A Proven Methodology for Developing Secure Software and Applying It to Ground Systems

2/29/16
Brandon Bailey
brandon.t.bailey@nasa.gov
304-629-8992

NASA’s IV&V Program
Safety and Mission Assurance (SMA) Office
Information Assurance/Cybersecurity Support
http://www.nasa.gov/centers/ivv
• Tutorial E Part 1:
  – Section 1: Cyber Threat – Who, What and Why
  – Section 2: Defense-In-Depth
  – Section 3: Secure Software Engineering Steps
  – Section 4: Errors, Weaknesses and Exploits
  – Section 5: Threat Modeling
  – Section 6: Testing
  – Section 7: Resources

• Tutorial E Part 2:
  – Section 1: Ground Systems Overview
  – Section 2: Secure Software Development
  – Section 3: Defense in Depth for Ground Systems
  – Section 4: What Now?
Defining “Ground Systems”

Spacecraft Ground Systems encompasses the entire system, beginning with issuing the command from the MOC up until it emits from the antenna to the reception of radio signals down at the antenna to displaying telemetry on the MOC computer.
• Tutorial will focus on the software developed for the
  • Mission Operations Center (MOC)
  • Mission planning area
  • Software development environment
CCSDS was founded in 1982 by the major space agencies of the world, the CCSDS is a multinational forum for the development of communications and data systems standards for spaceflight. 60+ standards published serving 500+ missions

Security Threats Against Space Missions was developed to provide mission planners with an overview on threat assessment as well as the common threats and threat sources that exist for various categories of civilian space missions.

**Space Elements**
- Space Debris
- Replay
- Link Jamming
- Unauthorized Access
  - Software Threats

**System / Network**
- Replay
- Link Jamming
- Interception (theft)
- Unauthorized Access
  - Software Threats
  - Traffic Analysis

**Control**
- Interception
  - Software Threats
- Social Engineering
- Physical Attacks

**Control**
- Replay
- Unauthorized Access
  - Software Threats
- Social Engineering
- Physical Attacks

Hardware Failures
Security Threats Against Space Missions was developed to provide mission planners with an overview on threat assessment as well as the common threats and threat sources that exist for various categories of civilian space missions.
### Applicable Threats to Space Missions

<table>
<thead>
<tr>
<th>Threats in Space</th>
<th>Impacts</th>
<th>Could Software Be Involved?</th>
</tr>
</thead>
</table>
| **Data Corruption** | • Modification of information  
• System damage | Yes; SW attacks could result in data corruption |
| **Ground Facility Physical Attack** | Loss of command, control and data | No |
| **Interception** | Loss of sensitive data | No |
| **Jamming** | • Loss of Command telemetry link  
• Loss of access to resources | No |
| **Denial-of-Service** | Loss of access to resources | Yes; SW DoS attacks are common and can affect both ground, flight and web applications |
| **Masquerade** | • Potential to disrupt operations (uplink)  
• Potential to receive false information (downlink) | Yes; SW protections can be placed to prevent |
| **Replay** | System damage (possible safety of life issues) | Yes; SW protections can be placed to prevent |
| **Software threats** | • Undesirable events  
• System damage  
• Enable other threats (i.e. Jamming, DoS) | Yes |
| **Unauthorized Access** | • Disruption of operations  
• System damage (possible safety of life issues) | Yes; SW protections can be placed to prevent or SW can be used to gain unauthorized access |
| **Tainted Hardware Components** | • Hidden, Malicious capabilities  
• System instability  
• System damage  
• Undesirable System effects | No |
Cybersecurity in the space domain
Isn’t ONLY an IT function
Security is a part of Mission Success

- Web sites/servers, email, workstation patching, etc.
- CIO infrastructure focused

Must counteract the threat landscape for Mission environments with Defense in Depth

Mission Targets / Enterprise Risk
- Software Security (COTS, FOSS, Custom, etc.)
- Network Layer (Routers, Firewalls, etc.)
- Computer Network Defense (IPS/IDS, Sensors, Continuous Monitoring, etc.)
- Industrial Control Systems (ICS)
- Supply Chain...

- Multiple stakeholders (CIOs, Network Engrs, SW Developers, Project Managers, etc.)
Custom SW – Gets Exploited!

Source: Microsoft Security Intelligence Report, Vol. 17, June 2014
Multiple vulnerabilities could adversely impact Mission Operations (Architecture, SW, IT, etc.)

