Interested in learning more about security?

SANS Institute InfoSec Reading Room

Framework for Secure Application Design and Development

The practice of secure application design and development is an important and necessary attribute of a secure computing environment. Applications that protect data from unauthorized access or modification and ensure its availability are key advantages to companies with physical and information assets that require such an environment. But, as crucial as this practice is, applications often are not being designed and developed with security in mind. As such, these applications are contributing to the burgeoning miasma of...

Framework for Secure Application Design and Development
Foundation, Principles and Design Guidelines
GIAC GSEC Practical Assignment Version 1.4
Chris McCown
November 12, 2002

Abstract

The practice of secure application design and development is an important and necessary attribute of a secure computing environment. Applications that protect data from unauthorized access or modification and ensure its availability are key advantages to companies with physical and information assets that require such an environment. But, as crucial as this practice is, applications often are *not* being designed and developed with security in mind. As such, these applications are contributing to the burgeoning miasma of potentially insecure interconnected systems providing an open door to the possible compromise of companies' sensitive and valuable information or systems.

In order to produce robust and secure applications that enable and promote a secure computing environment, developers must know and adhere to some fundamental tenets of security doctrine. The goal of this paper in one reference work is to:

- Illustrate the importance of secure application development.
- Provide background/history on why this practice is not as pervasive as it should be today.
- Present a *framework* to assist developers in the practice of secure application design and development.

Introduction

Why Secure Apps Are Important

Companies have information assets. In order to maintain and cultivate a competitive advantage, those assets must be shared intelligently with customers, business partners, and employees. It follows that those assets must be protected from threats that could cause financial loss or other harm to the company. Examples of such loss can be disclosure of trade secrets, damaged reputation, decreased customer confidence, etc.

The purpose of computer security is to contribute to the mission of the organization by protecting those assets through the selection and application of appropriate safeguards. [1] Successful companies deploy a computer security strategy known as Defense-in-Depth, a "layered" approach that relies on people, operations, and intelligent application of multiple techniques and technologies to achieve the desired level of *information assurance*[i]. [2] By deploying the effective safeguards intelligently, companies are able

to manage risk effectively by reducing vulnerability to threats, ultimately lowering the probably of compromise and financial consequences.

In order to share information assets, companies rely upon computing services and applications. These services and applications are generally a mixture of *COTS*[ii] software and custom applications that provide solutions to meet business goals. These custom applications are often developed in-house and are increasingly interconnected with companies' intranets and the Internet. While this creates and fosters business opportunity, it also means these applications are susceptible to compromise!

The practice of secure application design and development is clearly a vital component of a strategy to ensure a secure computing environment. Applications that are security "enabled" or "aware" contribute to the defense of that environment, and ultimately the mission of the company.

A Paucity of Secure Apps

If developing secure applications is so important, why aren't there more secure applications? When presented with a case for secure application development, many developers understand the value and need to write secure code and program defensively. In fact, most see it as a responsibility. The trouble is that many developers don't know how or where to begin with regard to security. Why? A host of factors in higher education, public and private sector, professional and market environments have influences—or lack thereof on the developer. As we'll see, in general developers haven't been given the training, tools, resources or time, or in many cases the motivation to meet secure coding objectives.

The State of Things

Diversion: A Little History Lesson

The evolution of mechanical/electronic computing could be said to have begun around five thousand years ago with the invention of the abacus in Babylonia. Some four thousand, six hundred years and some change later, people became seriously tired with doing math by hand. The first computing machines were invented to *grind*[iii] numbers. Computers evolved slowly over the next couple hundred years with discoveries in the nature of crystals and current, becoming larger and more complex but basically still just crunching numbers. It wasn't until 1945 that the concept of a "stored program" was first introduced by John von Neumann[iv]. Programming became a reality in 1948, when the first computer program was run on an electronic computer at the University of Manchester, England[v]. Its purpose was to compute highest common factors—still crunching numbers, but doing it in style. From that point on, programs and programming languages began their own evolution.

Of course, the state of things became a whole lot more interesting from the security standpoint when engineers at Xerox Parc frantically chased down the first "errant

program" that ran amok on their network in 1979. Developed by John Shoch and Jon Hupp, the program was developed as an exercise in leveraging idle computing resources on a network. The basic concept was to have a program that spanned machines by replicating itself to look for idle machines. The segmented nature of the program gave rise to a term used to describe its functionality: the first "worm" was born[vi].

The Developers Lot

A new breed is born. As machines were designed to solve increasingly more complex problems, *programmers*—people who wrote code, or structured programs to solve those problems came into existence. As machines evolved to have multifunctional capabilities, writing that code became a sort of arcane art, and *programmers* took on additional attributes: not only did they write code, they began to assemble and reuse code to accomplish even greater tasks. At some point, programming became as much an art as a science. The terms *developer* and *computer* rolled into the vernacular.

Today, developers are expected to work their "magic", pounding out millions of lines of code that do everything from instructing your equipment to record *Sex In The City* to enabling billion-dollar e-Commerce transactions to running power plants. The code is often so complex that it cannot be comprehended any one developer or cadres of developers. And therein lies a BIG problem: how can we be sure that all that code is secure—that no untoward souls with fiendish designs can get in and wreak havoc in that code for fun, profit or otherwise? Are developers ready and able to ply their trade defensively?

The State of Things - Part 2: The Hapless Developer

As mentioned, many factors have influence or no influence as the case may be on developers and the development of secure applications. Developers are faced with many challenges in gaining relevant knowledge, skills, data and best practices from higher education, public and private sector and professional organizations. It's not an easy puzzle to piece together.

Education: A Missing Link

Education and vocational institutions have only recently begun to focus on computer security as a discipline or program. As recently as 1996, there were only four dedicated computer security research programs at degree-granting university departments in the United States. [3] And at the time, there were relatively few textbooks on the subject, many of which were devoted to cryptography. To wit, college graduates finding work as code jockeys likely have the skills to write programs that meet objectives and have cool features, but are generally armed with little knowledge how to do it securely. A "killer" piece of "tight" code might throw a jazzy GUI up that grabs some user input, queries a database and returns results with multiple choice options in detachable drop-down lists—in 3-D and 16 million colors! That same piece of code may also allow

unauthorized direct access to the back-end data, or allow an attacker to execute *code of choice*[vii] on the system through poor memory management (i.e. buffer overflow and stack manipulation). Ouch.

Professional Orgs & Societies: Connect-the-Dots

Professional organizations and societies dedicated to the varied and evolving aspects of computer information security have existed in some form throughout the evolution of computers and networks. Organizations such as SANS, CSI, NSI, IEEE, IETF, and CERT to name a few, have been a wellspring of information security systems, methodology, best practices, tools, and more recently advanced training in target areas of significance in today's global computing environment.

While so much security related information is published and available, it is difficult at best to find the right resources for secure application development. Often, developers must jump from org to org and glean or string together methods and best practices that have anything to do with secure development that are relevant to the developers' tasks. Membership in these organizations and the associated training costs are sometimes cost-prohibitive in the eyes of managers looking to "save" money (more on this later).

The Government: We're Here to Help

The United States Government and organizations that support it also offer up volumes of unclassified security-related information through a variety of organizations such the NSA, NIPC, DOE-CIAC, and especially the NIST CRC.

As with professional organizations, it can be difficult and frustrating to sift through the voluminous resources to find content relevant to secure application development. And unfortunately, in this day and age, there's a general feeling of distrust and a lack of confidence in government in many sectors. What developer actually thinks about the government as a resource right offhand?

