ASTES

Advances in Science, Technology and Engineering Systems Journal Vol. 4, No. 2, 357-369 (2019)

www.astesj.com

Special Issue on Advancement in Engineering and Computer Science

ASTESJ

ISSN: 2415-6698

Computer Security as an Engineering Practice: A System Engineering Discussion

Robert M. Beswick*

Mission Design and Navigation Section, NASA, Jet Propulsion Laboratory, California Institute of Technology 91109 USA

ARTICLE INFO

Article history:
Received: 29 June 2018
Accepted: 03 April 2019
Online: 24 April 2019

Keywords:
Computer Security Engineering
Security Fault Tolerance
Ground Data System
Defense in Depth
Defense in Breadth
Least Privilege
Vulnerability Removal
Absolute Security
Sterility in Implementation
Time Based Security
Communications Security
Operations Security
Complementary Intersection

ABSTRACT

We examine design principles from more than 20 years of experience in the implementation and protection of mission critical flight systems used by the Mission Design and Navigation Section at NASA's Jet Propulsion Laboratory. Spacecraft navigation has rigorous requirements for completeness and accuracy, often under critical and uncompromising time pressure. Fault tolerant and robust design in the ground data system is crucial for the numerous space missions we support, from the Cassini *orbital tour of Saturn to the Mars rover* Curiosity. *This begins with the examination of principles learned from fault tolerant design to protect against random failures, and continues to the consideration of computer security engineering as a derivative effort to protect against the promotion of malicious failures. Examples for best practice of reliable system design from aviation and computer industries are considered and security fault tolerance principles are derived from such efforts. Computer security design approaches are examined, both as abstract postulates (starting from cornerstone principles with the concepts of Confidentiality, Integrity, and Availability) and from implementation. Strategic design principles including* defense in depth, defense in breadth, least privilege, *and* vulnerability removal *are target points for the design. Additionally, we consider trust in the system over time from its sterile implementation, viewed against the backdrop of Time Based Security. The system design is assessed from external access data flows, through internal host security mechanisms, and finally to user access controls. Throughout this process we evaluate a* complementary intersection *– a balance between protecting the system and its ease of use by engineers. Finally, future improvements to secure system architecture are considered.*

1. Introduction

This paper examines approaches used in the process of securing the computer systems employed by the Navigation Ground Data System. This text is meant to serve as a discussion about best practices in computer security engineering. It is based on twenty years of the author's practical, "in the trenches", field experience on systems involved in this effort. These systems comprise a multi-mission network that encompasses the navigation elements of more than forty current and previous interplanetary flight missions here at the Jet Propulsion Laboratory. This paper does not seek to be a prescriptive document (do this one thing, buy this product, etc.), but instead seeks to examine a process of how secure systems are designed – i.e., what general security principles we have found valuable [1].

Few systems require as much resiliency or have as much risk of causing negative (and final) outcomes as the computer systems used in support of Flight Operations. Scant resources are often available for the maintenance of complex hardware and software architectures, and these high-availability/high-reliability systems are often expected to function without (and moreover cannot *tolerate*) the regular sorts of software updates expected in other computational environments.

As an example, one of these missions, Cassini, from its launch in October of 1997, would spend seven years crossing the solar system, arriving at and entering into Saturn orbit in June, 2004. It would then orbit Saturn nearly three hundred times over thirteen years, conducting hundreds of targeted flybys of Saturn's largest moons during its mission lifespan, finally coming to a fiery end in Saturn's atmosphere in September of 2017. This operational effort would be conducted on a network of computer systems having a requirement for no more than two minutes of unplanned downtime

*Robert M. Beswick, 4800 Oak Grove Dr., Pasadena, CA 91109 USA, 1-818-393-0539, Robert.M.Beswick@jpl.nasa.gov

a year (99.9995% availability) [2]. For a network that could not be upgraded (except in minor incremental steps) during the length of the entire 13-year Saturn orbital tour, maintaining and keeping such a computer network secure, as well as supporting backwards compatibility (and obsolescent hardware) would be a challenging problem. In fact, the systems used over the launch and interplanetary voyage across the Solar System contained less processing power, storage, and *total* memory than the author's current iPhone – and these systems would have to be kept functional and secure in case there was a need for backwards comparison.

While some organizations may have little need for a computational environment larger than a client-server configuration of a few systems, others may require a more complex environment. The Ground Data System for Navigation is constructed in the manner of a classic Development and Operations model with development and production networks of several hundred servers and workstations. This network is used by teams of engineers working, often under critical time pressures, with rigorous requirements for accuracy [2].

For this environment, it should be clear that a particular cryptographic protocol, a set of software, or even an appliance will not solve these security challenges. This is not an application problem but instead a systems problem. As in other engineering disciplines, in such an environment the design must be based on good *principles*, because, like with the pouring of a foundation of a building, you only get one such opportunity.

Part of the effort in this discussion stems from the author's own struggles to find a good systemic set of definitions and guidance while endeavoring to build a more secure network. What you see here is a reflection on the insights I was trying to find for my own efforts. Furthermore, many current efforts in computer security research are based not on building a secure system but tearing a system down. Indeed, a great deal of present investigation in computer security is based around the twin ideas of "if it ain't broke, don't fix it", and "try and see if can be broken". Accordingly, extensive research across the spectrum of computer security is being conducted in the field of penetration testing, where the primary methodology is to break into an existing setup and then fix the discovered problems. While this can be helpful in discovering specific flaws, it does not provide much help in trying to understand a more general architecture model of security.

An example of this is perhaps best exemplified by the software tool known as "Chaos Monkey" – part of an open source suite of tools called "The Simian Army" that was originally designed for the cloud infrastructure of the streaming media service Netflix [3]. This software set comprises a series of tools that help the design of resiliency in a set of virtual machine instances. It does so by randomly shutting down members of the set of virtual machines. By forcing developers and systems engineers to prepare for unexpected failure, a more fault tolerant network design will emerge (it is hoped). This has been further expanded and generalized to even more powerful tools that comprise "Chaos Gorilla", which randomly simulates the shutdown of an entire Amazon Availability Zone, or the even more devastating "Chaos Kong", which simulates the shutdown of an entire Amazon Region [4]. This design is an example of "survival of the fittest", incorporating a genetic algorithm-like approach to the design of secure systems.

Critical questions that should be raised in conjunction with this software are, "How does a developer or systems engineer build a more stable and secure system? What principles should be used? What methods should be avoided?"

Continuing with the metaphors given above, consider an analogy to the above (genetic) algorithm: a team of stone-age architects trying to build a bridge across a river, first by using a captive monkey, then a captive gorilla, and then a giant mythical beast to try throwing stones and batter a pile of rocks into a working bridge. While such a method will produce some results (albeit very slowly), what about other engineering approaches to design? What of the arch and the use of suspension? What about the consideration of tension and compression and the use of different materials in the construction of a bridge?

Clearly other engineering principles can be useful here in order to produce an initial design and improve upon it before bringing someone to attempt to tear it down. In like manner, this paper is a study on security architecture design, and hopes to add to such efforts by discussing principles on "how to build a (better) bridge".

A word about the expected audience of this paper: the design principles discussed in Sections 2 and 3 are aimed at top level design of secure systems, and may be of greatest use for project management and systems engineers (it could also be titled "how to avoid buying crap"), while Section 4 covers an example implementation targeted more for computer systems architects and systems administrators. The paradigms covered in Section 2 and 3 are observed derivations from fault tolerance and have wide applicability in systems engineering, while Section 4 applies this methodology to securing a specific computer system.