- **Preventing vulnerabilities**
  - Levying requirements from the top in policy, contracts, etc.
    - PPPs, A&A process, SW development, etc.
  - During mission design/planning
    - Designing security in
    - Secure software development
    - Rigorous mission assurance (SW Assurance, IV&V, etc.)
    - Awareness, training, tooling
  - Supply chain – know the parts you are building with...
    - Hardware
    - Software (i.e. COTS and Open Source)

- **Discovering vulnerabilities**
  - Once vulnerabilities are introduced into operation – then what?
    - Continuous monitoring
    - Vulnerability assessments
    - Penetration testing
• Examples of requirements government agencies may invoke
  – DOD
    • [Program Protection & System Security Engineering](#)
  – NASA
    • NPRs [2810, 7150.2B, 7120.5E](#), and the SW Assurance Standard/Handbook (under draft)
  – NIST [800-53](#)
    • Example control for SW:
      – [RA-5](#) Vulnerability Scanning
  – European Space Agency (ESA) - (under draft)
    • ESSB-ST-E-008 - Secure Software Engineering Std
    • ESSB-HB-E-007 – Secure Software Engineering Handbook

• Other resources to help identify requirements
  – [Security Quality Requirements Engineering (SQUARE)](#)
  – [Microsoft Security Development Lifecycle](#)
Not Baking in Security

- Traditional cost of change curve depicts how discovering defects last impacts cost

Cost of Change

Requirements Analysis and Design Coding Testing in the Large Production

“Bake It In”

If Bug=Exploited
damage could be more than monetary

Loss of Mission Obj(s)
Loss of Mission
Loss of Life
Secure software development begins where all software begins!

Secure software is an end-to-end development concept, not patchwork.

Secure Development

• Utilize **Best Practices**
  – List is from NASA’s Secure Coding Portal

• Coding Standards (Ex. CERT C, C++ or JAVA Stds)
  – Ex: Don’t use unsafe functions (**Flawfinder**)

• Integrate tools into development environment
  – Code Analyzers (i.e. **Klockwork**, Fortify, Flexelint, CodeSonar, Sonatype, BlackDuck, etc.)
  – Great resource for identifying tools
    • **Report** | **Spreadsheet**
Secure Development (cont.)

- Use information from DHS:
  - Common Weakness Enumeration (CWE), Common Vulnerabilities and Exposures (CVE), and Common Attack Pattern Enumeration and Classification (CAPEC)
  - Plan for Defense in Depth and not solely on protective perimeter
    - Historically developers depend/plan for Firewalls to protect vice designing in SW
    - Securing the development environment (i.e. prevent injecting of malicious code)

- Training
  - Free:
    - FedVTE  Ex: Software Assurance Executive Course (SAE)
    - SAFECODE
    - Secure Coding and Standards Tutorial (NASA Only)
  - Paid: (Ex: Cigital, Pluralsight)
Not Baking in Security
Ex: Actual Ground Software

- Insecure random number generator was used to generate passphrases that control access to VPNs and other network resources
- Could enable someone to monitor or interfere with a system and be undetectable
  - If this code was deployed with weak symmetric keys, the supposedly "secure" data-links between these devices would be vulnerable to a "man-in-the-middle" attack.
- There were several instances throughout the code
  - Klockwork discovered these during static code analysis

```java
for (int i = 0; i < randomStringLength; i++)
{
    // randomly select a char for the random list
    int selChar = (int) (Math.random() * (randomList.length() - 1));
    if (allowSpecialCharacters) {...}

    // make sure we pick a non special character
    while (specialCharacters.indexOf(randomList.charAt(selChar)) != -1)
    {
        selChar = (int) (Math.random() * (randomList.length() - 1));
    }

    // make sure we pick a special character from the list
    while (specialCharacters.indexOf(randomList.charAt(selChar)) < 0)
    {
        selChar = (int) (Math.random() * (randomList.length() - 1));
    }
```

Resource Describing Math.random:
Not Baking in Security
Ex: Actual Ground Software

• Code calls a generic exception handler
  – Typically is done when a developer assumes they can only get known types of exceptions
  – However, depending on the source of the exception (input stream for example) someone can try to cause a different exception resulting in unpredictable behavior (i.e. DoS)
  – Also with a ground system, you want to fail-fast
    • Catching and ignoring fatal exceptions makes a program less robust since it will try to carry on as if nothing happened in the worst of conditions
    • Immediately report at its interface any failure
    • Don't pretend like nothing happened, because it's going to get worse