Online Resources: The Proverbial Needle

A universe of information provided by educational institutions, professional and government organizations and companies can be found online, but with a major drawback: the majority of material readily available on the web focuses primarily on network security. Oodles of information are at the fingertips for virus detection and removal, firewalls, IDS, system hardening, etc. Most of this can be boiled down and categorized into threats, vulnerabilities, incidents, response and fixes, with emphasis on hacking/anti-hacking, countermeasures and information warfare. These are the "glamorous", headline-grabbing topics that no news writer will get dinged for writing about.

As an exercise to illustrate the difficulty in locating germane info on secure coding practice, try typing the following (and variations) into a favorite search engine:

The results will be many; the relevance to actual secure coding practices very subjective; the references to industry or professional standardized methods and best practices almost non-existent…! (Author's Note: Perhaps this is why papers of this nature are beginning to proliferate.)

The Lifecycle Systems: Security is an Afterthought

With or without the skill sets, tools and resources necessary to develop secure applications, another set of roadblocks can occur within companies themselves. Various "program" and "project" lifecycle methodologies exist to facilitate the evolution of a program or project from inception to EOL[viii] in a consistent and standardized fashion. Managed software design and development is also governed by similar methodologies, and is often a component of the program or project lifecycle. Program/project lifecycle administration is generally overseen or administered by a team responsible for delivering a quality product on time and under budget—often mutually exclusive goals!

One big problem in terms of security is that unless members of the team have a security background or are accountable for it, security considerations are usually absent from the equation. Consequently, security is not considered during the appropriate phases of the lifecycle—i.e. right up front in *exploration* and *planning* all the way through to *design* and *development*. Often companies will jump directly from the determination of a business driver or goal directly to writing code—foolhardy at best even without regard for security. Systems that are designed according to requirements are better than systems that are "thrown together". And with regard to security, it is always preferable to design security in from the beginning, rather than bolting it on afterwards. [4] Developing a whiz-bang application that does unspeakably cool things with the company data without at least minimal security *requirements*, a *functional specification*, and a *design specification* before development begins amounts to building a house without blueprints. Oh, and extra locks and an alarm system were added after the house was built. Never mind the fact that the window frames weren't screwed in…or the door hinges were installed on the outside… It's also worth mentioning that everyone knows where the house is.

Another problem lending to the absence of security considerations in the lifecycle is that it is unless you're a company making money on security products security is not a revenue stream. Thus, it is often deemed secondary—or worse, not important by Management. In another typical scenario, Management may recognize the importance of security, but in order to keep the budget in line and the shareholders happy and smiling, might only spend money on technologies and products and not on the appropriate training for the team on how to use, configure and deploy this "hot new stuff". What Management often misses is that while security doesn't generate revenue, it can be considered an insurance policy against threats to the company's information

assets. The cost of security is directly related to the amount of perceived risk. [5] This forces the company to evaluate the worth of its assets, and choosing the right level of insurance requires a risk assessment as well. This exercise produces measurable value to the company. Unfortunately, risk assessments are also often overlooked as less important than delivering the product.

Compilers & Interpreters: Friends or Accomplices?

Development tool vendors have succeeded in producing exceptional tools in the form of programming languages, compilers and interpreters for developing applications of all shapes, sizes and functions. But along the way, many of these did not provide the safest ways for programmers to use or manage system resources.

The earliest of programming involved raw *machine language*—so named because you were entering instructions to the computer in a format the machine itself understood. Second-generation programming was through *assembly language*, which allowed for the use of *mnemonics*, or "*op codes*" which were more easily understood by humans—but only slightly! The *op codes* were interpreted and compiled into *machine code* to be executed. Not a lot of room for compromise here. In general, a bad register shift or interrupt call would produce a core dump or knock over the machine. Kaput—nothing left to access.

Third-generation languages (3GL) were developed to be much easier for humans to read and use, but with a price. The abstraction of the actual operation of the code through high-level APIs and functions added layers between the programmer and the operating systems. This had the effect of essentially masked the underlying interactions from all but the most technical or experienced programmer. Coding with these languages became a bit like driving a car by watching a video of the road on the dashboard. And, 3GL's have some great features like pointers and references—direct and indirect addressing of memory, which easily and quite conveniently allowed developers to shoot themselves in the foot[ix]. The earlier versions of these compilers were fairly liberal in allowing for direct memory access and calling any conceivable address whether it was in or out of range or bounds or whatever. One bad *malloc* or forgotten *delete object* could leave your program in a bad way, perhaps even allowing itself to be overwritten by code introduced to the program by an external source (attacker).

This is important. As machines and operating systems have evolved, they have been designed with a certain amount of *survivability* built in. In addition to operating systems that could isolate and recover from mild to moderately serious errors, languages and compilers began to offer developers ways to recover from errors or *exceptions*, and assist in the continued operation of a program to keep it running in spite of the errors that have occurred. This has actually become an integral method of programming called *exception handling*. But here's where things get interesting. If that code can be compromised, then potentially any code can be run—and, if all the other programs on the machine are still running, nearly anything might be accomplished, including

transmitting any available data off or out of the system into the waiting arms of less-than-honorable parties.

Today's compilers and interpreters do a much better job of memory management, and even prevent some common insecure programming practices through judicious use of options and settings. But in terms of security, we still have a ways to go. Take C# for example. It offers up something dubious called "unsafe" mode. Can't beat that with a stick! (Should.)

The Developers Quandary

So where does all this leave the developer? The odds of developing secure applications appear stacked in favor of the odds-makers! In most cases, the burden of implementing application security winds up becoming the job of the developers however and whenever they can squeeze it in, if at all. And as noted, developers may not have the education, professional resources, skills or tools to implement the necessary measures needed to contribute to the protection of the company's information assets.

Plainly, security at any level cannot be ignored. The 2002 CSI/FBI Computer Crime and Security Survey, an annual survey of the scope of computer crime incidents reported across the spectrum of government, private sector companies and education, reports that the overall trend of computer breaches and resulting financial loss has increased for the seventh year in a row. Theft of proprietary information is one of the most serious contributors to financial loss over the past few years, and it continues to mount. [6]

In order to contribute to the defense of companies' information assets, developers must develop secure applications. Virus, firewalls, IDS protection measures can't be relied upon as the only measures to protect the data. Any compromise of any of these measures can open a direct path to the applications and data—i.e. the company jewels. Secure applications are a must to provide another layer of defense against such compromise and the ultimate consequences of exposing sensitive data.

Help for the Hapless but Determined Developer

In the absence of some/any/all positive influences, developers do stand a chance of developing secure applications. How is this possible?

Answer: By adhering to a common framework that encompasses and adheres to basic tenets of information security. The proposed framework includes:

- **Foundation** – The basics; what you need to know before writing a single line of code.
- **Principals** – Fundamental rules to be followed when writing the code.
- **Design Guidelines** – Best practices with bite. Proven and successful methods for implementing the code.

The benefits of a common framework are manifest. The proposed framework:

- Arms developers with a basic reference—bedrock for developing secure code to protect information.
- Provides universal time-tested principals and design guidelines to follow in the development of that code.
- Promotes understanding and comprehension of security policy, programming language and tools.
- Provides a documented framework that can be used as a basis for influencing company development efforts, best methods, standards and procedures, and security policy.
- Raises visibility to management that security risks are being considered and efforts made to mitigate them at the application level in enterprise[x].
- And more.

What the proposed framework is not/does not do:

- This framework is not a replacement for a company's development or security policies and standards.
- This framework is not a soapbox from which to promulgate security righteousness.
- This framework is by no means an all-inclusive security guidebook. It is a synopsis of best practices and time-tested knowledge.
- This framework does not instruct the developer on how or what code to write. This skill set already exists. Rather, the framework serves to impress upon the developer *what should/must be considered* in order to write that code securely. It is then the developer's responsibility to put those considerations into practice.

