Common Aerospace Threats from Coding Practice to Full Exploit: Tracing Threat Angles.

Bryce L. Meyer[1]
AIAA Associate Fellow, O'Fallon, MO, 63366, USA

Aerospace decision makers are often aware of the need to devote effort in software testing and quality assurance at various stages in an Agile/Development Security Operations (DevSecOps) process. A risk of Agile however is the rush to production, which raises the specter of pernicious attacks, an example is lapses in coding practice onto real-time Linux based systems. Many of these exploits are no longer easy on modern desktop or server-based systems but are still valid on limited processing environments within aerospace systems deployed in orbit or on uncrewed aerial systems. These small processors and byte-limited systems are often needed due to radiation hardening or limited electricity. Every system deployed to sky or orbit is now a target by both amateurs and professionals, who are always looking to get control. This paper examines this risk within the scenario of memory overflow exploits. This use case is not difficult to plan as tools that enable this exploit are available to beginners (e.g., script-kiddie from GitHub). To prevent these issues, Aerospace managers should monitor three areas of risk : the rush to include code, the rush to compile, and ignoring warnings from tools and compilers to hit completion dates. This paper digests the cybersecurity language and methodology into a diagram format to enable decision makers readily understand the contributors of overflow attacks and shows simplified processes in a way understandable to non-cybersecurity professionals. The intent of this paper is to serve as initial training for additional in-depth instruction.

I. Introduction

Many exploits originate from three avenues, but all start with a similar flow of techniques. Before each is explored herein, we should understand the overall chain of danger, see Figure 1.

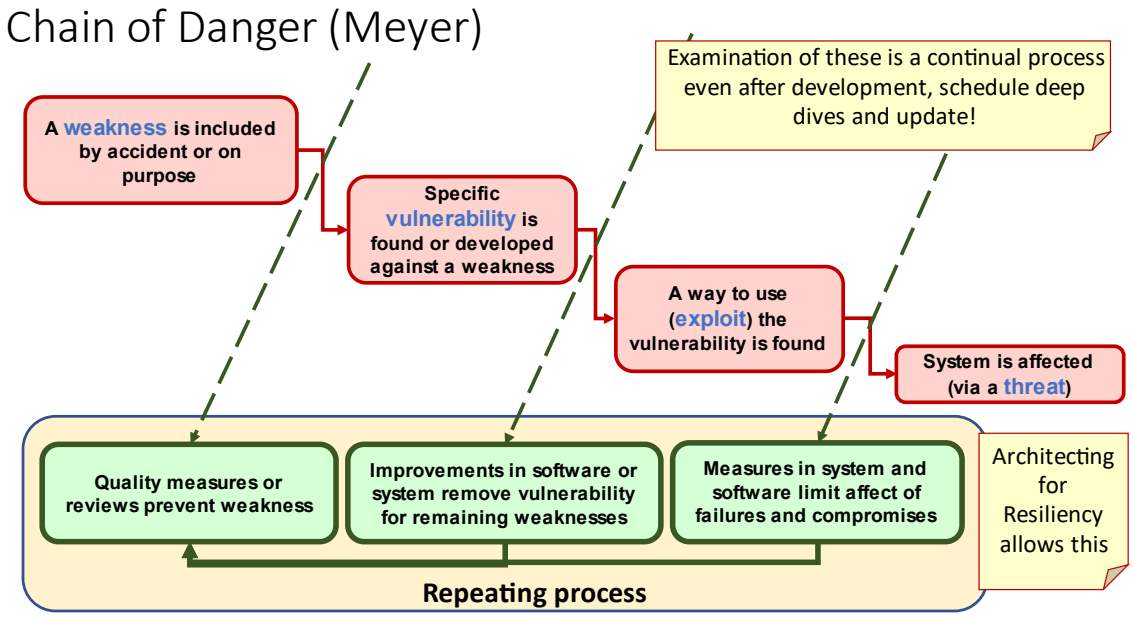

Figure 1. The Chain of Danger that allows (or prevents) vulnerabilities.

[1] AIAA Associate Fellow and member of the AIAA Aerospace Cybersecurity Working Group.

The chain starts when a threat is included in the developed code and then deployed. The threat becomes a weakness that exposes a specific vulnerability which is a general avenue to attack or manipulate the code. The vulnerability is found, and matched to a tactic that results in a specific method to exploit the code, control it, or stop it. As a result, the deployed system is compromised.

In the first section of the paper, I will describe in simple terms and at a high level, the example threat's flow from attack to system alteration (e.g., executing shellcode using return-oriented programming), resulting in system compromise (i.e., use by an adversary in a command-and-control network).

The most common overflow threat pattern uses a classic method defined by Aleph One in the last century [1], which has since been expanded, become tooled [?, you mean automated], for diversified use. At a high level, all the memory for programs running on a system like Linux or Windows (and their derivatives like RedHat, VxWorks, and others), specifically the variables in program functions, operate in memory. One type of runtime memory structure, usually for string or array variables, is called a stack. A stack is a first in, last out structure, the last item loaded is the first item pulled out. This stack includes memory for the values of variables in the function, but also the spot in memory that points to the next available instruction line. This next spot is called the stored instruction pointer, or stored return. So, if a program's main function called a another function in line 50 (see Figure 2), then the function runs in memory lines 5000-5010, then the stored instruction pointer goes to 51 so main function can continue running the program. These 'line numbers' are virtual hexadecimal/binary memory addresses like 0xffff ffacf, but the analogy holds.

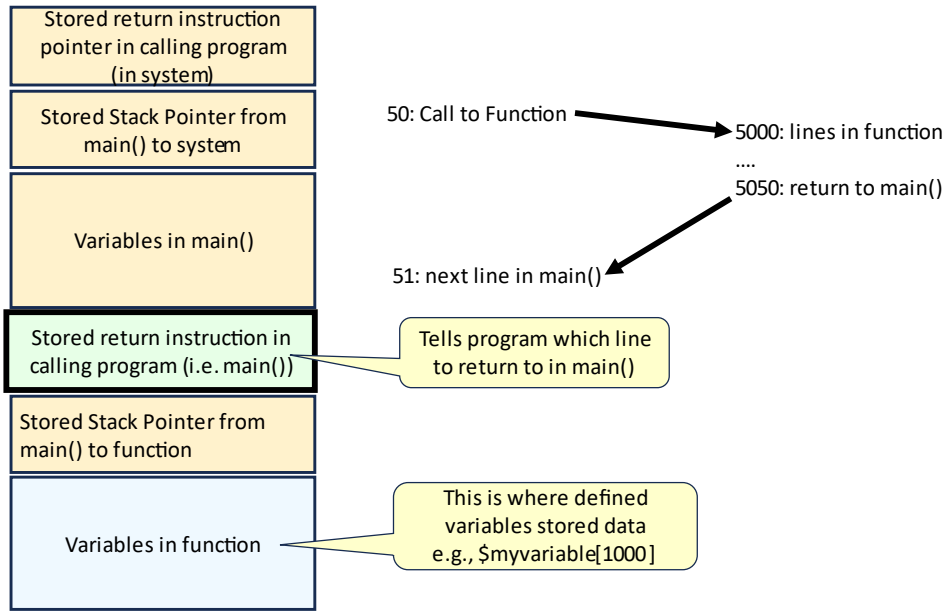

Figure 2. Simplified memory layout for a running function of in a program.

If the function has a place where one of the variables specifies a smaller bit of memory, say 1000 letters, but allows input to the function of 1100 letters, the stack memory can be overflowed as shown in Figure 3. One of the consequences of the overflow is that the extra letters can be used to overwrite that stored instruction pointer so that instead of returning to line 51 in main as the function finishes, the program 'jumps' to a wrong memory position (whatever was put into the stored instruction pointer).

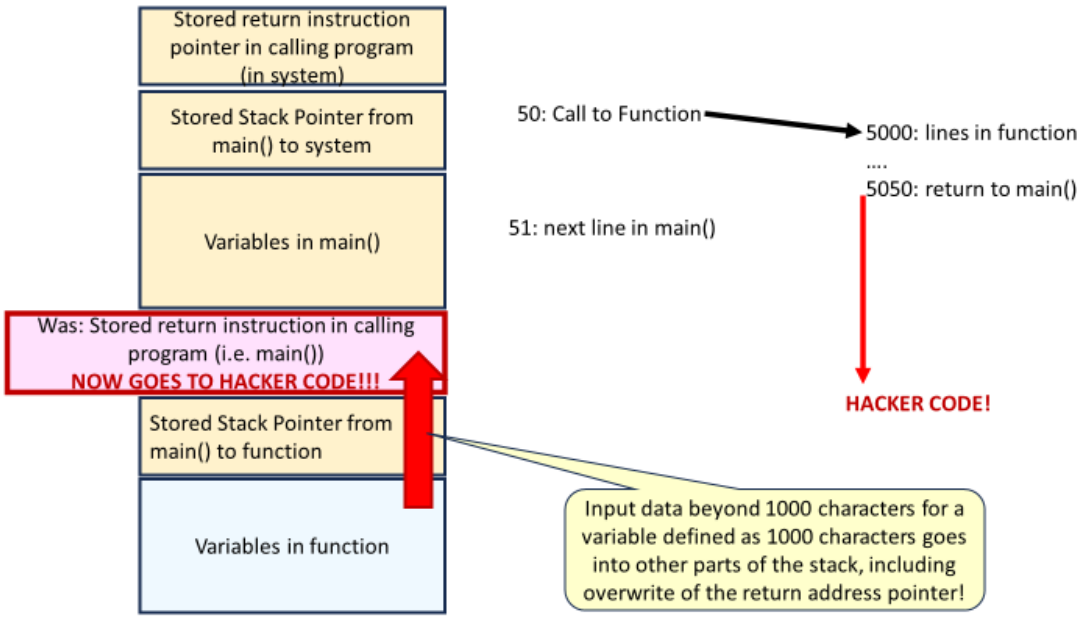

Figure 3. Redirection in a memory overwrite. As a stack or other memory location is overwritten, a hacker can insert a pointer to another memory location or shellcode.

Exploiting the overflow, a crafty hacker can then load a location in the stored instruction (return) pointer that leads to their pernicious code. Their code, often called 'shellcode', can do a variety of things, such as taking control of a system, 'twisting' databases, hiding code in the background until triggered, capturing data, etc. A hacker trick is to use running system code to assemble the hacker code. The hacker selected segments of existing system code, called gadgets, are then assembled at run time into the desired shell code. The hacker just hides the pointer to the gadgets in the stack, and a pointer back into the stack in the stored instruction pointer. Once the pointer hits the hacker inserted pointers, the hacker can then run whatever code they want, as an administrator, in a command shell. This is known as the ability to run arbitrary code.

This chain of events sounds difficult to implement, but even high schoolers can use GitHub tools to run this hack. Low level hackers like these are called 'script kiddies', and most of the tools they need can be found in packages used by legitimate red teams. Red teams are professionals hired or employed by the system owner to test a system or network as if they were hackers. These red teams use many packages to perform their work. These packages can be whole suites like Kali Linux which includes a host of tools or the packages could be single tools or toolsets found on GitHub. These tools are publicly posted to enable as many friendly testers as possible, but there is a side effect. Low level hackers can use the same tools to find example exploit scripts, and simply modify them without needing to know how the script works. A few example tools are in Table 1.

Table 1. A few example tools that can be used to research attacks.

Tool	purpose
Kali Linux	Red Team/Penetration Testing aggregation of tools to run exercises and make code more secure

PwnTools	Capture The Flag framework of tools, Linux.
Ropper	Tool Implement Return Oriented Programming (ROP) tests, by finding gadgets, largely Windows.
Metasploit	Penetration Testing Framework, includes methods to create a command-and-control channel into an exploited system, typically for ground systems
HackRF/Universal Radio Hacker (URH)	Toolset for Software Defined Radio (SDRs) to enable wireless exploitation

Nation-state or well-funded expert hackers have better tools and methods. In response, operating system and chip manufacturers have improved the security by adding methods to stop the memory overflow/redirect pattern, discussed later in this paper. This 'cyber arms race' has not ceased, however, both penetration testers and hackers have found ways to circumvent these measures. This is illustrated in Figure 4 below. The initial flood of overflow vulnerabilities from the 1990s and 2000s was slowed by countermeasures, but these countermeasures were not fully effective as to prevent discovery of new vulnerabilities. For the past 5 years, roughly, 3-5% of vulnerabilities per year are in the category of overflows as per CVE data [2]. These are but one attack type, and an older attack at that, but its viability hints at the dangers to aerospace systems. Many other attacks exist as well. In a system that might be difficult to update to prevent attacks, including ground based support systems, this attack may still be a threat. Many of these types attacks come in by ignoring the Software Engineering Institute (SEI) Top 10 Secure Coding Practices [3].

