NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A SOFTWARE ASSURANCE FRAMEWORK FOR MITIGATING THE RISKS OF MALICIOUS SOFTWARE IN EMBEDDED SYSTEMS USED IN AIRCRAFT

by

Robert C. Ginn

September 2011

Thesis Advisor:	John Osmundson
Thesis Co-Advisor:	Janet Gill

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704–0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2011	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE A Software Assurance Framework for Mitigating the Risks of Malicious Software in Embedded Systems Used in Aircraft		**5. FUNDING NUMBERS** N/A	
6. AUTHOR(S) Robert C. Ginn			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943–5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

Malicious software represents a significant and growing threat to Department of Defense systems. Threats to airborne systems in particular can be characterized not by system vulnerability to Internet based exploits but rather by the risk posed by malicious code already present in the system's software. Although there are software techniques to detect and prevent certain types of attacks, a Systems Engineer has access to system level information and system design techniques that can quantify and in many cases mitigate the risks posed by potential malicious code present in the system. These techniques are especially applicable to malicious code in embedded airborne system although they can be applied to other systems that share certain traits.

This thesis provides an overview of the types of threat involved; techniques that can be used to detect malicious code in individual aircraft Weapons Replaceable Assemblies (WRAs); risks and mitigation strategies related to a generic aircraft software development process; system level techniques to prevent embedded malicious software from causing harm in aircraft; and a technique for documenting Software Assurance (SwA) arguments being made about the system and the individual WRAs.

14. SUBJECT TERMS Systems Engineering, Software Assurance (SwA), Malicious Software, Malicious Code, Exploit, Mitigation Strategies, Software Custody Chain, Goal Structuring Notation			**15. NUMBER OF PAGES** 119
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01–280–5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A SOFTWARE ASSURANCE FRAMEWORK FOR MITIGATING THE RISKS OF MALICIOUS SOFTWARE IN EMBEDDED SYSTEMS USED IN AIRCRAFT

Robert C. Ginn
Civilian, United States Navy
B.S., Lehigh University, 1983
M.S., Penn State University, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2011

Author: Robert Ginn

Approved by: John Osmundson, PhD

Janet Gill, PhD

Clifford Whitcomb, PhD
Chair, Department of Systems Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Malicious software represents a significant and growing threat to Department of Defense systems. Threats to airborne systems in particular can be characterized not by system vulnerability to Internet-based exploits, but rather by the risk posed by malicious code already present in the system's software. Although there are software techniques to detect and prevent certain types of attacks, a Systems Engineer has access to system level information and system design techniques that can quantify and in many cases mitigate the risks posed by potential malicious code present in the system. These techniques are especially applicable to malicious code in embedded airborne system, although they can be applied to other systems that share certain traits.

This thesis provides an overview of the types of threat involved; techniques that can be used to detect malicious code in individual aircraft Weapons Replaceable Assemblies (WRAs); risks and mitigation strategies related to a generic aircraft software development process; system level techniques to prevent embedded malicious software from causing harm in aircraft; and a technique for documenting Software Assurance (SwA) arguments being made about the system and the individual WRAs.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Many of today's (2011) systems are software intensive. This represents a threat to the safety and security of those who use such systems because:

- Software directly controls many safety critical system elements—there is often no "man in the loop"
- It is easy to write software that will fail in predetermined ways
- It is difficult to detect malicious code in software
- It is possible to compromise most existing software if it is accessible across a network.

Since embedded systems used in aircraft control many critical aspects of an aircraft (the engine, flight controls, navigation), malicious software has the potential to cause catastrophic system failures. Failure of software in a Full Authority Digital Electronic Controller (FADEC) is believed to be responsible for the crash of a Chinook helicopter resulting in deaths of 25 intelligence personnel and a four-person Special Forces crew. According to General Shepperd, when F-22 Raptors crossed the international dateline for the first time, they lost most of their systems:

> All systems dumped and when I say all systems, I mean all systems, their navigation, part of their communications, their fuel systems. They were—they could have been in real trouble. They were with their tankers. The tankers—they tried to reset their systems, couldn't get them reset. The tankers brought them back to Hawaii. This could have been real serious. It certainly could have been real serious if the weather had been bad. It turned out OK. It was fixed in 48 hours. It was a computer glitch in the millions of lines of code, somebody made an error in a couple lines of the code and everything goes.

A software failure in the Air Data Inertial Reference Unit (ADIRU) of an Airbus A330–300 caused Qantas flight 72, from Singapore to Perth to change attitude violently "throwing passengers around the cabin." This resulted in the injury of seventy passengers, ten of whom required hospitalization.

Although none of these incidents are believed to be the result of malicious code, they illustrate the risks posed by software in airborne systems. Had an attacker with access to the source code wished to cause problems such as these, they would have been able to—and the problems would almost certainly have been blamed on a simple coding error.

Writing software that is resistant to these threats is not taught as part of standard computer science curriculums. Further, even well written software provides no protection against malicious code that is part of a system's source code.

Malicious software built into the source code of a system is difficult to detect and can pass typical software testing procedures used for safety critical elements of airborne systems such as those mandated by the FAA's DO-178B. The skills required to write such code are widely available and programmers without these skills can using existing tools to add malicious code to software. Such tools are designed to insert malicious code in such a way as to be difficult to detect by both programmers and static analysis tools.

Much of the research today (2011) centers around how to prevent software that is connected to the Internet from being exploited and little information is available on techniques to prevent or detect malicious code already present in source code. Existing static analysis tools do a poor job of detecting malicious code and cannot provide assurance that no malicious code remains. While quite powerful, dynamic analysis techniques also lack the ability to guarantee that no malicious code has gone undetected.

This thesis breaks malicious code down into two primary categories: vulnerabilities in the system (where an attacker with access to the system can exploit it in real time) and pre-exploited software (where malicious code is already present in a system). The pre-exploited case is further broken down into two main categories: overt

malicious code (where the dangerous code is clearly visible in the source code); and covert malicious code (where the dangerous code is not obvious to a casual review of the source code—and may not even be in the source code but will nonetheless be present within the Weapon Replaceable Assembly (WRA) load image). The WRAs are also categorized based on how they are used within the system architecture. WRAs with no external (off aircraft) connections can only have potential vulnerabilities exploited by other WRAs.

The extremely limited external connectivity of many aircraft WRAs means that in most cases, potential vulnerabilities can be discounted. Further, specific safety related WRA and system level testing used for many WRAs within an aircraft prevent some types of malicious code embedded in the WRA.

The safety critical nature of many aircraft WRAs limits our ability to use more aggressive run-time detection and prevention techniques. As a result, these systems are ideal targets for covert malicious code. However, due to the safety critical nature of the WRAs, it is possible to leverage existing development requirements to obtain secure systems at a lower cost. Safety critical systems will tend to have a MIL-STD 882 hazard analysis performed that identifies which system elements can cause the most damage. Commercial requirements such as DO-178B have specific development requirements for each element that depend on similar risk analysis approaches (e.g., ARP 4145A).

Since attackers need pre-exploited WRAs to fail when the system is in use, and to not fail during WRA or system test (where the damage would be contained), the attack will depend on some type of “trigger” based information available to the malicious code in the WRA. This trigger will be used by the malicious code to determine when to fail. As Systems Engineers, we are able to prevent malicious code from causing harm by limiting the information it can receive. For example, if an aircraft engine controller tracked the number of hours it had been in use, malicious code might cause it to fail only after 200 hrs. A Systems Engineer can often prevent this type of information from being available to the WRA. If such malicious code is present in a WRA and it cannot determine how long the WRA has been in use, it has two options: always fail; or never

fail. In the “always fail” case, the problem will be detected during normal testing. In the “never fail” case we cannot tell if malicious code is present—but if it is present it does no harm. In practice things are not quite so simple due to the possibility of statistical based triggers.

Unfortunately, some WRAs require access to potential trigger information. For example, a flight management computer needs to know many pieces of potential trigger information (altitude, location). For WRAs where we are unable to limit information that could be used to trigger malicious code, the WRA is decomposed into its elements to reduce the amount of software that needs other, more complicated techniques to mitigate the risk. An overview of static and dynamic analysis techniques is presented—unfortunately these techniques are not able to guarantee the absence of malicious code (although dynamic analysis in particular is quite powerful). Thus such elements within these WRAs represent residual Software Assurance risk.

For programs where the software has not yet been developed, a basic software development process is discussed and analyzed yielding eleven places in the process where an attacker has an opportunity to insert malicious code. Mitigation strategies are then provided for each threat type.

Once the software baseline and toolset is believed safe to use, it is important to be able to detect any malicious changes to this baseline. To this end a technique using cryptographic hashes and digital signatures is presented that is able to identify all changes in the source code, libraries and tools used to build the system software. Two simple shell scripts to implement this approach on UNIX and UNIX-like systems are provided.

Finally, a formal method to document Software Assurance (SwA) cases based on Goal Structuring Notation (GSN) along the lines used for documenting System Safety cases is proposed.

LIST OF ACRONYMS AND ABBREVIATIONS

A/C	Aircraft
ADIRU	Air Data Inertial Reference Unit
ASIC	Application-Specific Integrated Circuit
B.S.	Bachelor of Science
Bi	Dual
BSS	Block Started by Symbol
CAS	Column Address Strobe
CD	Compact Disc
CIA	Central Intelligence Agency
CM	Configuration Management
CNC	Computer Numerically Controlled
CPU	Central Processing Unit
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
CVE	Common Vulnerabilities and Exposures
DAL	Design Assurance Level
DC	Direct Current
DRAM	Dynamic Random Access Memory
EEC	Electronic Engine Control
FADEC	Full Authority Digital Engine Controller
FPGA	FPGA – Field Programmable Gate Array
GCC	GNU Compiler Collection
GNU	GNU is Not UNIX
GSN	Goal Structuring Notation
GSN	Goal Structuring Notation
I/O	Input/Output
IC	Integrated Circuit
IDD	Interface Design Description
IP	Internet Protocol
JPEG	Joint Photographic Experts Group
MC/DC	Modified Condition/Decision Coverage
MMU	Memory Management Unit

MoD	Ministry of Defence
NVM	Non-Volatile Memory
OEM	Original Equipment Manufacturer
Opcodes	Bit pattern representing CPU operation
OS	Operating System
PC	Personal Computer
POD	Ping Of Death
RADAR	Radio Detection And Ranging
RAM	Random Access Memory
RAS	Row Address Strobe
RF	Radio Frequency
ROM	Read Only Memory
SLOC	Source Lines Of Code
SwA	Software Assurance
Uni	Single
UNIX	Not an Acronym, an Operating System name
USB	Universal Serial Bus
VDC	Volts DC
VHF	Very High Frequency (30–300 MHz)
VOR	VHF Omni-directional Range
W^X	Write exclusive or Execute
WRA	Weapons Replaceable Assembly

I. INTRODUCTION

A. BACKGROUND

1. Overview

Many of today's (2011) systems are software intensive. This represents a threat to the safety and security of those who use such systems because software implements safety critical functionality within many systems; it is easy to write software that will fail in predetermined ways; and it is possible to compromise most existing software if it is accessible across a network.

Software that fails in predetermined ways (malicious code) is difficult to detect and can pass typical software testing procedures used in safety critical elements of airborne systems such as those mandated by the FAA's DO-178B. The skills required to write such code are widely available and only the knowledge of how high-level code actually runs on the hardware is necessary to implement malicious code. Computer science curriculums include assembly language programming that provides such knowledge, as do many classes in electrical or computer engineering that include programming microcontrollers. Samples of such code are available on the Internet so the required skills can be acquired if desired. There are even contests to see who can write the best malicious code that can pass a security review (e.g., http://underhanded.xcott.com). Note that the capability is language independent, although different languages require different techniques.

Although there has been increased awareness of threats to software across a network interface, much of the software in use today contains vulnerabilities that make it possible for an attacker to exploit the software if it is connected to a network. Figure 1 was generated from data available from the Common Vulnerabilities and Exposures (CVE) security vulnerability database (http://www.cvedetails.com). This data represents

software bugs that can be used to exploit the system the software runs on. Although this data cannot be used to rate the safety of specific vendors,[1] it is clear that vulnerabilities are an ongoing problem.

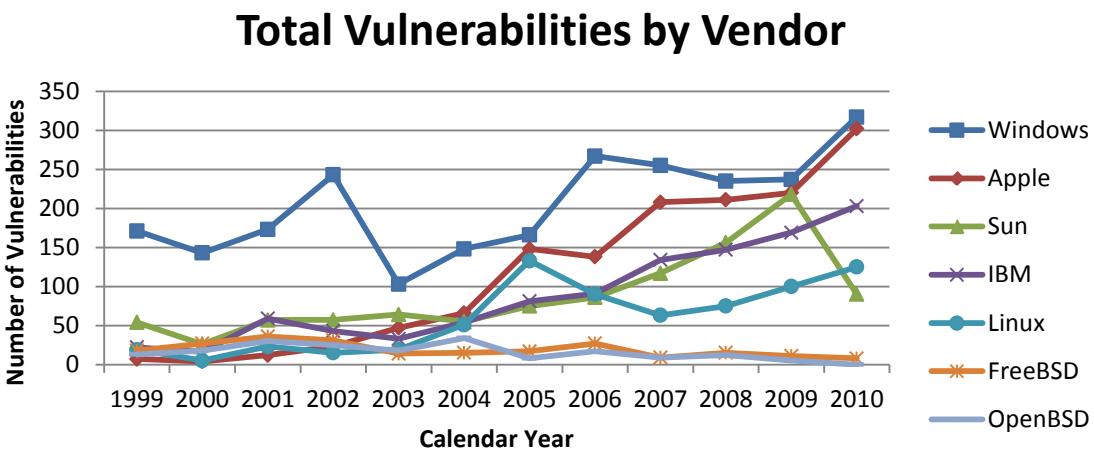

Figure 1 - Vulnerability Statistics

Writing software that can survive a deliberate attack requires skills that are not generally taught as part of a computer science degree. Stanford University offers a set of six courses in Advanced Computer Security; however these require not only a B.S. in Computer Science but also a background in security (http://scpd.stanford.edu/search/publicCourseSearchDetails.do?method=load&courseId=1284836).

At any given time, there are a number of known vulnerabilities in software that are not publicly disclosed nor fixed. One vulnerability research site listed 125 such vulnerabilities as of June 2011 (Zero Day Initiative, 2011). Some of these vulnerabilities were known for two years prior to being fixed by the vendor (Tipping Point, 2011).

There is a financial incentive for hackers to attack systems. Although the size of the underground economy is unknown, researchers and cybersecurity firms pay for

[1] The data not only groups software errors across product families (e.g. Windows OSes) but also includes programs that come with the operating system (e.g. Free Operating Systems such as Linux and BSD come bundled with hundreds of free applications) most of which may not even be installed.

security flaws. Greenberg (2010) states that Netragard (a cybersecurity firm) pays between $15,000 and $115,000 for a security flaw in the Apple Mac.

Some security flaws are used to steal information such as credit card information, bank account access information, etc. A Trojan Horse is a type of malware that pretends to be a useful program (and may actually be useful) while it steals information from the computer it runs on. Panda Security (2010) states that "Five years ago, there were only 92,000 strains of malware cataloged throughout the company's 15-year history. This figure rose to 14 million by 2008 and 60 million by 2010, which gives a good indication of the rate of growth." Of these instances of malware, 49% were Trojans overall and for new 2010 malware, 71% were Trojans.

Even if when vulnerability is known, it is not clear that a vendor will report it (and thus protect the product users). Vendors have a disincentive to report vulnerabilities. According to Telang and Wattal (2007), vendors lose on average $0.86 billion (in terms of stock price) on the day a vulnerability is announced.

An attack across a network could also compromise embedded software that will not be used in a networked environment. For example, Ragan (2011) reports that at least three government defense contractors (including Lockheed Martin, L-3, and an unconfirmed breach at Northrup Grumman) have been attacked through the use of SecurID technology. SecureID is a security token provided by RSA that allows employees to access their internal network securely (RSA 2011). Theoretically, without access to the security token, access is not possible due to the encryption used. However, a security breach at RSA in March allowed the attackers to penetrate the networks. RSA (2011) states that "we were able to confirm that information taken from RSA in March had been used as an element of an attempted broader attack on Lockheed Martin, a major U.S. government defense contractor."

2. Risks Posed by Malicious Code in Embedded Systems

a. In General

One question that might arise is "how much damage could malicious software do"? The answer to this is not as simple as a worst case failure mode. Instead, we must assume that the software will actively control hardware to yield the worst possible outcome. As an example, software that controls a pipeline may not seem to represent a significant threat, yet the damage can be enormous. According to Weiss (2000), the CIA implemented a program to sabotage plans and software that they expected the Soviets to steal. Safire (2004) reported on one example of this program where software used to control a pipeline was sabotaged with malicious code prior to its theft. ''The pipeline software that was to run the pumps, turbines and valves was programmed to go haywire,'' states Reed in Safire (2008), ''to reset pump speeds and valve settings to produce pressures far beyond those acceptable to the pipeline joints and welds. The result was the most monumental non-nuclear explosion and fire ever seen from space.''

Malicious code can also be used to sabotage industrial development. One recent (2010) example is the Stuxnet worm. The Stuxnet worm infected Windows PCs via USB memory sticks. Once present, it looked for a specific configuration of Siemens Programmable Logic Controller (PLC) units. The PLC units control motors and other hardware. If an exact match was not found, the worm did no damage (Fildes, 2010).

According to Symantec (cited in Fildes, 2010), nearly 60% of all Stuxnet infections were in Iran. President Mahmoud Ahmadinejad (quoted in Reuters, 2010), stated that enemies of Iran used computer code to make "limited" problems for centrifuges involved in uranium enrichment at some of its nuclear sites. "They succeeded in creating problems for a limited number of our centrifuges with the software they had installed in electronic parts." Nonetheless, it is believed that Stuxnet specifically targeted the model IR-1 centrifuges used at the Fuel Enrichment Plant at Natanz and destroyed one thousand (1,000) of the approximately nine thousand (9,000) in use at the site (Albright, Brannan, & Walrond, 2010).

According to Broad, Markoff, & Sanger (2011), the Stuxnet worm was "a joint American and Israeli effort to undermine Iran's efforts to make a bomb." The Dimona complex in the Negev desert duplicated the nuclear centrifuges Iran was using to enrich uranium. The worm had two major functions. "One was designed to send Iran's nuclear centrifuges spinning wildly out of control. Another seems right out of the movies: The computer program also secretly recorded what normal operations at the nuclear plant looked like, then played those readings back to plant operators, like a pre-recorded security tape in a bank heist, so that it would appear that everything was operating normally while the centrifuges were actually tearing themselves apart."

This type of targeted attack represents a serious threat. According to Ralph Langner (quoted in Broad, Markoff, & Sanger, 2011), Stuxnet represents a "new form of industrial warfare, one to which the United States is also highly vulnerable." In 2009, the U.S. Government admitted that Chinese and Russian spies had infiltrated systems controlling the Nation's power grid and left software behind that could destroy software infrastructure (Gorman, 2009).

b. Risks Posed to Embedded Airborne Systems in Particular

Airborne systems contain many critical systems. One of the most critical is the engine. In many aircraft, a computer called a Full Authority Digital Engine Controller (FADEC) controls the engines, not the pilot. A FADEC has "no form of manual override available, placing full authority over the operating parameters of the engine in the hands of the computer" ("FADEC," n.d.). According to (Safram, 2011), an Electronic Engine Control (EEC) manufacturer, "FADECs currently equip over 3000 commercial planes." It is important to note that although a simple software failure could result in an engine shutdown, malicious code in an EEC or FADEC could cause far more damage. Because the EEC or FADEC control all engine parameters, it is possible for them to cause the engine to be permanently damaged and in some cases, operate with reverse thrust. One method used by System Engineers to guard against EEC and FADEC failures is to use two systems in parallel, the idea being that if one fails, the other can

continue to operate the engine. However, in the case of malicious code, both controllers can be made to fail at the same time (i.e., the redundancy will not provide increased safety).