These Foundation, Principles and Design Guidelines can serve as a core—a common reference to be used by developers as a foundation for any development work requiring any level of security to protect information assets. The developer is encouraged to internalize their importance, commit to their support and propagation, and code to the best of his/her ability within and extending their bounds.

Framework for Secure Application Development

What: A common framework including **Foundation**, **Principles** and **Design Guidelines** encompassing and adhering to basic tenets of information security for the development of secure applications.

Why: Secure applications protect information assets and contribute to the defense of the company's computing environment and the mission of the company.

Framework Outline

Foundation

The basics; what you need to know before writing a single line of code.

- Know your Security Policy, Standards, Guidelines and Procedures
- Know your Development Methodology
- Know your Programming Language and Compiler/Interpreter

Principles

Fundamental rules to be followed when writing the code.

- Security is part of the design.
- Assume a hostile environment.
- Use open standards.
- Minimize and protect system elements to be trusted.
- Protect data at the source.
- Limit access to need-to-know.
- Authenticate.
- Do not subvert in-place security protections.
- Fail Securely.
- Log. Monitor. Audit.
- Accurate system date and time.

Design Guidelines

Some basic best practices with bite. Proven and successful methods for implementing the code.

- Input Validation
- Exception handling
- Cryptography
- Random Numbers
- Canonical Representation

Body of the Framework

Foundation

The basics; what you need to know before writing a single line of code.

Know your Security Policy, Standards, Guidelines and Procedures

Every company must have a Security Policy. Security Policy refers to the high-level overall plan representing computer security decisions made by the company in respect to protecting the company's systems and information assets. This policy is developed through discussion and difficult decisions involving resource allocation, competing

objectives, and organizational strategy related to protecting information resources and guiding employee behavior. Effective policies ultimately promote healthy and successful computer security programs and protection of systems and information assets. Some companies provide a security Code of Ethics that provides additional guidance to promote assuming responsibility for security issues and "doing the right thing".

By nature, policy is written at broad level, generally describing behavioral expectations. Companies develop standards, guidelines and procedures to provide a clear picture of the "what" and "how" associated with that policy to meet organizational goals. Standards provide specific minimum requirements (what to do); guidelines provide guidance where standards may not always be achievable but should not be overlooked (what can or should be done); procedures provide approved methods for meeting the standards or guidelines (how to do it).

Companies may promote and publish policy, standards, guidelines and procedures on or through various mediums, for example: security manuals, handbooks, websites, videos, training courses, incentive programs, etc. Each employee is personally responsible for learning and adhering to these to these to safeguard the company's information assets. For developers, this also means understanding how they are applied to the practice of application design and development. Many companies clearly call out practices that are relevant to the developer's role and responsibilities, but some may not. In the unfortunate case of the latter, it is up to the developer to determine what is significant and applicable.

Action Items:

- Learn the company Security Policy.
- Learn the company Security Standards, Guidelines and Procedures.
- Be compliant; adhere to them.
- Relate the policy, standards, guidelines and procedures to every day work; think about how these promote security and if they are successful or not.
- Provide feedback—both positive and constructive as it relates to application design and development.
- Commit to the Code of Ethics. If the company does not have one, seek one out that compliments the company's methodology and contributes to the mission of the company.[xi]

Know your Development Methodology

In developing applications, successful companies use some form of a software development life cycle, or SDLC. An SDLC is a framework generally adhered to by systems analysts, software engineers, and programmers for developing software applications to ensure they meet business requirements. Frameworks generally consist of multiple defined phases that include activities that are germane to each phase. Common phases include Planning, Analysis, Design, Build/test, Implementation Maintenance and EOL. A computer security plan and awareness spans the entire life cycle. Developers are involved primarily in the Design and Built/test phases. All

phases are important, but it is here that fundamental security principals and design guidelines fall primarily within these phases.

Action items:

- General
  - Learn the company SDLC and how it relates to the project.
  - Prepare for each phase/step, knowing which security principles, guidelines and technologies are appropriate.
- Design
  - Know the development process specified in the SDLC—i.e. is it Waterfall or Linear? Iterative? Etc.
  - Know the security technologies, tools, methods that will be used; understand the ramifications.
- Build/test
  - Keep the SDLC objectives in mind during coding.
  - Adhere to security Principles and Design Guidelines during coding
  - Keep status vs. objectives; note the impact of each principle and design guideline for future evaluation.
  - Perform code reviews and walk-throughs to validate code/objectives.
  - Test for vulnerabilities.
- Implementation
  - Do not expose any information assets during the implementation; it should be as transparent as possible, affecting no other systems' security.
- Maintenance
  - Monitor the application and users' activity for adherence to security policy, enforce by the security design.
  - If a security gap is discovered, adhere to principles and design guidelines when make changes.
  - Do not bolt on security measures; measures must be integrated—even if it means redesign.
- EOL
  - If applicable, dispose of applications and information assets in a secure manner.

Know your Programming Language and Compiler/Interpreter

The language and compiler/interpreter are the tools of the programmer. Like any artisan in a skilled trade, a developer must know the in's and out's of the tools of the trade. In general, programming language code is compiled to assembly code for execution on a specific computing platform. A compiler converts all the code it has been given into assembly language code (instructions to the machine) according a set of rules, regardless of what the code actually does. It does what it's told, safe and secure or not. In order to achieve secure software, the code must be safe to start with. Fortunately, a sort of silver lining has emerged in recent years to aid developers.

Most current compilers provide at least a small feature set that promotes some form of secure programming practice. Some compilers allow for extensions (plug-in like capabilities) that can assist in checking code for security errors. Some even embed code to check for security breaches like stack-smashing[xii] and raise an alert during operation. Some tie compile switches directly to security checking. Set one of these switches, and the compiler will search for common programming errors for security. These can be generic, geared toward the application such type checking, range checking, correct object allocation/deallocation, etc., or more specific to the nuances of the actual platform. If the compiler finds such errors, they are noted by the compiler for the developer to correct.

It should be noted that this new compiler feature set is not a panacea or "silver bullet", and does not absolve the developer from good programming practice. These feature sets are additional complimentary functions to aid the design and development of secure applications.

Action items:

- Learn the security in's and out's of the language, compiler/interpreter of choice.
- Know what security features each provide—e.g. is the compiler type-safe? Does is allow for extensions? Etc.
- Know the compiler options/switches that affect security. Which can check:
    - Stack checking (buffer overflows)
    - Memory/pointer management
    - Type-checking
    - Range-checking
    - Boundary conditions
    - Run-time checks
- Know how to force these types of errors. (The knowledge of how an error occurs provides the ability to prevent it from occurring.)

Addendum to Action items—Resources. There are vast resources available

- Vendor Training (often with an associated cost)
- Product documentation (always purchase at least one complete set of manuals)
- Online resources (e.g. technical articles, websites, newsgroups, etc.)
- Third-party Books and How-to's

And remember: compilers have security flaws too, so stay frosty!

Principles

Fundamental rules to be followed when writing the code.

Security is part of the design.

Security must be considered in application design and development. It is very difficult to implement security measures properly and successfully after an application system has been developed. It is a feature that simply cannot be bolted on after development is

completed. It has been estimated that the cost to make a fix after the fact can be as high as 50 to 200 times more expensive. [7]

Security compares to other system-wide design and development attributes that require advance planning such as functionality, dependability, and reliability. Successful software design and development is achieved through a well-structured process of requirements, detailed specification, design and development, and implementation. But as we've seen, this does not always occur for a variety of reasons. Even in the absence of this process, developers can do their part to design airtight code from the beginning of the development process through design and implementation. Airtight code from beginning to end leaving no holes is the goal. The penetrate-and-patch approach is risky and potentially costly because attackers have a nasty habit of exploiting holes in software. Know what is necessary beforehand to incorporate security.

