interests include systems-of-systems, enterprise architecture, enterprise logistics, supply chains, process improvement methods, and modeling and simulation.

Jeffrey A. Ogden is an Associate Professor of Logistics and Supply Chain Management within the Department of Operational Sciences at the Air Force Institute of Technology (AFIT). He earned his MBA and Ph.D. in Supply Chain Management from Arizona State University. His purchasing and supply chain management research has been published extensively, where his in research interests include purchasing strategies, insourcing/outsourcing, contracts, information systems, and supply chain performance measurement.

REFERENCES

1. E. Z. Maass, "Enterprise Engineering: U.S.Air Force Combat Support Integration," CrossTalk: The Journal of Defense Software Engineering, pp. 16-20, 2003.
2. G. Raines, "Leveraging Federal IT Investment With Service-Oriented Architecture," CrossTalk: The Journal of Defense Software Engineering, pp. 4-8, 2009.
3. F. Roghe, A. Toma, R. Messenbock, S. Kempf and M. Marchingo, "Breaking free of the silo: Creating lasting competitive advantage through shared services," BCG Perspectives, 2013.
4. M. Grebe and E. Danke, "Simplify IT: Six ways to reduce complexity," The Boston Consulting Group, 2013.
5. H. Bidgoli, The internet encyclopedia, John Wiley& Sons, Inc., 2004, p. 707.
6. E. J. Umble, "Enterprise resource planning: Implementation procedures and critical success factors," European Journal of Operational Research, vol. 146, no. 2, pp. 241-257, 2003.
7. G. R. Bliss, "Root cause analysis of the expeditionary combat support system program," Performance Assessments and Root Cause Analyses (PARCA), Washington DC, 2013.
8. U.S. Government Accountability Office, "Homeland security: Department-wide integrated financial management systems remain a challenge. (GAO Report No. GAO-07-536).," GAO, Washington DC, 2007.
9. U.S. Government Accountability Office , "Department of homeland security: Progress made and work remaining after nearly 10 years in operation," GAO, Washington DC, 2013.
10. Inspector General of the United States Department of Defense, " Enterprise resource planning systems schedule delays and reengineering weaknesses increase risks to DoD's auditability goals," DoD IG, Washington DC, 2012.
11. G. Baxter, "Key issues in ERP system implementation," University of St. Andrews, School of Computer Science, 2010.
12. J. Riposo, G. Weichenberg, C. K. Duran and B. Fox, "Improving air force enterprise resource planning-enabled business transformation.," RAND Corp, 2013.
13. M. M. Ahmad and R. P. Cuenca, "Critical success factors for ERP implementation in SMEs," Robotics and Computer-Integrated Manufacturing, vol. 29, no. 3, pp. 104-111, 2013.
14. K. Al-Fawaz, Z. Al-Salti and T. Eldabi, " Critical success factors in ERP implementation: A review," in Proceedings of the European and Mediterranean Conference on Information Systems, Dubai, 2008.
15. E. T. Wang, S. Shih, J. Jiang and G. Klein, " The consistency among facilitating factors and ERP implementation success: A holistic view of fit," Journal of Systems and Software, vol. 81, no. 9, pp. 1609-1621, 2008.
16. S. D. Eppinger and T. R. Browning, Design structure matrix methods and applications, Cambridge, MA: The MIT Press, 2012.
17. R. E. Thebeau, "Knowledge management of system interfaces and interactions from product development processes.," MIT, 2001.
18. T. R. Browning, "Applying the design structure matrix to system decomposition and integration problems: A review and new directions," IEEE Transactions on Engineering Management, vol. 48, no. 3, pp. 292-306, 2001.
19. R. Helmer, A. Yassine and C. Meier, "Systematic module and interface definition using component design structure matrix," Journal of Engineering Design, vol. 21, no. 6, pp. 647-675, 2010.

Upcoming Events

Visit ‹http://www.crosstalkonline.org/events› for an up-to-date list of events.

STAREAST Software Testing Conference
1-6 May, 2016
Orlando, FL
https://stareast.techwell.com

The 38th International Conference on Software Engineering
14-22 May, 2016
Austin, TX
http://2016.icse.cs.txstate.edu

Agile Dev West Conference
Las Vegas, NV
5-10 June 2016
https://adcwest.techwell.com

2016 IEEE Symposium on VLSI Technology
Honolulu, HI, USA
June 14-16, 2016
http://www.ieee.org/conferences_events/conferences/conferencedetails/index.html?Conf_ID=18215

ICITS 2016: The 4th International Conference on Information Technology and Science
Tokyo, Japan
17-19 June 2016
http://icits.org

ISCC 2016- IEEE Symposium on Computers and Communications
27-30 June, 2016
Messina, Italy
http://iscc2016.unime.it

2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science
New York, NY
5-8 July, 2016
http://www.ieee.org/conferences_events/conferences/conferencedetails/index.html?Conf_ID=38877

20th International Database Engineering & Applications Symposium
Montreal, QC, Canada
July 11-13, 2016
http://confsys.encs.concordia.ca/IDEAS/ideas16/ideas16.php

2016 IEEE International Conference on Automation Science and Engineering (CASE)
Fort Worth, TX
21-25 August 2016
http://www.ieee.org/conferences_events/conferences/conferencedetails/index.html?Conf_ID=35762

ICSEA 2016: The Eleventh International Conference on Software Engineering Advances
Rome, Italy
21-25 August, 2016
http://www.iaria.org/conferences2016/ICSEA16.html

Mobile Radio Integration, Interoperability, and Frustration

No. Not that kind of mobile radio. It's not a secret that I am "mature." I graduated high school in 1973 and my parents gave me a choice of a graduation gift – they would either send me to the college of my choice, or they would buy me a new car. I opted for the 1973 Chevy Impala (and the deal that I would go to college locally). It turned out to be a wise choice on my part. I ended up dropping out of the University of Central Florida in 1974 to join the Air Force. My college degrees were many years in the future.