The Chinook's Full Authority Digital Electronic Control (FADEC) software controlled the helicopter's engines and is implicated in a 1994 crash of a Boeing Chinook helicopter that killed 25 intelligence personnel and a four-person Special Forces crew. A 1993 review of the FADEC software by EDS-SCICON found 485 anomalies after examining only 18 per cent of the software code. "Errors caused unexpected engine shutdowns, as well as surges in power that resulted in engines completely blowing out" (King, 2011). Although the crash was initially blamed on pilot error in 1995, a recent (2010) independent inquiry, commissioned by the British Government assigns blame to the FADEC. Since the original findings, additional documents have come to light including one "written on the day of the crash," which "contained a warning by IT experts and airworthiness assessors at MoD Boscombe Down that the Chinook Mk2 should not be in the air" (King, 2011).

In another example, ten F-22 Raptors suffered a navigational failure during their first foreign deployment (Johnson, 2007). According to General Shepperd (as stated in Roberts, 2007), when F-22 Raptors, on their way from an Air Force base in Hawaii to an Air Force base in Japan, crossed the international dateline they encountered a serious software error.

> All systems dumped and when I say all systems, I mean all systems, their navigation, part of their communications, their fuel systems. They were—they could have been in real trouble. They were with their tankers. The tankers - they tried to reset their systems, couldn't get them reset. The tankers brought them back to Hawaii. This could have been real serious. It certainly could have been real serious if the weather had been bad. It turned out OK. It was fixed in 48 hours. It was a computer glitch in the millions of lines of code, somebody made an error in a couple lines of the code and everything goes.

Problems in one subsystem can affect the system as a whole in dangerous ways. For example, a computer failure injured over 70 people including "14 people with serious, but not life-threatening, injuries" when Qantas flight 72, from Singapore to Perth, caused the Airbus A330–300 to suddenly change altitude. "… the sudden drop sent the plane into a nosedive, throwing passengers around the cabin" (Packham, B., Dunn, M., 2008). According to the Australian Transport Safety Bureau (2008), the incident was the result of a software failure in an Air Data Inertial Reference Unit (ADIRU). Although the ADIRU detected and reported an internal fault (causing the auto-pilot to disengage), two minutes later "ADIRU 1 generated very high, random and incorrect values for the aircrafts angle of attack." These faulty angle of attack values caused the flight control computer to immediately drop the nose of the aircraft by 8.5 degrees (although the auto-pilot was disengaged, the fly-by-wire system still overrode the pilots). The ADIRU continued to generate "random spikes" of lesser magnitude. Another ADIRU failed in a different Airbus A330–303 two months later however the pilots were able to turn of the unit before it could cause harm.

Although these problems are believed to be the result of accidental coding errors, they illustrate the risks posed by malicious software in airborne systems.

3. Embedded Software used in Airborne Systems

a. Software Safety Considerations

Unlike general purpose software, software intended for use in airborne systems has specific safety considerations. Because software in certain parts of the aircraft has the ability to result in loss of life, both the process employed during software development and the testing requirements are necessarily more involved than software intended for some other purposes (e.g., a video game, a word processor, …). Although these safety considerations do not eliminate software assurance issues, in many cases the steps required for software safety issues can be leveraged to reduce the software assurance effort. For example, the Configuration Management (CM) approach used for software safety related issues can be leveraged.

One common commercial approach to software safety is the FAA's DO-178B document.

b. DO-178B Requirements

DO-178B (Software Considerations in Airborne Systems and Equipment Certifications.) is a widely used commercial standard for developing airborne software. The document specifies a set of Process Objectives that should be complied with when developing the software. The specific Process Objectives to be met are based on the results of a safety assessment. This assessment results in a Design Assurance Level (DAL) assignment from level A through level E as follows:

Level A – Catastrophic

Level B – Hazardous/Severe-Major

Level C – Major

Level D – Minor

Level E – No Effect

Level A implies that the aircraft will be unable to continue flying and unable to land safely. The failure of a level A box can be considered likely to result in the death of one or more people. Level B implies that death may result but it is likely that serious injuries will be sustained. Level C implies discomfort with the possibility of serious injury. Level D implies that some inconvenience may occur and E (as named) will not affect the operation of the aircraft (RTCA, Inc, 1992).

Because of the severity of WRA failures with Level A through Level C software, DO-178B requires that every line of the software be executed as part of the acceptance test suite (statement coverage). Level B and Level A add progressively more rigor to this process by requiring extra testing of all the branch points in the software. If the software is thoroughly tested, with every possible line being executed and the software behaved normally, it is assumed that the software is reliable. The implicit

assumption being made by DO-178B is that any software errors are accidental. Note that if an attacker wants to insert malicious code into a Level A through Level C box that is tested according to DO-178B, they must either insert the malicious code after testing or they must write the malicious code in such a way that it will work normally during testing yet fail in operation. This second approach is discussed later in the section titled "Trigger Concept" (p. 54).

B. PURPOSE

The purpose of this thesis is to develop strategies that can be used to generate software releases that are safe and secure for their intended use. Specifically, the software must work safely and reliably and be free of "exploitable vulnerabilities."

In order to achieve this, we need to have high confidence that either there is no malicious software present in the WRA, or that if malicious code is present, that it cannot do any damage. Note in particular that in most cases we are not trying to guarantee that no malicious code is present, but instead have simplified the problem by requiring that if malicious code is present that it can do no damage. If malicious code is present in the system but never does anything malicious, the system will operate correctly. This approach requires the Systems Engineer to be aware of and control the environment in which the software – that potentially contains malicious code – operates.

In addition, we need to have confidence that the WRA software is not exploitable, or if it is exploitable, that there is no way for an attacker to exploit it. Again, this second simplification means that we may use software that might contain vulnerabilities but that the Systems Engineer must assure specific criteria are met in order to use the software safely.

Although there are many techniques to find actual and potential software errors, finding (or not finding) errors does not tell us anything about how much (if any) malicious code remains in the software. We need to be able to state with some degree of certainty that the software is safe to use in our specific environment.

C. RESEARCH QUESTIONS

Based on my research, I will answer the following questions in this thesis:

- How can a System Engineer mitigate the risks posed by malicious code in embedded systems used in aircraft?
- What methods can we use to detect the presence of malicious code in embedded systems used in aircraft?
- How can we reduce the risk that malicious code will be inserted into the source code for embedded systems used in aircraft?
- How can we prevent malicious code in embedded systems used in aircraft from causing harm?
- How can we maintain the integrity of the software in an embedded system?

D. BENEFITS OF STUDY

This study allows a Systems Engineer to bound the problems associated with malicious code in embedded systems in aircraft. By more fully understanding the issues, risks, and threats, a Systems Engineer can determine which risks are acceptable and which must be mitigated. This thesis also provides an approach for categorizing the risks along with specific approaches that can be used as a framework to mitigate each of the risk categories. Threats during development and mitigation strategies to mitigate these threats are also presented. Methods of verifying that a software codebase is unchanged—and identifying any changes is presented. Finally, a technique to document the Software Assurance (SwA) effort is presented.

E. SCOPE AND METHODOLOGY

This section describes the scope and methodology for this thesis. The scope specifies the specific areas researched and the applicability of the thesis to specific concerns related the malicious code. The methodology describes the approach used to answer the research questions

1. Scope

The analysis presented here is limited to embedded software used for airborne applications. Although the framework can be extended to other types of systems, many of the assumptions made are specific to aircraft systems and would need to be re-visited for other types of software.

2. Methodology

The main research question was broken down into four sub-questions. Each of these questions was then independently researched. In order to answer each of the research questions, the same basic approach was followed:

- Gather background information – Even before performing literature searches on the specific question, it was necessary to understand certain background material. For example, it was necessary to understand computer architectures prior to researching memory segment layouts and types of malicious software attacks. Similarly, it was necessary to understand basic software development processes before researching development processes related to airborne software development.
- Literature Searches – Once the background was understood, it was necessary to search the literature for existing research on both the scope of the problem and current approaches to dealing with the problem. For example, searches were performed to determine current (2011) approaches to detecting malicious code.
- Analysis – Based on the results obtained from the literature searches, it was necessary to analyze the results to determine which elements were

adequately covered and which were missing. For example, it became clear that existing techniques to locate malicious code did not yield complete or even quantifiable coverage results.

- Alternate approaches – When specific shortcomings were detected in the available approaches, alternate approaches were developed and presented. For example, the method of blocking "trigger information" was developed as an alternate approach. In some cases, alternate techniques from other disciplines were adopted such as the use of Goal Structuring Notation (GSN) for documenting Software Assurance (SwA) cases.
- Synthesis – Elements obtained from literature searches were combined with elements developed as parts of alternate approaches to provide an overall approach to parts of the overall problem. The overall approach to mitigating malicious code in a system and the method of maintaining trust in the software after verification are examples of this technique.
- Testing – In some cases, it was necessary to test an approach. For example, the simple scripts to generate and verify a cryptographic "snapshot" were tested to verify that they operate correctly.

Since there was significant overlap in the results, this thesis presents the results in terms of: the threat; the model used to perform the analysis; mitigation strategies; gaining and maintaining software integrity; and documenting the Software Assurance (SwA) process rather than ordering according the research questions. It is hoped that this ordering will be easier to follow and more useful to a System Engineer.

II. OVERVIEW OF SYSTEM PROBLEM SPACE

A. NOTIONAL SYSTEM ARCHITECTURE

For the purposes of this thesis, a generic system architecture for airborne systems and use this as a basis for understanding the types of threats, vulnerabilities, and mitigation strategies that apply to the problem. This architecture will initially be considered in terms of the physical architecture comprised of Weapons Replaceable Assemblies (WRAs)—(aka Boxes) and their connections. Figure 2 shows a notional generic physical architecture. WRAs are categorized as either external or internal depending on whether they have an interface that crosses the system boundary. Interfaces between WRAs are categorized as either bidirectional or unidirectional depending of possible information flow across the interface. It is important to distinguish between interfaces where information is only *intended* to flow in a single direction and interfaces where information is only *capable* of flowing in a single direction. A unidirectional interface is one where information physically cannot flow in two directions, regardless of intended use (e.g., an RF transmitter without a corresponding receiver).

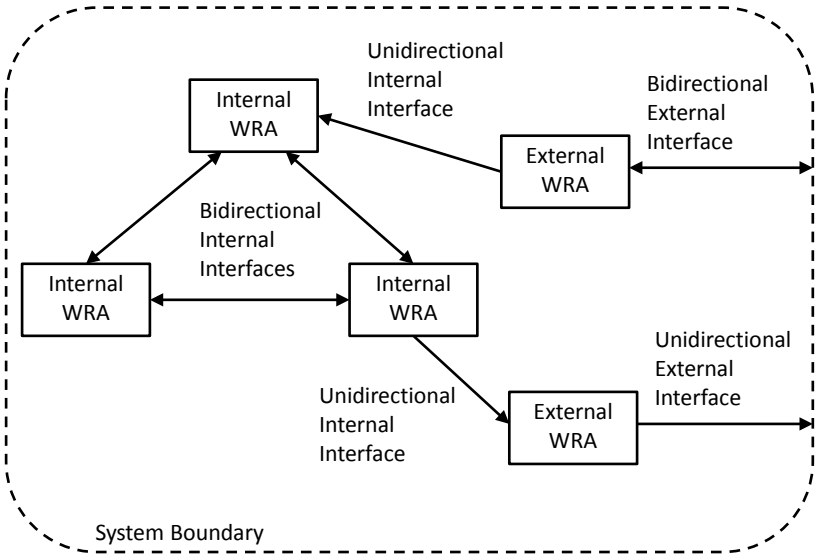

Figure 2 - System Architecture (Physical)

Interfaces where it is physically possible to send any information in both directions will be treated as bi-directional interfaces. This will be the case even when the logical interface would be considered a uni-directional interface. Consider a simple (logically) unidirectional parallel port interface. The computer will set eight bits on the output and then strobe a transfer line to tell the receiver that the data is ready. Although this might appear to be a uni-directional channel, the receiver controls an output line that is an input to the computer. This output line is intended to tell the computer whether it can setup for the next eight bits. However, there is nothing to physically prevent the receiver from switching this line high and low to transmit information back into the computer. In practice, a parallel port also includes other lines that are intended to pass information back such as the "out of paper" line. Special cables have been built that allow this "uni-directional" interface to act as a bi-directional network connection. These are even available commercially (http://www.nullmodem.com/LapLink.htm).

The architecture can also be thought of in terms of Computer Software Configuration Items (CSCIs). Since it is possible to load multiple CSCIs onto the same hardware, it is not possible to analyze input channels based solely on the CSCI architecture. Nonetheless, the CSCI architecture within each WRA can be useful when we wish to assess the pedigree of the software in each CSCI.

In some cases, a WRA consists of multiple hardware modules that contain software. In these cases, the WRA can be sub-divided into sub-WRAs (where each sub-WRA contains a processor) and the inputs for each sub-WRA can then be analyzed (as was done with the WRAs in the system).

B. SYSTEM LEVEL THREATS

This section discusses external threats to an aircraft in flight, discussion of attempts to insert malicious code into the software baseline during development and maintenance are covered in Chapter V (p. 65) and Chapter VII (p. 78).

In flight, an aircraft has a limited number of inputs. Many of these inputs are not capable of being used for an attack. For example, most radio communication represents

analog signals. The radio that a pilot uses to talk to the tower has a voice interface (note that we are concerned with attacks on the aircraft systems here, not potential "social engineering" attacks on the pilots—even though the pilots can be considered to be part of the overall system). A VHF Omni-directional Range (VOR) navigation system uses a phased analog signal to indicate bearing. Although an attacker could modify the VOR transmitter, the aircraft system would simply read the transmitted value, and not be damaged per se. The same logic applies to RADAR systems. Of course, RADAR systems that receive embedded digital data might be vulnerable since unexpected data sequences could theoretically exploit a vulnerability in the receiver. This is unlikely but each RADAR receiver must be evaluated to make sure that no combination of inputs could result in a compromise.

This is not to say that purely analog inputs can be considered safe. In the late 1960s and early 1970s, a hacker named John Draper alias Cap'n Crunch used a toy whistle from a Captain Crunch cereal box to place free phone calls. This was possible because AT&T used a 2600 Hz tone to signal that a trunk line was available and ready to accept a call. This signal disconnected one end of the trunk so Draper could act as the operator. The toy whistle happened to emit this tone ("John Draper," n.d.).

Thus to be certain that the aircraft will be free of potential attack vectors, every input must be scrutinized. A method of formalizing this process is presented in Chapter VIII.

Although the Internet may appear to represent the largest potential threat, as a general rule, aircraft system Avionics are not connected to the Internet so external attacks via the Internet are generally infeasible. However, this may not always be the case. When Boeing designed the 787 Dreamliner, their design "for the first time, connects a passenger Internet network with networks that control the plane's navigation and maintenance systems" (Zetter, 2008). This implies that a passenger laptop could theoretically affect the aircraft control systems.

According to Kuhl (2008), Boeing stated that "the problem was fixed before the FAA issued its warning." However, this is not really adequate. Jonathan Ezor (as quoted in Kuhl, 2008) states that "Any time you have a physical connection (between computer networks), there is a possibility someone could bridge from one to the other"

C. WEAPONS REPLACEABLE ASSEMBLY (WRA) THREATS

WRAs can be attacked from outside of the system boundary or from inside the system boundary. In some cases, a WRA can be directly attacked, and in others, due to a lack of physical access, only indirect attacks are possible. We'll refer to an attack that originates from outside the system boundary as a direct or first level attack. If an external WRA must be first be compromised in order to attack another (usually internal) WRA, we'll refer to the attack as an indirect or second level attack. Obviously each WRA, once compromised, could be made to attack another WRA that could not be directly attacked due to the lack of a physical interface. This is shown in Figure 3.

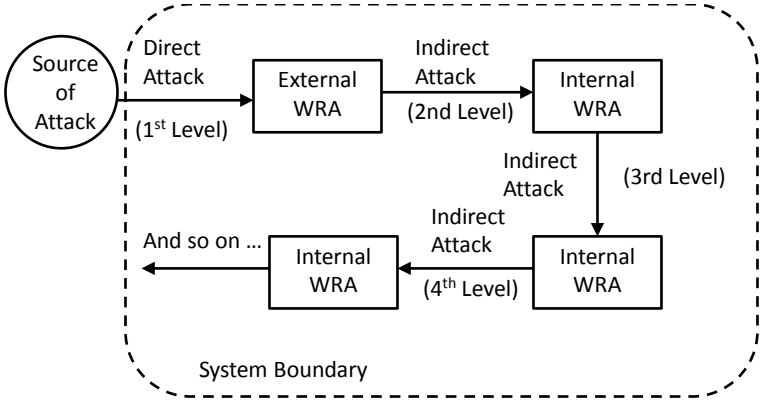

Figure 3 - External Direct and Indirect Attacks

In the case of an external attack, the externally accessible WRA must be vulnerable to the specific attack being used. Furthermore, in the case of a multi-level attack, not only must the first WRA be vulnerable, but each WRA in the chain must be vulnerable. In other words, to attack an internal WRA, an attacker must exploit a vulnerability in the external WRA and then exploit a (probably different) vulnerability in the internal WRA. Thus multi-level attacks are significantly more difficult than single

level attacks. As a consequence, it may be significantly less effort for an attacker to simply insert malicious code directly into the internal WRA, either in the source code or by replacing the WRA image with a malicious image. We'll refer to these WRAs as pre-exploited.

Attacks may also come from within the system boundary. If a pre-exploited WRA contains malicious code, it can do direct damage (by making that WRA behave maliciously or simply fail) or attack another WRA. This is shown in Figure 4.

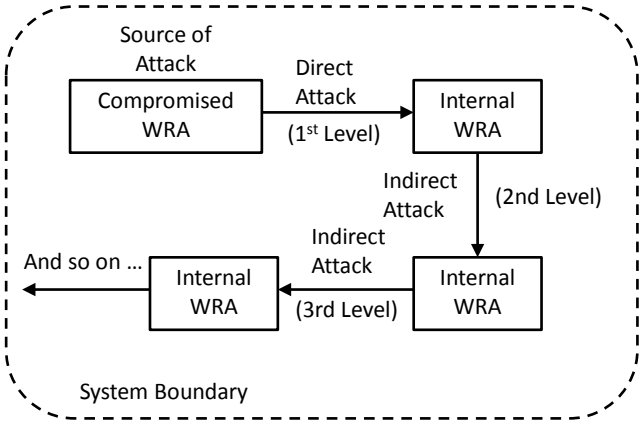

Figure 4 - Internal Direct and Indirect Threats

We'll use the same terminology for direct and indirect attacks when the source of the attack is an internally compromised WRA.

D. ANALYSIS

Airborne systems have a number of architectural characteristics that limit the ability of an external attacker to compromise the WRAs that comprise such a system. An attacker needs both a vulnerability and a physical connection to a WRA in order to successfully attack it. Further, this physical connection must be, at a minimum, uni-directional allowing information to flow from the attacker to the WRA. It is important to realize that it is not necessary for a bi-directional interface to exist. If the goal of an attack is to damage the system, no response from the compromised WRA is required. As

an example, consider the “Ping Of Death (POD).” This is a specially formatted Internet Protocol (IP) message that is sent to a computer that causes the computer to crash (“Ping of death,” n.d.).

Not all WRAs used in airborne systems are vulnerable to attack (although they may contain malicious code). Some WRAs only provide an output channel (e.g., altitude sensors) and in some cases this is implemented with a physical interface that only passes information out of the WRA. In these cases, an attacker will not be able to attack the WRA during use. Even when an input channel is present, it may not represent a vulnerability. Consider the wing de-icing system in Figure 5;

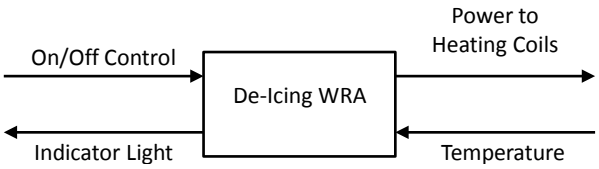

Figure 5 - De-Icing System

Although there are two inputs to the WRA, the On/Off control may be implemented as a discrete (e.g., 0 VDC or 5 VDC). The Temperature input may be the output of a thermocouple and provide a linear voltage (e.g., 0–12 VDC) as an input. In this case, even though there are two inputs to the WRA, it is likely that neither is suitable as a way to attack the WRA. In the case of the on/off switch, the WRA can be expected to survive having the pilot turn the system on and off, even rapidly, without causing a problem. In the case of the temperature input, this originates from hardware such as a thermocouple that an attacker will not be able to manipulate in real-time – even if such manipulation would be capable of compromising the WRA. Note that although it is unlikely that simply turning the system on and off quickly would be capable of compromising it, it is at least theoretically possible (such manipulation might, for example, overflow the interrupt stack). However, as shown later in “Attacker Requirements” (p. 53), this type of attack will not usually be acceptable to an attacker.