Action items:

- Know the value of what is being protected.
  - What is the worth of the data to the company?
  - How is the worth measured? (e.g. brand name, reputation, lost production, etc.)
- Know the threats.
  - Understand categories of threats such as network traffic monitoring (*passive*), exploits of vulnerabilities (*active*), tampering or destruction by gaining access (*close-in*), or through authorized access (*direct*), can compromise the application or data.
  - Understand how Mal-ware (i.e. viruses, Trojans, worms, and other "malicious" software) can compromise the application or data.
- Know the vulnerabilities of in-place and planned protection measures.
  - Leverage vendor support and technical info.
  - Leverage vulnerability databases such as BugTraq[xiii] and CVE[xiv].
- Know the consequences of being compromised. If the data is disclosed, tampered w/ or destroyed:
  - What will stop working?
  - What will it cost the company?
  - What will it mean to competitors?
- Know your enemy:
  - Why would an attacker want the data? (e.g. challenge, disruption, sabotage, theft, espionage, political agenda, etc.)
  - What skills would an attacker need to get the data? (i.e. does the attacker need be a true *cracker*[xv] or just a *script-kiddy*[xvi] to do the deed?)
- Know your limits. (a Zen-like moment)
  - The ability to recognize what one knows and does not know, or the level of one's own competence is crucial in secure software design and development. Consider the adage, "The more we learn, the less we know", and apply its meaning. Without the impetus to question and explore and pursue objective analysis, it is dangerous to assume that everything is known and secure.

    - If it's not clear, don't "take a stab at it". Assume responsibility for it: Stop, ask and learn.
    - Frequently compare knowledge to industry best practices.
- Develop code to protect the assets accordingly.
- Subject code to risk analysis and testing (e.g. vulnerability assessment).
- Remember, it's not over 'til the code EOL's.

Consider these factors before writing a single line of code. Although this is a veritable laundry list of what-to-know that can be daunting to internalize, the message is quite simple: think about security like any other system attribute, and understand as much as possible before design and development begin.

Assume a hostile environment.

"In general, whoever occupies the battleground first and awaits the enemy will be at ease; whoever occupies the battleground afterward and must race to the conflict will be fatigued."[xvii]
–Sun Tzu [8]

Applications function in an "information domain"—a group of information resources partitioned according to requirements such as business need, access control, and levels of protection required. Organizations implement security measures to protect the authorized flow of information between information domains. An external domain to the developer is one that is not controlled by the application. Developers should presume the security measures of an external system are not the same as those that influence the application. Given the growing trend of interconnectedness, developers should assume that an application may one day run in an environment other than what it was designed to run in and design security appropriately.

If an application is designed to interact with the Internet, developers absolutely must be prepared for anything. The "World Wild Web" is not a place for insecure e-commerce applications or otherwise. Insecure systems or applications can be compromised within hours of being made available. For example, statistics from The Honeynet Project indicate that the estimated life expectancy of a default installation of Red Hat 6.2 server fired up on the internet to be less than 72 hours—with the quickest recorded compromise occurring in 15 minutes, meaning the system was scanned, probed, and exploited within 15 minutes of connecting to the Internet! [9]

Action items:

- Until proven otherwise, all external input should be distrusted.
- Until an external system has been deemed "trusted", presume it is not trustworthy.
- Assume an application designed in one environment (ex. intranet) may not be secure in another (ex. internet)
- Assume the company firewalls and virus software will not protect the application and data. (Remember these too are software and could be flawed!)

- Don't assume the application will always communicate with trusted front-ends—these may be compromised.
- Client-side security does not exist.
- Assume that an attacker knows everything you know, and more.
- Take the time to understand the *motives* and *means* of attackers. These cannot be affected. What can be controlled is the *opportunity*—deny it.

Use open standards.

In distributed computing environments, it is critical to implement/leverage open and trusted (time and industry tested) standards to achieve interoperability, portability and continued viability of security measures. Untested and proprietary security solutions add unnecessary complexity to the computing environment. Such complexity could allow malicious or flawed subsystems to remain invisible until compromise occurs. [10]

Action items:

- Avoid homegrown cryptography; do not "roll your own".
- Do not rely on other companies' proprietary solutions.
- Use open-source or standard libraries.
- Check the implementation of standards by code reviews and comparison to industry best practice.
- Join an organization that promulgates and promotes industry standards.

Minimize and protect trusted system elements.

Applications should be designed such that a minimum number of system elements are employed in order to provide and maintain appropriate protection. These elements should provide the most secure functions that foster a known secure state. To maintain that state as best as possible, external modification should be minimized or disallowed. It is important to reduce the number of components/systems that provide security to embrace simplicity and avoid unmanageable complexity ("Keep it simple"). Simple and self-contained systems are much easier to protect themselves, and, complexity can allow flawed or insecure systems to go unnoticed until those systems are compromised.

Action items:

- Consider: If how it all works together can't be explained simply, it's too complex.
- Leverage built-in protections afforded by operating systems and the network.
- Use choke points to minimize and control access to data.
- Reuse trusted components.
- Limit the use of extensibility and *mobile code*[xviii]—the ability of an application to be "taught" or evolved to perform new functions by the user. If not closely guarded and scrutinized, this is a potential avenue for the introduction of malicious code into a system that could create insecurities or wreak havoc.

Protect data in process, transit and storage.

Information assets require protection wherever they exist. The risks of unauthorized modification or destruction of data, disclosure, or denial of access to data must be considered at all times. Developers should implement security measures to preserve, as needed, the integrity, confidentiality, and availability of data and the application software during process, transmission and source/storage.

Action items:

- Understand the value of the data to be protected.
- Understand the protections afforded by the operating system and network, native database and web security services
- Know the standard and approved tools used by the company.

In process

- Run applications and processes under required privilege only.
- Do not allow users and other processes unfettered access to application via open and listening ports.
- Implement authentication and authorization to access data (role-based).

In transmission

- Implement secure communications paths between source and destination via encryption tools/protocols (e.g. SSH, SSL, VPN)

In source/storage

- Store data in files protected by strict permissions so that users and other process cannot reach them outside of the application.
- Encrypt data in storage if required.

A common mantra is "protect data at the source." This essentially means that security controls should be placed as close to the resource as feasible and intelligent to do so.

Limit access to need-to-know.

Enforce the concepts of *least privilege* and *default deny*. Restrict user and process access according to business needs and acceptable business risk. Provide only the necessary authorizations to perform required functions. Unless the permission is granted explicitly, users and processes should not be able to access protected resources.

Action items:

- Identify data and resources that need to be accessed.
- Identify functions that need to be performed.
- Implement role-based access for the data, resources and functions.
- Assign each role only those permissions needed to access data or resources and perform necessary functions.
- Do not disclose data without access control.
- Do not disclose security measures used.

Authenticate.

For applications, authentication is the process of establishing the validity of a user or process so that it may carry out some action. The intent of authentication is to ensure that security is not compromised by an untrusted source. For most applications, identifying and authenticating logins is the first line of defense. It is essential to provide authentication for applications in order to implement access control and provide accountability. Access control, in terms of least privilege for example, is granting only those permissions needed to access data or resources and perform necessary actions. Accountability requires the linking of those actions to specific users or processes. Additionally, strong accountability provides for a measure of nonrepudiation—the ability to prove that a user or process did in fact perform an action, usually an illegal activity since this generally intended to provide proof of submission, reception and non-tampering of data[xix].

Action items:

- Identify the type of authentication required for the access to be granted (e.g. some may be anonymous—still requiring mapping to an id and logging, while others are strict and require detailed auditing such as in e-Commerce or payment applications).
- Implement unique identities to represent users or processes; avoid shared or group accounts that do not identify individual users or processes.
- Require a user or process to log in with a password.
- Enforce a strong password policy.
- Use/reuse available pass-through authentication and trusted components.
- Prevent unauthorized users from masquerading as an authorized user; do not allow aliasing to another user or process.