• Klockwork discovered these during static code analysis
• Lacking the appropriate code in a finally block (java exception handling)
  – Using something and then call a close, doesn’t mean it will actually close if an exception is encountered either in the use or the close call
  – A finally block helps assure proper closure and deallocation
  – This can be for any type of resources (file, database, etc.)
• Resource leak could use up all resources, causing the system to become unresponsive after excessive or continued use, reducing dependability (i.e. DoS)
• Klockwork discovered these during static code analysis

```java
public ICommunicationProcessorXXX connectUsingRemoteXXX()
    throws GeneralSecurityException, NamingException
{
    try
    {
        InitialContext context = new InitialContext(properties);
        m_ctmProcessorRemote =
            (ICommunicationProcessorXXX) context.lookup(CommunicationProcessorXXXClientUtils.PROCESSOR_REMOTE_JNDI_NAME);
    }
    catch (NamingException e) {...}

    return m_ctmProcessorRemote;
}
```
Low Hanging Fruit
Unsafe Functions

• Stop using known unsafe functions and always do bounds checking if you are copying to a buffer
  – Even if you think you know what you are copying from and it’s limited, defensive coding is best.
• Some samples of unsafe functions due to allowed writing with no regard to buffer size
  | memset | memcpy | strcat | strcmp | strcpy | strlen |
  |   |   |   |   |   |   |
  | sprintf | strncpy | _iota | sscanf | wcslen |
• Most of these are unsafe due to allowed writing with no regard to buffer size
  – strncpy, _iota, sscanf, & wcslen have safer _s varieties (ex. _iota_s) that require a buffer size to be specified
  • Resource: Security Development Lifecycle (SDL) Banned Function Calls
  • Resource: Stack Overflow Post
• Free tool to help find unsafe functions - Flawfinder
Demo

• Demo Flawfinder
Low Hanging Fruit
CERT Rules

• For legacy code:
  – MSC00-C. Compile cleanly at high warning levels
    • The process of fixing compiler warnings will probably quash some other vulnerabilities.
  – ERR33-C. Detect and handle standard library errors
    • Include any program functions that give some kind of error indication
      – If a function returns some special value on error, such as NULL, your calls to that function should always check its return value
• For new code
  – ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy
    • This is where programs fail the most easily. They fail to check for errors because the developers
don't know what to do if an unexpected error occurs.
  – MEM00-C. Allocate and free memory in the same module, at the same level of abstraction
    • A design issue, but not following it will get your code into hot water quickly.
  – MEM12-C. Consider using a goto chain when leaving a function on error when using and releasing resources
    • More specifically, make sure your code frees resources even if errors occur.

• For both new and existing code: execute static code analysis tools to determine weaknesses
  • Free ones are a good place to start; See slide 14 for commercial ones
    – Cppcheck
    – Rosecheckers
    – Splint
    – Find Bugs
    – RATS
    – Flawfinder
    – SWAMP ✭
Info from DHS

**CWE:**
- Serves as a common language for describing software security weaknesses in architecture, design, or code
- Provides a:
  - Standard measuring stick for software security tools targeting these weaknesses
  - Common baseline standard for weakness identification, mitigation, and prevention efforts
- Utilize CWE to better understand, identify, fix, and prevent weaknesses and vulnerabilities

**CVE:**
- Identifies publicly known information security vulnerabilities and assign them a CVE_ID.
- Scored 1 to 10 on CVSS scale

**CAPEC:**
- Community-developed list of common attack patterns
- Comprehensive schema and classification taxonomy
- International in scope

Taking into account attack pattern and any other factors to generate list of CWEs that are critical. Tools report findings in CVEs (known) and CWEs (potential) -> Identify then Fix!
For NASA, research & analysis has been performed by the IV&V Program to identify the Top 25 CWEs for Ground Systems.

The following categories are part of the formula for CWSS:

- **Base Finding**
  - Technical Impact
  - Acquired Privilege
  - Acquired Privilege Layer
  - Internal Control Effectiveness
  - Finding Confidence

- **Attack Surface**
  - Required Privilege
  - Required Privilege Layer
  - Access Vector
  - Authentication Strength
  - Level of Interaction
  - Deployment Scope

- **Environmental**
  - Business Impact
  - Likelihood of Discovery
  - Likelihood of Exploit
  - External Control Effectiveness
  - Prevalence

Each factor in the category is assigned a value. These values are converted to associated weights and a category sub-score is calculated. The three sub-scores are multiplied together, which produces a Common Weakness Scoring System (CWSS) score. Higher the score, higher it ranks.
## Top 25 CWEs