Because it is difficult for an attacker to compromise an airborne WRA using an external attack, an attacker could choose to "pre-exploit" the WRA by inserting malicious code into the WRA prior to its use. Since there are many ways for an attacker to insert malicious code into a WRA, and many of these methods are relatively easy to use, this thesis will concentrate on this type of threat. The CVE – Common Vulnerabilities and Exposures website provides a compendium of vulnerabilities that are relevant to WRAs connected directly to the outside world via the Internet (http://cve.mitre.org/). For other types of input, each input must be scrutinized for its potential to exploit a WRA.

Unfortunately, there are many places an attacker can hide malicious code within an airborne WRA. The WRA contains one or more Computer Software Configuration Items (CSCIs). Each of these contains executable code and data. The executable code consists of directly translated source code plus:

- Library code that the source is linked to
- Glue Code inserted by the compiler
- Anything in the memory of the WRA that the program can be "tricked" into transferring control to.

THIS PAGE INTENTIONALLY LEFT BLANK

III. OVERVIEW OF WEAPONS REPLACEABLE ASSEMBLY (WRA) PROBLEM SPACE

A. ENVIRONMENT MODEL USED

Today (2011), within each WRA that can potentially be compromised through the use of malicious code, there is at least one Central Processing Unit (CPU) and one physical memory space. These may be implemented as multiple components or integrated into a single chip (as could be the case for a microcontroller, ASIC, or FPGA application). A simple model of a WRA is shown in Figure 6.

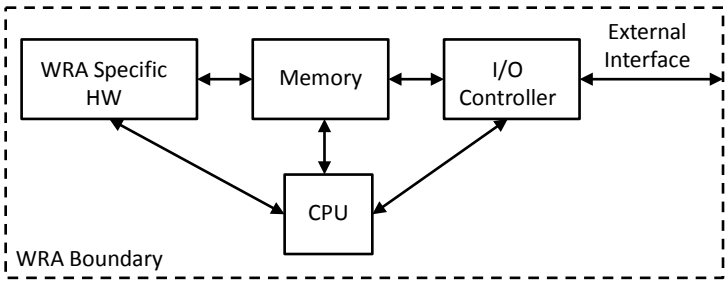

Figure 6 - Simple WRA Model

Some WRAs may contain multiple CPUs and physical memory spaces. When CPUs and Memories are isolated within the WRA, each can be treated as a set of WRAs in the sense that the external sub-WRA would need to be vulnerable for an attacker to successfully attack an internal sub-WRA. This model is shown in Figure 7.

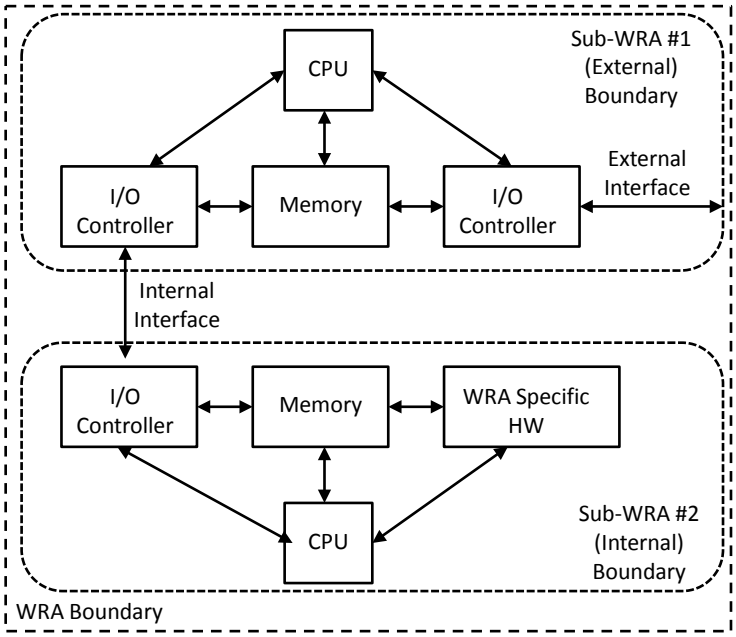

Figure 7 - WRA Model with Multiple Sub-WRAs

There is also a more complicated case where multiple CPUs can address the same physical memory. Execution threads are discussed are below and this case can be treated as a single CPU with multiple execution threads.

1. Memory Spaces

One key aspect of WRA memory usage with regards to malicious code involves how the physical memory is utilized. Three main models are considered: a Von Neumann architecture; a Harvard architecture; and a WRA that includes some type of memory management hardware that limits access to a smaller segment of physical memory. For our purposes, the key difference between these has to do with whether or not the data space can be "executed" by the CPU. Figure 8 shows an example of memory space utilization inside a WRA. The memory is typically divided into five segments:

- Text – this holds the compiled executable program that the WRA is executing. In general, this does not change during execution unless an error has occurred or there is malicious code involved. Although such

"self modifying code" can be written, it is typically not allowed for airborne systems

- Data – this holds initialized data for the program that the WRA will run. Its contents may or may not be altered by the program during execution

- BSS – this holds uninitialized data for the program the WRA will run. Its contents will be altered by the program as it runs.

- Heap – this holds dynamically allocated memory. The Operating System (or Executive) will allocate data as requested by the WRA program during execution.

- Stack – this holds temporary data used by the program as it executes. Typically this holds local variables associated with functions and procedures as well as keeping track of return locations as one function or procedure calls another function or procedure.

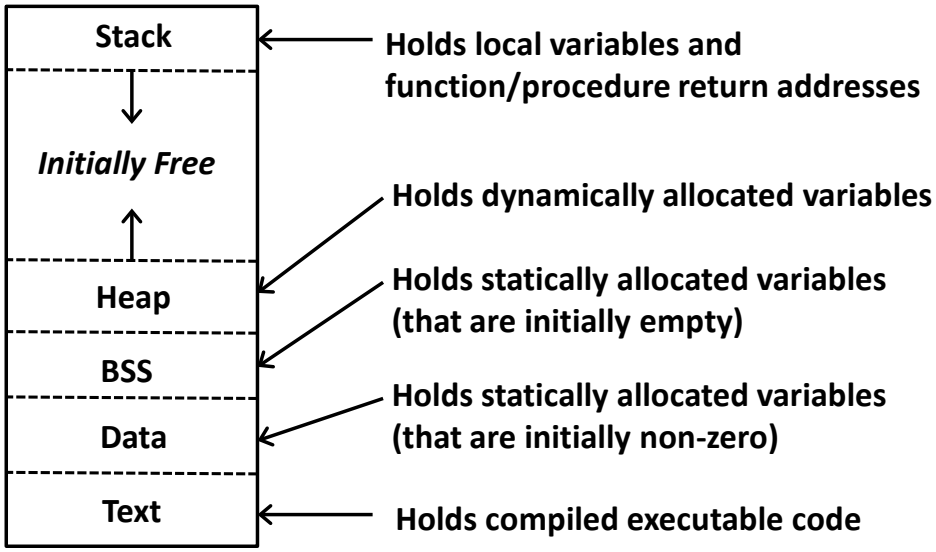

Figure 8 - Memory Segments

The dashed lines in between the memory segments in Figure 8 may represent either an actual physical boundary or simply a design/run-time choice made by the programmer for each segment of memory. In the case of a Von Neumann architecture, there is nothing to prevent executable code from being stored in any of the segments. In the case of a Harvard Architecture, the memory containing the executable code is physically separate from the data, BSS, heap, and stack memory. In the case of a WRA that has a Memory Management Unit (MMU), the sections of memory may be separated physically by programming the MMU to limit access outside of certain addresses for certain purposes. For example, the MMU could be programmed to allow read access but not write access to the code space. It could allow the CPU to read executable instructions from the text segment but not from the data segment. A typical protection scheme would be to specify that segments are W^X (Write exclusive or Execute) which means that any memory segment that the CPU can read executable instructions from cannot be written.

These techniques were implemented primarily to limit the possibility of accidentally executing data or modifying the executable software – it is possible for an attacker to get around each case. Neither limiting segment access and use nor putting the software into Read Only Memory (ROM) provides adequate protection against malicious code exploiting a vulnerability. For example, in (Checkoway, Halderman, Feldman, Felten, Kantor, & Shacham, 2009), the authors demonstrate the use of Return-Oriented Programming to execute arbitrary code on a Sequoia AVC Advantage voting machine utilizing a Harvard Architecture. They accomplished this without modifying the ROM based executable code and introduced the malicious code to the machine using a modified voting card. Thus, even though they did not change the executable program embedded in the voting machine's ROM, and the system only executed instructions that were in ROM, they were still able to execute any program they wanted through manipulation of the stack.

2. Execution Threads

Although a WRA may contain a significant amount of software, until a Central Processing Unit (CPU) actually reads and executes the software, the software is nothing more than data stored in memory. Consider the section of WRA memory shown in Figure 9. The same section of memory is shown twice, with the left side showing the bytes that are contained in each address, and the right side showing how the memory will be used.

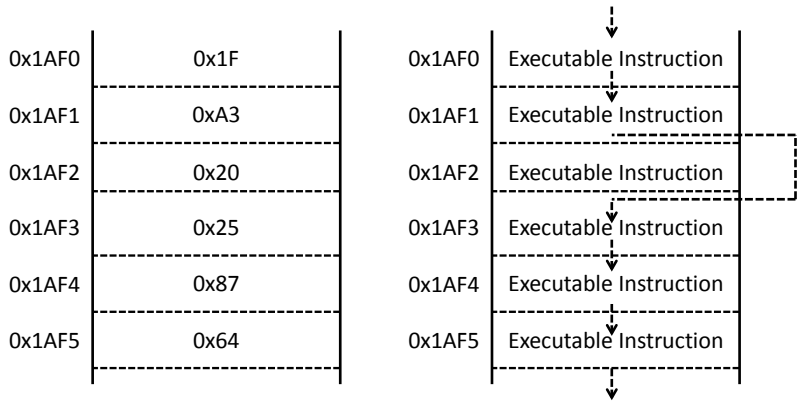

Figure 9 - Data vs Executable Instructions

Note that we cannot tell by examining address 0x1AF0[2] whether the byte (0x1F in this case) represents data, an executable instruction, or part of an executable instruction. There are several reasons for this. First, certain processors mix instructions and data. Second, some processors use a variable number of bytes per instruction. A simple instruction (such as to clear a register) might be a single byte while a complex instruction (such as an indirect memory reference) might use many bytes.

If the byte is intended to be read by the CPU as an instruction and executed, then it is an instruction, otherwise it is data. Also note that it does not matter where the byte is stored (i.e., Text, Data, BSS, Heap, or Stack segment). If it is loaded into the CPU as an instruction, then it is an instruction and will be executed. In a typical

[2] The leading "0x" indicates that the number is in hexadecimal format.

scenario, the CPU will read an instruction, execute it, and then read the next instruction. Certain instructions cause the CPU to read the next instruction for some location other than the next sequential address. A trivial example is shown on the right hand side of Figure 9. The flow of instructions that are actually executed represents the execution thread. This example shows an execution thread that contains a branch instruction in that some instructions are skipped (rather than simply executing all instructions in order).

The execution thread need not simply execute or skip instructions, it can also go to another location in memory and then return. This behavior is shown in Figure 10. Instead of proceeding directly from address 0x100 to address 0x101, the execution thread goes from 0x0100 to a subroutine at address 0x153. When the subroutine is complete, then execution continues at location 0x101. In order to return to the next location in memory past the location that the subroutine was called from (in this case 0x101), the CPU needs to store the location to return to somewhere in memory. In general, this is stored in the memory space allocated to the stack. Note that subroutines can call other subroutines so a number of memory locations must be available to hold these return addresses.

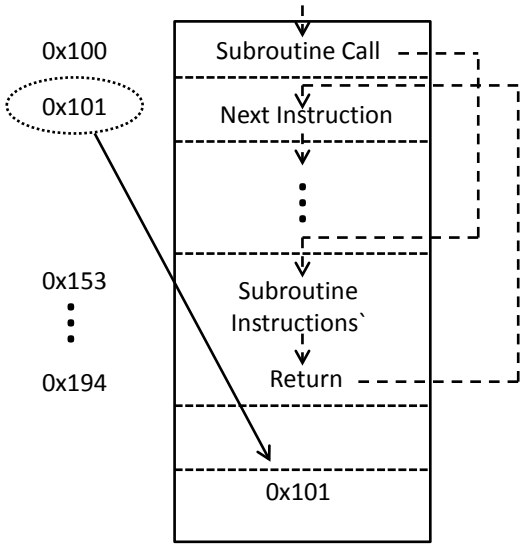

Figure 10 - Subroutine Call

As can be seen, the execution thread traces the path through memory showing which addresses in memory are executed and in what order. In addition, it should be clear that unless the execution thread traces through an address, the instruction at that location will not be executed.

Since bytes in memory are just data until executed, unless the execution thread passes through the malicious code, the malicious code cannot do any harm. Figure 11 shows a section of memory that contains malicious code and an execution thread that bypasses this code. Since the malicious code is not executed, it cannot do any harm.

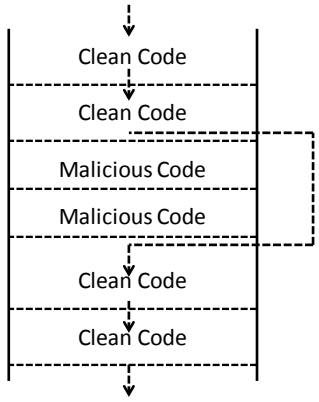

Figure 11 - Inert Malicious Code

Since the WRA operates correctly most of the time (otherwise it would be immediately obvious that the WRA was broken), only a small percentage of the bytes contained in the WRA will cause malicious behavior. Thus, in order to cause a problem, an attacker must make sure that the malicious code (data stored in memory that will do something malicious) is loaded into the CPU as an executable instruction.

B. VULNERABILITY THREATS

A vulnerability associated with software in a WRA represents an error in the software that allows an attacker to insert malicious code into the WRA during operation. These are generally the result of a programming error based on an assumption that the input will never fall outside of some expected range (although no mechanism is in place

to enforce this behavior). As long as the inputs fall within the expected range, the program will operate normally. By deliberately supplying the program with illegal input values, an attacker can cause the program to fail, often in predictable ways.

Each vulnerability represents an opportunity for an attacker to exploit the software. Note that the vulnerability may exist due to a programming error or may have been deliberately introduced by an attacker. This concept is show in Figure 12.

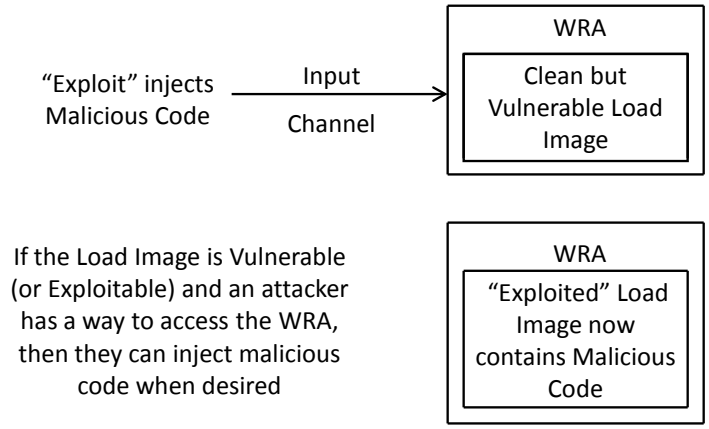

Figure 12 - Injected Malicious Code

One common type of vulnerability involves the size of data buffers allocated within the software. A data buffer refers at an array of values and has a fixed size[3]. If the program copies user input into the data buffer but does not verify that the amount of data copied fits within the buffer, a buffer overflow can occur. In practice, this means that an attacker can have data is written into memory outside of the buffer. According to Erickson (2003, p. 23) if enough extra data is written past the end of the buffer, the memory being overwritten will eventually be in the "stack" segment. As discussed in the subroutine call overview, the stack segment holds the address that the execution thread should continue at when the subroutine ends. Figure 13 illustrates a stack based type of buffer overflow attack. In this example, the attacker uses malicious code followed by the address of the buffer being overwritten as the data written into the buffer. When the

[3] Arrays can also be dynamic, but at any point in time they have a fixed size.

current subroutine ends, instead of returning to address 0x101, execution continues at address 0x357 (the address of the array). Since the array actually contains executable malicious code, the malicious code now has control.

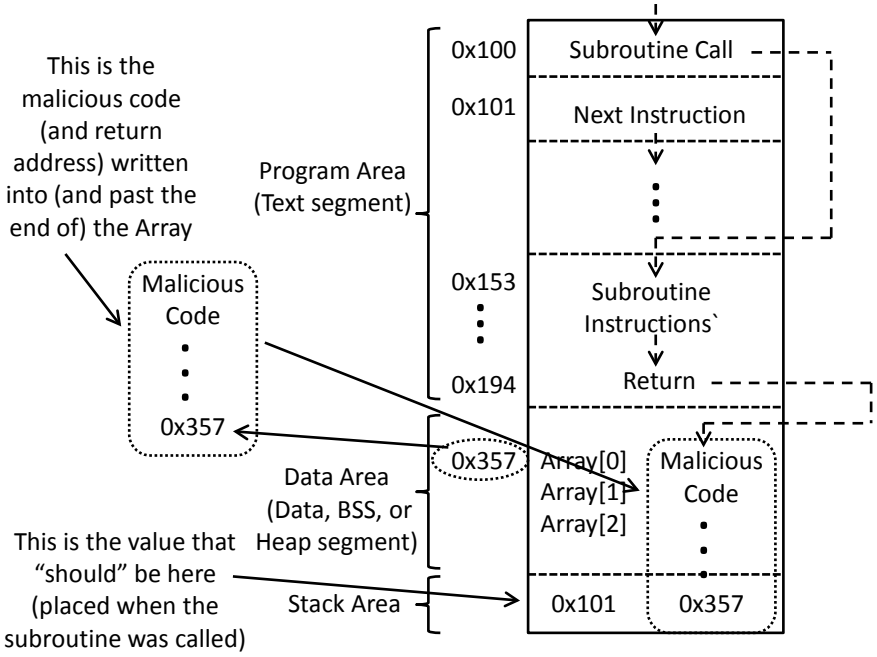

Figure 13 - Stack Based Buffer Overflow

There are many variations of this type of attack but the details are beyond the scope of this thesis.

There are many examples of this type of attack. Microsoft (2004) describes an interesting version of this attack in a security bulletin where the malicious code was inserted into a JPEG image file. Because the Microsoft code assumed that the file size information was correct in an image file, they only allocated a buffer large enough to hold the reported image size, not the actual image size. By deliberately corrupting the image size information in the file, attackers were able to take advantage of this programming error. When the image size information was set to a smaller value than required, the "image data" overwrote the return address and caused a jump to the

malicious code. Because programs such as Microsoft Outlook® used the broken routine to display images, it was possible for an attacker to compromise a Windows machine simply by sending an e-mail with an attached image file. Outlook's automatic preview feature tried to display the image and automatically transferred control to the malicious code embedded in the image.

Since these vulnerabilities are often easy to overlook and if found can be mistaken for simple programming errors, an attacker may choose to maliciously insert such a vulnerability so that it can later be exploited. However, in the case of embedded airborne systems, only WRAs with an external interface (input channel) are directly vulnerable. It is possible for an attacker to attack a WRA with an external interface and then use that WRA to attack a WRA with only internal interfaces, but as mentioned this type of multi-level attack is much more difficult as it requires knowledge of more than one WRA. Note also that the attack requires a successful attack on any intervening WRAs as well.