Do not subvert in-place security protections.

A company's security policy is generally developed to administer the protection of the people, process, and information assets that give that company its competitive advantage. Security controls are put in place to reduce or mitigate potential risks. With any luck, a balance has been struck between reducing risk and the associated costs and any decrease in operational effectiveness those protection measures render. In any case, the measures should be tailored to the company's unique needs, providing protection where it is deemed necessary/critical.

Action items:
Do's

- Understand the importance of the company's information assets.
- Be aware of the company's in-place and available security measures (e.g. authentication, encryption, abstraction, firewalls)
- Follow all company security guidelines for access to data.

Avoids/Don'ts

- Avoid running under elevated privileges (i.e. a level above the application's intended access and functionality).
- Avoid using "live" or production data for testing/debugging.
- Do not allow Everyone/World to access data without considering CIA[xx].
- Do not allow direct access to databases; use views and stored procedures/triggers permitted by role.
- Do not allow direct access to operating systems commands.
- Do not allow direct access to application configuration information or metadata (e.g. through special commands, shares, URLs).
- Do not enable back doors in software.
- Do not poke holes in firewalls.
- Do not operate without security measures!

Fail Securely.

Bugs happen; plan for failure. Design applications such that in the event of failure, that failure is a known state and secure. The definition of "failing securely" is dependent on the acceptable business risk and requirements for continued functioning. Applications should have a secure initial state that reflects these risks and requirements, and a well-defined (and automated) path to either a secure failed state, or a recovery procedure to the known secure state.

Action items:

- Design in secure defaults.
- Reject all requests for access to the data after a failure; allow no re-authentication.
- Don't "blab-vertise" details describing the application architecture, code, configuration or data in an error message.
- Do not expose/disclose data that would not otherwise be available.
- Save state information to support troubleshooting/investigation.

Log. Monitor. Audit.

"The price of freedom is eternal vigilance."
–Thomas Jefferson

Applications should be designed to include logging mechanisms that can be monitored and audited both manually and automatically. The primary reasons for this is to allow for the detection of unauthorized use and to support incident investigations. [11] The ability to identify misused resources and reconstruct events in an application or system provides traceability and accountability and can assist in identifying vulnerabilities that could ultimately lead to compromise. There should be sufficient data captured to identify who or what did what and when.

Action items:

- Determine audit requirements (e.g. legal, security, monitoring, etc.).
- Identify the events that need to be logged and for what purpose (e.g. logins, resource access, changes, etc.).
- Identify the appropriate level of audit information required to log (e.g. time of event, process or owner, success or failure, etc.)
- Only allow appends to log file; do not allow deletions.
- Maintain and store logs as required for audit and legal purposes (i.e. encrypt, store centrally, write once to CD-R to safeguard if necessary).
- Review audit logs regularly for unauthorized use.
- Review process regularly to ensure logging is functioning as expected.
- Use and state up front consent to monitor policy if applicable.

Accurate system date and time.

Accurate system date and time are essential to all security functions. Program execution, logging and monitoring often include and depend on date and time functions for synchronization and time-stamping[xxi]. Many enterprises depend on a complex, highly automated system of time-dependent program interactions due to business or industry requirements. In terms of security, audit accountability and event reconstruction are dependent on placing events sequentially and accurately by date and time to support incident investigation and gathering of forensic evidence.

Action items:

- Synchronize time server to UTC with standard clocks (NIST, U.S. Naval Observatory)[xxii]
- Employ network time synchronization tools/protocol.
- Protect access/availability to central time server.

Design Guidelines

Some basic best practices with bite; proven and successful methods for implementing the code.

Input Validation

Feeling good about some input code? Try banging the old elbows on the keyboard or letting the family pet walk across it to test input handling. This isn't quite the same as penetration testing, but it can certainly help to highlight the capability of an application to accept bad input. Serious attackers will try all manner of input at any prompt to scan for exploits.

The Problem

Accepting any input from users or processes (or pets) without some form of checking or validation is dangerous. Attackers know how to circumvent and exploit systems without protections to perform such actions as alter program execution, execute code of choice

or gain access to sensitive data. In general, poor programming practice is the culprit. Many developers don't do any type of checking to prevent common exploits.

- Static and heap buffer overruns where the data input is larger than the allocated buffer.
- Array indexing errors are less common than static and heap overruns, but allow the same exploits. This mechanism to achieve this is through a lack of bounds checking.
- String errors or overflows are generally caused by unanticipated special characters, not checking for the format or number of arguments passed as input.

The Solution

Applications must be designed to only accept input in the expected format, and reject all other input. Writing robust code to achieve this is not possible 100% of the time, but employing best practices will ensure a better chance of protecting the data the application is designed to serve up. The key to validating input is to evaluate it prior to passing it to parsing methods and program operation. The idea is that if bad input is passed to the application without inspection, it is too late; the application (language code) will blindly act on the data, whatever it is and whatever might happen. The input must first be filtered to assess whether or not it matches expected input. If it does not, appropriate responses might be to discard it, log the attempt, raise an exception, etc.

Best Practices

General

- Never trust incoming data; don't assume it is in an expected format.
- Validate input before passing it to the application.
- Filter data to assess whether it matches expected format (e.g. use regular expressions).
- Check for input that is too large (i.e. bigger than target program buffers, beyond array bounds, etc.)

C++

- Use the Standard Template Library which provides robust string handling functions (in lieu of the *strings* family of function such as *strcpy*).

Resources

Maguire, Steve. Writing Solid Code. Microsoft Press, 1993.

Howard, Michael, and LeBlanc, David. Writing Secure Code. Microsoft Press, 2002.

"A Guide to Building Secure Web Applications | Preventing Common Problems | The Generic Meta-Characters Problem." The Open Web Application Security Project. Verson 1.1 Final. 22 Sep 2002. 93.
URL: http://www.owasp.org/guide/v11/index.html

Exception handling

"A computer lets you make more mistakes faster than any invention in human history, with the possible exceptions of handguns and tequila."
—Mitch Ratcliffe

The Problem

In today's complex computing environments, applications often fail after an unanticipated error. In the best case, the application simply crashes leaving nothing but unintelligibly scrambled bits on the computer room floor. In the worst case, the application may crash but expose the information assets it is designed to protect.

In order to protect information assets, applications must have the capability to deal with unforeseen circumstances such as hardware, software or operating system errors in such a manner that regardless of the operational status of the application, the data is protected.

The Solution

Exception handling is a standard mechanism for application response to run-time errors or "exceptions". It provides the ability to identify (or "catch") unexpected events during application execution and handle them appropriately. This is a necessary component of any application protecting valuable information assets.

Most high-level languages provide built-in support for handling anomalous situations that may occur during the execution of applications. The mechanism for this is handled through classes and/or methods that generally have hooks to run-time stacks or execution contexts outside the normal flow of control set up specifically to watch for trouble. For example, an application written in C++ can communicate exceptions to an execution context whose sole function is to provide facilities for recovering from unexpected events. These execution contexts can be invoked by wrapping code in need of protection with exception blocks. The developer can use the classes and methods provided and write only responses to exceptions as needed within these blocks.

A generic example of an exception block (C++ but exemplary)"

```
try {
  // code to be tried
  throw exception;
}
catch (type  exception)
{
```

```
  // code to be executed in case of exception
}
```

This example actually "throws" an exception to illustrate what happens: code execution jumps to the "catch" block where appropriate response code is executed. Note that as a matter of good practice, applications generally do not throw exceptions intentionally.