**Ground Systems v2.0**

<table>
<thead>
<tr>
<th>Rank</th>
<th>CWE ID</th>
<th>CWE Title</th>
<th>Rank</th>
<th>CWE ID</th>
<th>CWE Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>312</td>
<td>Cleartext Storage of Sensitive Information</td>
<td>13</td>
<td>403</td>
<td>Exposure of File Descriptor to Unintended Control Sphere</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>('File Descriptor Leak')</td>
</tr>
<tr>
<td>2</td>
<td>88</td>
<td>Argument Injection or Modification</td>
<td>14</td>
<td>314</td>
<td>Cleartext Storage in the Registry</td>
</tr>
<tr>
<td>3</td>
<td>77</td>
<td>Improper Neutralization of Special Elements used in a Command</td>
<td>15</td>
<td>835</td>
<td>Loop with Unreachable Exit Condition ('Infinite Loop')</td>
</tr>
<tr>
<td></td>
<td></td>
<td>('Command Injection')</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>23</td>
<td>Relative Path Traversal</td>
<td>16</td>
<td>833</td>
<td>Deadlock</td>
</tr>
<tr>
<td>5</td>
<td>73</td>
<td>External Control of File Name or Path</td>
<td>17</td>
<td>764</td>
<td>Multiple Locks of a Critical Resource</td>
</tr>
<tr>
<td>6</td>
<td>798</td>
<td>Use of Hard-coded Credentials</td>
<td>18</td>
<td>421</td>
<td>Race Condition During Access to Alternate Channel</td>
</tr>
<tr>
<td>7</td>
<td>353</td>
<td>Missing Support for Integrity Check</td>
<td>19</td>
<td>119</td>
<td>Improper Restriction of Operations within the Bounds of a Memory Buffer</td>
</tr>
<tr>
<td>8</td>
<td>732</td>
<td>Incorrect Permission Assignment for Critical Resource</td>
<td>20</td>
<td>318</td>
<td>Cleartext Storage of Sensitive Information in Executable</td>
</tr>
<tr>
<td>9</td>
<td>22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory</td>
<td>21</td>
<td>242</td>
<td>Use of Inherently Dangerous Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>('Path Traversal')</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>78</td>
<td>Improper Neutralization of Special Elements used in an OS Command</td>
<td>22</td>
<td>497</td>
<td>Exposure of System Data to an Unauthorized Control Sphere</td>
</tr>
<tr>
<td></td>
<td></td>
<td>('OS Command Injection')</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>290</td>
<td>Authentication Bypass by Spoofing</td>
<td>23</td>
<td>772</td>
<td>Missing Release of Resource after Effective Lifetime</td>
</tr>
<tr>
<td>12</td>
<td>20</td>
<td>Improper Input Validation</td>
<td>24</td>
<td>681</td>
<td>Incorrect Conversion between Numeric Types</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>25</td>
<td>192</td>
<td>Integer Coercion Error</td>
</tr>
</tbody>
</table>

Rankings are currently under peer review.

Version 2.0 of Top 25 now includes Common Attack Patterns.
• Demo Fortify w/ CWE Reporting
Origin Analysis: Secure SW Supply Chain

• From Institute for Defense Analyses (IDA) **SOAR Report** – “Origin analyzers are tools that analyze source code, bytecode, or binary code to determine their origins (e.g., pedigree and version).”

• Origin Analysis can be used to reduce the software supply chain risk
  – Identifies **CVEs** that may be present in re-used open source libraries/code
  – Also identifies potentially licensing issues

• Examples of tools
  – **Sonatype**
    • Binary scanner; Works best on JAVA
  – **Black Duck HUB**
    • Provides binary and source tree scanning; Support C/C++ as well has JAVA
  – **OWASP Dependency Check**
    • Currently Java, .NET, Ruby, Node.js, and Python projects are supported; additionally, limited support for C/C++ projects is available for projects using CMake or autoconf.
# Examples from Ground Systems