C. EMBEDDED MALICIOUS CODE THREATS

Four types of malicious code are considered for embedded airborne systems. These include: overt malicious code, covert malicious code, a malicious load image, and injected malicious code. Each type is detailed below.

1. Overt Malicious Code

The easiest way an attacker can use to include malicious code into software is to simply place it directly in the source code for the software. The small notional code fragment show in Figure 14 illustrates this approach. Essentially, the attacker needs to determine when the program should behave maliciously. If the program always acted maliciously, it would immediately be detected the first time the program was run. Thus, if an attacker wants the program to pass some basic level of testing, the program can only fail under certain conditions. Note that there is also an implied requirement that the execution thread must pass through this malicious code or the malicious code will never even get an opportunity to check if the condition is met.

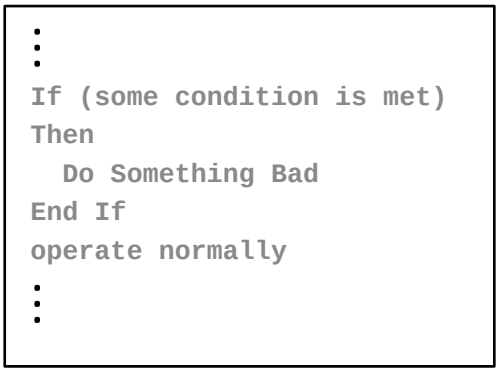

Figure 14 - Example of Overt Malicious Code

One specific problem for the attacker that wants to use overt malicious code is that anyone reviewing the software will notice it. In many cases it is common practice to have a team review software after it is written but before it is used. In this case, the malicious code would be detected.

In the specific case of airborne avionics, it is common practice for every single line of software to be executed during testing. For example, the DO-178B standard requires this for any software designated as Level A, Level B, or Level C. As a consequence, the "Do Something Bad" code would be executed during testing and be detected. This implies that under a DO-178B development effort, we have a significant defense against overt malicious code for any safety critical (level A-C) WRA. Note that it is possible for an attacker to replace the tested code with code containing malicious code after testing is complete. This is discussed in Chapter V. (p. 65). Although this type of testing is expensive, it is an effective way to mitigate the risk of overt malicious code.

According to Hilderman (2011), a DO-178B level C development effort costs approximately 30% more than a level D development effort (that is considered roughly equivalent to "any non-certified commercial software process"). Further, due to earlier bug detection, integration tends to be 50–75% faster than non-DO-178B efforts. Thus

requiring statement coverage (or even a full DO-178B level C development process) may be a reasonable alternative for certain applications.

2. Covert Malicious Code

Because overt malicious code is difficult for an attacker to use in the airborne environment, they may choose hide the malicious code instead. They might do this by writing software that looks normal but is actually malicious. They may also choose to hide the malicious code outside of the text memory segment (e.g., in the data segment) such that it does not even look like code. In these cases, an experienced programmer may examine the code and not notice a problem. Unfortunately, this is not that hard to do in practice. The Underhanded C Contest (http://underhanded.xcott.com) is a yearly contest to see which programmer can write the most innocuous looking malicious code.

The threats posed by covert malicious code are divided into two broad categories: compromised execution thread; and compromised data

a. Compromised Execution Thread

There are multiple ways for a compromised execution thread to cause a problem. We've broken these down into three threat types:

- Controlled execution thread – In this type, the malicious code executes specific malicious instructions. An example of this is the Stuxnet worm discussed earlier.

- Uncontrolled execution thread – In this type, the malicious code does not do anything specific, it simply causes the software to crash. In many applications, this can be catastrophic. Consider a Fly-by-wire system where a pilot moves an electronic control and a computer actually moves the flight control surfaces. If the computer crashes, the pilot loses control of the aircraft.

- Hardware interference – In this type, the malicious software affects the behavior of hardware in the WRA to cause a failure elsewhere. Referring back to Figure 7, we can see that if the hardware that connects parts of the WRA internally fail, the WRA will fail. This might be accomplished by modifying the value of a hardware register on an interface chip. In a bus arrangement, the malicious code could request and not release the bus, preventing all of the other system components from communicating.

Each of these types requires some amount of malicious code to be embedded in the main body of the software. In the case of an uncontrolled execution thread, only a small amount of malicious code need be present. This code will determine if it is appropriate to crash the WRA. The other forms also require code to determine if it is appropriate to behave maliciously. Remember that if the WRA crashes or fails every time it is used, the malicious code will be detected during testing.

In each case, the code which determines whether it is appropriate to act maliciously must be executed (or it will never have the opportunity to cause the malicious behavior). Put another way, part of the malicious code must operate every time the WRA runs but not cause the WRA to fail every time. If this part of the code is never executed, then the malicious code will be inert – it will never be able to cause harm.

There are multiple methods an attacker can use to embed malicious code into a WRA. These revolve around where the code will be hidden and whether the code is simply placed in the space or created/inserted during execution. For example:

- Malicious code embedded in the text segment
- Malicious code dynamically inserted into the text segment
- Malicious code embedded in the data segment
- Malicious code dynamically inserted into data segment

- Malicious code embedded in BSS segment
- Malicious code dynamically inserted into BSS segment
- Malicious code embedded in heap segment
- Malicious code dynamically inserted into heap space
- Malicious code embedded in stack space
- Malicious code embedded into free memory
- Malicious code dynamically inserted into memory mapped hardware registers
- Malicious code dynamically inserted into free memory
- Combinations

Note that some Operating Systems/Executives will zero out unused memory (e.g., free memory, the stack segment, the heap segment). When this is the case, an attacker cannot pre-load these areas with malicious code but must dynamically copy it there at runtime (if they want to locate the malicious code there).

b. Compromised Data

By taking advantage of an implicit vulnerability in the software, an attacker can cause malicious behavior. This type of vulnerability need not be the result of a programming error. When the data being compromised is part of the program itself, it is assumed that it is written correctly. I.e., the programmer is assuming that they do not need to guard against other programmers and there is no need to guard against illegal input because the software is not receiving input per se.

An attacker may modify data that is used to program hardware in the WRA. For example, they might adjust bus timing parameters to lock-up communication

between hardware elements within the WRA. In some cases, communication messages are built using a template. Such a template might specify a standard message header and the software would fill in the template when it is time to send a message. Since the template may be required to specify things such as a message size, an attacker may be able to change the template to cause problems. A programmer using the template may not check the template for validity since it does not represent external input. In practice, all internal data may not checked each time it is used due to the processing overhead.

3. Malicious Load Image

In addition to the software itself containing malicious code, a System Engineer must be sure that the final software image used in a WRA is either free from malicious code or that any malicious software present can do no harm.

Even when the source code is clean (i.e., free from malicious code), an attacker may be able to replace the load image built from the clean source code with a different load image containing malicious code as in Figure 15. This image can contain overt malicious code since no one will review the source used to build the malicious load image. This applies whether we are obtaining the load image from a WRA vendor or developing the WRA software ourselves. By paying attention to the software custody chain (as discussed in Chapter V [p. 65] later in this thesis), a System Engineer can often eliminate this risk.

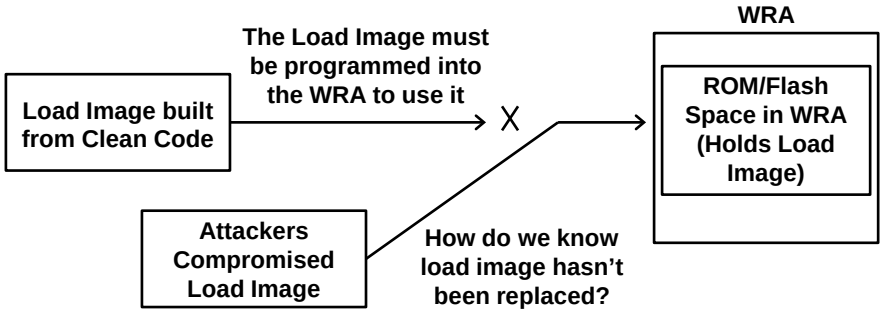

Figure 15 - Replacing Load Image with Malicious Version

Another way an attacker can insert malicious code into the load image is to modify part of the software toolchain that generates the load image. Thompson (1984) demonstrates that it is trivial to write a malicious compiler in such a way that even when the source code for the compiler is clean, if used to re-compile itself, the resulting compiler will still be malicious. He does this by having the malicious compiler executable insert malicious source code into the clean compiler source code as it is being compiled. Obviously, such a compiler could also insert malicious code into WRA load images as they are being generated.

4. Injected Malicious Code

Injected malicious code exploits a vulnerability in the WRA software to add the malicious code at run-time. Therefore, the issues associated this type and the mitigations recommended are the same as for the vulnerability threats discussed previously.

IV. MITIGATION APPROACHES

This chapter examines existing methods of mitigating the risks of embedded malicious code that are based on specific approaches for locating such code in individual system components. The main problem with simply looking for malicious code is that we cannot be sure we've found it all. Thus, we cannot make any statements about whether the software is safe, or the degree to which we believe it to be safe.

As these existing approaches prove inadequate to the problem, alternate system engineering approaches are proposed that utilize knowledge of the system (and the problem) as a whole to provide workable solutions in many cases.

A. TRADITIONAL DETECTION METHODS

From a software view (as opposed to a systems view), there are two main ways to detect malicious code. We can examine the source code and try to locate the malicious code, or we can execute the binary (compiled code) and try to detect anomalous behavior. Examination of the source code is referred to as static analysis and can be done by hand or with the aid of special software called static analysis tools. ("Static program analysis," n.d.) Detection during run-time is referred to as dynamic analysis ("Dynamic program analysis," n.d.).

1. Detection Using Static Analysis Tools

A static analysis tool is a software program that reads and automatically analyzes the source code for a program and tries to identify potential coding errors and vulnerabilities. Note that covert malicious code, if detected at all, will be identified as a programming error. Overt malicious code will not be detected at all—unless it happens to contain a potential coding error—because the tool cannot know what behavior is considered malicious.

The static analysis tool works by trying to identify potential programming errors and rule (e.g., coding convention) violations. For our purposes, one useful feature of a

static analysis tool is its ability to identify some locations in the source code where memory might be accessed incorrectly. This might be as the result of an error at a single point in the program, or as the result of multiple parts of the program accessing the same memory in different ways. This type of analysis is called a "pointer" or "points-to" analysis. Due to the way memory references are handled in software, in some cases it is not always possible to determine whether two variables refer to the same memory location. Thus the analysis tool has the choice of identifying all possible places where this might happen, or assuming that the potential cases are correctly coded (Livshits, 2006).

Zitser, Lippman, and Leek (2004) analyzed the performance of five static analysis tools by having them try to detect known buffer overflow problems in open source software. The tools were run on both the original (containing the known error) and patched (the error was no longer present) versions of the software. For three of the tools, detection was very poor, for the remaining two tools, detection rates were 57% and 87%. While this initially appears useful, the two tools with the high detection rate also had a very high false alarm rate. The tools produced one false alarm for every 12 to 46 lines of source code and were unable to distinguish between safe patched code and unsafe vulnerable code.

As previously mentioned, it is difficult for even an experienced programmer to examine source code to locate potential errors. Thus even when a static analysis tool flags potential problems with the software, it is not at all clear that a programmer will be able to detect the issue (if there is one). If the tool only generated one potential issue for every thirty-five (35) lines of source code, a one million Source Lines Of Code (SLOC) program would require an analysis of over 28,000 potential cases.

According to Charette (2009), a premium car in 1990 required approximately 100 million lines SLOC, which would translate into 2.8 million cases to examine—surely a daunting and impractical approach.

Malicious code can also utilize techniques that do not involve either programming errors or vulnerabilities. Although trivial types of vulnerabilities that might be employed were discussed, it is important to realize that the analysis tools are not able to flag all potential vulnerabilities as they rely on looking for specific programming constructs. Thus malicious code may be written that can bypass the detection capability of the tools. Further, malicious code is not limited to vulnerabilities to cause damage. As a trivial example, consider software that writes to specific register addresses in the hardware the software is hosted on. These registers control things such as the stack pointer (used to retrieve the return address for subroutines as well as the contents of variables—including pointers) and interrupt routines. Writing a value to a register does not necessarily result in a vulnerability (even when it enables malicious behavior). Since the hardware is outside the capability of a static analysis tool, the tool cannot analyze these accesses. Clean code may also be implemented from a malicious design.

Even when both the design and implementation is logically correct, the executable code will need to operate on some underlying system (e.g., hardware, software emulator). It is a common misconception that the software operates independently of the underlying environment, however this is not true. The influence of the hardware on the software can be extremely subtle. For example, consider the two pseudo-code fragments in Figure 16. The only difference is whether we access the array in column order or row order, so it may appear that it does not make a difference. However, it turns out that one method will be significantly faster if we are using Dynamic Random Address Memory (DRAM) on the underlying hardware. This is because of the way the physical memory is accessed. A DRAM Integrated Circuit (IC) is implemented as a matrix and is accessed by placing the upper half (row address) of the full memory address on DRAM IC's (smaller) address lines and pulsing the Row Address Strobe (RAS) line. Then the lower half of the full address (column address) is placed on the DRAM IC's address lines and the Column Address Strobe (CAS) line is pulsed. This causes the byte at that address in the DRAM IC to appear. However, the DRAM IC's have a feature that allows you to simply pulse the CAS line again to obtain the next sequential byte. If you access memory in column address order, the hardware will automatically use this capability. If you instead access

in row order, the hardware will need to transfer the address to the DRAM IC each time. In practice, this means that someone who is familiar with this can subtly affect the timing of the software.

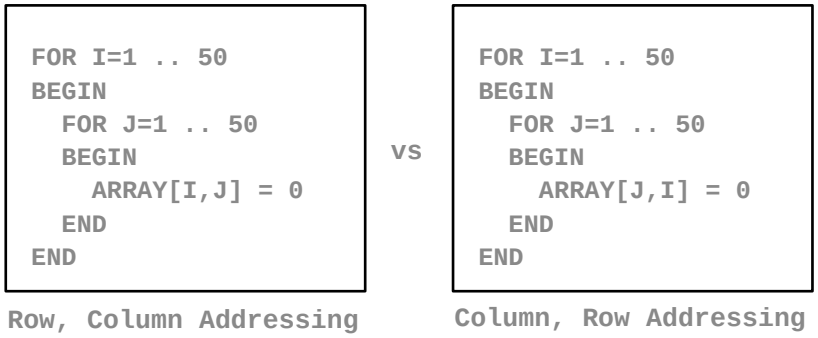

Figure 16 - Non-Obvious Hardware Dependency

Even when the code is clean, it can be made to fail by taking advantage of configuration parameters related to how the software will interact with specific hardware. These parameters are outside of the actual software but can cause failures for non-obvious reasons. Consider the pseudo-code in Figure 17. Both examples will work fine under most circumstances. However, the routine on the left will use more stack resources because it has more local variables. If there is not enough stack space allocated to the program, calling the subroutine will actually result in a stack overflow. On some platforms, this will cause a program crash, on others it will fail silently. Depending on the language and compiler, an attacker may be able to control what is overwritten and thus the behavior when the overflow occurs.

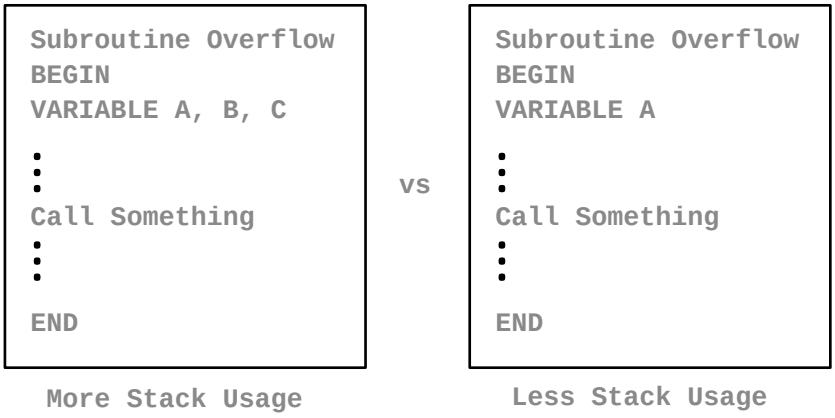

Figure 17 - Subtle Difference in Stack Usage

Note that the amount of stack space required by a program is often an estimate. Although it is often possible to determine what subroutines call each other and in what order, certain programming techniques such as recursion and trampoline code make this task impossible at compile time (and thus not detectable by static analysis). As a result, many systems have a default stack size. Under UNIX®, a user can change the amount of stack space allocated to a program. Under Windows®, a developer can tell the compiler how much stack space will be required (the value is embedded in the executable). Note that in both cases, the stack size (and thus whether an overflow will occur) is outside of the actual source code.

Static analysis tools process source files written in specific languages. There are literally thousands of computer languages ("Lists of programming languages," n.d.). Thus it is entirely possible that the WRA potentially containing malicious code is written in a language that is not supported by any static analysis tool. Keep in mind that it is possible to embed one language (typically assembly language) into source code written in a different language, so even if the primary language is supported, the embedded language(s) may not be.

Ballard, Chou, Chuang, Doshi, & Kimball (2004) describe two software tools that insert malicious code and vulnerabilities into software programs. These tools are specifically written such that specific static analysis tools will be unable to detect the

inserted malicious code and vulnerabilities. Note that a programmer need not be particularly skilled to use these tools. Thus it is possible for an attacker to simply write straightforward overt malicious code and then use the tool to hide it in the software. The tools even make the inserted errors look like careless programming errors to allow for plausible deniability.

Another potential issue is the unavailability of source code. Many applications are built using third party libraries and operating systems. The source code for these may not be available at any price due to trade secret concerns. Part of the source code cannot be scanned in these cases. This has implications not just for the possibility of vulnerabilities and malicious code in the libraries, but will also prevent a static analyzer from tracing potential problems through function/procedure calls that are part of the library.

Based on the above, static analysis is of partial, but limited usefulness in the detection of malicious code. Figure 18 shows the static analysis situation (note that the circles in the Venn diagram are not to scale).

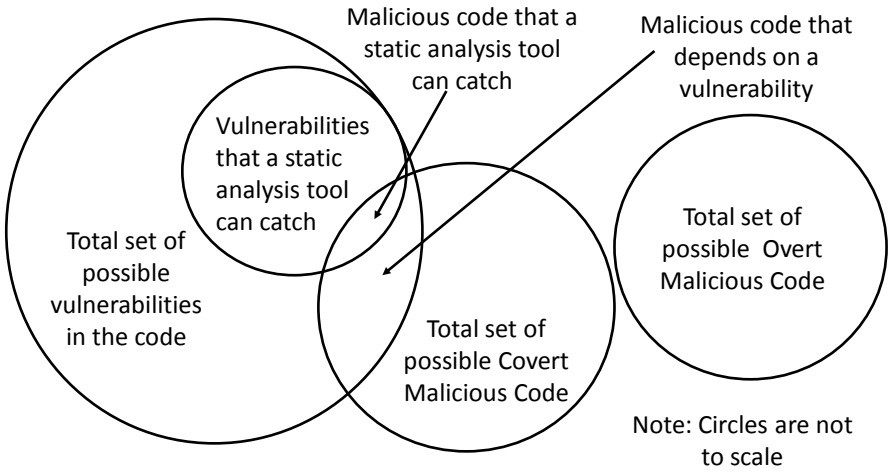

Figure 18 - Static Analysis vs. Malicious Code

As mentioned before, overt malicious code does not depend on any erroneous behavior so it will not be detected by an automatic tool (although manual

inspection would presumably catch some cases). Further, a static analysis tool is only going to catch a subset of the total vulnerabilities in the software. This is obvious not only from the literature but the simple fact that the news (2011) is full of examples of hackers exploiting vulnerabilities in code. If a tool was able to prevent this, these cases would not occur. Covert malicious code uses numerous techniques, and only some of them are based on vulnerabilities. Finally, since tools exist that insert vulnerabilities into software which these tools cannot detect, it should be clear that the tools cannot detect all vulnerability based malicious code. Note that although the case is not shown on the diagram, if the code is written in a language that is not supported by the tool, then nothing can be detected with the tool.