2. System design approaches: Fault Tolerance and Security

Fault Tolerance

2.1. Fault Tolerance

There are several valuable definitions for the concept of fault tolerance. Fault Tolerance, according to Carl Carson, comprises "...a design that enables a system to continue its intended operation, possibly at a reduced level, rather than failing completely, when some part of the system fails" [5, p. 167].

This has been categorized by Barry Johnson in the approaches taken for fault tolerance in microprocessor design as [6]:

- Minimize the number of points where a single fault will cause the whole system to fail.
- Graceful degradation – known also as "Fail-Gently", a system's ability to continue operating in the event of a failure, having a decrease in operating capacity no worse than necessary for the severity of the failure.
- Redundancy in components – both in space, having multiple parts that can be utilized, and in time, with the repeatability of an operation.

Jim Gray and Daniel Siewiorek help characterize such steps to provide high availability in computer systems by examining single points of failure to promote [7]:

- Independent failure – where each module functions so that if it fails, it does not impact other modules in the system.
- Design diversity – using hardware and software from differing organizations to promote independent failure modes (e.g. differing *types* of failure).

There are abundant examples of fault tolerant design. This is a well understood area of systems engineering. From the 135-year old Brooklyn Bridge in New York [8], to the current Eastern Interconnection power grid of the United States [9], approaches such as these are needed where one must trust the behavior of a system to work, and to work in predictable ways.

As an aspirational example, consider one such computer design further at the highest end of the reliability spectrum. The primary flight control system of the Boeing 777 achieves ultra-high reliability metrics. It is a highly redundant, highly available system comprising the fly-by-wire avionics controls and is a rare example of a true Triple Modular Redundant (TMR) system (with little exception the highest level of redundant design, having three redundant components for each single point of failure), in both computer nodes, software, and hardware. It has service metrics requiring a maximum rate of failure of the flight computers of 1.0×10^{-11} failure/hours (i.e. a failure of the flight computers less than 1 in 100 billion flight hours) [10]. This is an example of what can be done with sufficient effort and due diligence – a computer system millions of passengers a year put their trust in.

2.2. Security Fault Tolerance

In like manner, this paper considers computational systems that one can trust – as is done with the physical systems described above. From fault tolerance, we can derive similar approaches to deal with the actions of intelligent actors[1] rather than random chance or stress failure modes. Such an approach was described affectionately by Ross Anderson and Roger Needham as "Programming Satan's Computer" [11]. While the application of these principles we apply here to the flight computer and network security of our systems, the principles are applicable across the board to security design. Such design promotes:

- Secure systems should be resilient from random chance and predicted stress modes of failure.
- Secure systems should also be resistant to direct action.
- Simplicity of design, known as the popularized KISS principle, is a golden virtue in secure systems [12].

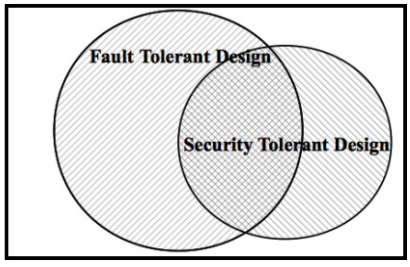

Figure 1: Relation of Fault Tolerant and Security Tolerant Design

These principles can be considered in four approaches:

2.2.1. For the most critical systems, use a machine that has only one function.

This approach has the benefit that in the event of a failure, you have lost only that one function. There are numerous examples in security design. Indeed, "appliance" IT systems, such as web servers, email servers, firewalls, and similar systems use this approach. As a good analogy: in (most) kitchens, refrigerators and ovens are not a part of the same appliance – even though they have the same task of changing and maintaining food temperature.

2.2.2. For redundancy in a security context ideally differing configurations should be used.

This technique considers that the systems in a given setup should not all have the same potential vulnerability (and therefore not truly redundant against a threat). It is an example of the previously discussed design diversity. Examples of this include the use of multiple arrays of web servers – running different OS/web server software, or application servers that use differing configurations and password sets, or multiple (different) backup systems – e.g. tape, disk, and cloud service providers. As an analogy, consider the commute from a major urban center: it is better to have several differing options for transport, be it freeway, commuter train, bus, or even surface streets. In the event one method is impacted, other options are available to return home.

2.2.3. Single points of failure of a system should be few, and truly independent.

As observed above, in a security context, systems that are single points of failure should be truly *independent* of other points of failure. Such single points of failure should be examined closely to ensure that they are actually single, independent, points of failure.

This goes hand in hand with the first point about single use. One example of this can be seen in network file server "appliances" that perform the function of serving files – other application sets such as virus checkers and configuration management tools may be run on the files themselves, but they are seldom loaded on the file servers. This design approach is often misunderstood in poor implementations, especially where security may be seen as a software package or an add on feature. As an analogy consider tires that serve as critical single points of failure for a modern car – in a similar manner, it would be absurd if a failure of the GPS system caused the tires to fail!

2.2.4. Systems should Fail-Safe and/or Fail-Gently.

In case of failure, a security system should degrade safely (known as Fail-Safe), and/or compromise only a limited part of the overall system or otherwise take an "acceptable" amount of time to fail ("Fail-Gently"). Examples of this abound, many of them utilizing cryptography, such as disk and file encryption, password login for most computer systems, firewalls, network segmentation, and network Intrusion Prevention Systems (IPS). An analogy can be found in the mechanical realm in that high grade safes are rated

[1] To be clear: this term is meant here to describe human attackers, not intelligent software agents. They may also be described as malicious actors, or threat actors depending on context. The colloquial and overlapping terms of hackers, crackers, or security hackers are also used in the general news media.

for time against tools, cutting/welding torches, and explosives. Some safes may have mechanisms that break or cause the safe to be unopenable if drilled or otherwise tampered with.

3. Computer Security Design

3.1. Abstract Principles

This computer security design incorporates the above techniques through increasing grades of refinement. Abstract concepts of Confidentiality, Integrity, and Availability (the traditional trinity of security) help to determine the "who, what, when, and where" of the security needs for navigation computation. These principles provide definition for the key concerns for securing a system, not in terms of technique, or subsystem protected, but rather in terms of what features in the computational environment must be protected.

3.1.1. Confidentiality

Confidentiality is referred to as "the concealment of information or resources" [13, p. 4]. This can be of high priority for some financial systems, where customer data is not only a crucial part of business operations, but also where strong legal regulations may come into play for control of customer information. It may be critical in military computer systems (for some cases it may be more desirable to destroy the system than allow the unauthorized release of information). This is significantly less crucial in the field of navigation computation. Such information for navigation comprises mechanisms used to authenticate access to the systems, network and system configuration information (possibly of use to subvert security), and restricted navigation software. An example of a baseline concern for Confidentiality can be seen even in the naming of individual systems, as such host names may reveal a great deal of information about the underlying network design [14]. Consider a hypothetical, badly named example from JPL: the host name "cas-web-serv3", which immediately gives information the system in question is running a web server daemon, for the Cassini project, and that most likely there are (at the minimum), two similar servers.