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Affected File</th>
<th>Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2014-0003: Allows remote attackers to execute arbitrary Java methods via a crafted message.</td>
<td>camel-core-1.5.4.0-fuse.jar</td>
<td>Upgrade Jar file to 2.11.4 or newer</td>
</tr>
<tr>
<td>CVE-2009-4611: Allow remote attackers to modify a window’s title, or possibly execute arbitrary commands or overwrite files, via an HTTP request</td>
<td>jetty-6.1.14.jar; jetty-util-6.1.14.jar</td>
<td>Upgrade Jar file to 6.1.25 or newer</td>
</tr>
<tr>
<td>CVE-2011-2730: Allows remote attackers to obtain sensitive information</td>
<td>spring-web-2.5.5.jar</td>
<td>Upgrade Jar file to 3.2.9 or newer</td>
</tr>
<tr>
<td>CVE-2014-0107: Allows remote attackers to bypass expected restrictions and load arbitrary classes or access external resources via a crafted messages</td>
<td>xslt.jar; xalan.jar</td>
<td>Upgrade Jar file to 2.7.2 or newer</td>
</tr>
<tr>
<td>CVE-2013-4002: Allows remote attackers to affect availability via unknown vectors.</td>
<td>Xerces2.6.2_xercesimpl.jar; xercesimpl.jar</td>
<td>N/A (new versions exist but also contain vulnerabilities). Implement host based restrictions (i.e., IP tables, file integrity detection, Host based IDS)</td>
</tr>
<tr>
<td>CVE-2010-1244: Allows remote attackers to hijack the authentication of unspecified victims</td>
<td>activemq-web-5.2.0.2-fuse.jar</td>
<td>Upgrade Jar file to 5.9.0 or newer</td>
</tr>
</tbody>
</table>
Demo

• Demonstrate Sonatype
OpenSSL is an open-source implementation of the SSL and TLS protocols. The core library, written in the C programming language, implements the basic cryptographic functions and provides various utility functions. Wrappers allowing the use of the OpenSSL library in a variety of computer languages are available.

- Its free!
- As of 2014 two thirds of all webservers use it.
- From Heartbeat to Heartbleed
  - Defect could be used to reveal up to 64 kilobytes of the application's memory with every heartbeat
  - The affected versions of OpenSSL allocate a memory buffer for the message to be returned based on the length field in the requesting message, without regard to the size of actual payload in that message.

Reference: (http://en.wikipedia.org/wiki/Heartbleed)
Example: Heartbleed
How does it work?

Reference: (http://en.wikipedia.org/wiki/Heartbleed)
Example: Heartbleed
How do I find it?

- One way to find it would be to execute Origin Analyzer across source tree that includes your open source code as well.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Published</th>
<th>Base Score</th>
<th>Exploitability</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>VulnDB 107729</td>
<td>Jun 5, 2014</td>
<td>6.8</td>
<td>8.6</td>
<td>6.4</td>
</tr>
<tr>
<td>VulnDB 113251</td>
<td>Oct 14, 2014</td>
<td>2.6</td>
<td>4.9</td>
<td>2.9</td>
</tr>
<tr>
<td>NVD CVE-2014-0160</td>
<td>Oct 21, 2015</td>
<td>5</td>
<td>10</td>
<td>2.9</td>
</tr>
<tr>
<td>NVD CVE-2015-1788</td>
<td>Jun 15, 2015</td>
<td>4.3</td>
<td>8.6</td>
<td>2.9</td>
</tr>
<tr>
<td>NVD CVE-2015-1789</td>
<td>Jun 15, 2015</td>
<td>4.3</td>
<td>8.6</td>
<td>2.9</td>
</tr>
<tr>
<td>NVD CVE-2015-1790</td>
<td>Jun 15, 2015</td>
<td>5</td>
<td>10</td>
<td>2.9</td>
</tr>
<tr>
<td>NVD CVE-2015-1791</td>
<td>Jun 15, 2015</td>
<td>6.8</td>
<td>8.6</td>
<td>6.4</td>
</tr>
<tr>
<td>NVD CVE-2015-1792</td>
<td>Jun 15, 2015</td>
<td>5</td>
<td>10</td>
<td>2.9</td>
</tr>
<tr>
<td>NVD CVE-2015-4000</td>
<td>Jul 22, 2015</td>
<td>4.3</td>
<td>8.6</td>
<td>2.9</td>
</tr>
</tbody>
</table>
• Demonstrate BlackDuck Hub
Finding Vulnerabilities in SW

- It is difficult to determine what types of tools and techniques exist for analyzing software, and where their use is appropriate.
  - Institute for Defense Analyses (IDA) created the SOAR report and matrix to assist
    - NASA has slight modified version to include tool names <-> contact brandon.t.bailey@nasa.gov
  - Ideally developers will institute static source code and binary analysis to assist in identifying weaknesses
    - Development activities should include analyzing source code before it is compiled to detect coding errors, non-secure coding constructs, and other indicators of security vulnerabilities or weaknesses that are detectable at the source code level
  - Developers should perform software evaluations throughout the software development lifecycle to address potential security vulnerabilities early in the process
  - Use research from NSA’s CAS and Institute for Defense Analyses to establish a blend of tools that will provide the most value
**Demo**