2. Detection Using Dynamic Analysis

Another approach is to use some form of dynamic analysis. Dynamic analysis consists of executing the compiled program on a real or virtual processor ("Dynamic program analysis," n.d.).

As mentioned earlier, some software development methodologies, especially for embedded airborne software, include some type of code coverage analysis. This is a form of dynamic analysis in that it works by instrumenting (adding little code snippets) to the source code to allow someone to tell what parts of the program have actually been executed. Although it may be impossible to fully evaluate the code (due to issues such as multiple languages and hardware limitations), by providing assurance that all of the available source code has been executed we can be reasonably sure that there is no obvious overt malicious code present (in the code we were actually able to instrument). Note that there are levels of code coverage analysis. Figure 19 illustrates a code snippet with the code "blocks" and branch statement identified. On the simpler end of the spectrum, we verify that each code "block" has been executed during testing. More complicated methods require that branching constructs (e.g., "if" statements) be fully exercised such that malicious code buried in the branch itself would be detected.

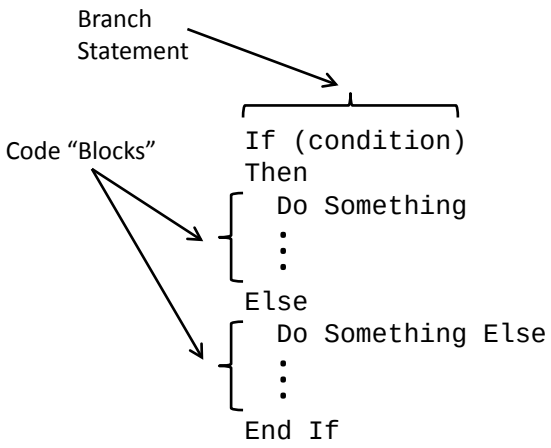

Figure 19 - Code Coverage Analysis

It is also possible to detect many types of covert malicious code using other source code modification techniques. For example, the ProPolice homepage provides a "GCC extension for protecting applications from stack-smashing attacks" (http://www.research.ibm.com/trl/projects/security/ssp/). The GCC compiler extension works by adding "canaries" to the stack around buffers and other key parameters. For example, when a return address is placed on the stack, it is surrounded with specific values. Before using the return address to return to the calling routine, the canaries are checked to make sure they have not been altered. Thus a simple buffer overflow attack would be detected as the canaries surrounding the buffer would be modified by the attack. This adds a very slight amount of overhead to each function call.

Although the canary technique seems quite powerful, it is not adequate to stop a determined attacker. Bulba & Kil3r (2000) describe a simple method to bypass canary protection techniques and allow buffer overflow attacks. They attack the stack indirectly through other variables.

Many types of dynamic analysis detection methods involve changing the underlying environment. This is because in order to be successful, malicious code depends on the underlying environment to operate "correctly." Examples of changes that may result in detecting the malicious code include:

- Change the opcodes used by the CPU – for example, if the malicious code consists of executable instructions hidden as data in the data segment of the program, when the opcodes change, the "executable" instructions will become meaningless garbage. This will generally result in a program crash.

- Changes to the stack layout – for example, if the malicious code overwrites a subroutine return address in the original stack layout, in the modified layout some other value will be overwritten. This may cause a crash, it may cause the program to run incorrectly (presumably detected during testing), or it may do no harm (e.g., if the value overwritten is not used again).

- Changes to the memory layout – for example, if the malicious code tries to transfer the execution thread to a specific location in memory (where other malicious code is hidden), changes to the memory layout will result in a transfer of control to a random location in the program resulting in a crash or odd program behavior. There is a small chance that the location will immediately return and the program will run normally.

As an example of how changing the underlying execution environment can detect malicious code, consider the attack shown in Figure 20. The attacker has placed malicious code on the stack (in the form of data) with an expectation that it will be at a specific location in memory. The attacker has also included a hidden jump to this location. In order for the attack to work, the malicious code must be where the attacker thinks he put it. By randomly choosing the start of the stack frame (or making other changes such as the direction of stack expansion), the attacker is unable to use a fixed offset. If he did use a fixed offset and the offset is now changed the, the program will crash during testing.

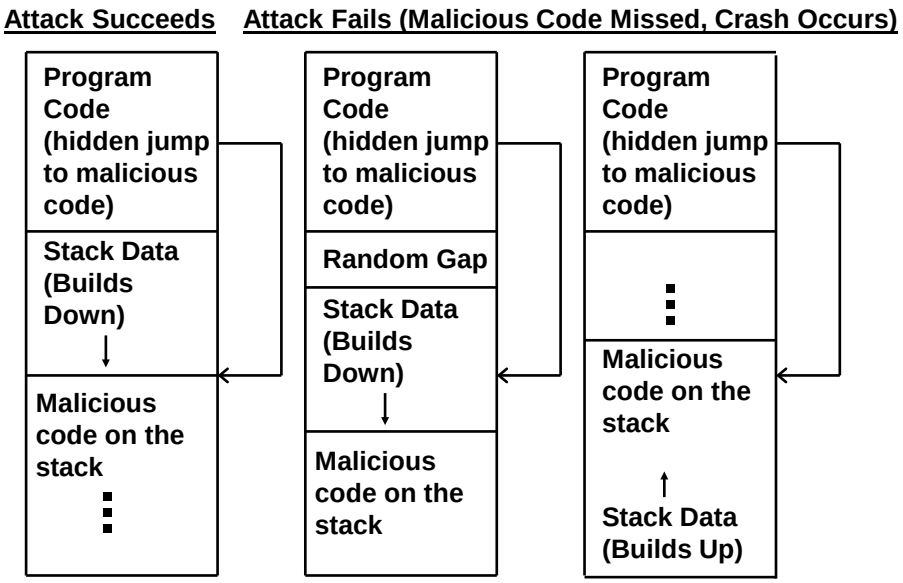

Figure 20 - Affect of Randomizing Stack Location/Characteristics

Note that certain classes of subtle programming errors may also be detected with this approach. For example, if a programmer allocated dynamic memory on the heap, and then uses it after it has been freed, the program might appear to operate normally (depending on what happened to be in memory at that location). By moving the heap location, the program might fail because different bad values may be referenced. Of course, malicious source code may be disguised as a programming error for deniability so we cannot tell if the error was malicious.

Not all covert malicious source code attacks rely on hard-coded addresses. Attacks that are "portable" may survive different environments. These attacks are harder to write because they need to use references (e.g., variables) to calculate the required offset. In other cases, an attacker might need to know the relative offsets between two buffers. For example, an attacker might choose to add a covert channel to an encrypted data stream, deliberately leaking the decryption key by embedding it into "random" noise used to pad out the encrypted message. Simply moving the start of the stack would not affect this attack if the buffers were in the same compilation unit. This case is not addressed in detail although other randomization techniques can be used in this case.

Both attacks involving malicious code injection (where an attacker utilizes an input channel and a vulnerability to "inject" malicious code into the program) and covert malicious code that uses pre-compiled malicious code rely on the knowledge of what opcode (bit pattern in memory) causes which behavior. For example, if 0x1234 means add register A and register B and we suddenly redefined 0x1234 to mean subtract register A from register B (and perhaps redefined 0x2134 to mean add register A and B), any program compiled using the old set of rules would fail to run. Thus if we randomize the opcodes (i.e., redefine what each bit pattern in memory means), any pre-compiled malicious code would fail. This approach is shown in Figure 21.

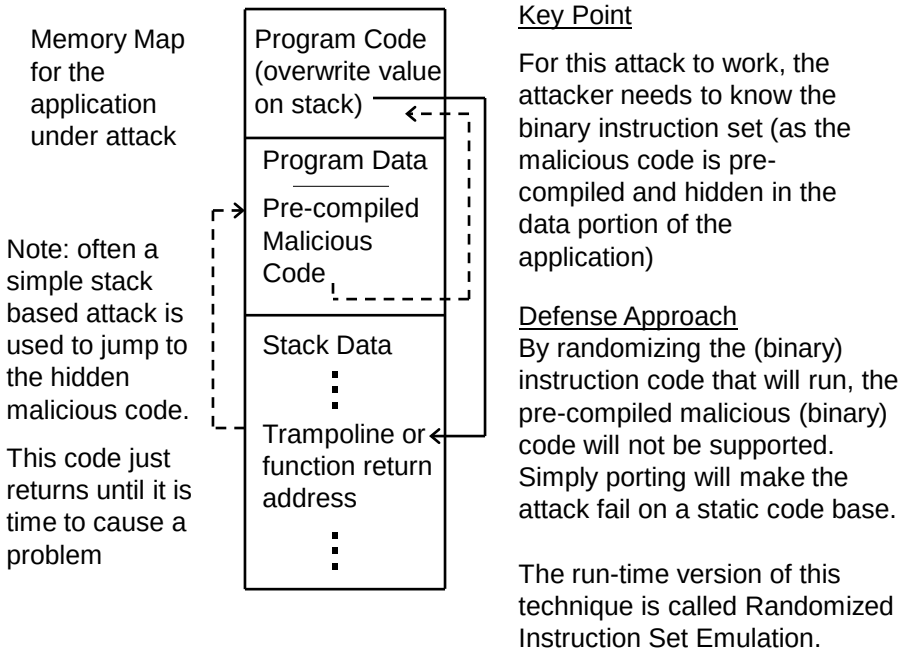

Figure 21 - Effect of Changing Opcode Definitions

Embedded airborne software is not able to utilize many dynamic techniques such as these in the final production version of the software. This is because detection of an accidental problem (e.g., not malicious software but instead a coding error) during operational use might be worse than the response to the detected problem. For example, in the case of a canary, if a buffer overflow was detected, the result would be a software crash. If this was an accidental problem (or even an incorrectly implemented malicious

attack), the software might continue to run normally despite the problem. The same logic applies to modifying the software's environment, although here the problem "detection" would most likely result in a crash. In a safety critical WRA, a software crash might be catastrophic. Thus techniques that involve a virtual environment may be unacceptable for operational usage due not only to unacceptable performance loss but also safety concerns. Safety issues are also involved if we were to change the opcodes used. Nonetheless, porting the software from one platform to another can both effectively change the opcodes and also affect other execution environment factors.

The Systems Engineer must weigh the damage caused by detecting an accidental programming error when the system is in use against the possibility of detecting malicious code. In general, the risks posed by dynamically altering the execution environment are unacceptable. Most programs contain numerous errors but may contain no malicious code. Thus for safety critical software we will tend to be more concerned with accidental software errors than malicious code.

In order to minimize the likelihood of exposing a programming error when the WRA is in use, safety critical applications tend to minimize any dynamic characteristics of the execution environment. Dynamic memory allocation may be prohibited, process execution order may be fixed, etc … As a result, we are providing the attacker an ideal environment for malicious code.

Many of the techniques involving changing the run-time environment may only be used under controlled conditions (when the WRA may fail without serious consequences), although there are exceptions. Techniques such as using Write Exclusive-Or Execute (W^X) memory pages should never cause an issue for legitimate safety critical software since they would indicate that the software had already failed in a very serious way. Most other techniques require making a change, testing the code under the change, and then changing back to the original.

One simple method of partially randomizing the run-time environment is to re-compile the software with a different compiler. This may result in changes to the stack

and heap layout as well as resulting in size changes (e.g., due to optimization differences) throughout the code and changing CPU register usage.

How effective is simply using a different compiler? Sun Microsystems chose to support multiple compilers on their OpenSolaris platform, in part to allow for the execution environment changes to detect programming errors. Wesolowski (2006) writes "These 28 bugs are tangible evidence of code quality improvement demanded by the use of multiple compilers; many of these defects would be expected to affect customers. Note that these defects still existed even after most common kernel and library code was fixed by the amd64 team, so the actual number of bugs was likely much higher.."

If it is feasible to port the software to another hardware platform, this is even more effective (especially with a different compiler). The stack and heap may be arranged differently, the wordsize may be different, the system libraries will be different, the instruction set will be different, etc. Each of these changes makes it less likely that any embedded malicious code will still operate "properly."

A complete port to new hardware is seldom feasible because the embedded systems often depend on special hardware and interfaces. Even so, porting of CSCs combined with unit testing can provide much of the benefit of a complete port for those portions of the code. Some of the benefits of porting may be obtained by simple modifications of the code base. Although this does not change the instruction set (opcodes) and uses the same libraries, we can alter the layout and position of the stack and heap as well as re-order the layout of code blocks within the text segment and data blocks within the data and BSS segments.

One simple technique is to change the order that object files are linked together. This is shown in Figure 22. Any covert malicious code that depends on absolute addresses or that depends on a fixed distance between locations in different source files will break. Note that in this example, the first object file was not moved because often the program entry point must be in the first object file. Similarly, the library files,

although re-ordered, still follow the object files because many linkers require this. Thus only a limited amount of re-ordering is possible.

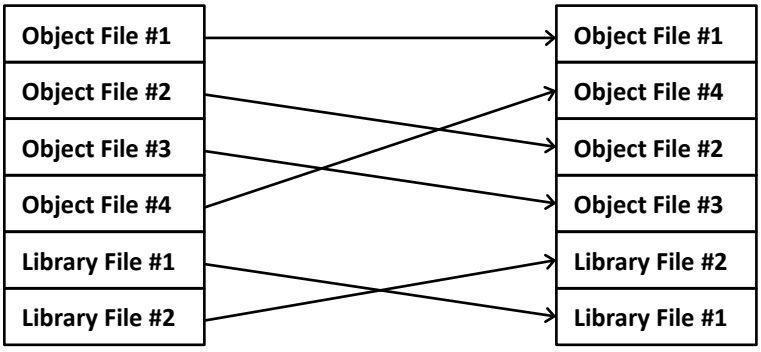

Figure 22 - Link Order Randomization

Covert malicious code might also depend on the relative spacing between procedures or the data associated with them within the same source (and object) file. Figure 23 shows a simple method to impact this. Note that this is labor intensive and might introduce errors.

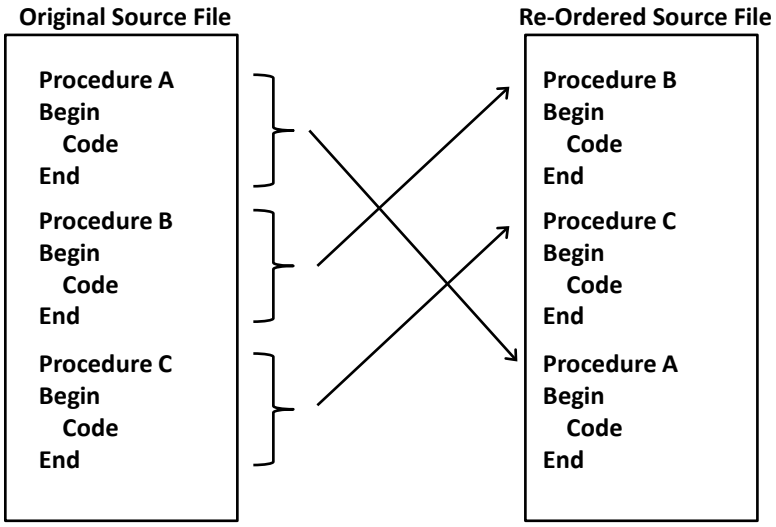

Figure 23 - Randomize Code Order

Similar techniques can be used to randomize Data and BSS segment contents. It is also possible to add extra dummy arrays with randomly determined sizes. This last approach can be done in an automated way as part of the toolchain.

Keep in mind that most of the detected issues will be a result of a programming error and that the program may be able to operate normally despite the error. In some cases, this failure mode (assuming it is the result of an accidental programming error) may be preferable in actual use to a complete program failure. As a result, when these techniques are used it is necessary to make the modifications to the code, compile and test the code, then remove the modifications and re-compile and re-test. This concept is embodied in the NASA principle "test what you fly, fly what you test." It would be possible to use some of these techniques (e.g., the canary approach) and have detected errors reported to a Built In Test (BIT) system rather than crash but I have not seen this approach implemented.

Unfortunately, none of the techniques presented here can guarantee that no malicious code remains in the source code – or toolchain for that matter. Figure 24 shows the dynamic analysis situation (note that the circles in the Venn diagram are not to scale).

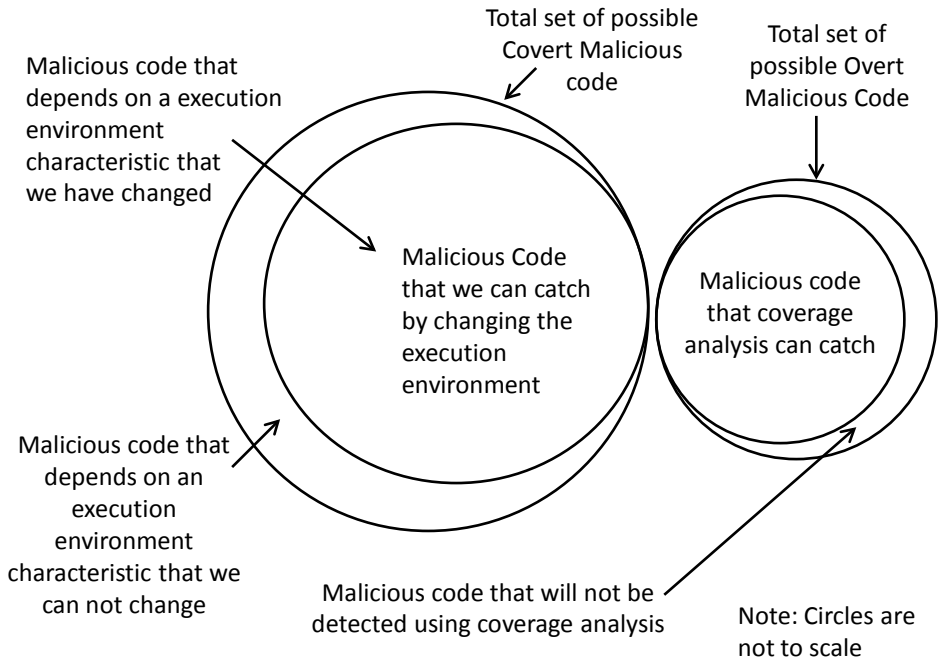

Figure 24 - Dynamic Analysis vs Malicious Code

B. SYSTEMS ENGINEERING MITIGATION APPROACH

A Systems Engineer works on a problem as a whole, rather than being limited to techniques that apply to a single WRA. Although a Software Engineer may have insight into these approaches, it is unlikely that they will be able to control the selection of all the WRAs used or the information flow between them to the level available to a Systems Engineer. In addition, the Systems Engineer can control system level testing – typically not an option for a Software Engineer. This section uses this holistic view as the basis for an alternate approach to the problem of malicious code.

It is assumed that the attacker wants the malicious code to cause a problem in an airborne system after it is put into service. If the malicious code is detected during testing, it will be extremely limited in the amount of damage it can cause. Thus the challenge for the attacker is to insert malicious code in such a way that it will pass testing but fail later during use.

It is further assumed that the goal is to have the system operate safely, whether or not it actually contains malicious code. As such, the goal is not to make sure that there is no malicious code present, but instead to be able to say that if there is malicious code present it cannot cause harm.

1. Attacker Requirements

What does an attacker need in order to insert malicious code into the software? It depends on whether they want to attack the source code or the binary load image.

For overt malicious source code, they only need basic programming skills and access to the source code. For covert malicious source code, they need a higher level of programming skill (or a tool that will insert the code for them), knowledge of the specific system environment, and access to the source code.

For a malicious load image, they need the ability to build a malicious load image. Thus they need basic programming skills, access to the source code, and the compilation toolchain necessary to build the image. Alternatively, they need an existing binary, compilation tools, and enough knowledge of the binary that they can patch it to link in the malicious code. They also need the ability to replace the clean load image with the malicious load image. For example: access to the CM system; access to a load device; or access to the WRA after programming.

In the above cases, it is assumed that the software will be tested (a solid assumption for the case of safety critical embedded airborne software), thus the attacker needs a way to have the software operate normally during test yet fail later

For injected malicious code, the attacker needs a higher skill level than basic programming knowledge, as well as knowledge of the specific system environment, and knowledge of the vulnerability. If they have access to a hacking tool for the system they are trying to attack, then only basic programming skills and knowledge of the vulnerability are required. Depending on when the code will be injected, it may be necessary for the injected malicious code to know when to act maliciously.