3.1.2. Integrity

Integrity is "the trustworthiness of data or resources ... usually phrased in terms of preventing improper or unauthorized change" [13, p. 5]. Integrity is particularly critical for navigation computation, as improper or unauthorized modification of the environment could cause very serious problems. Corrupted data sets, results or software could cause terminal errors in spacecraft control. With regard to the accuracy required for navigation of spacecraft missions, the sensitivity placed on the accuracy and integrity of the data, software, and corresponding results cannot be overstated. One such example of the critical nature of Integrity is seen in the previous case of the Cassini project. Integrity was critical on such a mission, which consisted of one of the largest teams of navigation engineers ever assembled, flying on the navigation computer system one of the most complex orbital and interplanetary trajectories ever designed [15].

3.1.3. Availability

Availability is "the ability to use information or resources" [13, p. 6]. Conceptually, it is the most direct derivation of principles coming from fault tolerant design. Availability in the context of navigation computation, has an additional concern that someone could deliberately try to deny access to a system service or data sets. The 24-hour-a-day, 365-day-a-year activity calendar of spacecraft navigation indeed makes this a crucial concern [15]. An event which causes the system to be unavailable, which in other industries might be cause for concern, could lead to the premature end of a multimillion dollar space mission.

3.2. Implementation

From these abstract principles, there are a number of paradigms that are employed for the implementation of better protection. Accordingly, strategic archetypes are used in the work toward good security design.

3.2.1. Defense in Depth

Defense in depth is a military security term with a long historical pedigree. Instead of a single defensive stronghold or chokepoint, this strategic principle is based around a series of overlapping defenses, forming a type of fault tolerant redundancy [16, pp. 40-44] in that a failure of one part of the system does not lead to the failure of the whole system. Considering a networked computer system, this can be seen on a variety of levels in overlapping defenses: starting at the network perimeter firewall, the network segmentation rules used on the network switches, the host based firewall, the protections applied to the processes that connect to the network, and then the internal host security measures that must be bypassed by a potential attacker. This is analogous to a defensive perimeter made up of a series of overlapping walls and fortifications, much like that seen in that of a medieval castle, or World War I trench warfare, presenting so many obstacles to entry that an adversary, having a choice, will pick a less well defended target [17, p. 259].

3.2.2. Defense in Breadth

Defense in breadth is a term serving as the converse for *defense in depth*. It involves the concept that a secure system design must consider the whole system in its implementation, or an attacker will simply ignore a well fortified part of the system and instead chose an easier route into the system. All of the previous paradigms of overlapping defenses, minimization, elimination, and security over the system life, must be considered for *every* part of the system. This is a difficult task and it does lay bare the challenge of the security engineering process. By definition this is an effort to examine the totality of all of the possible attacks against the system. Indeed, this defensive principle is referred to by Sami Saydjari as *Ensure Attack-Space Coverage*. He acknowledges that achieving this comprehensive coverage is a large and challenging problem, and one that is usually not well understood, but he underlines its necessity as a security engineering principle when he remarks "Depth without breadth is useless; breadth without depth weak. " [18, pp. 133-135] Consider an analogy from a failure of this principle: putting a well-locked door with a security system on a Hollywood film set facade; while it is possible to break through the front door, it is much easier to simply walk around the side or back of such a facade because most such sets do not even have a back wall. History is rife with excellent examples of the failure to consider the whole scope of ways an adversary could carry out an

attack. One such example can be seen in the WWII failure of the Maginot line on the French-German border. [2]

3.2.3. Least Privilege

The principle of *least privilege* is that subjects (processes and users by extrapolation) should be given only the rights and accesses for the tasks the subjects need to perform, and no more [13, p. 457]. This avoids many possibilities for the manipulation of a process (or manipulation by a user) to get them to do things they were not intended to do. There are many similarities to ideas in finance and accounting where financial mismanagement and corruption in corporations are prevented by not allowing any one person to take charge of all the corporate finances – also known as the "two man" rule. As an example, consider a hypothetical printing utility that can write to a central spooling directory as a privileged user. It is discovered that by manipulating the input files sent to the printer, it is possible to write to other directories on the system. With some effort and cleverness it may become possible to overwrite critical system files or even get remote access to the system. By changing the permissions of the spooling directory and removing the ability to run as a privileged user, the printing utility can continue to function, without enhanced access, thereby removing its capability to be misused. A similar analogy is a classic safe deposit box repository in a bank, where two people (each with a different key) are required for access to a given safe deposit box.

3.2.4. Vulnerability Removal

When we remove potential vulnerabilities, we can elude the many problems that come with those vulnerabilities, and sometimes avoid other issues as well. Matt Bishop refers to this as *The Principle of Economy of Mechanism* [13, pp. 459-460]. An example of *vulnerability removal*: a self-replicating worm that attacks email server processes can do little damage if you do not have an email server on your system. Likewise, if you do not have a web server running on your system, there is no need to monitor for such and keep up with attendant patches. Making sure that (along with the software and its own attendant maintenance) possible vulnerabilities in buggy software are not installed, can save tremendous resources and time over the system lifetime. As an analogy, this is akin to not just locking a side door into a building, but instead never constructing the door into the building in the first place.

3.2.5. Sterility and Absolute Security

We now consider the idea of *sterility.* This is a point along a spectrum of levels of trust in a given system. Understanding how much one can trust such a system may help determine how it should be used and where it should be placed on the network. A few (very rare) systems we consider to be sterile and have *absolute security*, the top of this spectrum, or "security beyond any reasonable doubt". [3] To be clear: this is a theoretical initial state, a starting point for definition, and *no* connected or exposed system can be said to be sterile and in this state. This is a valuable concept, especially as we consider the next paradigm. It is comparable to a definition of an initial starting point (a *t0*) that is useful when considering how that system might evolve over time.

Let us further consider, that a system implemented on a closed network, built from a secure image is secure, or *sterile* to the limits of confidence one has in the hardware (and the hardware vendors), network, and the software media (DVD or secure image and/or patch servers, etc.) used to construct the system. On such a spectrum of trust this system would be seen as approaching *absolute security* considering these limits of confidence. This useful idea can be seen as extrapolated (as systems in miniature) from credentials, passwords, or certificates that also have not been exposed. We can liken this to issues seen in medical care, or food preparation and handling, where much effort for hand washing, cleanliness, or cooking is undertaken to produce sterile areas, or clean food for consumption.

3.2.6. Security over time or robustness

The previous paradigms examine the implementation of a static system. Considering the evolution of a system state over time, one should also consider how the system is secure during its implementation and over its whole lifecycle. How robust is this system? How much can it be trusted and how does that trust change over time? This is similar to principles of Communications Security (COMSEC) and Operations Security (OPSEC). These are military security terms that examine the communications and background operations processes of military organizations, working to prevent compromise of communications, and the processes of such organizations. As in a military organization, our trust in a system will change over time. This is strongly dependent on the environment and networks that connect to this system. It is very important to examine these connections and method of contact to characterize the *security over time* of a system.

This concept is akin to what Winn Schwartau called "Time Based Security" [18, p. 34]. In Time Based Security, a system would be considered "secure" if the time to break through the system protection (Pt) was greater than the time required to detect (Dt) and to respond (Rt) to the intrusion. This can be expressed as:

$$Pt > Dt + Rt \quad (1)$$

Time Based Security suffers in the great difficulty of quantifying these values in (1), for the system protection, detection, and reaction time. Nonetheless Time Based Security does help illustrate well that security over time is a derivation from the fault tolerant concept of Fail-Gently, as discussed above.