- Demo SOAR Spreadsheet

![SOAR Spreadsheet](attachment:SOAR_Matrix.xlsx)
• Tutorial E Part 1:
  – Section 1: Cyber Threat – Who, What and Why
  – Section 2: Defense-In-Depth
  – Section 3: Secure Software Engineering Steps
  – Section 4: Errors, Weaknesses and Exploits
  – Section 5: Threat Modeling
  – Section 6: Testing
  – Section 7: Resources

• Tutorial E Part 2:
  – Section 1: Ground Systems Overview
  – Section 2: Secure Software Development
  – Section 3: Defense in Depth for Ground Systems
  – Section 4: What Now?
Defense in Depth (DiD)

• Secure software development is extremely important but DiD is key to protecting mission assets
• In Mission environments, DiD can be difficult
  – Older architectures/technology
    • Unsupported operating systems, older hardware, etc.
  – Shared architectures/technology
    • Mission X doesn’t own all layers of the defense
• Sometimes vulnerable software depends on something that is out of their control
  – Do you trust the Network Engineers? Should you?
  – Do you control the host level configuration?
Compounding Problem

- Work with Network Engineers to implement enclaves/network zoning and/or encryption
  - Build a “zero trust” architecture
    - Vulnerabilities injected by Mission X may affect Mission Y
    - Network layer encryption
- Understand and eliminate pivot points
  - From networking perspective, software security perspective, host level security
- Increase attack depth or eliminate all together

Utilize tools like RedSeal Networks, Skybox, etc. to understand network topology and threat exposures
Example SW Impacting Mission

Developers can’t assume protection from Firewall. Need “Defense in Depth”. Can’t assume if knocking on door, that they are supposed to be there.

Signs onto Rogue Wifi, Click Phishing Link, Etc. **Then Signs onto VPN**

Exploits Custom S/W Establishes persistent foothold on Mission Asset

Mission Asset

Launch Attacks (DoS, Brute Force, Extract Data, etc.)

Mission Control

This example will depict how vulnerable software within a network can potentially impact critical mission assets
Demonstrates that a pathway exists from the VPN Landing Zone, Internet, Or "Untrusted" to a vulnerable piece of software.
Sample Exposure

Demonstrates all outbound access paths (Pivoting) from the vulnerable asset.
Demonstrates potential vulnerabilities that could be exploited from this server
• Example depicted how vulnerable software in a network that doesn’t employ zero-trust architecture can expose mission assets

• Network encryption is another layer of defense that provides protection
  – Protocol machinery below the Network Layer including the Network Layer protocol (e.g., IP, Data Link, Physical) is exposed

• Data Link Layer services can provide additional protection
  – Upper-layer protocols become opaque
  – Data Link Layer security may be useful when a threat assessment indicates a heightened risk of exposure of the underlying protocols across an RF link or when traffic analysis is a concern.
  – Data link layer is usually under complete control of the Mission and vulnerabilities within the shared architecture can be mitigated by added this layer of defense
Data Link Layer

- DiD: Network layer unsecure/non-encrypted or frame data is sensitive post network layer (i.e. RF)
- Secure at data link layer
  - Data Link Layer security services may be able to provide all of the mission’s security needs, which could include authentication, integrity, and confidentiality, but only on the specific link over which the security services are provided.
• Without protection:
  ▪ Spacecraft are vulnerable to spoofing attacks from rogue ground stations
  ▪ Telemetry could be received and processed by rogue ground stations
  ▪ …
CCSDS realized that a standardized protocol to integrate security into space missions with a simple network topology could be proposed at the data-link layer

- Space Data-Link Layer Security Protocol (SDLS)

Space Data Link Security (SDLS) [Blue Book](#) Published September 2015

- *Protections implemented via software!*
SDLS Capabilities

Network Layer

- Space Packet Protocol: SCPS-NP

Data Link Layer

- TC Space Data Link protocol
- TM Space Data Link protocol
- AOS Space Data Link protocol
- Secured TM/TC Protocol
- TC Synch and channel coding
- TM Synch and channel coding

Physical Layer

- RF and Modulations Systems
• SDLS supports two main security services
  1. Authentication only – providing authentication and integrity
  2. Authenticated encryption – Adding confidentiality

Security trailer (MAC)
(computed over full transfer frame minus OCF, ECF and masked subfields of TF headers)
• To promote multi-mission implementations and interoperability:
  - 3 recommended profiles have been defined covering security requirements of most missions w.r.t. TC, TM and AOS links

• Baseline mode for TC
  - **Authentication only**, using AES/CMAC, 128-bit key, 32-bit ARC, 128-bit MAC (22-octet overhead (8%))

• Baseline mode for TM and AOS
  - **Authenticated encryption**, using AES/GCM, 128-bit key, 96-bit initialization vector, no seq. # needed, 128-bit MAC (30-octet overhead (2.5%))
What To Do Now?