2. Trigger Concept

As discussed earlier, in order to do something malicious, the execution thread must pass through a section of malicious code that determines whether to actually behave maliciously or not. Figure 25 shows this as an "if" statement (although the implementation of the "if" will not be explicit in the case of covert malicious code). This "if" statement is executed each time the WRA operates—otherwise, the malicious code would be unable to do any harm.

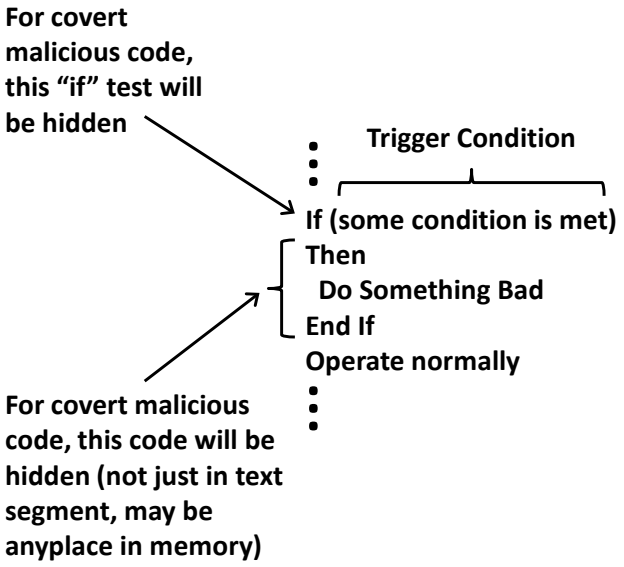

Figure 25 - Trigger Concept

The "if" statement must evaluate some condition in order to determine if the malicious behavior should occur, otherwise the malicious behavior would always occur—including during testing. This condition is referred to as a "Trigger Condition" in this thesis because it is used by an attacker to "pull the trigger." For example, a trigger condition might be related to usage hours. If the WRA has been operated over 200 hours, then cause a problem. The trigger condition might be related to an altitude. If the A/C exceeds 10,000 ft, then cause a problem.

Thus in order to cause a problem, the trigger condition must be met. The trigger condition will be based on some input condition. In the above example, the software

cannot trigger based on the usage hours unless it has a way to determine those hours. In the case of an altitude based trigger, unless the software has a way to determine the altitude it cannot use this as a trigger.

A Systems Engineer can often control what information is available to a WRA (and thus the software embedded in the WRA). For example, if a WRA does not need to know the altitude, there is no reason to provide it this information. The information available to a WRA can be evaluated to determine if any of it is useful as a trigger. If no trigger information is available, then a reasonable amount of testing can ensure that the WRA does not contain malicious code. If there are pieces of information that can be used as a trigger, but the WRA does not require them to be used in the particular system being developed, a Systems Engineer can prevent those pieces of information from reaching the WRA. In this case, a reasonable amount of testing can ensure that either there is no malicious code present or, if there is malicious code present, that it will never be triggered (and will thus not do harm)[4].

In the case of an internal usage counter, the Systems Engineer will probably not be able to prevent this input from being used as a trigger as the counter is likely to be stored in some section of Non-Volatile Memory (NVM). However, if the System Engineer were to test the WRA for (say) 200 hrs and then reset the usage counter every (say) 100 hrs, any trigger based on the usage hours would be blocked. Thus a certain amount of testing would allow us to conclude that either the WRA contained no malicious code or that the code was based on a trigger value that required a value greater than the amount of testing hours to cause the malicious behavior.

Note that it is also possible for an attacker to trigger based on some statistical occurrence. If a source of random information is available, an algorithm could be devised that would cause malicious behavior very rarely. By doing some statistical analysis, an attacker could estimate the mean time to trigger and adjust it appropriately.

[4] Note that this is not completely true as a statistical trigger could be used. This is discussed later.

This is not a particularly attractive approach for the attacker as there are two undesirable outcomes: the behavior might occur during testing; or the behavior might never occur.

Statistical triggers can be partially mitigated by increasing the testing time in order to reduce the probability that an undiscovered trigger is present. For example, if the expected lifetime usage of a WRA is 100 hrs, and it is tested for 10 hrs, the odds of detection would be low (but not zero) that a hidden problem would be detected. If it is tested only 50 hrs, the odds of detection would still be low (but higher than before).

3. Information Limiting Concept

A Systems Engineer needs to evaluate all the information available to a WRA in order to assess potential trigger information. This involves examining all potentially available information, not just information that the WRA requires to operate. For example, many airborne avionics systems use ARINC429 to communicate. ARINC429 is a two wire bus system that allows one transmitter and up to twenty receivers (AIM, 2010).

A WRA receiving information through an ARINC429 bus may only need a fraction of the information available on the bus. However, the WRA has access to all information on the bus and could use any information from the bus as a trigger.

A Systems Engineer can control what information is available to WRAs by controlling what busses the WRA is connected to. Since some WRAs may transmit information that is not needed (because they are general purpose and do not know what receivers will be using it), it may be possible to modify these WRAs to eliminate the potentially risky information from the bus.

4. Safety Assessment

Not all WRAs can cause the same degree of damage if they fail. WRAs on airborne systems are commonly categorized by the danger posed by a failure. For example, DO-178B assigns Design Assurance Level (DAL) values of A-E to WRAs based on a safety assessment. Since a Level D WRA only has a potential impact of

"Minor" and a Level E WRA only has a potential impact of "No Effect," it may not be necessary to worry about the possibility of malicious code in these units. Keep in mind, however, that these levels are only related to safety issues. The failure of a Level E WRA could result in a mission failure or compromise of classified/sensitive information. The DAL values can be used as a starting point and then augmented as necessary.

5. Testing Considerations

Software in airborne avionics systems is typically thoroughly tested, however these tests are designed to locate unintentional problems in the software. Testing is typically performed on the WRAs themselves, then subsystems, then the entire aircraft. In order to mitigate the risk of malicious code, it is necessary to augment these tests based on the system risk posed by each WRA and the triggers available to each WRA.

The actual choice of what augmentation (if any) will be required is a function of the potential triggers identified. For example, if usage hours is the only available trigger, then we might require that the number of testing hours exceeds the amount that will be incurred during normal usage.

In practice it may not be cost effective to mitigate a potential trigger since it is generally not adequate to simply test each possible value of the potential trigger condition. For example, assume the attacker is using altitude information as a trigger. The attacker is unlikely to trigger when the altitude exceeds or equals a specific value as this is likely to be caught in testing. They might, however consider triggering after an altitude is exceeded for a specific amount of time, or after it is crossed a certain number of times, or on a specific combination. Keep in mind that the attacker will need to identify a scenario that will not occur during test yet will occur in use. Thus, as a general rule, if access to trigger information cannot be blocked, testing will be adequacy cannot be guaranteed.

If a WRA is available that has been used successfully in the past, this usage can be leveraged as additional assurance that the WRA contains no malicious code—since it is unlikely that a WRA that has not been triggered during use elsewhere will cause a

problem for our effort. Note however that the environment in which the WRA was used needs to be similar. For example, if the WRA has an input that was not used in the past, which is planned to be used in the new application, it cannot be assumed that no malicious code is present – since the malicious code might trigger based on the previously unused input. Assuming the environment was similar, there is a need to verify that the source code, which was used successfully in the past, is the same source code that will be used as the starting point for the new effort. I present a method to accomplish this in Chapter VI. (p. 74). Any required changes can then be made and our effort is limited to the risk imposed by the new code only.

C. MITIGATION RECOMMENDATIONS

Based on the previous analysis, this section provides guidelines that can be used as a framework to mitigate the risk of malicious code in a WRA. Start by bounding the potential risks posed by each WRA by categorizing it. Specifically, categorize each WRA based on:

- How much risk it represents (can use DAL values as a starting point)
- Whether it has an external interface
- What potential trigger information is available

Potential approaches for each of these cases is listed in Table 1.

Characteristics	Observation	Actions
Low Risk	If WRA contains malicious code, damage is limited	No action necessary (assuming willingness to assume the risk)
No Potential Trigger Information	Could use a statistical trigger	Specify amount of testing to reduce risk to acceptable level
Potential Trigger Information	Significant risk	Determine if trigger information can be eliminated; examine WRA components to see if risk can be isolated to a subset of the WRA
External Interface	Risk depends on input channel specifics	Determine potential for triggers; determine potential for vulnerability based attacks
Internal Interfaces Only	Low risk of vulnerability (another WRA must be compromised first)	Take action based on other characteristics

Table 1. Table 1 - Potential Mitigation Approaches Based on WRA Characteristics

Based on Table 1, we can see that the primary challenges for mitigating malicious code are WRAs that both have high risk and also have either external interfaces (because they are potentially vulnerable to an attacker) or have triggers available (because they may contain trigger-able malicious code). In each case, we can sub-divide the problem further.

1. WRAs with External Interfaces

In order to deal with WRAs that have external interfaces, we can categorize the types of input to determine both the potential for taking advantage of vulnerabilities and

also for use as a potential trigger. For the case of airborne avionics, many of the external interfaces only represent a voltage level. For example, a temperature sensor, or an airspeed indicator may output a voltage depending on the current value. The WRA may use this value directly or may simply send it (via an internal interface) to another WRA in the system.

Potential approaches for various types of external inputs are listed in Table 2.

External Input Type	Observation	Actions
Analog Level (e.g. engine temperature, not encoded data)	Very low vulnerability risk, potential trigger	Handle based on trigger information
Specific Message Format (irrespective of carrier, i.e., analog carrier of data is not an analog level)	If we already read (say) six bytes, vulnerability risk is low. If we read two bytes that then tell us how many bytes to read, vulnerability risk is high. Potential Trigger.	Determine potential for abuse of message format by examining message layout, treat as freeform if potential exists; take steps to mitigate trigger risk
Freeform Message Format (including Ethernet connections)	Risk of vulnerability, potential trigger	Take steps to expose vulnerabilities and potential malicious code, examine WRA components to see if risk can be isolated to a subset of the WRA

Table 2. Table 2 - Potential Mitigation Steps for WRAs with External Interfaces

For WRAs that have external inputs likely to pose a problem, it is necessary to roll off to static and dynamic analysis methods and accept the residual risk. Dynamic techniques are recommended when possible. The degree of testing would be determined by the potential risk posed by that particular WRA. Note that in some cases, only part of the WRA must be tested as the WRA can sometimes be decomposed into WRA components and our efforts can be targets to components with external connections.

2. WRAs with access to Trigger Information

Some WRAs have access to information that can be used as trigger information and we cannot block this information from the WRA, either because it is impractical to do so or because the WRA needs the information to operate. If a WRA represents a significant risk and has trigger information available we will need to determine what portions of the WRA represent a risk and then take steps to identify potential malicious code. Since trigger information cannot be blocked, I recommend using static and dynamic analysis techniques for these WRAs.

For the case of Overt Malicious Code, we can gain significant benefit from the use of structure coverage testing as required by DO-178B. Figure 26 shows the situation for Overt malicious code. Note that DO-178B cannot completely eliminate this risk because some parts of the source code will not be tested. Level C WRAs will only be tested with statement coverage – allowing for malicious behavior in the decision tests themselves. Even Level A WRAs with MC/DC testing cannot cover all cases since not all possible program states are represented. Static analysis testing is shown on this diagram because there is significant overlap between static analysis testing and DO-178B testing.

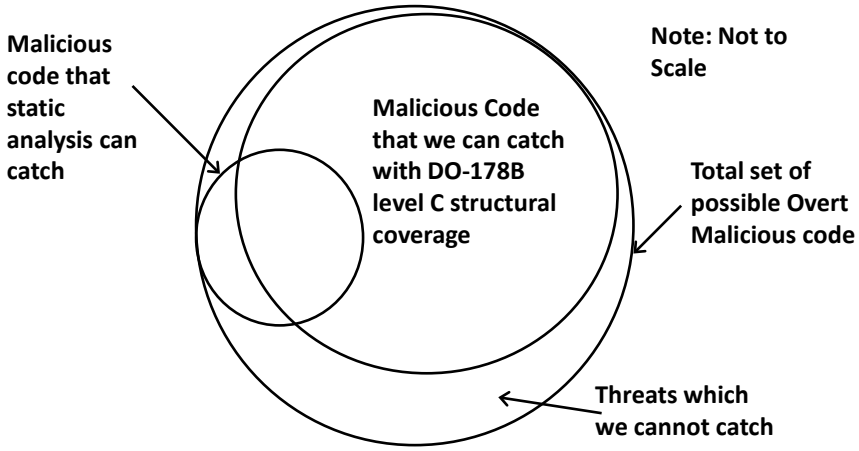

Figure 26 - Leveraging DO-178B Testing for Overt Malicious Code

For the case of Covert malicious code, some set of dynamic testing is recommended. Figure 27 shows the situation for both Overt and Covert malicious code and the effect of changing one or more factors in the underlying execution environment. In some cases we will already be planning to make changes that will affect the execution environment. For example, during the development process we may plan to move software between partitions to balance the system load. We might plan on shrinking the stack size late in the development process once we know how much memory will be required. We might already plan to implement some safer run-time detection methods such as using a Memory Management Unit (MMU) to limit access to memory segments.

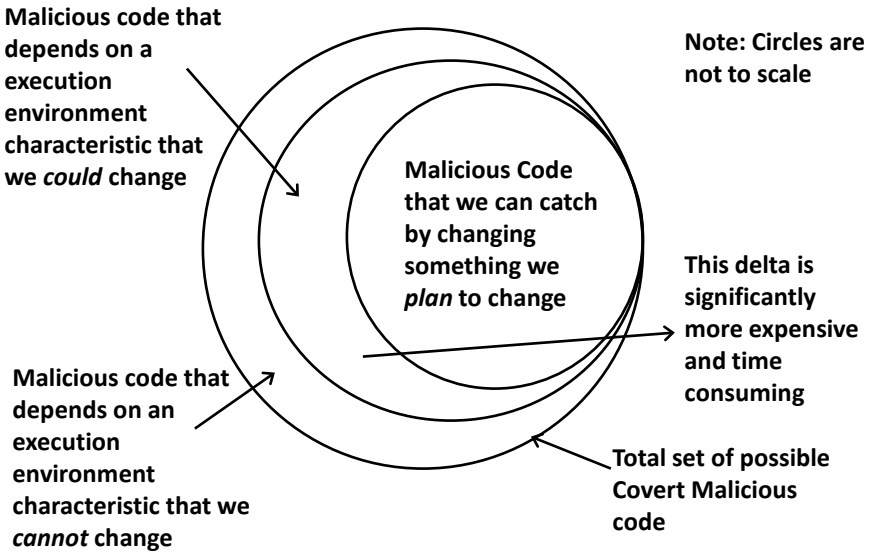

Figure 27 - Changing the Execution Environment

Unfortunately, there is insufficient information to set the relative size of the circles so we cannot determine the remaining risk that there is malicious code present. Note however that the risk posed by the specific WRA within the overall system can be determined. The WRA has a limited ability to cause harm at the system level. Similarly, we can sometimes bound the risk within the WRA.

The problem can be approached by examining the architecture of the WRA to identify both the physical structure of the WRA (different CPUs and memory spaces) and the individual CSCIs have access to the trigger information. In many cases, there may only be a small subset of the software in the WRA that poses a risk. Obviously it is an advantage to identify this situation if it exists. In terms of physical sub-assemblies, each assembly can be treated as if it were a WRA. Depending on the internal topology of the WRA, the maximum risk posed by the WRA may be further reduced. For example, if the trigger information is only used by a small subset of the WRA that does not interfere with the more critical functions provided by the WRA, the risk posed may be low.

Since software within a WRA (or WRA sub-assembly) may contain multiple CSCIs, it may prove useful to examine the pedigree of each CSCI. If, for example, it is

discovered that only one CSCI is involved and it has been widely used in the past, it may be possible to eliminate this as a risk (although there would be a need to verify that the CSCI source code is in fact what has been safely used in the past). I present a method of doing this verification in Chapter VI. (p. 74).

If the remaining risk posed by software in the WRA (or WRA sub-assembly) is unacceptable, I recommend using statement coverage (execute every line) to eliminate overt risks and various types of execution environment randomization to increase the probability of malicious code detection.

In the case of DO-178B level C and above WRAs, we will obtain the statement coverage testing for free. When this is not the case, the same techniques would be used even though they would not be required from a System Safety standpoint.

V. TRUSTED SOFTWARE CUSTODY CHAIN

When software is initially developed, it may contain malicious source code. However, even if the original source code is clean, there exists the possibility that malicious code will be added later. This chapter traces the source code from development to its final use in an attempt to identify and mitigate the risks of this occurring throughout the development process.

A. BASIC DEVELOPMENT PROCESS MODEL

The generic software development life cycle for embedded avionics systems shown below is used as the basis to discuss the potential for inclusion of malicious code during software development and deployment:

- The source code is developed
- The source code is placed under CM control
- Source code is pulled from CM control for build
- Source code is compiled, assembled, linked, etc ... into a load image
- The load image is loaded into firmware by the OEM, or
- The load image is delivered to the customer/user
- The load image or hardware with loaded firmware is transferred to customer control
- The customer/user maintains control of the software

Note that the design effort that would typically proceed the development of the software has been deliberately excluded as a malicious system/subsystem design is beyond the scope of this thesis. Also note that the software development process

assumes the use of a Configuration Management (CM) system. Many design methodologies (including DO-178B) expect this basic level of process control to be in place.

B. THREATS

Each stage of the development process contains opportunities for an attacker to insert malicious code. Thus even if we trust the software developer completely, the source code may still be at risk. Multiple places where the WRA may be compromised during the software development and delivery process are identified below. Each threat is detailed along with specific mitigation recommendations.

1. Source Code Development

This threat consists of a programmer deliberately writing malicious code that can pass various types of screening used by an organization using good engineering practices —such as we expect would be used to develop embedded avionics software. Such organizations typically employ techniques such as peer code reviews, code analysis tools, unit testing, and integration testing. Code compromised in this way must be able to pass such screens.

Since it is assumed that the code must pass code review, the programmer will need to embed a subtle bug in the program that will cause the program is misbehave. I.e., it is not likely that code can be inserted that explicitly causes the failure.

Since the software will be tested, the corrupt code must not fail under normal conditions or have the failure detected. This means that code that contains a “payload” must have a “trigger” that causes the software to fail. Such code is sometimes referred to as a “logic bomb.” Other failures are not obvious, and might always be present but not noticed. For example, in some cryptography algorithms, noise is added to the signal during the encryption process. When the signal is decoded, the noise is removed and discarded. If covert malicious code added the decryption key to the noise in the

encryption algorithm, it is not likely that this would be noticed—and the algorithm operate correctly; yet an attacker could read all the encrypted traffic.

Unfortunately, if trigger information is available (or, as in the cryptography example, the code can act maliciously all the time without detection), blocking trigger information is not a viable approach. Instead, a combination of static and dynamic analysis may be used. Since many individual Computer Software Components (CSCs) can be tested independently of the entire application, it may not be particularly expensive to port both the CSC containing the problematic source code and the testing framework to other environments and use other dynamic techniques while unit testing. Unit testing can be done by a separate test group to maintain independence. Combined with formal code walkthroughs and control of the CM system, this will reduce the risk – although the actual remaining residual risk will not be known.

2. Source File Replaced (e.g., Pre-CM System Check-In)

This threat consists of replacing code that has been inspected and unit tested prior to its insertion into the CM system. It can be carried out by anyone with access to the code or filesystem. This includes the programmer, anyone on the programming team (assuming that a shared work area is used), or system administrators.

An inexpensive way to mitigate this threat is to setup the development process so that the software is checked into the CM system prior to code inspection. The code inspection should be done on copies of the software obtained from the CM system.

3. Configuration System (CM) System Compromised

This threat consists of replacing/altering code that is under CM control without leaving a record of this change. This is a trivial task for many CM systems. Files stored under CM are often stored as a parent file (original version or current version) and then a series of differences that can be used to re-construct specific versions of the file. Under many CM systems, all the differences are stored as simple text files that can be read or modified by anyone with access to the filesystem where they are stored. This includes

the CM administrators and system administrators. Often programmers and even users have access as well. Certain CM systems (e.g., ClearCase) mangle the contents and references to file differences. This can stop a casual effort to modify files under CM control but a determined attacker can still make changes. Note that this type of attack is undetectable without additional measures since it is not hard to change the timestamp on the file as well as its contents.