Full disclosure time. Fresh out of high school in 1973, I was not motivated to attend college. I needed to see the value of college. Seven years later, the advantages of a college education had become obvious to me. I returned to the University of Central Florida and got my B.S. in computer science (and commissioned a 2nd Lt.). Motivation really makes a difference! Mind you, I'm *not* saying students shouldn't go to college straight out of high school – I'm just saying without motivation, somebody can waste a lot of money.

I drove that 1973 Chevy into the ground – putting over 150,000 miles on it during the 12 years I owned it. It was a wonderful car – except for the low end AM/FM radio. 1974 was a bit late for 8-track tapes, and CDs were not invented yet. What I really wanted was a stereo AM/FM/cassette. The dealer wanted WAY too much to swap the radio out saying that the car was not wired for stereo. My goal (over my father's violent objections) was to replace the radio with a $79 "easy-install" upgrade.

The long story has been told before (see my BackTalk column in the February, 2009 issue of CrossTalk titled, "Two, Four, Six, Eight! Software and Systems – Integrate!") but the experience taught me several things:

- As an 18-year-old, I didn't give my dad enough credit for being smarter than me.
- Dropping the dashboard of a 1973 Impala is not a job for the inexperienced.
- You can burn through a box of 5-amp fuses during one afternoon, and then realize that maybe you should upgrade to a 10-amp.
- 10-amp fuses in a 5-amp circuit will cause wires to smoke and burn.
- Did I mention my dad was a heck-of-a-lot smarter than me?
- The enjoyment he got jumping my car six times over three days as I drained the battery gave him many happy memories to last over the next 38 years.
- The old AM/FM radio was a REALLY great radio and (once reinstalled with a few burnt wires replaced) worked well and faithfully for the next 12 years.

The lessons I shared with readers seven years ago were that a new radio integration into a 1973 Chevy was hard, and software systems integration is *really hard*. The topic of this issue is, "integration and interoperability" – which is similar, and I am so happy to say that, thank God, we have managed to make great inroads into solving the integration problem over the last seven years. And, if you believe that, I have swamp land in Florida to sell you, along with a great deal on the scrap from the soon to be demolished Brooklyn Bridge. I'll even (for a few extra dollars) throw in some soon-to-be-valuable shares of Enron. You think I can solve integration and interoperability problem in a (well-written) Backtalk column of 800 – 1,200 words? Not likely.

Integration and Interoperability are the dark, scary monsters that lurk in the nightmare of all systems developers. It's not enough to built a single software system – NO – we want to build and field a *system* of *systems*! Some of my most-referenced series of books in my office library include several books by Steve McConnell – *Software Estimation*, *Code Complete*, and *Software Project Survival Guide.*

If you are a software developer, *Code Complete* will help you get past "code and fix" programming. And if you are a manager, *Software Project Survival Guide* covers useful lessons gleaned from lots of experience – potentially saving you time, money and heartache. Or you could spend 20 years or so learning the lessons yourself.

In an awesome blog (entitled *Coding Horror* by Jeff Atwood) discussing Steve's work on estimation (see http://blog.codinghorror.com/diseconomies-of-scale-and-lines-of-code), it is pointed out that a program that is 10 times larger takes *much* more than 10 times the effort to write and integrate (the article refers to the "effort" – the addition of "to write and integrate" was added by me). How right Steve is!

One solution given is to keep projects "small." Unfortunately – in the DoD especially – "small" is not a reasonable option. When I started coding for the Air Force back in 1974, a large program would require *two* boxes of punched cards (2,000 cards per box, for you youngsters). One of the last projects I consulted on before returning to academia was estimated to top 90 million lines of code (LOC) when (if?) complete. And we don't even know how to measure code size – or even what we are counting. To quote the above-mentioned article, McConnell is quoted as saying **"The LOC measure is a terrible way to measure software size, except all the other ways to measure are worse."** I feel the urge to shout "Amen!"

There is a non-linear relationship between size and effort – and the exponential curve is NOT in our favor. We're not sure what we are counting in measuring size. We're unable to agree on standardized languages or operating systems. We can't even agree on whether to use 8 or 16-bit character sets for applications.

Is there hope for the future? Well, please note that the title of this journal is *Crosstalk, the Journal of Defense Software Engineering*. Not the Journal of Software Development, nor the Journal of Programming. In every class on software engineering I have taught over the last 30+ years, I have taught my students to strive to develop systems that meet four criteria: Reliable, Understandable, Modifiable and Efficient. If you don't have these four – well, it's not going to operate or integrate, is it? Different software engineers (plus various IEEE documents) have their own list – some different. My four-item list comes from *Software Engineering with Ada*, by Grady Booch.

Software Engineering skills are REQUIRED to have any chance of successful integration and interoperability, or to have any chance at all of successfully fielding a system of systems.

We've returned to where I was when I decided to return to college back in 1981. Now I'm motivated to improve.

Having troubles with integration and interoperability? Want to improve? Consider continuing your software engineering education. Maybe read a journal (for example, CrossTalk) or something.

David A. Cook, Ph.D.
Professor of Computer Science
Stephen F. Austin State University
cookda@sfasu.edu

CrossTalk / 517 SMXS MXDED
6022 Fir Ave.
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

CrossTalk thanks the above organizations for providing their support.