• In mission environments (esp. mission with extended ops) you may not be able to patch code; therefore for vulnerable code that can’t be fixed the “host” owner can
  – Harden the servers and hosts by disabling all ports, protocols and services that are not explicitly required for operations
  – Install file integrity software (i.e., TripWire, Aide) to alert to changes made to the file system
  – Install and finely tune a host-based IDS that will alert to any anomalous traffic
  – Utilize IP tables/IPFilters to limit data flow to specific IP addresses, ports, protocols and services
What To Do Now?

• To prevent future deployments of vulnerable code
  – Participate in secure code training
    • Educate developers, PMs, Authorizing Officials, Security Personnel (ISSO, ISO, etc.) on the importance of eliminating vulnerable code from architecture
  – Pick the low hanging fruit (see slides 20/21)
  – Utilize Best Practices and Secure Coding Standards
    • Ex: Best Practices from NASA’s Secure Coding Portal
    • Ex: Coding Standards (Ex. CERT C, C++ or JAVA Stds)
    • Apply the tools within the development activity (i.e., as an add-on to the developer's Integrated Development Environment (IDE)) as well as in the Independent Test and Evaluation (IT&E) activities
    • Top 25 CWEs for Ground Systems
      – Use NASA’s or create you own based on your mission and threats
Help Assure
Ground System Security

• Expand independent assessments of ground systems for vulnerabilities using latest technologies
  – Perform “Red” and “Blue” Teams across the entire ground system – End to End

• Promote integration of security early in acquisition and development life-cycles

• Integrate cyber security activities to dependably *Know, Prevent, Detect, Respond, and Recover*
Backup Slides
Links

Slide 5/6:

- major space agencies of the world - http://public.ccsds.org/participation/member_agencies.aspx
- multi-national forum - http://cwe.ccsds.org/

Slide 11:

- 7120.5E - https://foiaelibrary.gsfc.nasa.gov/Assets/doclibBidder/tech_docs/1. N_PR_7120_005E_.pdf
- Security Quality Requirements Engineering (SQUARE) - http://www.cert.org/cybersecurity-engineering/products-services/square.cfm?

Slide 14:

- C - https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard
- C++ - https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637
- JAVA - https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java
- Sonatype - http://www.sonatype.com/
- Spreadsheet - http://www.acq.osd.mil/se/docs/P-5061-AppendixE-soar-sw-matrix-v9-mobility.xlsx

Slide 15:

- Common Weakness Enumeration (CWE) - https://cwe.mitre.org/
- Common Vulnerabilities and Exposures (CVE) - https://cve.mitre.org/
- Common Attack Pattern Enumeration and Classification (CAPEC) - https://capec.mitre.org/
- SAFECode - https://training.safecode.org/
- Cigital - https://www.cigital.com/services/training/elearning/
- Pluralsight - https://www.pluralsight.com/search?q=security&categories=course
Links (cont.)

Slide 16:
• http://franklinta.com/2014/08/31/predicting-the-next-math-random-in-java

Slide 19:
• Stack Overflow Post - http://stackoverflow.com/questions/6747995/a-complete-list-of-unsafe-string-handling-functions-and-their-safer-replacements
• Flawfinder - http://www.dwheeler.com/flawfinder/

Slide 22:
• Cppcheck - http://cppcheck.sourceforge.net/
• Rosecheckers - http://sourceforge.net/projects/rosecheckers/
• Splint - http://www.splint.org
• RATS - https://code.google.com/p/rough-auditing-tool-for-security
• Flawfinder - http://www.dwheeler.com/flawfinder
• SWAMP - https://continuousassurance.org
• Find Bugs - http://findbugs.sourceforge.net/

Slide 23:
• CWE - https://cwe.mitre.org/
• CVE - https://cve.mitre.org/
• CAPEC - https://capec.mitre.org/

Slide 27:
• Sonatype - http://www.sonatype.com/
• Black Duck HUB - https://www.blackducksoftware.com/products/black-duck-hub
• OWASP Dependency Check - https://www.owasp.org/index.php/OWASP_Dependency_Check

Slide 30/31:
• http://en.wikipedia.org/wiki/Heartbleed
Slide 34:

Slide 46:

Slide 51:
- C - https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard
- C++ - https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637
- JAVA - https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java
Best Practices from NASA’s Secure Coding Portal

1. **Validate input.** Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files.