There is no simple mitigation approach for this threat unless there is a willingness to trust the system administrators. Basic steps can be taken such as locking down the filesystems so that only the CM administrator (and system administrator) can access them, but both the CM and System administrators would still have the ability to insert malicious code. In addition, there is no guarantee that the CM system will remain uncompromised (and if it was compromised, that the breach would be detected). For example, Microsoft discovered that their network was breached in 2000 and that hackers had access to their CM system over an extended period of time (Lopez, 2000). Microsoft (as cited in Lopez, 2000) stated that the hack-in could have been an act of industrial espionage. Thus it is possible that source code was infected with malicious code.

If deemed necessary, the CM system can be segregated from the development system. Thus code checkin/checkout would be done across a network. If this model is adopted, either redundancy or hash verification can be used to reduce the chance that an attack will succeed. Figure 28 shows how a development system can be used to detect tampering (or errors) in a single CM system. By encapsulating the checkout/checkin routines (read/write from CM system) operations, this can be made transparent to the user. When each development system obtains source code to work on, they read it from both CM systems simultaneously. The results are then compared to make sure that both CM systems provide the same information. As shown, only error detection is possible, but with additional CM systems, error correction could be added. Multiple development systems are also required. If only one system was provided, then an attacker could simply modify that single system and an error would not be detected (or local substitution of clean source for malicious source could occur). Note that an attacker would either

need to penetrate both CM systems or multiple development systems. Obviously it does not make sense to use the same CM or system administrator for both CM systems or all of the development systems if this is deemed a risk.

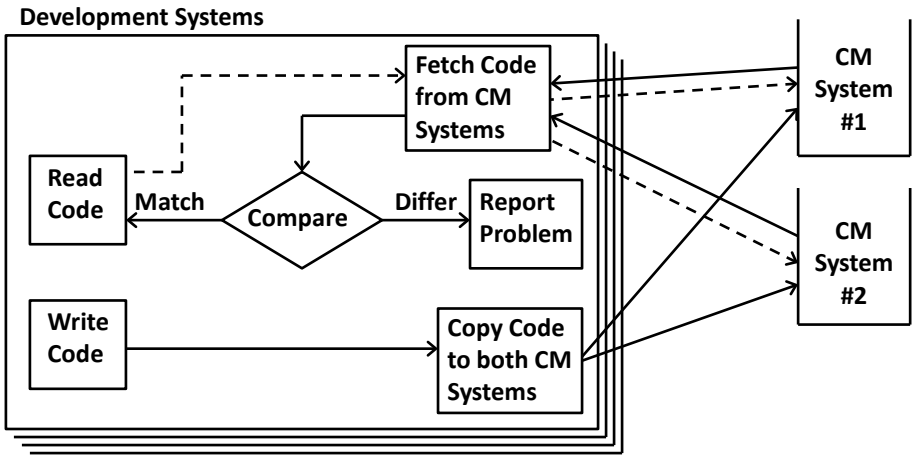

Figure 28 - Cross-Checking CM Systems

Another approach that could be used would be to archive cryptographic hashes of each source file as they are placed into the CM system and then verified as they are extracted. In this approach, instead of a second CM system, a system would store and retrieve the cryptographic hashes created and verified by the development systems. This approach eliminates the problem of re-syncing the CM systems should one crash.

4. Toolchain Compromised

This threat consists of replacing a tool (e.g., a compiler) that is used in the process of converting the source code into the final product. This change allows the compromised tool to insert whatever code is desired into the final product. Figure 29 shows a sample toolchain that might be used to convert the source code into a load image. Any of the tools that transform source code into object code or link object code and libraries into the final load image could add malicious code. Anyone who has write access to the filesystem where the tools are stored can replace/modify one of these tools. This includes system administrators and sometimes CM administrators (since tool versions must be tracked in addition to code versions).

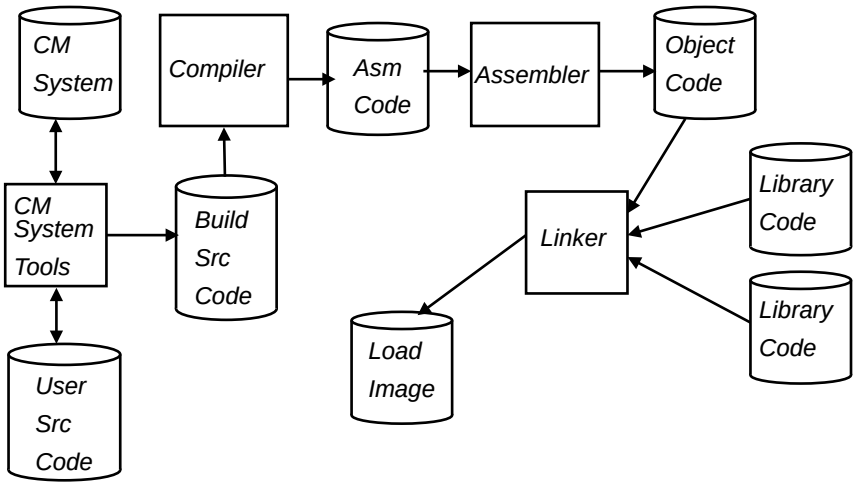

Figure 29 - Example Toolchain

Two approaches are described to mitigate this threat. First, the build tools can be validated as being unchanged since installation prior to the build procedure. This validation can be done by using a cryptographic checksum (e.g., SHA-512). This procedure guards against the possibility that the tools have been modified. It is not foolproof however because the checksums are being generated on a potentially compromised machine (which can report the correct checksums even if the files are compromised). Tools such as Tripwire (www.tripwire.com) and AIDE (aide.sourceforge.net) can reduce the odds of this being undetected. Redundancy can also be used similar to the redundant CM system approach detailed previously. A separate build system can build the installation in parallel and then the results can be compared. This will only work if the attacker has not compromised both machines.

5. User Build Control Compromised (e.g., Makefile Compromised)

This threat is similar to a toolchain compromise, but is done against programmer provided toolchain control files. Files such as Makefiles and scripts are written by programmers to control the build process. They typically specify which source files, object files, and tools are used when building a product. In addition, source file transformations are often performed in this manner (i.e., these files sometimes behave as

part of the toolchain). Some of these files are considered part of the CM system and are written by a CM administrator. These files may not be subject to code walkthroughs as they are not considered to be code. In addition, these build control files are sometimes not even under CM control.

It is recommended that build control files be treated as source code, undergo the same scrutiny as source code, and be stored under CM control. Then the CM controlled version can be used to build the load image.

6. Library File Replaced

This threat is similar to a toolchain threat, but in this case, instead of replacing executable code, a library file (that contains compiled source code that will be linked to the compiled source code) is replaced with a library file containing compromised code. It is called out separately since library files are sometimes managed differently from the toolchain.

The same mitigation approaches used for protecting the toolchain are recommended for this threat.

7. Load Image File Replaced (Assumes Image is not Stored under CM)

Once built, the correctly built load image can still be replaced with a load image containing malicious code. If the image is stored on a filesystem prior to delivery, anyone with write access to the portion of the filesystem containing the image can replace it with a malicious version.

It is recommended that the load image be stored under CM and using the same mitigation approaches used for protecting data in the CM system. Another way to maintain assurance is to use two or more cryptographic hashes of the load image. These cryptographic hashes can later be re-generated from the load image that is retrieved from storage and compared to the original hashes. If the hashes match, the load image has not been tampered with.

Using more than one hash is recommended to guard against the possibility of a flaw being discovered in a cryptographic hash method. Cryptographic hash functions have been broken[5] in the past. For example, SHA-1, GOST, and MD5 have been broken fairly recently (Schneier, 2005), (Mendel, Pramstaller, Rechberger, Kontak, & Szmidt, 2008), (Sotirov, Stevens, Appelbaum, Lenstra, Molnar, Arne Osvik, & de Weger B., 2008).

Since there exists the possibility that an attacker with access to the load image may also have access to the hash repository, I further suggest that the cryptographic hashes be digitally signed.

8. Wrong Load Image File Delivered

During the delivery process itself, the image file can be replaced with a different image file. This is viewed (2011) as an extremely common threat for files delivered electronically. It is also possible to physically switch the delivery media.

For electronic delivery, it is recommended that cryptographic techniques be used. Either the load image file can be digitally signed or a cryptographic checksum of the image file (such as was generated in the previous threat mitigation), delivered via an alternate path, can be used to verify that the image file received is what was sent. The digital signature works by using a public/private key pair. The load image is signed with the private key and the receiver of the load image verifies the signature by comparing against the public key. As long as the private key remains private and an attacker is not able to substitute both the load image and the public key, this approach will provide assurance that the load image has not been modified.

[5] Broken in this context implies that an attacker can generate the same hash result with a different file in computationally feasible time.

9. Load Image File Modified before Use

In this threat, an attacker has access to the user's copy of the load image file before it is loaded into the WRA. I recommend verifying the load image immediately prior to use.

10. WRA Loader Modified

Obviously if the system used to program the WRA with the load image is compromised, it can be made to load anything. Since the loader may be infrequently used, it is recommended that the loader never be connected to a network to reduce the chances that it can be compromised. Systems such as AIDE (aide.sourceforge.net) can be used in combination with a bootable CD to verify that the system is unchanged since installation. Note that many Windows bootable CDs actually use software on the Windows machine to run so it is important to make sure that no potentially compromised execution environment is used to verify that the execution environment is unchanged. One approach is to use a bootable Linux disc to mount and scan a Windows system.

11. WRA Software Modified (after WRA is Programmed)

Once the WRA is loaded, it may still be possible to modify the software and replace the load image with a malicious one. There are multiple ways to handle this but it depends on the specific characteristics of the WRA. Physical security can often be used. In some cases, the load image can be extracted and compared against an extract from a known clean WRA.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. USING CODE FROM UN-TRUSTED SOURCES OR WITH A BROKEN CUSTODY CHAIN

A. VERIFYING SOURCE CODE AND TOOLSET WITH A KNOWN CLEAN LOAD IMAGE

In some cases, there may be a clean load image but an uncontrolled source code and build environment. For example, there may be a wish to use a WRA that has been widely used and has not exhibited malicious behavior. Thus there may be an ability to claim that the pedigree of the WRA is good and that the WRA may be trusted. This in turn implies that the pedigree of the load image within the WRA may be trusted. It also means the original source code and build environment that was used to build the load image may be trusted. However, there is no guarantee that the source code and the build environment today are the same as when the load image we trust was built.

If a change is required to the load image, it would be desirable e to avoid the expense of full Software Assurance testing to re-validate the source code and build environment. Our trust in the binary load image can be leveraged to provide assurance for the current source code and build environment. The approach consists of the following steps:

- Verify that the existing software baseline and toolset used match the clean image
- Make the required changes to the software baseline
- Test the impacted portion of the software baseline

1. Verify Existing Software Baseline and Toolset

Any changes, malicious or not, to either the existing software baseline or the toolset can be expected to result in a malicious (or at least different) load image than the load image created from the clean baseline and tools. Any change to the source code

(other than formatting and comments) would result in a different binary image being generated. Any change to a library (assuming the modified function is utilized) will likewise result in a different binary image.

The reasoning associated with the actual toolchain is different. If an attacker changes the toolchain such that it always produces malicious code, then the binary image will differ. Unlike the library and source code however, the toolchain is an executable and thus it is possible that malicious behavior of the toolchain may also operate based on a trigger. However, if the attacker does not know that such a test will be made ahead of time, they have no ability to select an appropriate trigger. They cannot trigger based on time (be malicious after such a date as they will not be able to determine the date), they cannot trigger of a change in the baseline (as they do not know what change will occur). Thus this type of issue is unlikely, assuming the Systems Engineer does not give the attacker warning.

If it is possible to build the exact same, known clean binary image using the existing source baseline, libraries and toolchain, then (at the moment the source, libraries, and tools were used) we can conclude that these components are also clean. Note that this requires trust in the underlying Configuration Management (CM) system, Operating System (OS), and network environment—otherwise there is no assurance that the files are the same from one moment to the next.

In some cases, it is not convenient have a copy of a known good load image sitting on the shelf. This situation can be addressed by programming a new WRA with the new load image and then extracting the images from the trusted WRA and the new WRA. The images are then compared. Note that it is not generally possible to compare the new load image to the extracted old load image since artifacts of the extraction process are likely to be present (e.g., memory address offsets, checksums).

In some cases it will not be possible to extract a load image from a known good WRA. Some WRAs provide no ability to extract an image. In some cases, Anti-

Tamper[6] efforts take specific steps to prevent such extraction (http://at.dod.mil/). Thus for new efforts it is important that digitally signed copies of load images be kept to avoid the need to extract the image in the future. Note that Anti-Tamper methods that involve preventing image extraction (as opposed to those that rely on dynamic decryption of the load image) also prevent determining if a load image has been replaced with a malicious version once a WRA loaded with a clean load image leaves our control.

Once either the new load image and the old load image or the extracted versions of both the old and new load images are obtained we can compare them to determine what, if any changes are present.

If the load image does not match, bit for bit, then the conclusion is that something has changed. The Systems Engineer will need to determine what has changed and if the changes are malicious. There are techniques for determining which specific items have changed but they are beyond the scope of this thesis. Once malicious code has been ruled out, the build environment can be accepted as clean.

[6] Anti-Tamper is arguably misnamed. It refers to preventing reverse engineering and can be thought of as Anti-Reverse Engineering.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. MAINTAINING TRUST IN CODE BASE AFTER VERIFICATION OF CODE AND TOOL BASE

A. OVERVIEW OF THE APPROACH

We may need to be able to detect any changes in the known clean source code, libraries, and toolchain when changes are required at some point in the future. To this end, a series of cryptographic hashes should be run across all of the source code, libraries, and toolchain (e.g., compilers, linkers, ...) components. These cryptographic hashes can be used at a later time to verify that neither the source code nor the build environment has changed.

The source code is updated as required, with appropriate CM control. At this point, we can re-run the series of cryptographic snapshots and compare each snapshot on a file by file basis. Any hashes that do not match will indicate which files were changed.

For source file changes, a simple "diff"[7] can identify what lines of source code were modified. Changes in the binary files would not be expected unless libraries or tools were upgraded—in which case it is possible to verify that the changes were innocuous.

Once all the changes are accepted, the new set of cryptographic hashes becomes trusted.

Finally, the new load image can be built using the updated baseline, libraries, and toolset.

1. Generating Cryptographic Artifacts

In order to generate cryptographic artifacts, cryptographic hash for each file that is to be tracked is calculated. These hashes can then be collected into a container archive (e.g., Tar, Zip, …) and then digitally signed. Many systems have cryptographic hash

[7] A "Diff" tool compares two or more files and outputs the differences between them

software available. On UNIX and UNIX-like systems, standard tools can be used together to generate artifacts for entire directory trees. Figure 30 below shows a simple example for a UNIX system.

```
#! /bin/sh
#
# Demo program to show cryptographic signature
# generation on a UNIX system
#

SHA=/bin/sha256
CSDB=/tmp/csdb
CODEBASE=.

touch "$CSDB"
find "$CODEBASE" -type f -exec $SHA "{}" >>"$CSDB" \;
```

Figure 30 - Simple Script to Generate Artifacts in Directory Tree

2. Verifying Cryptographic Artifacts

The previously generated cryptographic artifacts can be used to verify that the build environment remains unchanged. When we need to verify that the build environment remains unchanged, we can re-generate the artifacts from the current (potentially different) codebase and compare them to the existing artifact set. If the artifacts match, then we can conclude that the build environment remains unchanged. Figure 31 shows a simple script to verify the artifacts generated earlier.

```
#! /bin/sh
#
# Demo program to show cryptographic signature
# verification on a UNIX system
#

SHA=/bin/sha256
CSDB=/tmp/csdb
CSDBNEW=/tmp/csdb2
CODEBASE=.

touch "$CSDB"; touch "$CSDBNEW"
find "$CODEBASE" -type f -exec $SHA "{}" >>"$CSDBNEW" \;

cat $CSDB $CSDBNEW | sort | uniq -u
```

Figure 31 - Simple Script to Verify Artifacts

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. DOCUMENTING THE SWA PROCESS

A. GOAL STRUCTURING NOTATION (GSN) FOR SOFTWARE ASSURANCE (SWA)

Kelly (1998) proposed using Goal Structuring Notation (GSN) as a method for "clearly expressing and presenting safety arguments." GSN is a type of goal-based assurance. Since 1994 there has been a trend toward explicit goal based approaches for system safety justification (Bishop, P, Bloomfield, R., & Guerra, S., 2004). As such, this type of approach should be familiar to Systems Safety personnel and Systems Engineers alike and thus provide a common framework for addressing and documenting potential issues and mitigation steps. I propose using GSN in a similar manner to document Software Assurance (SwA) cases.

The Goal Structuring Notation (GSN) approach is based on making a "claim" about the software assurance associated with a system. Each claim is then supported with some type of evidence to justify the claim. The basic elements are shown in Figure 32.

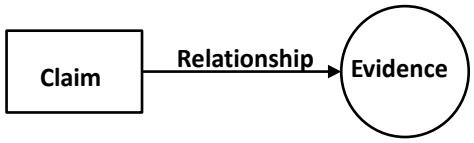

Figure 32 - GSN Elements

As the evidence can be a decomposition of one or more other claims, we end up with a hierarchical graphical "argument" supporting a SwA claim. This relationship is shown in Figure 33. Note that the "context" elements from GSN, although they could be used to provide amplifying information about the reasoning. Also note that when multiple sub-claims or pieces of evidence are presented, all sub-elements must be satisfied for the claim to be satisfied (i.e., this is a logical AND relationship).

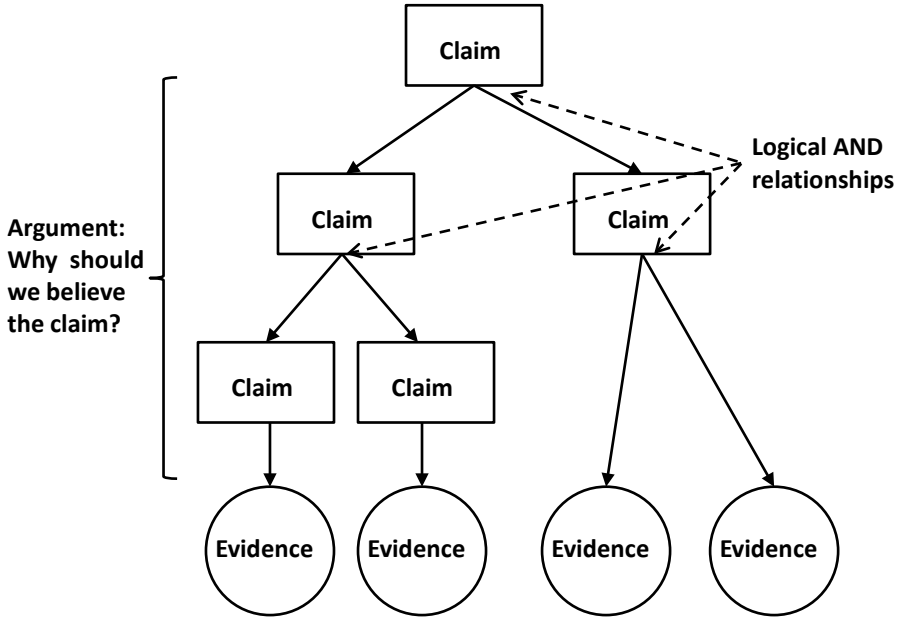

Figure 33 - Hierarchical Claim Argument

Figure 34 shows a sample system level SwA case. Note that the system claim is supported by claims about its components (WRAs that contain software).