A more representative example might be:

```
try
{
  do_something_unspeakably_cool ();
  do_something_potentially_bad();
  do_something_weird_that_might_have_enexpected_results();
  ...
}
catch( char * str )
{
  cout << "Exception raised: " << str << '\n';
}
```

The point being made is that code inside an exception block could be *any* code that deals with information assets (data) under protection. If there is an error or exception, the application can then take the necessary steps in the handler to protect the data as necessary, such as closing and encrypting any open files, or locking out all users.

Best Practice

Today, exception handling is achieved through platform and language specific implementations. The best practice is to learn the platform/language specific implementation.

Best practices for exception handling are typically well documented for each platform and language and can be found using the same resources as with the Know your Programming Language and Compiler/Interpreter section.

These include:

- Vendor Training (often with an associated cost)
- Product documentation (always purchase at least one complete set of manuals)
- Online resources (e.g. technical articles, websites, newsgroups, etc.)
- Third-party Books and How-to's

Sample Resources for C++[xxiii]

Kalev, Danny. ANSI/ISO C++ Professional Programmer's Handbook. 1999. Macmillan Computer Publishing.

URL: http://documentation.captis.com/files/c++/handbook/ch06/ch06.htm

"ISO/IEC 14882 Standard for the C++ Programming Language." American National Standards Institute. September 1998. Available for download at http://www.ansi.org/. Ref: INCITS/ISO/IEC 14882-1998.

Cryptography

Cryptography is the science of making data "secure" through cryptographic algorithms. Through cryptography, developers can:

- Keep data *confidential*, through encryption—specialized algorithms called ciphers that scramble data;
- Ensure the *integrity* of data by providing evidence of tampering through hashes (non-reversible transformations) or general encryption;
- Provide *authentication* to ensure that only authorized parties can supply or access data through the use of passwords, keys, one-way functions (hashes) or digital signatures.

The Problem

There are four major cryptographic mistakes:

- Failing to use cryptography when it's called for;
- Incorrect implementation of cryptography—even if need and proper algorithms are identified;
- Rolling your own;
- Assuming security by obscurity.

The consequences of the first two cases—the mis-application of cryptography can be huge. Applications potentially can become vulnerable to a variety of easily perpetrated attacks, leaving the data that was to be protected quite vulnerable. The consequences of the last two cases are related and should be self-evident. Unless you are Rivest, Shamir or Adleman, or have the educational and professional credentials and backing of industry, you shouldn't be creating new cryptographic solutions. And if you do decide to create a proprietary algorithm, keeping it a secret is no way to test it. (Recall that open standards are advocated because they are time and industry tested!)

The Solution

For *confidentiality* and *integrity* (as a sort of side-effect), *symmetric* cryptographic algorithms are the primary solution. *Symmetric* algorithms use a single key shared by two parties to both encrypt and decrypt data. The crux of the solution is that this shared key must be kept secret to provide true confidentiality. A common use is for communicating across insecure mediums such as the Internet. *Symmetric* algorithms come in two main flavors: block ciphers, which break up data into constant-sized chunks, and stream ciphers which operate on a single bit of data at a time. Stream

ciphers are much faster than block ciphers, however block ciphers are more commonly in use because they are well known and well studied. Public key cryptography is an enhanced form of symmetric cryptography that solves the problem of exchanging a shared key with intended parties over an insecure medium—such as the Internet. The basic premise is that two keys, or key-pair are generated: a public key and private key. One key is made available to anyone, while the other is kept secret. The beauty of the solution is that the public key is used to encrypt data while only the private key can decrypt the data. A downside to this approach is that public key cryptography is logarithmically slower than plain symmetric encryption.

For *integrity* and *authentication*, *one-way* cryptographic algorithms are an excellent choice. *One-way* algorithms, also called *hashing* algorithms, are one-way functions. that takes data—usually a plaintext string, and transforms it into a small piece of ciphertext—typically called a *hash value* that cannot be used to reconstruct the original plaintext. These *hash values* are the equivalent of cryptographic checksums because a good algorithm will only generate the same *hash value* for any two wildly random and intelligible plaintext strings.

Digital signatures also provide an excellent solution for providing *integrity* and *authenticity*. A digital signature is analogous to a hand-written signature whereby digital data or documents can be "signed" in a legally binding way. The existence of a digital signature should prove that it was intended to be signed. Additionally, it should be impossible to forge, and impossible to alter without being detected. Digital signatures are generally employ a combination of public key and hashing algorithms.

Best Practices

Hashing algorithms

- MD5 - http://www.rsasecurity.com/
- SHA-1 (developed by NIST for 160-bit or less hash length) - http://www.itl.nist.gov/fipspubs/fip180-1.htm
- GOST (developed in former U.S.S.R.; symmetric cipher adapted for hashing; 256-bit length) - http://vipul.net/gost/

Symmetric key algorithms (block ciphers)

- Blowfish - http://www.counterpane.com/blowfish.html
- IDEA - http://www.mediacrypt.com/indexExplorer.htm
- AES (NIST Standard for U.S. Government organizations) - http://csrc.nist.gov/encryption/aes/

Public key algorithms (for encryption and digital signatures)

- RSA - http://www.rsasecurity.com/

Resources

Schneier, Burce. Applied Cryptography. Second Edition. John Wiley & Sons. 1996.

McGraw, Gary, and Viega, John. "Make your software behave: Everything to hide | Knowing the right cryptographic algorithm to use, as well as when and how to use it, is essential to developing secure software." IBM developerWorks : Security. May 2000.
URL: http://www-106.ibm.com/developerworks/library/everything.html?dwzone=security

McGraw, Gary, and Viega, John. "Make your software behave: Tried and true encryption | The only path to secure software is to use established, rock-solid cryptographic algorithms."IBM developerWorks : Security. June 2000.
URL: http://www-106.ibm.com/developerworks/library/tried/index.html

McGraw, Gary, and Viega, John. "Make your software behave: Cryptography essentials | Using hashing algorithms for data integrity and authentication." IBM developerWorks : Security. July 2000.
URL: http://www-106.ibm.com/developerworks/library/hashing/index.html?dwzone=security

"Cryptograph." University of British Columbia, Canada, Maintained by Bill Unruh.
URL: http://axion.physics.ubc.ca/crypt.html

Random Numbers

"Computers, being completely deterministic machines, are particularly bad at behaving randomly (software missteps aside)."
–Gary McGraw & John Viega, Reliable Software Technologies

The Problem

Random numbers are not really random. The random() function does not return a truly random number; it is a call to a "pseudo-random" number generator (PRNG). The PRNG's job is to produce a seemingly random number from a range of possible numbers. Note that the range of numbers is relative limited, being the largest number the machine can store—typically 2^{32}-1 on most machines. (This jumps to 2^{64}-1 with newer processors.) To initiate this process, the PRNG generator is first passed a "seed"—an initial "random" number to "initialize" the PRNG. The PRNG then produces a sequence of seemingly random numbers. The catch is that the PRNG uses its previous result as input to the next call to itself. Given the limited size of the solution space and the deterministic quality of the machine, the primary weaknesses of this solution are: a) if the number used to compute any one value is known, then every subsequent value returned from the PRNG can be computed, and b) if the seed and algorithm are known, the sequence can be determined fairly easily.

The Solution

Real random-number generators are non-deterministic, meaning that even if the algorithm is known, the output cannot be guessed with any consistency just by

observance of the output. The best source is that which appears to be complete entropy, but which actually has some measurable property that simulates randomness. Naturally occurring phenomena in nature such as radioactive decay or fluxuations in thermal signatures are good examples. These models provide patterns nearly incomprehensible to humans. The trade-off is that these models are implemented through hardware and can be very costly and difficult to maintain across a distributed computing environment. Software solutions, though not as exceptional as hardware, provide adequate solutions.