2. **Heed compiler warnings.** Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code.

3. **Use Code Analysis Tools.** Use static and dynamic analysis tools to detect and eliminate additional security flaws. Dynamic analysis is the testing and evaluation of an application during runtime. Static analysis is the testing and evaluation of an application by examining the code without executing the application. Many software defects that cause memory and threading errors can be detected both dynamically and statically. The two approaches are complementary because no single approach can find every error. The primary advantage of dynamic analysis: It reveals subtle defects or vulnerabilities whose cause is too complex to be discovered by static analysis. Dynamic analysis can play a role in security assurance, but its primary goal is finding and debugging errors. The primary advantage of static analysis: It examines all possible execution paths and variable values, not just those invoked during execution. Thus static analysis can reveal errors that may not manifest themselves until weeks, months or years after release. This aspect of static analysis is especially valuable in security assurance, because security attacks often exercise an application in unforeseen and untested ways.

4. **Use Binary Analysis Tools.** Binary analysis creates a behavioral model by analyzing an application's control and data flow through executable machine code – the way an attacker sees it. Unlike source code tools, this approach accurately detects issues in the core application and extends coverage to vulnerabilities found in 3rd party libraries, pre-packaged components, and code introduced by compiler or platform specific interpretations.
5. **Architect and design for security policies.** Create software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.

6. **Keep it simple.** Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.

7. **Default deny.** Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted.

8. **Adhere to the principle of least privilege.** Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should be held for a minimum time. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges.

9. **Sanitize data sent to other systems.** Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.

10. **Practice defense in depth.** Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment.
11. **Use effective quality assurance techniques.** Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions.

12. **Adopt a secure coding standard.** Develop and/or apply a secure coding standard for your target development language and platform.

13. **Define security requirements.** Identify and document security requirements early in the development life cycle and make sure that subsequent development artifacts are evaluated for compliance with those requirements. When security requirements are not defined, the security of the resulting system cannot be effectively evaluated.

14. **Model threats.** Use threat modeling to anticipate the threats to which the software will be subjected. Threat modeling involves identifying key assets, decomposing the application, identifying and categorizing the threats to each asset or component, rating the threats based on a risk ranking, and then developing threat mitigation strategies that are implemented in designs, code, and test cases.

15. **Don't trust services.** Many organizations utilize the processing capabilities of third party partners, who more than likely have differing security policies and posture than you. It is unlikely that you can influence or control any external third party, whether they are home users or major suppliers or partners. Therefore, implicit trust of externally run systems is not warranted. All external systems should be treated in a similar fashion.
16. **Separation of duties.** A key fraud control is separation of duties. For example, someone who requests a computer cannot also sign for it, nor should they directly receive the computer. This prevents the user from requesting many computers, and claiming they never arrived. Certain roles have different levels of trust than normal users. In particular, administrators are different to normal users. In general, administrators should not be users of the application.

17. **Software Supply Chain.** IT managers should create and preserve a bill of materials, or a list of ingredients, for the components used in a given piece of software. The complexities and interdependencies of the IT ecosystem require software suppliers to not only be able to demonstrate the security of products they produce, but also evaluate the integrity of products they acquire and use. Ultimately this should lead to greater confidence through integrity checks incorporated in a defined secure development lifecycle.

18. **Avoid security by obscurity.** Security through obscurity is a weak security control, and nearly always fails when it is the only control. This is not to say that keeping secrets is a bad idea, it simply means that the security of key systems should not be reliant upon keeping details hidden. For example, the security of an application should not rely upon knowledge of the source code being kept secret. The security should rely upon many other factors, including reasonable password policies, defense in depth, business transaction limits, solid network architecture, and fraud and audit controls. A practical example is Linux. Linux's source code is widely available, and yet when properly secured, Linux is a hardy, secure and robust operating system.

19. **Fix security issues correctly.** Once a security issue has been identified, it is important to develop a test for it, and to understand the root cause of the issue. When design patterns are used, it is likely that the security issue is widespread amongst all code bases, so developing the right fix without introducing regressions is essential.