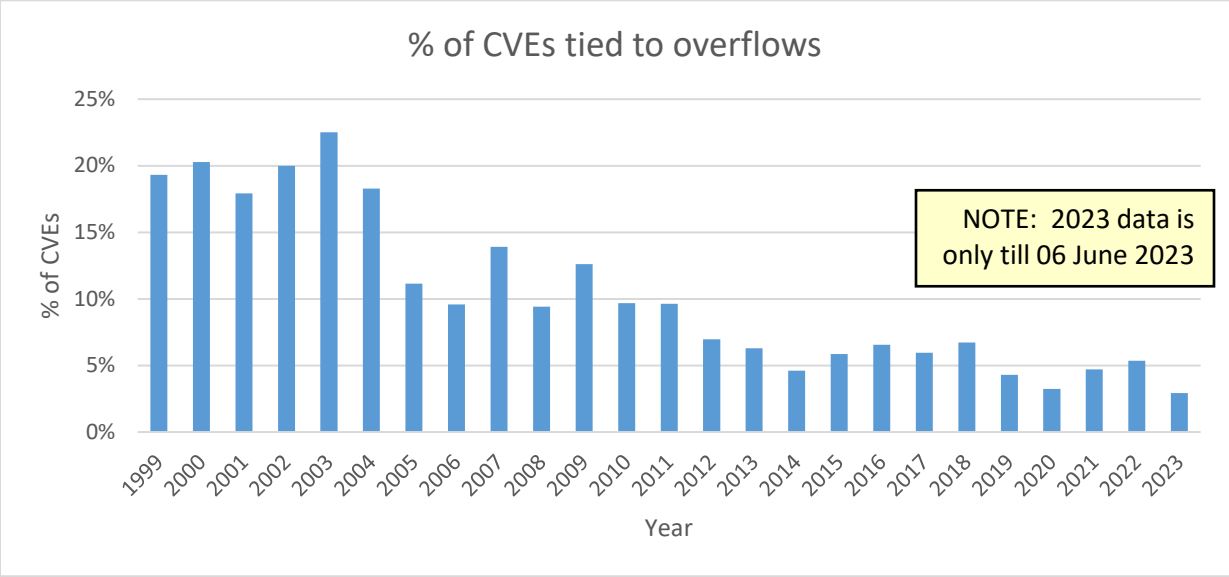

Figure 4. As per Common Vulnerabilities and Exposures (CVEs) database maintained by MITRE [2], even decades after overflows were first documented, the threat remains.

Having established a basic vocabulary, we can explore common risks that result in the inclusion of memory overflow vulnerabilities.

The first risk is the rush to include code to finish development. Drivers or libraries are included that may be innocuous per-se but were compiled or built with a sub-optimal software quality process. These drivers and third-party libraries are now the attack vector that enables access to the point of shell access. Assuming a vulnerable driver in 32-bit format example, many stack-overflow attacks become possible, using code already running on a system.

The second risk, the rush to compile, is where code is constructed, but choices that make the compiled binaries more secure are omitted, leaving exploit opportunities that accrue as technical debt. The skipped secure choices would cause more work for the developer team and add processing overhead at run-time but prevent attacks that use stack and other memory exploits.

A third risk, ignorance is bliss, is one where the whole toolset from inception to deployment is available with diagnostics, but warnings and errors are ignored. Compilers are the first line of defense, followed by static tests, yet in the rush to meet development schedule, memory warnings are ignored (again technical/cyber debt), or code is written only to pass initial tests, omitting code that would have added resiliency in operation.

Understanding these risks at a simplified level should help managers justify investment in Agile DevSecOps best practice, and reduce risks before a red team, or adversary, finds them.

II. Rush to Include Code Angle

For many aerospace systems, code is assembled from combinations of automatically generated code, code adapted from prior similar systems, code elements for drivers and interfaces, and newly constructed code. There is always a pinch in resources and time to meet project deadlines. Even in an environment where Agile/DevSecOps is spoken, steps can be skipped or rushed. The DevSecOps process in a software factory has many opportunities, by design, to stop exposures from entering a deployed system, see Figure 5 and Figure 6 below. Figure 6 is not all inclusive, but illustrates that testing is in many parts of the process, and if the tests are used well, can limit exposures in the code.

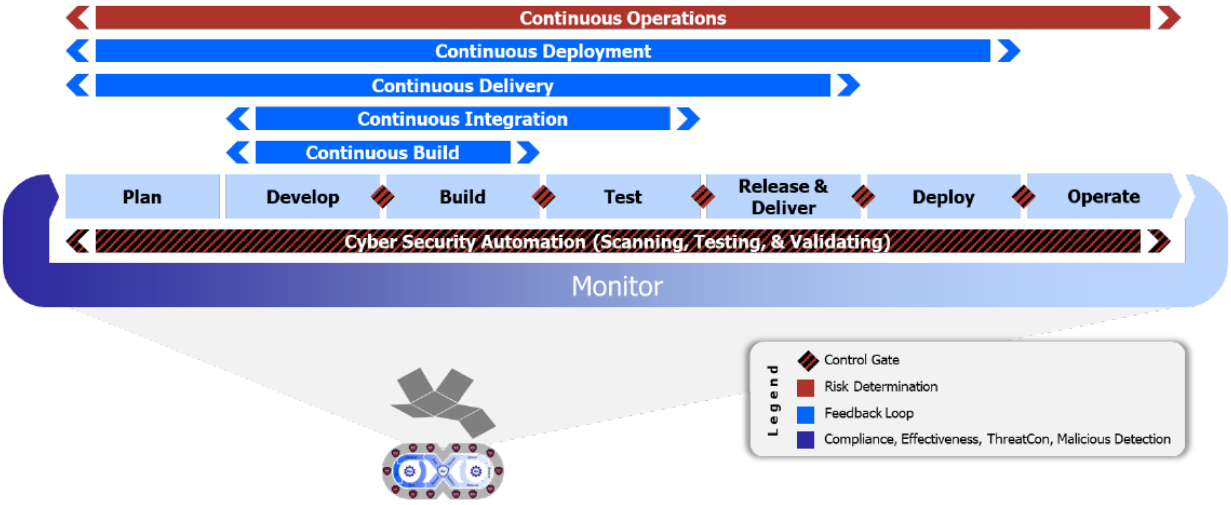

Figure 5. DevSecOps Phases and Continuous Feedback Loops (from [4]). This process shows the constant examination of code at multiple steps.

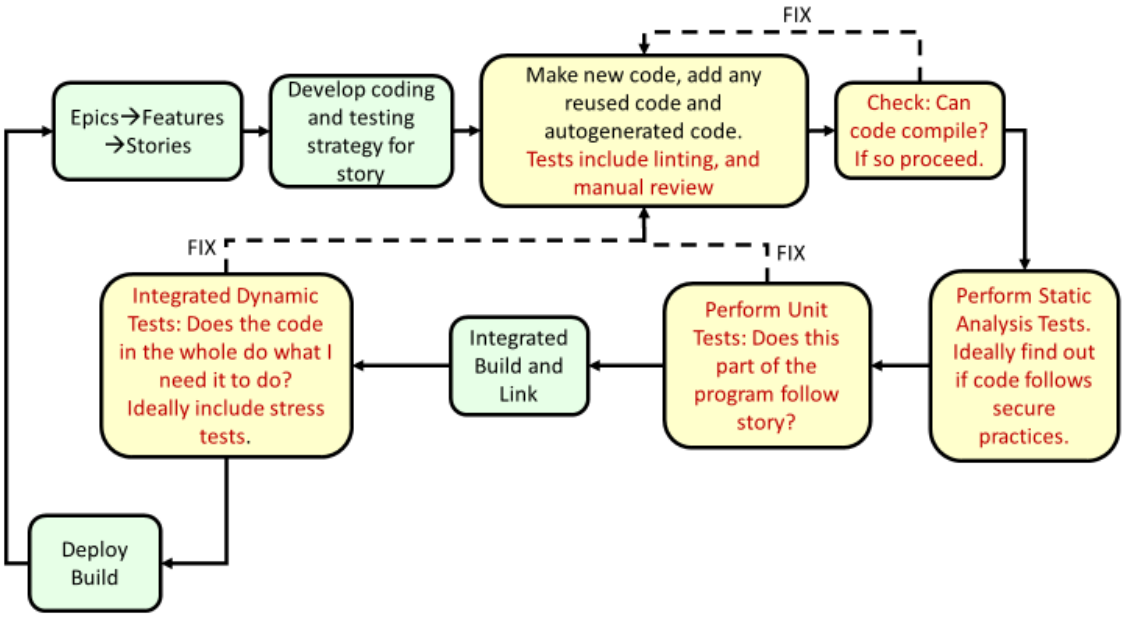

Figure 6. Selected parts of a DevSecOps process with Agile.

These steps, however, can be rushed or gamed. As a result, some shortcuts may be taken. Figure 7 shows the build-up of software on a system. Modern systems are a marriage of software from many sources, only one source is newly written software (from scratch), the rest of the code could be generated from algorithms (auto-generated code), code from prior systems, code brought in from libraries, code to use middleware, and code for interfaces to hardware. The resultant product can have many sources of risky and old code.

A code source is via auto generators e.g., Simulink®/MATLAB® to create both source code and hardware description languages, that may be complex to satisfy the algorithms for the design. These auto-generated segments, however, can be difficult to analyze. Other code sources can be reused code, off-the-shelf code, and newly developed code. Each can include instructions that have become dangerous.

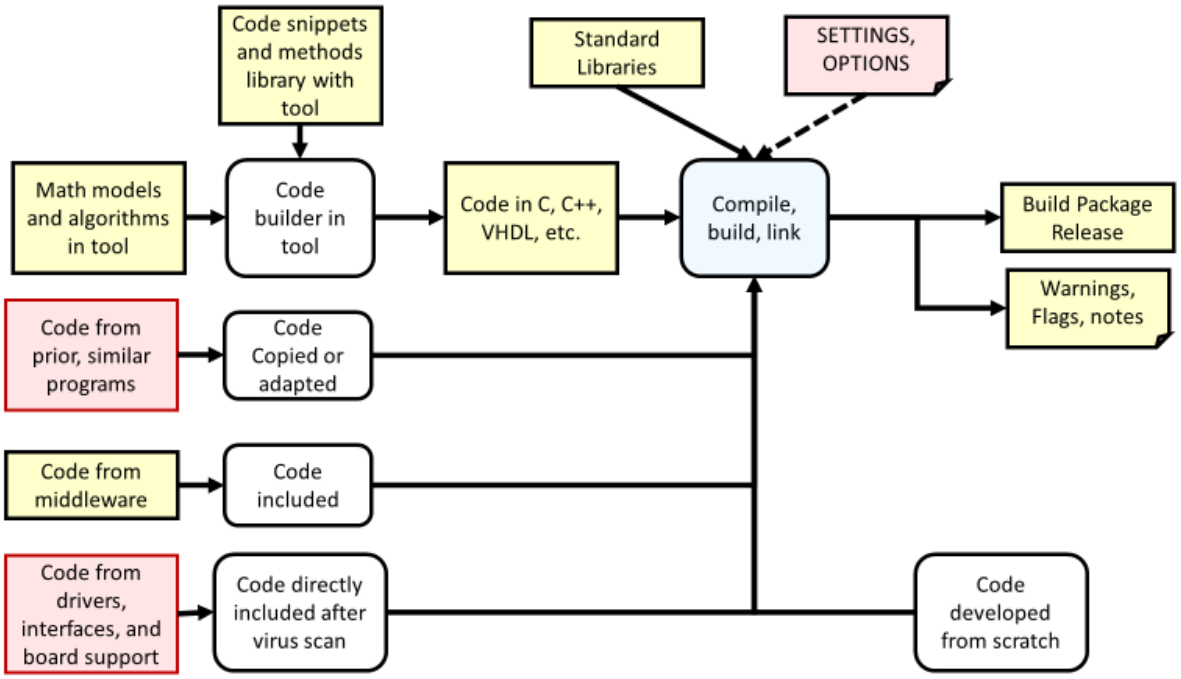

Figure 7. Code build-up from many sources for a typical aerospace system.

Many satellites and some aircraft fly with segments of older code, as do support systems on the ground. Older code is more likely to contain overflow opportunities (see Figure 4). Aside from the fact that some spacecraft are hard to patch, there are other reasons why older code exists. It can be the case that equipment makers have gone out of business, or that the elements are maintained only by a limited community. If the older code contains variable definitions smaller then as used in newer code, this older code now becomes an overflow opportunity. This opportunity exists especially if the code does not include error checks to prevent overflow (again technical debt gives rise to the exploit). As shown in Figure 8, some software in systems runs as planned under normal conditions but causes security issues when a low likelihood condition occurs.

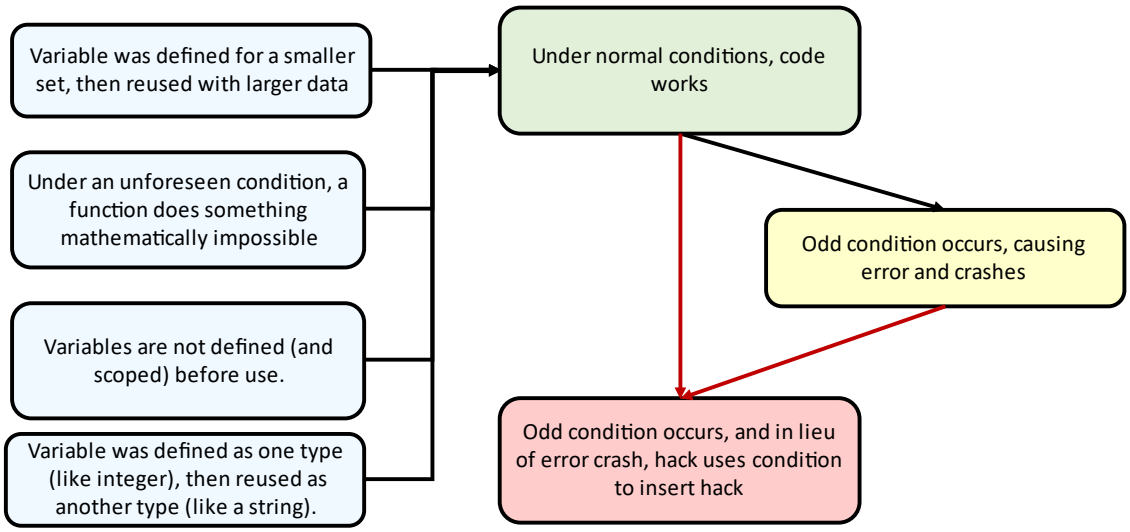

Figure 8. Included coding issues can run under normal conditions, but open a security problem in odd conditions.

Several of these practices are in the ISO/IEC 9899:2018 standard [5] and listed in the secure coding standards in OWASP [6] and SEI [7], [8], [9]. Below in Figure 9 is an example of how an older driver could lead to a hack.

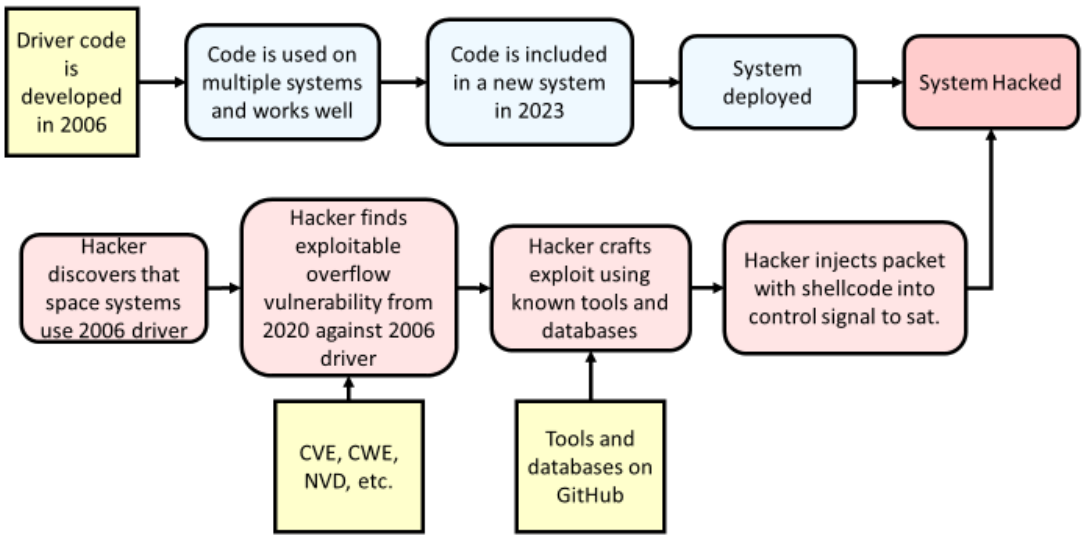

Figure 9. A notional example of how older code can result in an overflow hack. Assumes the 2006 driver could not be patched or modified, it becomes a threat opportunity by 2020, and the threat is included in 2023.

There is often an argument in aerospace systems that the systems will only be used on isolated networks, and will have limited interfaces, unlike a web-based application. From prior work [10] and by many others including Lukin

and Haselberger [11], it can be noted that most system are far more susceptible to attack then this assumption allows. This includes hacks employing control and communication links, using network protocols. These channels allow a way that a hacker can inject a hack.

It is also common in proposals for even complex systems to assume that previous project code will simply be copied in, with little modification, since the previous system was space- or air-worthy when fielded. Reuse is a task that requires effort, and since projects may ignore this effort, the pressure increases to just include the code to get the task accomplished. The code may compile and even run but may not include fixes for weaknesses, especially those found since the code was originally written. This same project schedule pressure also leads to the next angle.

III. Rush to Compile and Start Tests Angle.

Several decisions are made under schedule pressure from compile time through testing that leave vulnerabilities in code, that in turn opens opportunities for overflow to adversaries. Since make files and compile scripts are included with older code, choices during the build and compile process may require lower bit counts (e.g., 32-bit versus 64-bit), and memory protection measures may have to be switched off to make the older code work. So in lieu of going into the older code and updating it to handle a more secure set of compile options, the older compile choices are retained, or worse, compile switches to protect memory are turned off until the code runs. This pressure ripples down to coding teams who are then pushed to include the code, without a detailed examination, and if it still compiles and runs, they move to the next task in the project. The code could be just good enough to not trigger major errors and pass a limited set of unit tests. The coding team gets Agile credit for completing the story and moves to the next task. One magnifying problem in many real time systems is that some controllers and devices operate at 32-bit or lower levels, which eliminates some protections available in 64-bit and above architectures. There are often no work arounds for must-use chips, or critical components that must live in a 32-bit or less world. Even older ground equipment might be required, running a lower bit level. In 32-bit architectures defenses against memory overflows are limited simply by available memory space. Defenses like randomizing the addresses of core components only work to the extent of addresses allowed in 32-bit, meaning that adversaries can use tools to guess likely locations or indications of key code that can be reused in attacks. There are whole books on the detail of how memory protections work at 64-bit, but this paper will not go that deep. The biggest protections (as in Figure 10) are like so: non-executable bit marked memory, stack smashing/stack canaries, and address space layout randomization. Figure 10 depicts the ability to perform memory protection by most compilers, processors, and operating systems at compile, usually by default. However, developers may choose to turn-off these compile time memory protection options.

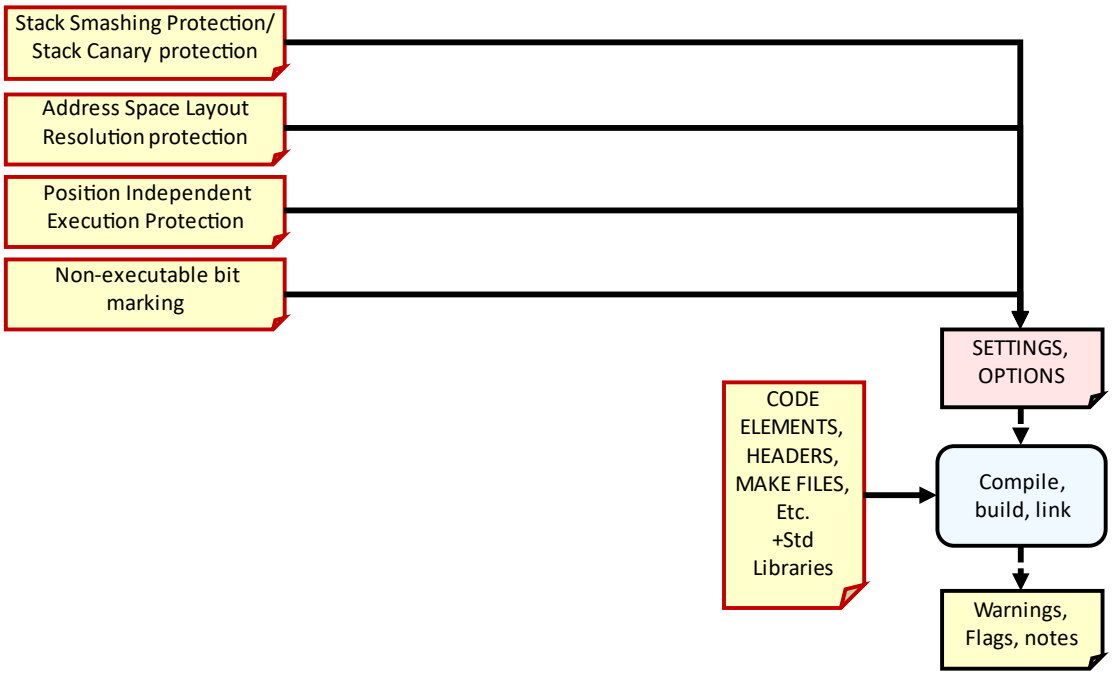

Figure 10. Memory protection compile time options as part of the release assembly process.

Non-executable marked memory attempts to prevent hackers from loading shellcode or other types of code in places it does not normally belong, like variable (stack) memory. An overflow that directs execution to these marked areas theoretically would just error out. Stack smashing protection, also known as stack canaries, uses bits of memory that if overwritten signal the operating system to error out. Like the canary in the coal mine, and memory overflow tries to write the canary address, which then call out to signal an error. Address space randomization relocates libraries and key elements of libraires that most programs use. As a result, a hacker that wants to use parts of these libraries to perform an exploit must find these libraries every time they run the attack. Hackers want to use libraries such as 'libc' because parts of the machine code can be used separately. Instead of loading a lot of hacker code, the hacker simply makes a chain of pointers to these parts to conduct mayhem. Address space randomization therefore makes this hack harder to achieve. Note that the hacks are still possible, just more difficult to perform. Knowing which compiler options to use, and when to use them requires experience and knowledge of both the code and the operating system, a level of expertise often lacking in coding teams composed of less experienced employees.

Given a 64-bit architecture and the full set of compiler, operating system, and processor options, those who build various components often ignore these memory protection options. One reason is that developers will have a build script for testing, and once the code works as designed, will reuse the same build script that turns off memory protection and security features. This chain of events is similar the process in Figure 11. These decisions result in deployed code that is exploitable. Some groups will also simply turn off options to silence the numerous warnings that ensue from compilers, sneaking their code through the process.

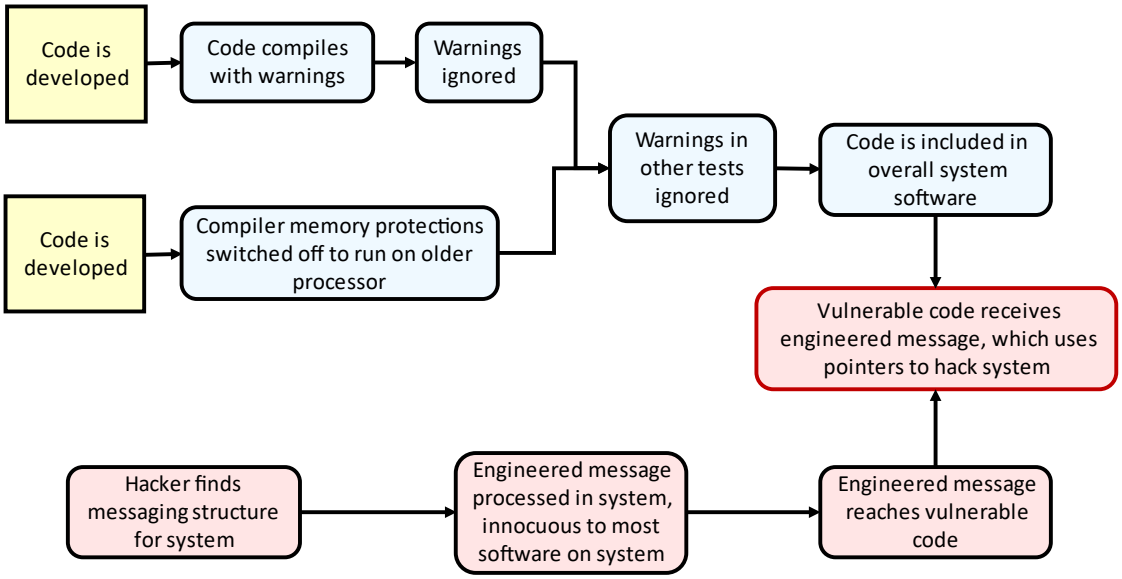

Figure 11. A notional example where vulnerable code is deployed ignoring compiler warnings, then a hack is introduced to system and passed around system messaging until it reaches vulnerable software.

Compiler warnings can be a first line of defense for code crafted outside of interactive development environments (IDEs). IDEs like Eclipse or Visual Studio often have features that work like the grammar checker in a word processor. These in line checks are part of Linting, a type of static analysis, and a hurried development group might ignore these checks if they don't prevent compile and run. The next line of defense however is the compiler itself, which checks for issues during the compile, and returns warnings even if the code successfully compiles. A group versed in secure coding will act on these warnings. The biggest warnings look like so: "warning: …writing xxx bytes into a region of size yyy overflows the destination…", but many warnings are far more subtle and must be examined in detail by those familiar with the code. Speeding past compile is often part of an 'if it runs, it is done' policy.

IV. Ignorance is Bliss Angle.

Once the build package runs, and meets a set of basic functionality tests, there is a strong push to get the software out to the system. Software can indeed be built to pass tests and analysis but lacks elements like robust error checking or extensive condition checking. The argument for such ignorance is that this system may not be critical, or like many CubeSats, disposable. Further, the intention may be to get past initial acquisition reviews, with the intension of adding robustness and security later. As a result, there is pressure to ignore the non-critical outputs from static analysis tools that are used past initial compile (see Figure 12).

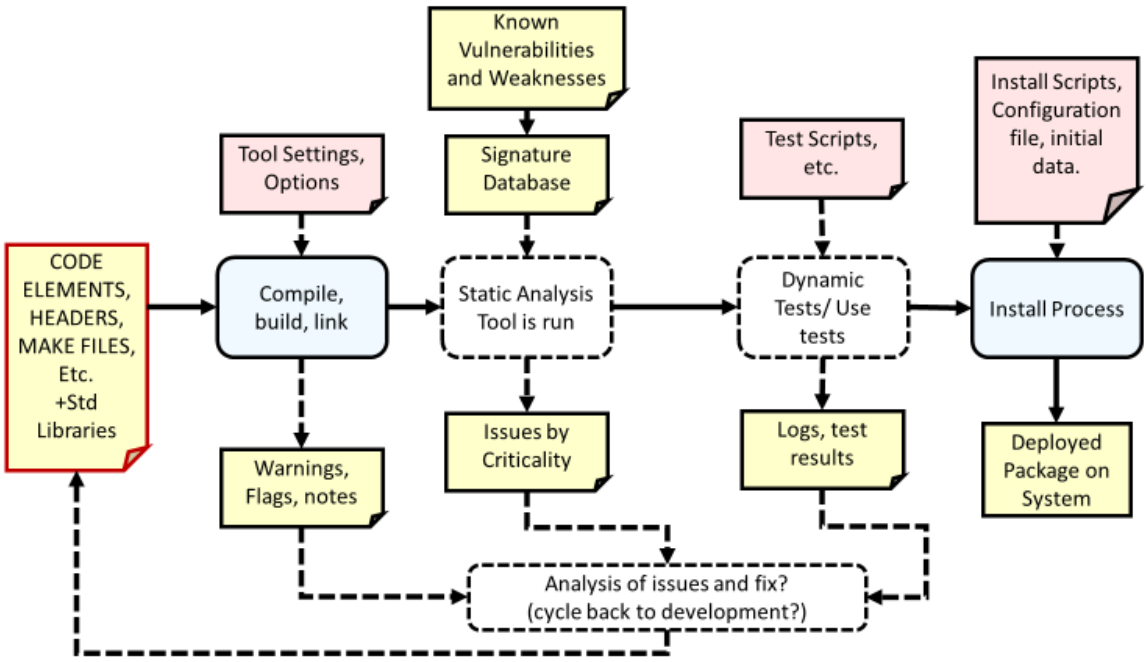

Figure 12. The process from testing to deployment is a series of choices to use test results or ignore them. If there is enough pressure, or simply a lack of expertise, warnings and cautions can simply be overlooked.

This drive to the finish line often results in a shortened testing cycle or overreliance on analysis tools. Many groups do run static analysis (i.e. pre-run time) tools though may ignore the plethora of warnings from the tools, many of which can be false alarms. The code could be written specifically to avoid major warnings from tools. But ignoring static analysis test warnings, even non-major warnings, are not the only risk. These tools are only as good as the data that feeds their scanners, and the scanner in these tools will miss some subtle items like undeclared or uninitialized variables. A manual code examination, that may take hours for every 100 to 1000 lines of code, is often considered a costly luxury. Once past the static analysis gauntlet, dynamic testing involves running code in part or whole and goes from unit testing to stimulate individual parts of the code, to full scenarios that test the integrated software in system as designed. Unit testing just tests the specific functions given to the developers, often nothing more. The code may run therefore just well enough to pass unit testing. Many enterprises ignore deeper stress testing that pushes in false data, data beyond intended parameters, and data beyond scale. It is an irony that the system's hardware is tested extensively for physical stresses due to temperature, vacuum, radiation, and other factors, but the software is not tested to see what happens if the sensors feed too much data to processors. In many cases if the code runs on the intended system and can pass some planned use cases, it is put into use.

Even with a full set of tests and examinations, time pressure might let configurations to define program behavior become a risk, see Figure 13. Installation scripts and files define the starting values and structure of the running software, and any faults or purposeful alterations affect installed the software on the target system. If variables are not correctly set, left uninitialized, or if added libraries or malicious code is included in the installation scripts, it becomes part of the running system.

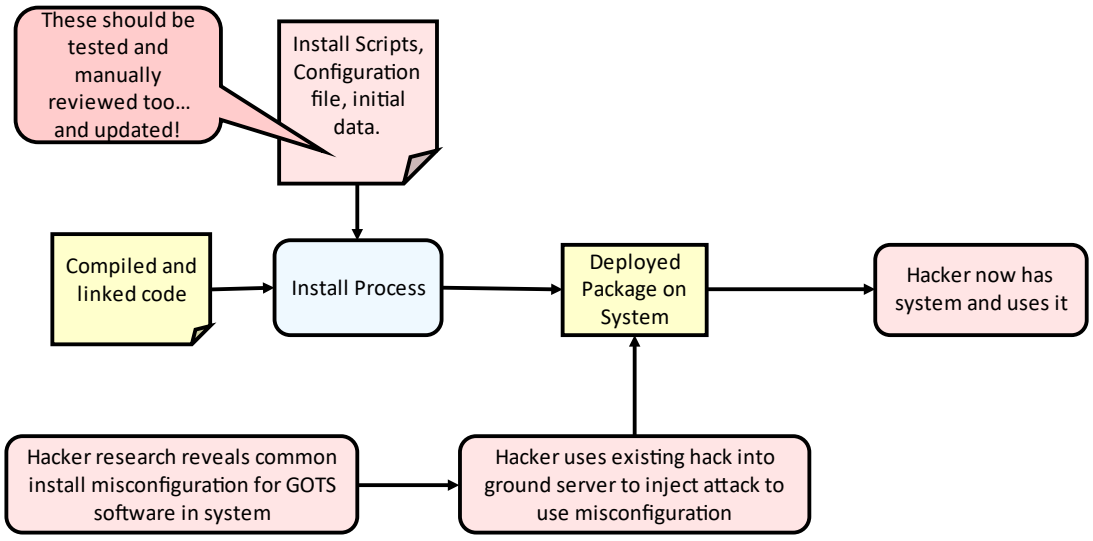

Figure 13. An installed misconfiguration can be capitalized by an adversary. In this case, the code may be well tested and securely developed, but a configuration choice allows an adversary an opportunity.

However, even tested and well configured software is not immune to overflows. The way adversaries perform overflows on even space and flight systems have evolved beyond simply filling in a form field or inserting text in a line of input or simple packet. For example, many vulnerabilities in 2023[2] (e.g., CVE-2023-1424, CVE-2023-22416, et al.) include network packets can be engineered to trigger overflows, then dropped into communication with t both the flying systems, and the on ground systems. This means that overflows, or other assumed older hacks, cannot be ignored even in the least interactive systems. Systems need to monitor internal activity and error handling, and control how errors can affect the software on the system. This magnifies the other angles above. Since wireless can be intercepted and sent, especially to spacecraft, making packets to insert into the communication stream to trigger an error that opens an opportunity to cause overflow are a likely attack method.

V. Best Practices to Prevent Overflows and Other Exploits

Best practices can lead to closing the three angles. The most difficult is realistic planning. Managers need to plan for enough resources to allow developers and testers time to examine the code design, and this may mean more hires and more realistic timeframes, thus more cost and schedule. The next best practice is knowing exactly what each code element needs to work, and then using means to limit the element to that memory and processing envelope, which requires analysis and testing for every included element. Autogenerated code may need to be isolated (in containers for virtual machines) or wrapped with developed code to prevent memory leak opportunities. If reused code must operate with older settings, it must also be limited and controlled to keep is operation limited enough to prevent overflows. Ideally, it may be time to consider the trade-off of including older code, versus rewriting the code to more modern coding practice and methods. Another code practice is to pay attention to warnings and flags tripped in compile, and again detailed analysis from linting and static analysis tools. Using compiled code with lots of warnings should not be standard practice. Every warning needs to have a reason, or even better, a fix. This means budgeting for a testing and verification exam by a team, preferably an independent team not influenced by acquisition pressure. None of these practices are new, rather they are well known quality measures that help security as a benefit. The better the software process and practice, the better the code quality, the lower long-term costs, and the better security.

As mentioned earlier, secure coding practices and methods such as OCTAVE [12] can be used to prevent issues from coming into code, implementation standards in the NIST series [13] and using the tools noted above, that draw from weaknesses as in [14].

Once a system is constructed a healthy red teaming approach to find security issues before launch and before adversaries find issues, is a must. It is an act of programmatic courage to expose your system to onslaught, and even more courage and funding, to fix what red teams find. The MITRE ATT&CK™ [15], Engage™[16], and d3fend™[17] methods provide likely tactics and methods of adversaries, as do books like Gray Hat Hacking [18] and many others. Program managers should plan from the beginning for red team activities, but even red teaming late in a program's lifespan, as before launch of successive satellite systems in a series, or even on orbit, can bring benefit. Better to have your friend point out your faults, than your enemy use the faults and you find out later.

VI. Conclusion

There are many ways bad actors could deny service or disrupt aerospace systems, and the memory overflow vulnerabilities shown in this paper are but a single way to cause trouble. Even this lower probability attack can be included from angles ensuing from a schedule driven over quality driven software development approach. Fully employing a DevSecOps methodology is not a panacea, but matched with managerial commitment to secure quality code, can mitigate these attacks by preventing the angles. Further work in projects must include planning and funding for updates to systems including space-based assets, since older code accumulates the vulnerabilities found each year. This paper is only a top-level overview and should be followed by deeper coursework and training across the aerospace system enterprise.

Acknowledgments

While this paper was written outside of work, the author would like to acknowledge his employer, Carnegie Mellon University Software Engineering Institute, and concepts from the SEI CERT Secure Coding Team.

References

[1] "Smashing The Stack For Fun And Profit" by Aleph One, 1996, Underground.Org, https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

[2] Common Vulnerabilities and Exposures (CVE): https://www.cve.org/

[3] Top 10 Secure Coding Practices, created by R. Seacord, found at: https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices

[4] "DevSecOps Fundamentals Guidebook: DevSecOps Tools & Activities", March 2021, v2.0, U.S. Department of Defense,

[5] ISO/IEC Standard 9899:2018, International Organization for Standardization, Chemin de Blandonnet 8 CP 401, 1214 Vernier, Geneva, Switzerland, © 2018, found at: https://www.iso.org/standard/74528.html

[6] Open Worldwide Application Security Project (OWASP) Security Knowledge Framework, found at: https://owasp.org/www-project-security-knowledge-framework/

[7] SEI CERT C Coding Standard, available online at: https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

[8] Seacord, R. C., *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2016, URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=454220.

[9] Seacord, R. C., *Secure Coding in C and C++ (SEI Series in Software Engineering)*, 2nd ed., Addison-Wesley Professional, Upper Saddle River, NJ, 2013.

[10] Bryce L. Meyer, "Perilous Paths: The Cybersecurity Issues of a Space-Based Cloud Imagery Processing System" AIAA 2022-4327, 15 Oct 2022, https://doi.org/10.2514/6.2022-4327

[11] K. Lukin and M. Haselberger, "Hacking Satellites With Software Defined Radio," 2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC), San Antonio, TX, USA, 2020, pp. 1-6, doi: 10.1109/DASC50938.2020.9256695.

[12] Alberts, C. J., Dorofee, A. J., Stevens, J. F. Stevens, and Woody, C., *Introduction to the OCTAVE Approach*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2016, URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=51546.

[13] National Institute of Standards and Technology, "Security and Privacy Controls for Information Systems and Organizations," Special Publication 800-53, Revision 5, NIST, Gaithersburg, MD, September 2020. URL: https://doi.org/10.6028/NIST.SP.800-53r5.

[14] Common Weakness and Enumeration (CWE): https://cwe.mitre.org/

[15] MITRE ATT&CK ™: https://attack.mitre.org/

[16] MITRE Engage ™: https://engage.mitre.org/

[17] MITRE D3FEND ™: https://d3fend.mitre.org/

[18] Allen Harper, Ryan Linn, Stephen Sims, Michael Baucom, Huascar Tejeda, Daniel Fernandez, Moses Frost, Gray Hat Hacking: The Ethical Hacker's Handbook, Sixth Edition 6th Edition, McGraw Hill, ISBN-10, 1264268947, ISBN-13, 978-1264268948