Best Practices

In hardware (if practical):

- Internal or external devices such as the ComScire QNG that provide random numbers. (See http://www.comscire.com/.)
- Newer chipsets such as those from Intel Corporation include thermal-based random number generators and developer guides. (See http://developer.intel.com/.)

In software:

- For a fairly simple solution, sample keyboard or mouse events from user input and combine with timing or positional data to derive a seed.
- Newer operating systems have good random number generators built-in.
  For Unix/Linux, use: `/dev/random`.
  For Windows, use: Yarrow, by Bruce Schneier and John Kelsey. (See http://www.counterpane.com/yarrow.html.)
- Current language platforms such as Java and C/C++ provide good random numbers via classes and
  For Java, use: `SecureRandom` class to seed `java.util.Random`.
  For C/C++: adapt `TrueRand` by Don Mitchell and Matt Blaze.

Resources

McGraw, Gary, and Viega, John. "Make your software behave: Playing the numbers | Truly secure software needs an accurate random number generator." IBM developerWorks : Security. April 2000.
URL: http://www-106.ibm.com/developerworks/library/playing/index.html

McGraw, Gary, and Viega, John. "Make your software behave: Beating the bias | How to approach truly random number generation through hardware." IBM developerWorks : Security. April 2000.
URL: http://www-106.ibm.com/developerworks/library/beating.html?dwzone=security

McGraw, Gary, and Viega, John. "Make your software behave: Software strategies | In the absence of hardware, you can devise a reasonably secure random number generator through software." IBM developerWorks : Security. April 2000.
URL: http://www-106.ibm.com/developerworks/library/randomsoft/index.html?dwzone=security

Canonical Representation

Merriam Webster defines the phrase *canonical form* as "the simplest form of something". In terms of computing, the *canonical form* of an object is its one standard name. *Canonicalization* is the process of reducing various equivalents to the one standard name.

The Problem

There are many different ways to represent the name of a computing object such as a file name, server name, user name, etc. Names can be represented by the default character set, in hexadecimal or in some case octal or other base systems, and with special characters that are either interpreted specially or disregarded by whatever mechanism is receiving the input. This is critical in programming because alternate representations of an object name can pose serious security issues in the absence of programming practices that protect against this. The biggest reason for this is that in many situations, applications try to determine if input matched a certain pattern, and if it did not—if alternate representations are not considered, the default response is insecure—potentially allowing unauthorized access to the system!

Examples:

- URLs[xxiv] can contain hexadecimal representations (or HTTP-escaped equivalent) of characters.
  Ex.
  http://www.aurl.com/secure/files.html
  is the same as
  http://www.aurl.com/%73%65%63%75%72%65/%66%69%6c%65%73.html
- IP addresses can be represented either with or without (dot-less).
  Ex.
  http://192.168.1.1
  is the same as
  http://3232235777
- Newer Windows file systems using long filenames automatically generate 8.3 formatted filenames for backward compatibility. The format is the first six characters of a long filename, plus a tilde (~) and an incremented digit, plus the first three letters of the file extension.
  Ex.
  SecretFile.ledger
  is the same as
  SECRET~1.LED
- File systems allow for movement termed directory traversal using parent paths (..), absolute and relative paths to filenames. Filenames can be represented by

any combination of the paths constructed with the special characters required to traverse directories.

Ex.
`C:\SecretFiles\TopSecret\foo.bar`
is the same as
`C:\SecretFiles\..\SecretFiles\TopSecret\foo.bar`
or even
`C:\SecretFiles\TopSecret\..\TopSecret\..\..SecretFiles\TopSecret\foo.bar`

If input is allowed to contain these special characters required for directory traversal, then without proper access controls, an attacker could gain access to critical files on the system.

Ex.
`..\WINNT\Repair\SAM`
`../etc/passwd`

- On some platforms, system devices are represented by filenames and available in the directory structure. Making calls to these devices as if they were files can cause unexpected results, potentially leaving the system open to compromise.

  Ex.
  On Unix/Linus: `/dev/console`, `/dev/tty0`, etc.
  On Windows: `CON`, `AUX`, `COM1-COM9`, etc.

In these examples, what if the input checker was not checking or filtering for the alternate forms? What if the files in question had critical information stored in them, such as credit card numbers, account numbers, or authentication information? If the coding was such that the input checker looked for known patters and the alternate forms didn't match those patterns, and the default response was to allow access, these files could then be accessed and subsequently "owned" by an attacker! In the case of system devices, systems might fail to an insecure state, allowing an attacker to own whatever is exposed.

The Solution

The solution is to distrust input names! Input should be canonicalized to a standard form before any programmatic decisions are made based on that input. Design so that if input does not match accepted patterns, no access is allowed—period. Operating systems and programming languages offer up some common solutions that can be leveraged to assist in this process.

The Practices

- The application receiving the input performs *canonicalization*. Reduce the input to the single standard form as determined by the intended use of the application.
- Use regular expressions to define patterns that are acceptable. Everything else is invalid input and allowed no access. (The .NET Framework, VB, Perl and C++

standard template library include or have available implementations of regular expressions.)

Ex.

`^[a-zA-Z0-9]+$` allows for only alphanumeric input.

`^([a-z\.]+)@([a-z\.]+[\.][a-z]{2,3})$` can filter for a valid e-mail address.

- Do not trust the a path. Again, regular expressions can help to *canonicalize* input to filter for correct file locations.

  Ex.

  `^[def]:\\\w{1,20}\.(doc|xls|ppt)$` can filter input to check for only specific document filenames up to 20 characters in length in the root of the *d*, *e*, or *f* drives.
- Do not trust file names to represent actual files. Instead leverage the operating system's ability to open a file and determine its type by its properties.

  Ex.

  Call the *stat* function (*stat.st_mode* variable) in C/C++ on many platforms to check if the file is a valid file or device.
- In Windows, disable 8.3 filenames.

  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem
  NtfsDisable8dot3NameCreation: REG_DWORD : 1

Resources

Howard, Michael, and LeBlanc, David. Writing Secure Code. Microsoft Press, 2002.

"Canonicalization." The Open Web Applicatoin Security Project. Version 1.1 Final. 2002.
URL: http://www.owasp.org/asac/canonicalization/unicode.shtml

---

Summary

Security is really about risk management. Risk is the possibility of events occurring that produce negative consequences. Risk management is the process of evaluating or assessing risk and taking steps to reduce or mitigate risk to an acceptable level. The process provides companies with the information to make decisions that balance the operational costs of protection measures vs. the benefits of protecting assets to support the mission of the organization.

Secure application design and development greatly assists in protecting the company's information assets and thus contributing to the mission of the organization. It is an important and necessary attribute of a secure computing environment—a key item successful strategy to reduce the level of the risk.

- From importance and gaps to history and the developer's role;
- From a romp through educational, professional, government and online resources to life cycle issues and compiler quirks and features;
- From the quandary of the developer to help and hope,

This paper has illustrated the importance of secure application development and provided background background/history on why this practice is not as pervasive as it should be today.

This paper has assembled some fundamental tenets of security doctrine into a reference framework in the form of a foundation, principles and design guidelines for the design and development of secure applications.

Go forth and design and develop securely!!