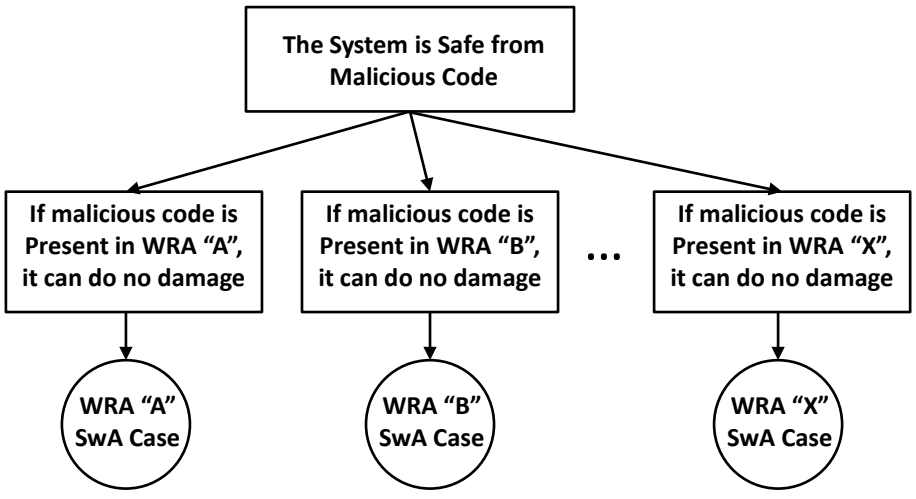

Figure 34 - Example Top Level SwA Case

The claims about each WRA are supported by evidence (in this case, a separate SwA case for each WRA). In some cases, the system itself may require additional supporting arguments to rule out system design threats.

Figure 35 shows a sample SwA case for WRA "A." In this example, WRA "A" has no external interfaces (so there is no need to worry about external vulnerabilities). This is supported by the System Architecture diagram (that will show that WRA "A" has no external connections). Note that any change to this piece of evidence would require re-evaluation on this claim. Internal WRAs can contain malicious code, so we verify that no "trigger" information (e.g., date, altitude, GPS location, etc …) is available to the WRA on its internal interfaces. Although a review of an Interface Design Description (IDD) or similar document might be trusted, the actual interface (in case non-documented data is present) may also require monitoring. The WRA could simply fail after a few hours of run-time (it is assumed that the WRA can tell how long it has been running), or may even contain overt malicious code so there is a need to verify that the testing was adequate to detect a problem in these cases.

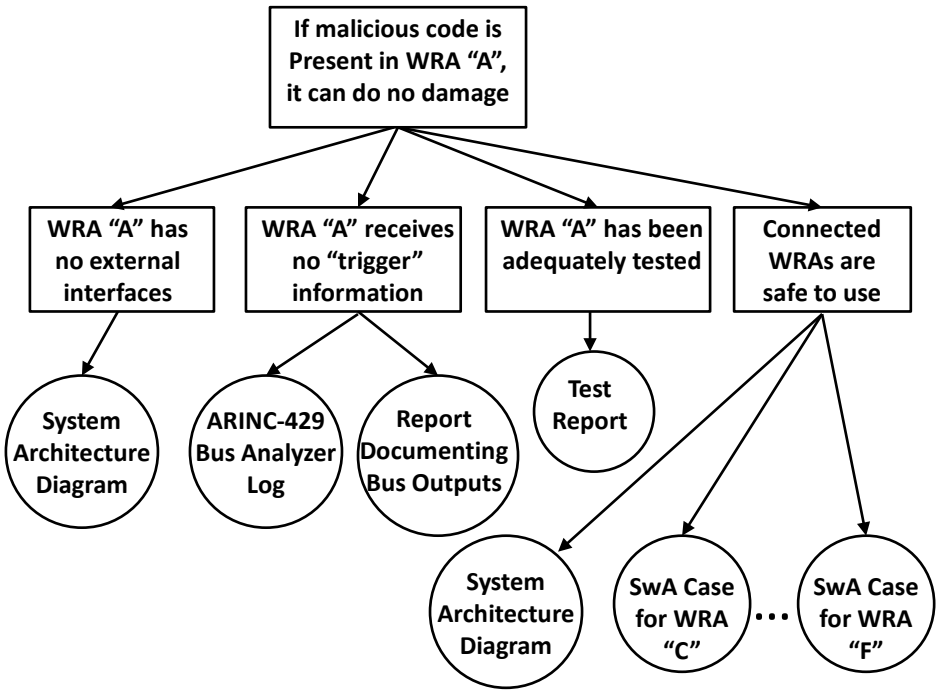

Figure 35 - Example SwA Case for WRA "A"

Since it might be possible for multiple-level attacks to exploit the WRA on an internal interface, there may also be a wish to have a SwA case for all the other connected WRAs. Since protecting against multi-level attacks greatly expands the work-load, there may be a wish to specify that such protection is excluded if the overall risk posed by malicious code to the system is limited.

IX. APPLICATION OF STUDY

A. RELEVANT SYSTEM CHARACTERISTICS

The approaches discussed in this thesis rely on a few key system characteristics. Other systems that share some or all of these attributes of embedded airborne systems may find the work useful. The key characteristics of embedded airborne systems that we depend on are:

- Embedded System – because the system is embedded, the software associated with the WRAs (system components) is static, with a limited ability for an attacker to replace the software.

- No network connectivity – because airborne systems are (for the most part) devoid of network connectivity, we tend to have a limited number of data sources to worry about.

- Limited connectivity – because airborne systems tend to have point to point connections, we can make a distinction between external and internal interface threats.

- Significant testing – because airborne systems have many safety critical components, special testing (specifically DO-178B statement coverage testing in this case) is required. Other systems may have similar testing requirements for other, non-safety reasons (e.g., Information Assurance critical subsystems)

B. EXAMPLES

Some examples of specific systems that may find the work in this thesis relevant are:

- Airborne embedded systems – this thesis was specifically designed to address these

- Industrial embedded systems – many industrial systems use embedded control systems. Devices such as Computer Numerically Controlled (CNC) Milling Machines and Lathes are extremely dangerous if a software failure occurs. As such, there is a specific safety risk for these systems that requires extensive testing. The systems have limited connectivity and are typically dedicated to the control of a single machine.

- Medical equipment – medical testing equipment, life support equipment, and implanted devices share the required characteristics. Some medical test equipment (e.g., an X-Ray machine) has the ability to cause harm if it malfunctions. Even when the system cannot directly cause harm, misinformation from the system can lead to inappropriate action (or inaction) based on the incorrect results. Life support equipment must operate correctly or death may occur. Similarly implanted medical devices such as pacemakers or drug delivery systems also carry the risk of death to a patient should they malfunction. Most of these systems have extremely limited connectivity and are associated with a specific device.

- Electronic voting systems – Electronic voting systems are typically embedded systems with extremely limited interfaces. Some systems (e.g., the Accuvote TS) have been compromised and de-certified. As a result, the systems are expected to undergo rigorous testing.

- Air traffic control systems – Many sub-systems within the nation's air traffic control system are embedded systems with limited connectivity. They have the ability to cause loss of life should they fail, as such they require significant testing.

- Nuclear power control systems – many of the system components are safety critical, control limited aspects of the system, have limited connectivity, and have significant testing requirements.

- National electric grid control systems – the national electric grid system is expected to provide real-time, fault tolerant control of the nation's electric supply. Embedded systems control generators and switches and can cause great damage to the infrastructure should they fail.

THIS PAGE INTENTIONALLY LEFT BLANK

X. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS AND RECOMMENDATIONS

This thesis answers the following questions:

- How can a System Engineer mitigate the risks posed by malicious code in embedded systems used in aircraft?

  A Systems Engineer can mitigate the risks by a combination of: malicious code detection methods; reduction in the chances that malicious code will be inserted into the software anywhere in the development, deployment, and maintenance life cycle; techniques to prevent extant malicious code from causing harm; and tight control of the integrity of software.

- What methods can we use to detect the presence of malicious code in embedded systems used in aircraft?

  A Systems Engineer can utilize a combination of static and dynamic detection techniques to locate malicious code. Although static analysis is not particularly effective, certain dynamic detection techniques can cause software that contains malicious code to fail during testing. Certain techniques such as code-walkthroughs and coverage testing can be leveraged to detect the presence of overt embedded malicious code.

- How can we reduce the risk that malicious code will be inserted into the source code for embedded systems used in aircraft?

  Simple changes to the software development process can be made to reduce the chances that malicious code is inserted into an embedded aircraft system. These include techniques such: as requiring software check-in to a Configuration Management (CM) system before code walkthroughs and testing; encryption techniques to verify the integrity of

the CM system and build toolchain; and formal documentation procedures to reduce the chance that a vulnerability will remain unaddressed.

- How can we prevent malicious code in embedded systems used in aircraft from causing harm?

  By controlling the information available to a Weapons Replaceable Assembly (WRA), a Systems Engineer can often prevent potential "trigger conditions" from occurring within the malicious code. This in turn will result in a condition whereby the malicious code (if present) will either always fail (and be detected during testing), or never fail (and thus cause no harm).

- How can we maintain the integrity of the software in an embedded system?

  The integrity of software throughout the development, deployment, and maintenance life cycle of a program can by maintained through the use of cryptographic hashes and digital signatures. These techniques can be augmented through the use of multiple independent CM systems. When properly employed, these techniques can provide assurance that the software has: not been modified; or that only known modifications have occurred.

A basic tenet of Systems Engineering is to bound the problem. In this case, it is necessary to be completely certain that software contains no malicious code. The goal should be to state that if any malicious code is present it cannot cause harm. This is an important distinction because it may well prove impossible to prove the negative. After all, if covert malicious code is present but does not actually do anything malicious, how would we detect it?

By using system level information combined with basic knowledge about how malicious code operates, a Systems Engineer has tools available that are not available to

WRA manufacturers, Programmers, or Software Engineers. This system level information can allow the Systems Engineer to quantify the risks posed by malicious software and in many cases completely mitigate them. Although certain WRAs (e.g., those that require "trigger" information to operate), cannot (using the techniques presented in this thesis) have their malicious code risk reduced to zero, decomposition of these WRAs into sub-elements and a combination of static and dynamic analysis on troublesome elements can greatly reduce the remaining risk.

By documenting a Systems Software Assurance case using a GSN approach, a Systems Engineer can effectively track and communicate the SwA mitigation steps along with the reasons why each of the steps are needed. This communication may help "sell" the expense that implementing the SwA cases will incur.

Certain characteristics of Embedded Airborne Avionics software can be leveraged to reduce the cost associated with implementing SwA. Many system and software safety requirements can be used as supporting evidence for SwA claims at no additional cost. For example, DO-178B statement coverage testing requirements can mitigate the risk of overt malicious software and the associated hazard analysis or safety assessment may be helpful in determining which system WRAs require SwA efforts and which do not. System testing requirements may be helpful in meeting some additional SwA evidence requirements.

The basic approach taken in this thesis is "divide and conquer" where we decompose the system risk down into risks posed by individual WRAs and in some cases further decompose individual WRAs to allow similar analysis techniques to be used on WRA elements.

It is recommended that a system level approach (such as presenting in this thesis) be used to allocate SwA risks to individual WRAs and that high risk WRAs be primarily handled by managing their inputs. For WRAs that cannot have potential "trigger" information removed, the WRA should be further decomposed (to limit the size of the

problem) and the techniques presented in the static analysis and particularly the dynamic analysis sections be used to reduce the risk posed.

B. AREAS FOR FURTHER RESEARCH

Strategies to deal with WRAs that depend on trigger information should be more fully developed and ways to more fully calculate the residual risk should be developed. Additionally, work to deal with statistical triggers may prove useful. Specific research into specific characteristics of other software systems such as those called out in Chapter IX may allow different cost saving strategies for implementing the basic concepts detailed in this thesis.

LIST OF REFERENCES

AIM. (2010). *ARINC 429 specification overview*. Retrieved from http://www.aim-online.com/pdf/OVIEW429.PDF

Albright, D., Brannan, P., & Walrond, C. (2010). *Did Stuxnet take out 1,000 centrifuges at the Natanz Enrichment Plant?* Institute for Science and International Security. Retrieved from ISIS Reports Online: www.isis-online.org/uploads/isis-reports/documents/stuxnet_FEP_22Dec2010.pdf

Australian Transport Safety Bureau. (2008). *Qantas Airbus A330 accident media conference.* Retrieved July 27, 2011, from http://www.atsb.gov.au/newsroom/2008/release/2008_43.aspx

Ballard, L., Chou, D., Chuang, J., Doshi, S., & Kimball, P. (2004). *600.643 - Group 2 report: Hiding code*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.7427&rep=rep1&type=pdf (DOI: 10.1.1.117.7427)

Bishop, P., Bloomfield, R., & Guerra, S. (2004). The future of goal-based assurance cases. *Proc. Workshop on Assurance Cases*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.1568&rep=rep1&type=pdf (DOI: 10.1.1.64.1568)

Broad W., Markoff J., & Danger D. (January 15, 2011). Israeli test on worm called crucial in Iran nuclear delay. *The New York Times*. Retrieved from http://www.nytimes.com/2011/01/16/world/middleeast/16stuxnet.html?_r=2&pagewanted=1 ital magazine/newspaper/journal titles

Bulba & Kil3r. (2000). Bypassing StackGuard and StackShield. *Phrack Magazine*, 0xa(0x38), May 2000. Retrieved from web.eecs.utk.edu/~dunigan/cs594-cns/p56–0x05.pdf

Charette, R. (2009). This car runs on code. *IEEE Spectrum*, Feb 2009, 7649. Retrieved from http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code

Checkoway S., Halderman J., Feldman A., Felten E., Kantor B., & Shacham H. (2009). *Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage.* USENIX 2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections. Retrieved from http://www.usenix.org/event/evtwote09/tech/full_papers/checkoway.pdf

Dynamic program analysis. (n.d.). In *Wikipedia*. Retrieved June 9, 2011, from http://en.wikipedia.org/wiki/Dynamic_program_analysis

Erickson, J. (2003). *Hacking: The art of exploitation*. No Starch Press. p. 23.

Fildes, Jonathan. (23 September 2010). Stuxnet worm targeted high-value Iranian assets. *BBC News*. Retrieved from http://www.bbc.co.uk/news/technology-11388018

FADEC. (n.d.) Wikipedia. Retrieved July 14, 2011, from http://en.wikipedia.org/wiki/FADEC

Gorman, S. (2009, April 8). Electricity grid in U.S. penetrated by spies. *Wall Street Journal*. Retrieved from http://online.wsj.com/article/SB123914805204099085.html

Greenberg, A. (2010). The Bounty For An Apple Bug: $115,000. Forbes: The Firewall, the world of security. Retrieved from: http://blogs.forbes.com/firewall/2010/03/25/the-bounty-for-an-apple-bug-115000/

Iran says cyber foes caused centrifuge problems. (November 29, 2010). *Reuters.* Retrieved from http://af.reuters.com/article/energyOilNews/idAFLDE6AS1L120101129

John Draper. (n.d.). *John Draper* in Wikipedia. Retrieved June 6, 2011, from http://en.wikipedia.org/wiki/John_Draper

Johnson, D. (2007). *Raptors arrive at Kadena*. U.S. Air Force. Retrieved July 27, 2011, from http://www.af.mil/news/story.asp?id=123041567

Kelly, T. (1998). *Arguing Safety — A Systematic Approach to Safety Case Management.* Doctoral thesis. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.8163&rep=rep1&type=pdf

King, L. (2011, July 11). *Chinook crash: The unanswered software questions*. Computerworld UK. Retrieved July 27, 2011, from http://www.computerworlduk.com/in-depth/public-sector/3291437/chinook-crash-the-unanswered-software-questions/

King, L. (2011, July 13). Chinook Mull of Kintyre crash pilots exonerated after 17 years. *Computerworld UK*. Retrieved July 27, 2011 from http://www.computerworlduk.com/news/public-sector/3291139/chinook-mull-of-kintyre-crash-pilots-exonerated-after-17-years/

Kuhl, J. (2008, February 12). How to hack into a Boeing 787. *Fox News*. Retrieved from http://www.foxnews.com/story/0,2933,331088,00.html

Lists of programming languages. (n.d.). In *Wikipedia*. Retrieved June 7, 2011, from http://en.wikipedia.org/wiki/Lists_of_programming_languages

Livshits, B. (2006). *Improving Software Security with Precise Static and Runtime Analysis* (Doctoral dissertation). Retrieved from http://research.microsoft.com/en-us/um/people/livshits/papers/pdf/thesis.pdf

Lopez, S. (2000). *The Real Effects of the Microsoft Hack-in*. SANS Institute. Retrieved June, 2011 from http://www31.giac.org/paper/gsec/188/real-effects-microsoft-hack-in/100662

Mendel, F., Pramstaller, N., Rechberger, C., Kontak, M., & Szmidt, J. (2008) *Cryptanalysis of the GOST Hash Function.* Advanced in Cryptology (Lecture Notes in Computer Science) 5157(LNCS), 162–178. Retrieved from https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=36649 (DOI: 10.1007/978–3-540–85174–5_10)

Microsoft. (2004). *Microsoft Security Bulletin MS04–028: Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution (833987) (Version 3.0).* Retrieved from http://www.microsoft.com/technet/security/bulletin/ms04–028.mspx

Packham, B., Dunn, M. (2008). Laptops and other devices feared linked to plunge. *Perth Now*. Retrieved from http://www.perthnow.com.au/news/jet-plunge-laptop-fear/story-e6frg12c-1111117687849

Panda Security. (2010). *The Cyber-Crime Black Market: Uncovered*. Retrieved from http://press.pandasecurity.com/wp-content/uploads/2011/01/The-Cyber-Crime-Black-Market.pdf

Ping of death. (n.d.). In *Wikipedia*. Retrieved May 27, 2011, from http://en.wikipedia.org/wiki/Ping_of_death

Ragan, S. (2011). *Three military contractors linked to post-RSA attacks.* Retrieved June 13, 2011 from http://www.thetechherald.com/article.php/201122/7225/Three-military-contractors-linked-to-post-RSA-attacks

Roberts, J. (anchor). (2007, February 24). *This Week At War*. [Television broadcast]. [Transcript]. CNN. Retrieved July 27, 2011, from http://transcripts.cnn.com/TRANSCRIPTS/0702/24/tww.01.html

RSA. (2011). *Open Letter to RSA SecurID Customers.* Retrieved June 13, 2011, from http://www.rsa.com/node.aspx?id=3891

RTCA, Inc. (1992). *Software Considerations in Airborne Systems and Equipment Certifications*. RTCA, Inc.

Safram. (2011). *Engine Control Units*. Retrieved from http://www.sagem-ds.com/spip.php?rubrique16

Schneier, B. (2005) *Cryptanalysis of SHA-1*. Schneier on Security. Retrieved from http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

Sotirov, A., Stevens, M., Appelbaum, J., Lenstra, A., Molnar, D., Arne Osvik, D., & de Weger B. (2008). *MD5 considered harmful today -- Creating a rogue CA certificate.* Retrieved from http://www.win.tue.nl/hashclash/rogue-ca/

Static program analysis. (n.d.). In *Wikipedia*. Retrieved June 9, 2011, from http://en.wikipedia.org/wiki/Static_code_analysis

Telang, R., & Wattal, S. (2007). An Empirical Analysis of the Impact of Software Vulnerability Announcements on Firm Stock Price. *IEEE Transactions on Software Engineering*, 33(8), 544–557.

Thompson, K. (1984). *Reflections on Trusting Trust.* Communication of the ACM, 27(8): 761–763.

Weiss, F. (2000). *Duping the Soviets: The farewell dossier*. White Paper. Retrieved from https://www.cia.gov/library/center-for-the-study-of-intelligence/kent-csi/vol39no5/pdf/v39i5a14p.pdf

Wesolowski, K. (2006). *OpenSolaris Supported Compiler Policy, V1 [Draft 2]*. Retrieved from http://mail.opensolaris.org/pipermail/tools-gcc/2006-March/000093.html

Zero Day Initiative. (2011). Retrieved June 28, 2011, from http://www.zerodayinitiative.com/advisories/upcoming/

Zetter, K. (2008, January 9). FAA responds to Boeing security story. *Wired.* Retrieved from http://www.wired.com/threatlevel/2008/01/faa-responds-to/

Zitser, M., Lippman, R., & Leek, T. (2004). Testing static analysis tools using exploitable buffer overflows from open source code. *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6. Retrieved from www.ll.mit.edu/mission/communications/ist/corpora/04_TestingStatic_Zitser.pdf

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California