Coming from the previous point about sterility, trust in a sterile system must change when it is attached to outside networks. Our confidence in such a system is then based on the security configuration of the external network, internal host, and user

[2] The Maginot line was designed to protect France from a German incursion in the event of a second world war. It was an excellent example of the principles discussed here, providing significant fault tolerant, defense in depth, with carefully considered access and both perimeter and internal security. A marvel of military engineering, its primary construction protected the French-German border. Most elements of the line were operational and fully prepared to continue fighting *after* the fall of France, and had to be ordered to surrender with the French capitulation. The designers famously did not consider (mostly due to the staggering costs to expand the fortification) an assault across the French-Belgian border.

[3] This is a functional term that in our model is akin to, but not the same as, the mathematical term *perfect security* – such as the unbreakable one-time pad [36].

controls of the system. Log mechanisms can help give indications of change to the system. If this trust is misplaced, the system could be compromised, and the system could become untrustworthy and itself a source of security concerns. As in the previous section we consider that this is akin to handling a compromised password or certificate, and in most cases the system must be rebuilt.

There is considerable kinship between these ideas and the study of infection and food-borne illness. The frequent admonition about hand washing before meals comes to mind. What things did one touch or pick up? Do you have a thermometer to see if the food is cooked? A brisket set out too long at a picnic may "go bad" and become a host to viruses, worms and other sources of contagion. It may be possible to sterilize it, or at least to clean the dish containing the item. Much care is required to safely clean dishes under food handling regulations, but having gone bad, food, no matter how savory, almost all of the time must be thrown out.[4]

4. System Evaluation

Keeping these principles of secure system design in mind, we consider an example case: a single computational node and the techniques and mechanisms put to use to create a secure system. Our model system is a server running Red Hat Enterprise Linux 7. Particular care is taken in its construction as it serves as a fundamental "building block" in our system plan. This security design takes on several design principles found in architecture in that the strength of a system can often be improved, not by what one adds, but by what one takes away. Also, this serves as an effective approach in considering the overall security of our environment. Moreover, many systems are actually allocated in a "one engineer, one workstation" ratio. Similar efforts can just as easily be a network segment, implemented in a network group.

It is useful to reiterate here: this is *example*. These security architecture guidelines can be applied in many other approaches and types of system setups. This is an outline that other system architects and system engineers can follow to create their own secure systems. It is hoped that this demonstration will help to make these key concepts and their application in design more clear. This system-design model is a simple one. It is to serve as a means to explain principles and their use in an architecture. As we extrapolate to a network, or a network-of-networks we get a clearer understanding of these principles and their use in a model. Complexity may be introduced in efforts to move towards a particular architecture model.

For example, one such effort called Zero Trust computing, moves towards a very atomic model of trust, that of so called "de-perimeterisation" where the outer defenses of a network system are assumed to be ineffective. Zero Trust uses the same design goals, but with an aggressive application of the principle of *least privilege* applied to every element of a network [20, pp. 2-3, 12-18]. It involves the hardening of every computational node, and every network connection and data access, and moves away from the traditional specialization of gateways, firewalls, and trusted subnets or zones. The model we discuss is agnostic to such efforts, as approaches we take here are equally valid for such designs. A more "classic" network security model is, simpler to explain.

Our fundamental design goal is to create a system that can be trusted and relied upon to perform the difficult task of spacecraft navigation. This design effort sought to find an optimal tradeoff between usability and security, the point of both maximal utility (for the users) and security: a point where users could work without being fettered by obtrusive security measures, while confident that their work is safe. Indeed, it is a mistaken, albeit popular, notion that security must interfere with usability, and that secure systems must necessarily be hard to use. Much of the time this misconception is due to a poor implementation of a secure feature, or a security control that was added on as an afterthought to a software system. Failing to consider these requirements and needs of the users also can lead to the perceived adversarial role of security in an organization. The idea of examining the security controls in consideration with the user environment is referred to by Matt Bishop as the *Principle of Psychological Acceptability* [13, pp. 464-466]. The strategy we consider here strives to find a complementary intersection between usability and security, an optimal point where these two groups of competing requirements reach their maximum effectiveness [21]. This is similar to the concept known as the Center of Percussion (known also as the so-called "sweet spot") as found in weapons, aircraft, sound engineering, and sports equipment design – where multiple factors combine into an optimum response from a given amount of effort [22]. Analogously, these competing needs are similar to the economic concept of the law of supply and demand.

Finding this complementary intersection requires the evaluation of both security and user requirements. For an example of this optimal intersection, consider the often maligned password change requirements of most institutions.[5] Usually eight or more alpha-numeric, and special characters are required, often with other "randomness" requirements. These passwords often have to be changed every 180 to 90 days (or less). As a matter of cryptographic security, the longer the encrypted password hash, the more difficult it becomes to conduct a brute force attack to obtain the unencrypted password. Hand in hand with this, changing the password more frequently helps ensure that even if the password is broken or obtained by some other means, the window of exposure is short before a new password is created. Evaluating these security requirements, without user input, leads to more "random" and longer passwords, approaching something looking akin to line noise, with changes occurring in shorter and shorter time intervals. Taken alone, this might be considered to be a positive trend.

However, it has been widely known for some time that the user response to such policies tends to decrease the security of the system [23]. Users, unable to remember their ever-changing passwords, proceed to create easily guessable password combinations, or worse still, write passwords down in easy-to-find locations such as next to computer monitors or under keyboards. Evaluating both security and user utility requirements can instead lead to an optimal solution for these concerns. This could include

[4] There are limitations to this analogy. One of my associates who is a professional chef cited several counter examples – however he was quick to note that this holds true for 99% of food handling concerns.

[5] A memorable and humorous example of this can be found at http://xkcd.com/936/

a longer password rotation with a longer password, or possibly a smart card and a password (for two- or three-factor authentication).

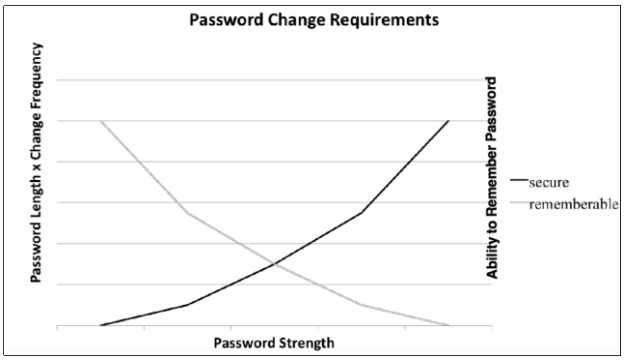

Figure 2: Complementary intersection of password controls.

But how can this optimal intersection point of security and usability for this system be determined? For this system design an initial setup is undertaken stemming from security design requirements and user needs. Numerous manuals for securing systems can be used *a priori* to determine a baseline configuration. We have used recommendations from the National Institutes for Standards and Technology (NIST) [24], the Information Assurance Directorate (previously named the System and Network Attack Center) of the National Security Agency (NSA) [25], or the Center for Internet Security (CIS) [26]. Using the security principles we discuss, these (and other) security approaches are evaluated, determining what user needs are crucial while seeking as secure a configuration as practicable. Areas of conflict (and overlap) are considered and from this iterative changes can be made.

Changes are then evaluated on the privilege, area and criticality of the alteration. Iteration towards an optimal state is a manual process of testing, examination, and consultation with users. For our model, there are three areas of concern: external network access, internal host security, and user level access controls. These areas have differing complementary intersections: the optimum trade-space differs between these areas of concern. Once this optimal state is determined, we then consider the evolution of this security state over time.

4.1. External Network Access

Under the greatest amount of scrutiny, the area of greatest care for the system design is the configuration of external network access. This is an exercise in optimization and extreme minimization. While users will be surveyed for input on desired services, security needs predominate in this area over almost all user concerns. Every access point (open network port or service) is of the highest concern as network access control is the first and best line of defense against intrusion. As we note above this approach can be generalized: this principle applies to both individual computer nodes, as well as networks of computer systems. For an analogy, if the network is compared to a street, an IP address would be the street address of the building and the ports would be the doors of that building. A firewall could be a wall surrounding that building (having its own doors/gates), or around several buildings in a compound. This analogy can be further extended much like that of a design of a military base (or a medieval fortress) with layered defenses. Again, this system design model is a simple one, and other design choices may factor into such an extension. In the area of external network access, the interplay between security controls and user needs will be very heavily tilted toward security because this represents the first, best chance to block an attacker (at the gate).

As mentioned previously this security design could be used in other system approaches. When we examine network access points to the model system, we should also consider other access points, such as physical connections to the system like the ubiquitous USB or Thunderbolt connectors. One can also consider similar methods by extrapolation to evaluate a database system, or a file storage system, by examining the controls on the data, and the access points to that data.

The approach utilized involves cutting down to the absolute minimum the number of open network ports, and then over those ports minimizing the content and amount of data transferred. Among the network testing tools we use, two include the Nmap port scanner (providing a comprehensive scan of all open ports on a machine) [27], and Nessus, a widely available network security scanner that helps probe for security vulnerabilities on the remaining open ports [28]. Indeed, although the user community may with regularity request the installation of numerous software applications that will have their own unique web server on an additional open port, such services will usually be disallowed. This is necessarily an iterative process. Sometimes if taken too far and too many ports are closed off and services shut down, the machine will not be able to communicate over the network. There may be times as well when certain user applications really do need to have a port opened for them. These should be examined most carefully. Those few services that are still available will be expected to be extremely secure against external subversion, such as the SSH remote login/file transfer/port forwarding service. In this approach:

- First test the system absent the host based firewall. This is to ensure the underlying system is secure without it. The firewall should not be the only point where outside connections are minimized and controlled. *Almost all network services can be shut down on a secure system with no real difficulties to the underlying function of the operating system!*
- Next test the system with the host based firewall engaged. In many cases stateful packet filtering can be used to trace (and allow) replies to network calls, which will eliminate the necessity for most holes in the firewall.

Table 1 depicts the results of this process for our example. From more than twenty open network ports on our initial image (which is a great improvement over previous Red Hat Linux versions) we reduced the number of open network services to a bare minimum of essential network services:

Table 1: Number of open network services with and without Firewall – Final Minimum Services

Firewall State	**TCP**	**UDP**	**ICMP**
No Firewall	3	6	2
Firewall	2	3	2

This is an excellent example of the point made earlier that the strength of a system can many times be improved by what is taken away from it. Contrasting these results with the hundreds or thousands of network services on a general home Mac or Windows computer can be enlightening as to the current default state of computer security.

4.2. Internal Host Security

A highly secure system that is impossible to use for its primary duties is not very *useful*. In evaluating our system's internal host security, the interplay between security controls and user requirements must be nearly equal due to the competing concerns to both keep it working as efficiently, and as secure, as possible for the users of the system.

By analogy, inside the building, the use of the building by its inhabitants is an important design consideration. Is it a home? A bank? A police station? Safe construction approaches are dictated for all buildings by fire codes, but how a building is used will determine much of a building's interior design.

For our own computation environment, this process of system hardening considered functions and services the user community required:

- The completion of operational duties such as supplying critical data sets to mission partners and utilization of mission critical flight software.
- The supply of required capability for a task or job role (e.g. visualization and formatting tools, viewers, editors, functional compilers, and third party software).
- The provision of *useful* utility functions and software (e.g. BR/DVD/CD archiving, external export of interesting graphical output, the capacity to safely load removable file systems, and network printing).

From these characterizations, a definition evolved of what behaviors were expected of a user and conduct they should *never* be expected to engage in. With such a set of definitions we removed from the system those privileges and programs a user was never expected to use, and added further constraints so users would have difficulty in performing anomalous activities.

Similar to the approaches used to secure the external network access of the machine, a software tool was used to evaluate the internal security settings of the machine and produce benchmarks for enhancement. Created by the Center for Internet Security (CIS), the CISscan package was used to evaluate the internal host security for the system [26]. It provides a set of metrics for assessing security configurations for a number of package distributions, as well as operating system platforms, allowing comparisons between variant software sets, alternate configurations, and operating systems. Representing an effort to create a "best practices" industry standard guideline to system security, the CISscan package is based around a set of consistent standards, that can be used to evaluate a given system against such metrics. With this tool, it was possible to create a security baseline, make sure that configuration changes and patching continued to meet that baseline, and iterate to a desired system configuration.

Categorical changes to the internal security settings included:

- Removal of extraneous executable permissions. The ability to change users in a program is an especially serious concern. SetGID and SetUID executables have the ability to change the group and user of the process that runs them. With this ability such programs present an extreme security risk. Some programs however, such as the ones that are required for users to log into the system need such a capability. Like our evaluation of open network services, almost all operating systems have a needlessly high amount of these types of programs by default. Even the latest versions of Red Hat Enterprise 7 contain numerous examples of programs with too many privileges. In our system, with effort we reduced the number of these very sensitive programs from 45 SetUID executables in the initial install to a total of 15 in the final secured configuration (see Table 2 for full discussion).
- Limit the installation of new software. Every new software installer has the potential to open up new vulnerabilities. It is far better to avoid such problems by not installing the software at all. As noted, this may minimize many maintenance issues as well, including keeping the software secure and patched. A related question for discussion is, "does your software really *need* its own web server?"
- Limit or otherwise shut down all unnecessary system services. For some minimal operating system configurations, the number of system services running on startup can be counted on one hand. This is a far cry from the average computer, which can have more than a hundred processes running at any given moment before anyone has logged in. Such extraneous processes can represent significant vulnerabilities on a system, especially if they have a network port open.
- Modify key network and kernel data structures. By changing the settings of key network and kernel configuration values, it is possible to significantly increase the protection of the system against denial of service attacks and other security failures with the occasional cost of small increases in memory use. Indeed, lowering user limits (number of users) and per-process (number of processes) on the system, significantly lower the odds an application can harm system function, blocking a denial of service attack by preventing such software from consuming system resources. Fortunately this area is one of the least likely to result in negative feedback and consequent user pushback.
- Restrict file system permissions on untrusted and/or critical file systems, secure application sets, and use caution for removable file systems. Crucial file systems are mounted read-only on most machines in our environment (especially the ones containing critical software). Also, file systems that are removable cannot execute programs with SetGID or SetUID privilege as it is trivial to create such programs that can give the user elevated privileges (thus bypass system security) on a different system and load them from a removable device.

These categorical changes are summarized both from the metrics of the CISscan tool and with direct metrics (where possible). Results of these efforts are shown in Table 2 and Table 3, taken from our model system image at four different revision times.

Table 2 examines direct metrics that can be compared across several differing system images. This includes the total number of SetGID and SetUID programs in the image, the number of "classic" System V and "new" Systemd services, as well as kernel settings (variables) changed for hardening the internal system and network from attack. For the first four rows of metrics, lower is better, for the last two rows, higher is better:

Table 2: Navigation OS Direct Metrics

Metric	Initial Install [6]	June 2016	April 2017	May 2018 [7]
(Lower numbers are better)				
Total SetUIDs	45	15	15	15
Total SetGIDs	23	18	19	19
Systemd Services	136	55	58	58
Sys V Services	5	2	2	2
(Higher numbers are better)				
Kernel variable changes	3	11	14	14
Network changes	2	4	4	4

Table 3: Navigation Red Hat Enterprise 7 Security Metrics (Center for Internet Security Benchmarks)

CIS category	May 2015	June 2016	April 2017	May 2018 [6]
Updates, Add. Security	38%	45%	45%	47%
OS	12%	11%	11%	11% (est.)
Special Services	67%	67%	67%	79%
Net Configuration	91%	92%	92%	91%
Log & Audit	50%	55%	55%	57%
System Access	45%	41%	41%	39%
User Account	14%	14%	29%	43%
Tests passed/failed	71/66	87/79	88/78	110/79 [8]

Table 3 tracks more general changes in *types* of configurations, and in order to track such changes over time, the CISscan tool is used. Breaking the OS into areas of concern, these metrics measure how secure such categories are compared to a reference model. For some cases, with the addition of software sets or new features requested by users, these scores went down. While the results are broad and vary by software sets and installed feature, they give the ability to check the aggregate state of security and compare changes over time. This may be invaluable when a software installer decides to reset a host of security settings. Such changes are very apparent in these results, and this tool also proves itself useful in providing aspirational (and safe) examples for academic papers. Relevant CIS categories along with numeric percentages for several system images are given. For these values a higher percentage is better, but it can be nearly impossible, and (mentioned previously) less than useful to reach a 100% score.

4.3. User Access Controls

The controls directly placed on user action in this security model is an area where considerable compromise is necessary in the interplay between system security and user needs. The domain is one where (for the Navigation systems) the need for usability is a greater priority than security. By analogy, consider the use of badge controlled rooms in a building. While such security may be appropriate for certain high security buildings where this type of security is required, it may not be as suitable for a supermarket or a home. Two such systems are in general available for most OS platforms.

4.3.1. Discretionary Access Control

In Linux this user control is based on standard process and file permissions, allowing for access control of critical system files and granting users the ability to manage access to data sets and user files. This mechanism allows for a separation of users and data, with these management controls based on Discretionary Access Control (DAC) mechanisms, such as traditional Unix process and file management. Such controls have been standard on almost all operating systems for some time. Care should be taken to avoid overly restrictive user settings. Unlike the previous areas, it is hard to come up with a general standard for these controls. There might be significant difference in file system and data permissions even for different teams on the same system. Indeed, all that we can recommend here is due diligence in the application of these protections.

4.3.2. Mandatory Access Control

Mandatory Access Control (MAC) is an alternate security mechanism that is available under Linux, referred to as SELinux [29]. Part of a series of research projects out of the United States' National Security Agency, this series of software modules seeks to bring aspects of highly secure operating system designs to the Open Source community [30]. These features, which allow for a second, very granular control over user actions, previously were available only on highly secure and expensive computer designs. These modules are included with the Red Hat Enterprise Linux distribution and are a part of our model image. User access can be granted or revoked to a specific file or process, categorized by role. For example, on a system without SELinux, the critical system password file (/etc/shadow), cannot normally be read, however the time it was last modified and the size can be read by an unprivileged user of the system. Such information, such as when the file was last changed (i.e. the last time a password was modified) could be useful to an attacker. For a machine running SELinux with restrictive settings however that file cannot even be listed by an unprivileged user of such a system: no information about that file will be returned *at all*.

[6] The initial install was not suitable for general use and underwent significant change. It is close but not equivalent to the May 2015 release of Table 3.

[7] It should be noted in both Table 2 and Table 3 (below) that the system contained almost five times as many software packages in the May 2018 release as the initial install due to new feature requests and open source changes.

[8] CISscan was updated and several tests were added and expanded. This category had changed significantly in the new version and the value was derived from the previous test method.

However, as we note, how well the users can use the system is a crucial concern for this area. Unlike the specialized intelligence and military computer systems that utilized Mandatory Access Control initially, users of general purpose engineering machines, like those of the navigation computing environment, typically do not have the same expectations or same security concerns about how such machines should function. The use of such constraints could potentially make the user experience miserable, or simply untenable. Furthermore, familiar applications could fail in unexpected ways. Such controls are also quite difficult to configure in a correct manner to support the Navigation user community. There are few use cases where this level of security restriction for these *User Access Controls* area, has been found useful, in comparison with the application of greater restrictions on the *External Network Access* and *Internal Host Security* subsystems.

A trial run of the SELinux system was undertaken on a limited set of operations workstations to evaluate the utility of these controls. The proposed idea was to apply an iterative process to determine the areas of the system utilized normally by the user base. SELinux was run in permissive mode over the course of several weeks, and the SELinux system was examined to determine those areas where the users would be blocked by the SELinux policy. From these conflicts a model of user activity was built up over several months, which allowed for changes to be made to the SELinux policy rules, or changes to be made to user software and/or user behavior. Over time the sum of these changes helped define *expected* user actions. Areas of the system that users did not use and did not need, would be blocked off from user activity.

In principle this is a good approach. However a formal effort examining this method noted two major difficulties [31] and these difficulties have not changed significantly:

- System analysis and configuration: Getting the computer system and its processes working, and working with the correct permission sets was significantly more technically challenging than expected. Many errors were only solved by trial and error. A considerable amount of time was spent to get the system to a working and stable state.
- User analysis: This process (even on earlier versions of SELinux) was not unusually difficult. However downtime could be required during the analysis iterations, and these iterations could make it difficult to accommodate system and user activity changes (users do not like having to log out of their systems repeatedly). This may be an area that functions better for environments that are more static.

At this time methods of Mandatory Access Control such as SELinux are still being evaluated for use in Navigation. Although it offers much improved control mechanisms, and even allowing the automatic design of user access from user actions, our current examination of the intersection between system security and user utility in this area does not recommend its use at this time.

4.4. Evolution of system security over time

Unlike the previous parts of this section which deal with the issues of creating an initial, secure system state, here we consider the evolution of that state over time. As we observed in our discussion on *security over time* how this state changes is strongly subject to the environments and networks that connect to our system.

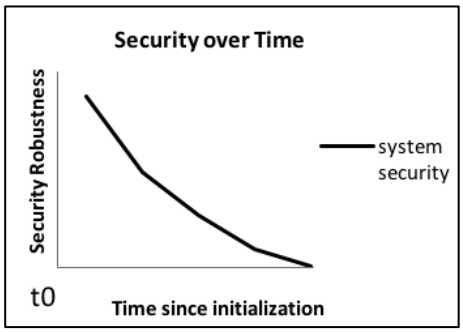

Figure 3: Security over time.

How this confidence in a system changes, and hence the *slope* of the security state in the figure above is very dependent on events in the local environment and the vulnerabilities that emerge over time. It may be nearly linear over time, or an exponential die-off due to newly discovered vulnerabilities. For example, a powered down system on a shelf (or offline storage for a VM) will not change at all. However once connected, many factors will strongly affect the slope of the curve, such as the age of the patching and software set, the security controls, and the environment to which it is connected. This is highly particular to the environmental factors acting on the system: even systems at the same site may have wide variance in this evolution. Understanding this, and remediating such changes is a large part of the maintenance lifecycle of a system. The only clear guidance that can be offered is that, as with a broken password, a system compromise will cause the curve to have a cliff (or stair-step) function in the trust of the system. This will require a system reset or reinstall to resolve.

5. Future Considerations and Conclusions

Considering the evolution of the cyber threat environment and trying to discern what new challenges will arise and how to meet these concerns is a subject of ongoing interest and research for this author. The navigation computational environment conducts critical activity on a 24/7/365 schedule, and cannot afford to be the victim of a security compromise. It is an important system that needs protection. While tips and techniques for security have dramatically changed since this author started as a system administrator, the underlying principles discussed here have changed little in two decades, or not at all. Such archetypes are examined in this paper, and while the particular technical points can change, the foundational ideas will likely remain the same.

One case in point, as noted “de-perimeterisation” approaches like Zero Trust computing may help our approach to computational and network security by encouraging rigorous security at every computational node and network connection. This is a refreshing model which brings to mind some of the most secure intelligence facilities and military fortifications of the last century. It is a vast improvement on the simple, single-defensive layer (sometimes called the hard exterior / soft interior) model which ends up being the most common approach taken in network security. Indeed, we advocate an approach in this paper of being hard on the exterior *and* hard on the interior!

While it is heartening to see such enthusiasm, and much good can come from this, it is important to remember that these efforts (still) need to consider both the complementary intersection of security and user design requirements, as well as the importance of the fault tolerant protection of *all* of the critical computational elements. As we mention above, solving the totality of this defense in breadth problem is a *hard* one. Indeed, not considering these two avenues of concern can lead to situations where the computational environment is perceived by the users as as more akin to something like a prison, or worse still, end up promoting a failure of protection akin to that seen in the previously mentioned Maginot line. Especially in the second case, aggressive efforts as Zero Trust are expensive and challenging to implement and can (from exhaustion of resources in time and effort) lead to the promotion of the failure they sought to prevent! If even a simple approach cannot be executed correctly, attempting to implement a more complex model (like Zero Trust) can be a dubious undertaking.

Another example showcasing this fundamental difference is the spate of attacks based in speculative execution on modern x86 processors (as well as methods to remediate them). These have been of tremendous concern and rapid development for the entire computer industry over the past 12 months [32]. However, the architecture problems at the source of these issues are fundamental design concerns that were identified in their implementation nearly twenty years ago [33].

In addition, a few observations may be derived from the principles espoused above. To the author's mind these should be regarded as obvious and self-evident. Unfortunately, that is often not be the case.

Of these, the most significant is *don't engineer in single points of failure*. For example, after considerable effort to construct a secure, fault tolerant network design, with a hardened firewall and a focused IDS, it is a contradictory effort to then layer a poorly secured Active Directory server for single sign-on for the network (or a poorly configured VPN to access such systems). Such a setup is self-defeating – at best. Fault tolerant and secure design should not be ignored for "just one service". In a similar manner the implementation of a poorly designed security tool designed to examine (or worse, modify) the entire enterprise environment can serve to help implement a large security hole for the entire enterprise. Indeed, as Sami Saydjari relates, there is particular concern with such enterprise security tools as such systems are the highest priority targets of attackers [19, p. 346]. Ease of management of an enterprise network is a nice goal. However, one of the questions that should come up is, "by *whom*?" Single points of failure must be closely, closely watched, as they can be doors that potentially open up the whole network to attack.

This goes with the concept that uniformity of design is not necessarily good. "Balkanization" or the promotion of multiple differing configurations (as long as each of those configurations are secure) promotes design diversity, as discussed in Section 2. While this may not be desirable from a management perspective, such diversity can lead to independent modes of failure, and hence increased security fault tolerance.

Similar to this is the desire to patch often, in the effort to be secure. These two concepts are not, as popularly believed, the same. While this may go against the grain of the currently popular DevOps and Agile software engineering paradigms, it is clear that, absent other testing schemes, it is wise to take the approach of "wait and see" when patching systems.[9] Such updates may introduce bugs, new vulnerabilities, or other unforeseen issues. This is especially true in times of stress after the announcement of major computer security bugs. Large companies can and do make mistakes, and a patch failure that in individual systems may cause pain and irritation could lead to a catastrophic failure or outage if on a critical system.[10]

Finally, one of the bigger trends in current computation lies in the increase in the use of virtual and cloud computing systems. Distributed systems such as these can offer significantly lower costs if (and this is still a big "if") their security and reliability metrics can meet the requirements of their customers. Architecture principles discussed here are (as much as they can be made to be) platform agnostic. One computer security researcher relates the major difference in cloud computing as, "Fundamentally, cloud security is a primary concern due to loss of control ... We've seen this before – [with] outsourcing..." [34, p. 24]. Does the cloud service provider follow fault tolerant and security fault tolerant design principles? Can such questions even be answered by such services?

This loss of control over the configuration for the security environment and for critical events, continues to be problematic for the Navigation use case for cloud computing, especially with concerns over Availability [34, p. 96]. With cloud computing, ownership of the systems and operational processes is by an outside organization. As a part of the operations of our computational environment, it has sometimes taken extraordinary effort to ensure the continuing functioning of our systems in times of stress. This has sometimes meant the difference between mission success and failure. In an emergency, wide latitude is given to operational staff to keep systems running and restore failed systems as fast as possible to meet the needs of flight operations. With an emergency in a remote cloud computing environment, can one even expect to get a responsible engineer on the phone? Without control over that environment, it is obvious we cannot, as Flight Director for NASA Chris Kraft said, ''...take any action necessary for mission success'' [35, p. 392].

It is hoped that the ideas in this paper will provide assistance to administrators and system engineers, and especially the astronautical community. Aspirational goals are presented here, in the hope of providing a guideline to follow for your own design

[9] A major point of defense in depth is that a single failure should not provide the opportunity for compromise on a wide scale. This should be one of the benefits of such an architecture focused approach. Many times a significant vulnerability in our systems crossed our desks with an organizational rallying cry of "patch now!", "patch now!" and we would discover that we had limited to no exposure from the vulnerability because of other defense in depth controls that were in place. Our goal is to be proactive rather than reactive. While we do have emergency patching mechanisms in place, we prefer not to use them unless truly necessary.

[10] It may be instructive to examine the release schedule (and following systems failure reports) for the firmware and software updates for the MELTDOWN/SPECTER vulnerability of 2018. Having a *running* server is in most cases preferable to one that is "currently patched", but "awaiting motherboard replacement."

efforts. These principles can be applied in Ground Data Systems design problems, as well as in other areas of systems engineering. This is particularly valuable in a community of austere budgetary realities where our bespoke systems engineering is based around missions with only a few, or (more often) only a single deliverable.

In an ideal world we would be able to trust our computer systems in the same way that when we drive a car, we trust its brakes. I believe that this optimistic idea is a badly needed one in the design of the increasingly complex and intertwined computer systems that comprise our world today. With the current state of computer security this can appear hopeless, however it is clear, as with the encouraging Boeing flight control system [10], there *are* computer systems that have been designed to be truly fault tolerant. A hard problem is not necessarily an impossible problem. There is no reason we can not also do this in computer security.

Conflict of Interest

The author declares no conflict of interest.

Acknowledgements

To the Systems Engineers and System Administrators who have helped in support of the security architecture discussed here I would like to give my sincere thanks. Additionally, of the editors who assisted with this paper, Zachary Porcu did tremendous service and to him a considerable debt of gratitude is owed. In addition it is important to note that a great deal of appreciation is also due Hal Pomeranz, contributing author of the Center for Internet Security, and senior Fellow of the SANS Institute, who spent valuable time reviewing the ideas in this paper, and raised some insightful questions that helped refine this effort.

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference to any specific commercial product, process, or service by trade name, trademark, manufacturer or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology. © 2019 California Institute of Technology. Government sponsorship acknowledged.

References

[1] This paper is an extension of work originally presented in, R. M. Beswick, "Computer Security as an Engineering Practice: A System Engineering Discussion," IEEE: 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT), 27-29 September, 2017. doi: 10.1109/SMC-IT.2017.18

[2] R. Beswick, P. Antreasian, S. Gillam, Y. H. Hahn, D. Roth and J. and Jones, "Navigation Ground Data System Engineering for the Cassini/Huygens Mission," AIAA 2008-3247, SpaceOps 2008 Conference, Heidelberg, Germany, May 12-16, 2008. doi:10.2514/6.2008-3247

[3] The Netflix Tech Blog, "The Netflix Simian Army," Medium, 19 July 2011. [Online]. URL: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116. [Accessed 3 September 2018].

[4] The Netflix Tech Blog, "Chaos Engineering Upgraded," Medium, 25 September 2015. [Online]. URL: https://medium.com/netflix-techblog/chaos-engineering-upgraded-878d341f15fa. [Accessed 3 September 2018].

[5] C. Carson, Effective FMEA's – Achieving safe, reliable and economical products and processes using Failure Mode and Effects Analysis, Hoboken, NJ: Wiley & Sons, 2012.

[6] B. Johnson, "Fault-Tolerant Microprocessor-Based Systems," *IEEE Micro,* vol. 4, no. 6, IEEE Computer Society Press, Los Alamitos, CA, pp. 6-21, 1984.

[7] J. Gray and D. P. and Siewiorek, "High-Availability Computer Systems," IEEE Computer Society, Los Alamitos, CA, p. 39-48, September 1991. doi: 10.1109/2.84898

[8] D. McCullough, The Great Bridge: The Epic Story of the Building of the Brooklyn Bridge, New York: Simon & Schuster, 2012.

[9] U.S. - Canada Power System Outage Task Force, "Final Report on the August 14th, 2003 Blackout in the United States and Canada – Causes and Recommendations," April 2004. [Online]. URL: https://energy.gov/sites/prod/files/oeprod/DocumentsandMedia/Blackout Final-Web.pdf. [Accessed 25 May 2018].

[10] Y. C. Yeh, "Safety critical avionics for the 777 primary flight controls system," IEEE - Digital Avionics Systems, Daytona Beach, FL, DASC. 20th Conference, October 14-18, 2001. doi: 10.1109/DASC.2001.963311

[11] R. Anderson and R. Needham, "Programming Satan's Computer," *Computer Science Today, Berlin, Springer, Lecture Notes in Computer Science,* vol. 1000, pp. 426-441, 1995.

[12] B. R. Rich, "Clarence Leonard (Kelly) Johnson, 1910-1990, A Biographical Memoir," in *National Academy of Sciences*, Washington, D.C., National Academies Press, p. 231, 1995.

[13] M. Bishop, Computer Security, Art and Science, 2nd. Ed., New York: Addison-Wesley, 2019.

[14] D. Libes, "RFC 1178 – Choosing a name for your computer," August 1990. [Online]. URL: http://www.faqs.org/rfcs/rfc1178.html. [Accessed 25 May 2017].

[15] I. Roundhill, "699-101: Cassini Navigation Plan, JPL D-11621," Jet Propulsion Laboratory, Pasadena, CA, 1 August 2003.

[16] S. Garfinkel, G. Spafford and A. Schwartz, Practical UNIX and Internet Security, 3rd Ed., Sebastopol, CA: O'Reilly and Associates, 2003.

[17] W. R. Cheswick, S. M. Bellovin and A. D. Rubin, Firewalls and Internet Security, Repelling the Wily Hacker, 2nd Ed., New York: Addison-Wesley, 2003.

[18] W. Schwartau, Time Based Security, Seminole, FL: Interpact Press, 1999.

[19] O. S. Saydjari, Engineering Trustworthy Systems, New York: McGraw-Hill Education, 2018.

[20] E. Gilman and D. Barth, Zero Trust Networks, Sebastopol, CA: O'Reilly Media Inc., 2017.

[21] B. Youn and P. Wang, "Complementary Intersection Method for System Reliability Analysis," *ASME Journal Mechanical Design,* vol. 134, no. 4, April 2009. doi: 10.1115/1.3086794

[22] R. Cross, "Center of percussion of hand-held implements," *American Journal of Physics,* vol. 72, no. 5, pp. 622-630, May 2004. doi:10.1119/1.1634965

[23] S. Chiasson and P. C. v. Oorschot, "Quantifying the security advantage of password expiration policies," *Designs, Codes and Cryptography,* vol. 77, Issue 2-3, pp. 401-408, December 2015.doi: 10.1007/s10623-015-0071-9

[24] National Vulnerability Database, "National Checklist Program Repository," National Instituite of Standards and Technology, [Online]. URL: https://nvd.nist.gov/ncp/repository. [Accessed 30 March 2018].

[25] Information Assurance Directorate, " Operating Systems guidance," National Security Agency, [Online]. URL: https://www.iad.gov/iad/library/ia-guidance/security-configuration/operating-systems/index.cfm. [Accessed 20 April 2017].

[26] Center for Internet Security, "CIS - Center for Internet Security," CIS, [Online]. URL: http://www.cisecurity.org. [Accessed 30 March 2018].

[27] NMAP, "Nmap," [Online]. URL: http://www.nmap.org. [Accessed 30 March 2018].

[28] Nessus, "Tenable Security," Tenable Inc, [Online]. URL: http://www.tenable.com/products. [Accessed 30 March 2018].

[29] SELinux, [Online]. URL: http://selinuxproject.org/page/Main_Page. [Accessed 25 May 2018].

[30] NSA, "SELinux," National Security Agency, [Online]. URL: https://www.nsa.gov/what-we-do/research/selinux/.[Accessed 25 May 2018].

[31] R. M. Beswick and D. C. Roth, "A Gilded Cage: Cassini/Huygens Navigation Ground Data System Engineering for Security," AIAA 2012-1267202, SpaceOps 2012 Conference, Stockholm, Sweden, June 11-15, 2012. doi:10.2514/6.2012-1267202

[32] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, "Spectre Attacks: Exploting Speculative Execution," *arXiv.org:1801.01203,*3 January 2018.

[33] Z. Wang and R. B. Lee, "Covert and Side Channels due to Processor Architecture," *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06),* 11-15 Dec. 2006. doi:10.1109/ACSAC.2006.20

[34] D. Shackleford, "Security 524: Cloud Security Fundamentals - Day 1," in *[seminar], SANS National Conference*, Orlando, FL, April 7th, 2017.

[35] G. Kranz, in *Failure Is Not an Option: Mission Control From Mercury to Apollo 13 and Beyond*, New York, Simon & Schuster, 2009.

[36] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical Journal,* vol. 28, no. 4, pp. 656-715, 1949. doi:10.1002/j.1538-7305.1949.tb00928.x