References:

[1] United States. National Institute of Standards and Technology. SP 800-12 An Introduction to Computer Security: The NIST Handbook. October 1995. 9.
URL: http://csrc.nist.gov/publications/nistpubs/800-14/800-12.pdf

[2] United States. National Security Agency. Information Assurance Technical Framework. Release 3.1, September 2002. 1.4 Defense In Depth, 1-14.
URL: http://www.iatf.net/framework_docs/version-3_1/file_serve.cfm?chapter=ch01.pdf

[3] "Précis: Research on Techniques and Tools for Computer Security: The COAST Project and Laboratory". Section: Academic Security Education in the U.S. 2 June 1998. Purdue University, Deptartment of Computer Sciences.
URL: http://www.cerias.purdue.edu/coast/precis/node2.html

[4] "Software Assurance for Security." Gary McGraw. 1 Oct 1999. Reliable Software Technologies.
URL: http://packetstormsecurity.nl/papers/java/ieee-computer-secass.pdf

[5] "Security is Essential to Running an e-Business." Rajeev Khanolkar. Internet Security Advisor. July/August 2000. 21.
URL: http://www.netforensics.com/documents/khanr01.pdf

[6] "Computer Security Issues & Trends, 2002 CSI/FBI Computer Crime and Security Survey." Richard Power. 2002.
URL: http://www.gocsi.com/press/20020407.html

[7] McConnell, Steve C. Software Project Survival Guide. Microsoft Press, 1997.

[8] Tzu, Sun. The Art of War. Trans. Ralph D. Sawyer. Westview Press, Inc., 1994. Vacuity and Substance, 191.

[9] "Know Your Enemy: Statistics, Analyzing the past ... predicting the future." The Honeynet Project. 22 Jul 2001.
URL: http://project.honeynet.org/papers/stats/

[10] Viega, John, and McGraw, Gary. Building Secure Software. Addison-Wesley Professional Computing Series, 2001. 7.

[11] United States. National Institute of Standards and Technology. SP 800-14 Generally Accepted Principles and Practices for Securing Information Technology Systems. September 1996. 50.
URL: http://csrc.nist.gov/publications/nistpubs/800-14/800-14.pdf.

Resources:

United States. National Institute of Standards and Technology. SP 800-27 Engineering Principles for Information Technology Security (A Baseline for Achieving Security). June 2001.
URL: http://csrc.nist.gov/publications/nistpubs/800-14/800-27.pdf.

searchSecurity.com. 2002. TechTarget. http://searchsecurity.techtarget.com/

Free On-Line Dictionary of Computing. Ed. Denis Howe. Supported by Imperial College Department of Computing. 2002. http://foldoc.doc.ic.ac.uk/foldoc/index.html

[i] The protection and defense of information and information systems by ensuring their availability, integrity, and confidentiality.
[ii] Common Off The Shelf.
[iii] The term grind is used literally, as these machines were built using gears.
[iv] Bio information on John von Neumann's life and contributions to computing can be found at http://ei.cs.vt.edu/~history/VonNeumann.html.
[v] The tale of the first program can be read at http://www.computer50.org/.
[vi] Most people assume Robert Morris's Internet Worm that wreaked havoc on the Internet in 1988 was the first of its kind. *A Short History of Computer Viruses and Attacks* provides some clarity.
[vii] A buzz phrase meaning an attacker can run any code on a vulnerable system—very bad.
[viii] End Of Life.
[ix] A common phrase referring to the ability of the programmer to easily write code that could crash a program.
[x] Understanding and communicating risks and mitigation to Management is speaking their power-dialect. Management understands ROI, Mission and Objectives. Presenting secure application development as risk mitigation makes sense to Management.
[xi] Organizations such as GIAC (associated w/ SANS) and (ISC)^2 provide excellent models for security Code of Ethics.

[xii] Smashing the stack refers to the practice of writing past the end of a declared storage array so that the program will then execute arbitrary code or jump to a random address in the program causing all manner of havoc.
[xiii] BugTraq is a full disclosure moderated mailing list for the *detailed* discussion and announcement of computer security vulnerabilities: what they are, how to exploit them, and how to fix them operated by Security Focus. See http://online.securityfocus.com/popups/forums/bugtraq/faq.shtml#0.1.1.
[xiv] "A list of standardized names for vulnerabilities and other information security exposures" maintained by the Mitre Corporation. See http://cve.mitre.org/about/.
[xv] For definition see http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci211852,00.html.
[xvi] For definition see http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci550928,00.html.
[xvii] Sun Tzu, Vacuity and Substance, p.191.
[xviii] A technique for distributing applications employing code that can be transmitted across a network and executed on disparate systems.
[xix] There is much discussion on the interpretation of the term nonrepudiation in the paper vs. digital world. For more, see http://searchsecurity.techtarget.com/sDefinition/0,290660,sid14_gci761640,00.html.
[xx] The CIA Triad: Confidentiality, Integrity, and Availability.
[xxi] Current operating systems now rely on accurate system time for authentication protocols, distributed and cluster computing synchronization.
[xxii] Universal Time Coordinated. See http://aa.usno.navy.mil/faq/docs/UT.html.
[xxiii] While C++ was not the first language to include exception handling, it was the first international standard to include the capability/functionality. It serves as a good reference source on the topic.
[xxiv] Uniform Resource Locator.

Last Updated: October 22nd, 2017

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS San Diego 2017	San Diego, CAUS	Oct 30, 2017 - Nov 04, 2017	Live Event
SANS Seattle 2017	Seattle, WAUS	Oct 30, 2017 - Nov 04, 2017	Live Event
SANS Gulf Region 2017	Dubai, AE	Nov 04, 2017 - Nov 16, 2017	Live Event
SANS Milan November 2017	Milan, IT	Nov 06, 2017 - Nov 11, 2017	Live Event
SANS Amsterdam 2017	Amsterdam, NL	Nov 06, 2017 - Nov 11, 2017	Live Event
SANS Miami 2017	Miami, FLUS	Nov 06, 2017 - Nov 11, 2017	Live Event
SANS Paris November 2017	Paris, FR	Nov 13, 2017 - Nov 18, 2017	Live Event
Pen Test Hackfest Summit & Training 2017	Bethesda, MDUS	Nov 13, 2017 - Nov 20, 2017	Live Event
SANS Sydney 2017	Sydney, AU	Nov 13, 2017 - Nov 25, 2017	Live Event
GridEx IV 2017	Online,	Nov 15, 2017 - Nov 16, 2017	Live Event
SANS San Francisco Winter 2017	San Francisco, CAUS	Nov 27, 2017 - Dec 02, 2017	Live Event
SANS London November 2017	London, GB	Nov 27, 2017 - Dec 02, 2017	Live Event
SIEM & Tactical Analytics Summit & Training	Scottsdale, AZUS	Nov 28, 2017 - Dec 05, 2017	Live Event
SANS Khobar 2017	Khobar, SA	Dec 02, 2017 - Dec 07, 2017	Live Event
SANS Austin Winter 2017	Austin, TXUS	Dec 04, 2017 - Dec 09, 2017	Live Event
SANS Munich December 2017	Munich, DE	Dec 04, 2017 - Dec 09, 2017	Live Event
European Security Awareness Summit & Training 2017	London, GB	Dec 04, 2017 - Dec 07, 2017	Live Event
SANS Bangalore 2017	Bangalore, IN	Dec 11, 2017 - Dec 16, 2017	Live Event
SANS Frankfurt 2017	Frankfurt, DE	Dec 11, 2017 - Dec 16, 2017	Live Event
SANS Cyber Defense Initiative 2017	Washington, DCUS	Dec 12, 2017 - Dec 19, 2017	Live Event
SANS Security East 2018	New Orleans, LAUS	Jan 08, 2018 - Jan 13, 2018	Live Event
SANS SEC460: Enterprise Threat Beta	San Diego, CAUS	Jan 08, 2018 - Jan 13, 2018	Live Event
SANS Amsterdam January 2018	Amsterdam, NL	Jan 15, 2018 - Jan 20, 2018	Live Event
Northern VA Winter - Reston 2018	Reston, VAUS	Jan 15, 2018 - Jan 20, 2018	Live Event
SEC599: Defeat Advanced Adversaries	San Francisco, CAUS	Jan 15, 2018 - Jan 20, 2018	Live Event
SANS Berlin 2017	OnlineDE	Oct 23, 2017 - Oct 28, 2017	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced