PDHonline Course E180 (3 PDH)

Software Security: Practical Defensive Strategies

Instructor: Warren T. Jones, Ph.D., PE

2020

PDH Online | PDH Center

5272 Meadow Estates Drive
Fairfax, VA 22030-6658
Phone: 703-988-0088
www.PDHonline.com

An Approved Continuing Education Provider

Software Security: Practical Defensive Strategies

Warren T. Jones, Ph.D., P.E.

Course Content

Module #1 Introduction and Definitions

According to studies by the Software Engineering Institute "over 90% of software security vulnerabilities are caused by known software defect types. Analysis of 45 e-business applications showed that 70% of the security defects were software design defects." [1].

Three trends are often cited as introducing security risks into systems and contributing to the magnitude of the security problem today.

The increasing complexity of systems make them more difficult to understand and hence more difficult to secure.

Increasing access to applications through various computer network technologies adds considerably to the security risks.

Software is being increasingly designed to be extensible with the incremental addition of functionality making it impossible to anticipate the kind of mobile code (updates) that may be downloaded.

Definitions:

Security policy – the rules established by an organization that govern how security is provided to protect assets of that organization.

Threat – a goal an adversary might try to achieve to abuse an asset in the system.

Vulnerability – a specific way that a threat can be exploited.

Security flaw – a software defect that represents a potential security risk.

Exploit – software or technique that is used to take advantage of vulnerabilities.

Mitigation – the means to prevent or limit exploits against vulnerabilities.

Can Security be Defined?

A good question is "Can we ever declare a software application secure?" Unfortunately security, like many other engineering goals, is a relative quantity and 100 percent security is unachievable. A better question is to be more specific and ask "Secure against what and from whom?" Some consider security is a subset of reliability.

Module #2 Approaches to the Security Problem

Penetrate and Patch

Often software is developed in an "Internet time" highly compressed schedule in order to be first to market. This approach considers security as an add-on feature after delivery. When vulnerabilities are found, frequently as a result of an attack, patches are developed and issued to the user community. There are many problems with this "penetrate-and-patch" approach to security. Here are a few:

- Developers can only patch problems which they know about. Attackers may find problems that they never report to developers.
- Patches are rushed out as a result of market pressures on vendors, and often they introduce new problems of their own to a system.
- Patches often only fix the symptom of a problem and do nothing to address the underlying cause.
- Patches often go unapplied, because system administrators tend to be overworked and often do not wish to make changes to a system that "works" [8]. It should also be noted that system administrators are often not security experts.

Build Security into the Software Development Life Cycle

The recommended approach is to incorporate software security as an engineering goal throughout the software engineering life cycle.

Several software development life cycle models have been developed. These models consist of a sequence of stages of development activities, some or all of which may be revisited during the development process, and usually include the following four stages:

1. Requirements Engineering and Analysis

This first stage is about communicating with the customer. The objective is to arrive at a written agreement describing the functionality of the software to be developed. The final product of this activity is usually called a specification and forms the basis for all the development activities that follow. The focus is on defining the operational characteristics of the software and has three primary goals of providing the following:

- behavioral description of customer requirements with the emphasis on the *what* rather than the *how*
- foundation for the software design
- operational system definition that can be used for system validation after the software development is complete.

2. Architectual Design

The design activity is the bridge between the software requirements and analysis models, and deliverable product construction. Design is the process of producing the "blueprint" for the coding and testing. It is also the activity which establishes software quality. The results of design activities are representations which can be assessed for quality.

3. Component Level Design

This level of design describes the data structures, interfaces and algorithms. Component level design can be represented in a programming language, but it is also often described in some other intermediate representation such as a program design language (PDL) for conventional module design and the Object Constraint Language (OCL) in the object-oriented design world.

4. Testing

After the software system is coded into a deliverable product, testing strategies are used to validate system requirements. Testing strategies are designed to detect errors in the system. Debugging is the process of finding the source of the errors for correction. Exhaustive testing is impractical. Therefore, no matter how much testing is done, it is never known with certainty if all bugs have been detected. Since testing is a process of detecting the presence of errors, the absence of all errors cannot be guaranteed by the testing process. A high percentage of project resources are expended on the testing phase. Testing usually proceeds in two phases, first at the component level sometimes called unit testing. Unit testing is followed by

integration testing in which increasingly larger groups of components are tested culminating in the total system. Unit testing is usually done by the developer and integration testing by an independent test group

See PDHOnline course **Software Engineering Concepts** for a more detailed discussion of the software life cycle.

Since many of the issues of software security are issues of risk management, the spiral model of software development is often mentioned as appropriate. The **Spiral Model** combines elements of the four stages described above and rapid prototyping to implement an evolutionary development process. Each traversal around the spiral, beginning with Objectives and moving through Risk Assessment, Production and Validation and Next Phase Planning represents a new more complete version of the system with a risk assessment each time around. Each version can be viewed as a system prototype during any phase of this evolutionary development. For example, one of the spiral traversals might represent the architectural design and another might focus on integration testing.

Given that a software development model such as the Spiral Model is in place, some security activities that should be added to each life cycle stage are listed below.

Requirements:

Add security specifications. Need to include information on what the system is not supposed to do. Use/misuse case scenarios can be helpful together with threat modeling. See module 4 of this course for more details on threat modeling.

Design:

Apply security design principles described in module 3. , e.g. “Design with the Enemy in Mind” [8].

Implementation:

Follow secure programming guidelines discussed in module 5.

Testing:

Add Security test plans and use random input testing (e.g. Fuzz Testing http://www.cs.wisc.edu/~bart/fuzz/fuzz.html) or vulnerability analysis using penetration testing (http://www.penetration-testing.com) .

Team Software Process for Secure Software Development (TSP-Secure)

The Team Software Process for Secure Software Development (TSP-Secure) of the Software Engineering Institute (SEI) – augments the earlier Team Software Process,

which is a set of defined and measured practices, with security practices throughout the software development life cycle. SEI has shown that it is possible to reduce defects by one or two orders of magnitude with the adoption of TSP. Training in these practices is available from SEI.

Module # 3 Principles for Software Security

It has been said that a very large percentage of security problems can be avoided if the following principles are followed:

1. Secure the weakest link: security is a chain.

2. Practice defense in depth: manage risk with diverse defensive strategies. Do not rely on just one strategy. Consider using layers of defense. e.g. combining secure programming techniques with secure runtime environments.

3. Fail securely. Failures are unavoidable. Plans should be made to anticipate them. For example, access decisions should be made based on permission rather than exclusion.

4. Follow the principle of least privilege: minimum access required to perform an operation and only for the minimum time necessary.

5. Compartmentalize: minimize the amount of potential damage by organizing the system into the smallest number of units as possible.

6. Keep it simple.

7. Promote privacy.

8. Remember that hiding secrets is inherently difficult.

9. Be reluctant to trust. Servers should be designed to distrust clients and conversely.

10. Use your community resources: Use security libraries and cryptographic algorithms that have been widely used and evaluated

Module #4 Threat Modeling

What is Threat Modeling?

Threat modeling views a software system from an attacker's perspective in order to define attack goals. It is assumed that an attacker will need to provide the system with data or interact with it some way. It is also assumed that systems are attracted to assets of interest. Threat modeling focuses on **system entry points** and is intended to provide a methodology for analyzing the security of a system. A threat model is fundamentally a plan for penetration testing, which is the actual attacking of a system in ways described in the model.

Identifying and Classifying Threats

The set of all threats identified constitute the threat profile of the system and should be classified using the six element STRIDE classification system described below [10].

- Spoofing – Allows an adversary to pose as another user, component, or other system that has an identity in the system being modeled
- Tampering – The modification of data within the system to achieve a malicious goal.
- Repudiation – The ability of an adversary to deny performing some malicious activity because the system does not have sufficient evidence to prove otherwise.
- Information Disclosure – The exposure of protected data to a user that is not otherwise allowed access to that data.
- Denial of service – Occurs when an adversary can prevent legitimate users from using the normal functionality of the system.
- Elevation of Privilege – Occurs when an adversary uses illegitimate means to assume a trust level with different privileges than he currently has.

Threats can frequently be classified into more than one of these categories. Elevation of privilege is often the highest threat risk in a system.

Analyzing Threats

One approach to the analysis of the threat profile of a system is the use of threat trees. These structures are used to determine if the conditions exist for a threat to be realized

and if so, to determine if they are mitigated or unmitigated. These conditions are associated with risk and are sometimes rated associated with the DREAD system of characterizing risk.

- Damage Potential – ranks the extent of the damage that occurs if a vulnerability is exploited.

- Reproducibility – ranks how often an attempt at exploiting a vulnerability works.

- Exploitability – assigns a number to the effort required to exploit the vulnerability. In addition, exploitability considers preconditions such as whether the user must be authenticated.

- Affected Users – a numeric value characterizing the ratio of installed instances of the system that would be affected if an exploit became widely available.

- Discoverability – measures the likelihood that a vulnerability will be found by external security researchers, hackers, and the like if it went unpatched.

It is recommended that a range no wider than from 1 to 3 be used to make the risk categories more meaningful and less ambiguous to the security team.

Threat modeling is typically viewed from two perspectives, feature-based and scenario-based. Each approach has its advantages and disadvantages.

Functionality-based Feature-Level Threat Modeling

Threat modeling at the feature level requires everyone on the software development team to be involved in security issues. These models are typically created at the same granularity level as specifications. Feature level models are particularly appropriate for high-risk components or subsystems. Since large systems may have a large number of features, it is important to select the right features for modeling. For example, it is very important to determine which features will process user-supplied data or otherwise interact with the system user.

Scenario-based Application-level Threat Modeling

Scenario-based threat models are developed at the application level and consider a specific deployment and attack scenario. This scenario would include specific attack goals, motivations as well as a profile of the adversary. This application level model can identify areas in the application that may require a more in-depth analysis, perhaps at the feature level. If resources are limited, the application level model should be developed first.

It is important to establish threat modeling completion criteria. The following three tasks are often used as exit criteria:

- Document all entry points
- Resolve all threats
- Review the threat model documentation

Also note that any modifications to the system should require a review of the threat model in terms of the impact of the modification.

Module #5 C/C++ Security

What would you guess would be the number one software security problem, by far? You might be surprised to learn that it is buffer overflow, a rather simple concept.

Buffer overflows as a security vulnerability have been discussed for forty years and yet this type of software problem continues to be one of the most frequently reported instances of system attacks. During 2004, 323 buffer overflow vulnerabilities were reported (http://nvd.nist.gov). This is an average of more than 27 new reports a month. The same source reported 331 during the first half of 2005 [6]. A buffer overflow is a condition caused by a write operation into a fixed-sized buffer in which the size of the data is larger than the size of the buffer. Most buffer overflows are the result of the properties of the C or C++ language. For additional information on reported vulnerabilities, see postings to the BUGTRAQ mailing list at http://msgs.securepoint.com/bugtraq.

Many factors influence the choice of a programming language to use for implementation. It is common for efficiency considerations to dominate the language selection process. One of the factors should be security considerations. Choosing the C programming language for efficiency should take into account the inherent security risks associated with a language that has no bounds checks on array and pointer references. The programmer must build these checks into the program code. C program efficiencies and low-level data manipulation capabilities come at the high potential risk of security vulnerabilities and very special diligence is required. Reference [8] is an excellent resource for more detailed information on secure programming in C/C++.

Some useful guidelines are listed below:

Off-the-Shelf Software Vulnerabilities

Future software systems will be reusing software components and libraries more extensively. Vulnerabilities in C libraries can be mitigated by statically linking safe

libraries with applications. Another approach is to use secure wrappers that perform validation checks for known vulnerabilities. Finally, applications can be executed in a supervised environment which enforces a defined policy. (e.g. systrace facility: http://www.citi.umich.edu/u/provos/systrace)

Compiler Error-Checking

Raise the level of compiler checking to be sure that some errors are detected automatically.

Format String Vulnerabilities

String representation and manipulation are the source of many security vulnerabilities since strings are used for most of the data interaction between users and the system. In the C language strings are not a built-in data type but consist of a contiguous sequence of characters terminated by and including the first null character. The length of a string is the number of bytes preceding the null character. The use of these C-style strings in C or C++ programs is error prone. The most common errors are unbounded string copies, off-by-one errors, null termination errors and string truncation. Some mitigation strategies are:

Use fgets() or gets_s() instead of gets()

Use memcpy_s and memmove_s() instead of memcpy() and memmove()

Use strncpy_s() and strncat_s() instead of strncpy() and strncat()

There is a Microsoft library of safe string handling functions called Strsafe.h. However, be aware that there are differences in semantics. For example, when strncat() detects an error, it sets the destination string to a null string. In the case of StringCchCat() in the Microsoft library the destination is filled with as much data as possible, and then null-terminates the string.

Integer Vulnerabilities

Integers are an increasing source of vulnerabilities primarily because boundary conditions for integers are frequently ignored. The prevention of these vulnerabilities depends on a comprehensive understanding of integer representations and the application of this understanding in systems design. The Howard library, that detects integer overflow conditions can be helpful (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure09112003.asp) and can be used in both C and C++ programs. It is one of several libraries for this purpose.

Dynamic Memory Management

Dynamic memory management can be quite complex and therefore is a source of defects. Programming errors to be avoided include the following:

- initialization errors
- failing to check return values
- writing to already freed memory
- freeing the same memory multiple times
- improper paired memory management functions
- failure to distinguish scalars and arrays
- improper use of allocation functions.

Some good practices are as follows:

Set the pointer to NULL after the call to free()

Use the same patterns for allocating and freeing memory. e.g. define create() and destroy() functions that perform equivalent functions.

Allocate and free memory in the same module and at the same level of abstraction.

Match memory allocation and memory freeing.

Make use of runtime analysis tools such as Purify, Dmalloc Library and Electric Fence.

Validation of Inputs

Identify all sources of input including user-controlled files, environmental variables, network interfaces and command-line arguments. Design the system to handle any range or combination of valid input data. Specifications for all inputs can be defined by use of a data dictionary. An even better strategy is to include and validate not only all input variables but all variables that hold data from a persistent data store.

Static Scan of Source Code

Tools for scanning source code for potential vulnerabilities are available. Good practice dictates that this scan should not replace manual code review. Also it should be remembered that static scan tools will not find all security problems. A downside for these tools is that they tend to produce warnings that are not helpful.

A list of example tools is given below:

Freeware

- ITS4 (http://www.cigital.com/its4/)
- RATS (http://www.securesw.com/rats/)
- Flawfinder (http://www.dwheeler.com/flawfinder/)
- -Lint (a unix utility) (http://lcint.cs.virginia.edu)

Commercial

- Prevent (earlier version was called SWAT) available from Coverity.
- Prexis available from Ounce Labs
- Fortify available from Fortify Software
- PREfix and PREfast available from Microsoft

Cyclone: A Safe Dialect of C

Cyclone (http://www.cs.cornell.edu/projects/cyclone) has been designed "to prevent buffer overflows, format string attacks and memory management errors that are common in C programs, while retaining C's syntax and semantics". [11] The performance cost of bounds checking can be as high as 100 percent.

Modified Compilers

Automatic buffer overflow detection can be added to a program by using modified compiler with this capability. Examples are StackGuard, ProPolice, StackShield and Return Address Defender (RAD).

Race Conditions

Race condition induced vulnerabilities are very likely to increase in the future since the hardware community is turning increasingly to multiple processors and distributed systems to achieve their performance goals. A race condition is a consequence of the use of concurrency in today's computer systems. Concurrency is the running of two or more separate execution flows simultaneously. Uncontrolled concurrency can produce undesirable program behavior and can be the source of vulnerability. Static analysis tools for race detection include ITS4, RacerX, Flawfinder and RATS. Examples of dynamic analysis tools are the following: Eraser, Thread Checker, RaceGuard and Alcatraz.

Race condition detection has been shown to be a very computationally intensive (NP-Complete) problem. Static tools produce both false positives (incorrectly identified vulnerabilities) and false negatives (vulnerabilities not identified). Dynamic analysis tools have large execution time costs and cannot detect race conditions outside the actual execution flow.

Possible Future Solution to the Performance Penalty for Buffer Overflow Detection

Any buffer overflow technique will exact a performance penalty that can range from 4% to over 1,000% as reported by the research community. One approach to reducing this penalty is to move the buffer detection operations from software to hardware. The SmashGuard Project (http://www.smashguard.org) proposes changes to the CPU microcode to accomplish this reduction.

Module # 6: Java Security

The object-oriented (OO) approach to software development is epitomized by the Java language at the implementation stage. The OO approach is based on modeling of the problem domain using classes and objects.

Class : defines the data and procedural abstractions for the information content and behavior of some system entity.

Package : a group of related classes.

Method : representation of one of the behaviors of a class.

Object : instance of a specific class. Objects can inherit the attributes and operations defined for a class. Classes are sometimes illustrated as "cookie cutters" and the associated objects as "cookies".

The goal of the OO approach is the design of all classes and associated methods that are appropriate for the application being developed. Java applications can be developed using a development environment.

The three primary components of the Java Development Environment are as follows:

- A programming language that compiles into byte code
- The Java Virtual Machine (JVM) that executes the byte code
- A JVM execution environment containing a collection of base classes that are foundational for applications development

Notice that the byte code runs on the JVM, so Java code can run on any machine to which the JVM has been ported.

Some of the important features of the Java language are enumerated below.

Strongly Typed

Memory access is limited to particular controlled locations that have particular representations and variables cannot be used before they are initialized.

No Pointers

Memory is managed by reference and therefore cannot be manipulated arithmetically. Pointers are often cited as the source of many bugs in C and C++ programs.

Garbage Collection

Java provides a garbage collector which runs automatically in background mode.

Multi-threaded

Synchronization primitives are provided in support of Java's built-in multithread capability. Multimedia application performance is enhanced by use of this capability.

Dynamic Linking

Java classes are linked together as they are needed at run time.

Bounds Checking

Bounds on arrays are automatically checked for all array accesses.

Object and Method Protection

Objects and methods cannot be changed or overridden if they are declared final.

The above list of features is only one part of the security-related aspects of the Java language. In order to address the security issues of mobile code, a Java Security Model has been developed. The original default model is called the sandbox. The idea is to provide the capability to safely run untrusted code. The protection mechanisms of the original sandbox have been extended by more recent versions of the security model. The kind of threats the designers of the Security Model had in mind are attacks that

- modify the system
- invade a user's privacy
- deny legitimate use of the machine by monopolizing resources
- antagonize a user.

The Base Java Security Model: The Sandbox

The purpose of the sandbox is to impose strict controls on what certain Java programs can and cannot do. The default sandbox consists of three components: the Verifier, the Class Loader and the Security Manager.

The Verifier guarantees that: the class file has the correct format; stacks will not be overflowed or under flowed; byte code instructions all have parameters of the correct type; no illegal data conversions occur; private, public, protected and default access are legal and all register accesses and stores are valid. For example, the Verifier functions as the primary gatekeeper in the security model. Any downloaded byte code must first satisfy the rules of the Verifier.

All Java objects belong to classes. Class Loaders provide the class loading function when needed by the VM and define the namespaces seen by various classes and the interrelationship among these namespaces.

The Security Manager is a single object whose function is to perform runtime checks on dangerous methods.

In the original JDK, 1.0 security policy was a black and white situation, with external code treated as untrusted and processing by the Verifier. Built-in local byte code from the local disk is automatically trusted and bypasses the Verifier.

JDK 1.1 security extended the JDK 1.0 security model to allow code digitally signed by a trusted party to be treated as trusted in the same way as local built-in code. JDK 1.1 also introduced certificate technology which provides an authentication mechanism, allowing one site to securely recognize another. Site recognition provides the opportunity for establishing trust.

Java 2 security allowed digitally signed external classes to be partially trusted as determined by the user. This change provides the development of fine-grained security policies that grant privileges according to identity. Java 2 security also includes secure socket layer (SSL) communications by use of packet encryption and transmission over an untrusted channel. Additional details on Java security, including the more recent refinements of Java 5, can be found at http://java.sun.com/security/.

As mentioned at the outset of this course, there is no 100 percent secure system. However, there are application developer and user and guidelines that can help assure more secure systems. The following lists are taken from reference [7], a rich source of additional information on Java security issues:

Guidelines for Java Developers

- Do not depend on initialization. There are ways an object can be allocated and not initialized. Write classes so that before any object performs any action, it verifies that it has been initialized.

- Limit access to your classes, methods and variables. By default, make everything private. Change to private only if a good reason arises and when it does, document that reason.

- Make all classes and methods final, unless there is good reason. If a good reason arises, document it.

- Do not depend on package scope. Classes, methods and variables that are not explicitly labeled as public, private or protected are accessible with the same package. Do not rely on this apparent limited access for security. Java classes are not closed, so an attacker could introduce a new class inside your package, and use this new class to access the things you thought you were hiding.

- Do not use inner classes. Java byte code has no concept of inner classes, so inner classes are translated by the compiler into ordinary classes that happen to be accessible to any code in the same package.

- Avoid signing your code.

- If you must sign your code, put it all in one archive file.

- Make your classes uncloneable. The object-cloning mechanism can permit an attacker to generate new instances of classes that have been defined without using any of the constructors.

- Make your classes unserializeable. Serialization can make it possible to access the internal state of objects.

- Make your classes undeserializeable. Even if your class is not serializeable, it may still be deserializeable.

- Do not compare classes by name. This is a potential problem since several classes can have the same name in a JVM.

- Secrets stored in your code will not protect you, for example, do not store cryptographic keys in the code of your application library.

Guidelines for Java Users

- Know what websites you are visiting.

- Learn as much as you can about Java security.

- Know your Java environment.

- Use up-to-date browsers with the latest security updates.

- Keep a lookout for security alerts. Subscribe to the CERT Alert List at http://www.cert.org.

- Apply drastic measures if your information is truly critical.

- Assess your risks.

Smart Card Security

Smart cards have the appearance of a credit card but actually contain an embedded computer system with nonvolatile memory, storage and card operating system. Some of the uses of smart cards are listed below:

- Security cards for advanced authentication algorithms

- Electronic cash cards

- Memory cards that can function as portable databases

- Cards that contain the Java Virtual Machine and can run Java applets

Initially, programming languages for smart cards have been special-purpose assembly languages. However, Java's appeal as a cross-platform language has led to the development of Java Card (see http://developers.sun.com/techtopics/mobility/javacard/articles/javacard1), a scaled-down

card-size version of Java. Applications written by different vendors can be deployed on a single card and new applications can be added after issuance to an end user. Gemplus (http://www.gemplus.com) and Schlumberger (http://www.cyberflex.slb.com/) distribute commercial Java Card development environments called GemXpresso and Cyberflex, respectively.

Since smart cards are frequently used in security-critical applications, the security research community has studied smart card vulnerabilities and made some surprising discoveries. Among the most noteworthy is that it is possible to deduce the values of cryptographic keys hidden in the smart card by introducing computational errors. These errors can be introduced by methods as simple as introducing variations in temperature, input voltage or clock input.

In addition to these physical vulnerabilities, there is what is known as the terminal problem. Since smart cards have no built-in display, they need a way to interact with their users. Card acceptance devices (CAD) have been developed to serve as smart card readers. Some are single function devices which simply display the balance on a card. More sophisticated CADs can be directly connected to a port of a standard PC and allow the development and downloading of code for smart cards. PCs represent an attractive candidate for consumers as a client-side CAD. However, this approach is very risky given the normal exchange of documents and programs through a PC.

The removal of some of the features of Java to make card scale-down possible has introduced security vulnerabilities. For example, the security manager and garbage collection is missing. No automatic garbage collection capability means that the system may be subject to memory leaks, a condition of full memory as a result of the inefficient management of memory. To prevent these type problems means special diligence in programming as well as extensive testing and analysis of the code. The latest information on Java Card Technology security is posted at http://java.sun.com/products/javacard/index.jsp.

Module # 7: Perl Security

Although Perl has been around since 1987, originally as a utility programming language for the UNIX operating system, it has become popular for use in developing Common Gateway Interface (CGI) programs for web applications. Some features that account for this popularity include: Perl is free and available on the Internet; programming is relatively easy, especially for string manipulation; Perl provides a CGI module; and Perl applications are portable. Nevertheless, these attractive features bring with them security vulnerabilities.

A mitigating strategy for many of these vulnerabilities is the use of the special security mode called taint mode. In taint mode, Perl monitors all information entering the program from outside and issues warnings when it detects situations it considers

dangerous. User input, environmental variables and program arguments are targets of this monitoring activity. Data that is considered tainted cannot be used to do any of the following:

- Execute system commands
- Modify files
- Modify processes
- Invoke any shell
- Perform a match in a regular expression using the (? { … }) construct
- Execute any code using string eval

Invoking taint mode does not guarantee that the program is secure. Its purpose is to make it more difficult for the programmer to do something unsafe. Also note that taint mode is a run-time check. You should check your program carefully to make sure that turning on taint mode does not inhibit any intended normal processing. All logical paths of execution should be checked. For more detailed information about Perl security issues see website http://www.perl.com/CPAN/doc/manual/html/pod/perlsec.html.

Module #9: Common Criteria

In the United States, the National Institute of Standards and Technology (NIST) and the National Security Agency (NSA) have promoted security in commercial off-the-shelf IT products. NIST and NSA activities included working with government and industry to develop effective IT security criteria and to evaluate products developed by industry in response to those criteria for over 20 years. The Common Criteria (CC) is the culmination of the combination and consolidation of U.S. efforts with similar work in Canada, United Kingdom, France, Germany and the Netherlands to develop criteria for the evaluation of IT security that are widely used in the international community.

The U.S. activity is called the Common Criteria Evaluation and Validation Scheme (CCEVS) and is jointly managed by personnel from NIST and NSA. The purpose of CCEVS is to establish a national program for the evaluation of information technology products for conformance to the International Common Criteria for Information Technology Security Evaluation. The Validation Body approves the participation of security testing laboratories and provides technical assistance to those laboratories and validates the results of IT security evaluations for conformance to the Common Criteria. These evaluations are carried out in laboratories accredited by NIST. The Validation

Body assesses the results of a security evaluation, and when appropriate, issues a Common Criteria Certificate for the IT product being evaluated.

NIST and NSA have the following objectives in the operation of the CCEVS:

- To meet the needs of government and industry for cost-effective evaluation of IT products
- To encourage the formation of commercial security testing laboratories and the development of a private sector security testing industry
- To ensure that security evaluations of IT products are performed to consistent standards and
- To improve the availability of evaluated IT products.

The structure of the CC provides flexibility in the specification of secure products. Security functionality can be specified in terms of standard profiles and a scale of increasing evaluation assurance levels from EAL1 to EAL7 is available. The Protection Profile is defined for prospective consumers or product developers to create standardized sets of security requirements which will meet their needs. Examples of profiles now available include a commercial security profile template, a role-based access control profile, smart card profiles, a relational database profile and firewall profiles for packet filters and application gateways.

A more detailed description of the CC and related activities is available at http://niap.nist.gov/cc-scheme. The schedule for the regular meetings of the International Common Criteria Conference is also at this site along with a listing of certified products.

Course Summary

This course presents an introduction to software security with the objective of providing practical strategies for addressing security challenges. The risks of the popular "penetrate and patch" approach to software security along with the advantages of the recommended approach of integrating security considerations into the software development life cycle are discussed. Tools and techniques are presented that can enhance security at each stage of the life cycle as well as general principles for more secure design. Specific practices are recommended for programming in C/C++, Java and Perl. Security issues of smart cards and the certification of IT products are also discussed. Additional book and web resources are given for a more in-depth follow-up study.

Web Resources

Software Engineering Institute (SEI) http://www.sei.cmu.edu

SEI at Carnegie Mellon University is a federally funded research and development center sponsored by the Department of Defense. The SEI's purpose is to help others make measured improvements in their software engineering capabilities. The Software Engineering Information Repository (SEIR) is a community based web site that provides both information and the opportunity to participate in a free forum on software engineering improvement activities. Software Process Improvement Network (SPIN) is a network of individuals with an interest in improving software engineering practice. These individuals are organized into regional groups called "SPINs" that meet and share experiences. They meet annually at the Software Engineering Process Group (SEPG) Conference (http://www.sei.cmu.edu/collaborating/spins/spins.html) which is co-sponsored by the SEI and a regional SPIN.

SEI is also the location of CERT/CC, a group that researches Internet security vulnerabilities and issues security advisories in cases of large security risks. Website: http://www.cert.org.

Textbook and Other Resources

1. Davis, Noopur, "Developing Secure Software", *The DOD Software Tech News*, Vol. 8, No. 2, 2005.

2. Graff, Mark G. and van Wyk, Kenneth R., **Secure Coding: Principles and Practices**, O'Reilly and Associates, Inc., 2003.

3. Hansmann, Uwe, Nichlous, Martin S., Schack, Thomas, Schneider, Achim, Seliger, Frank, **Smart Card Application Development Using Java**, Springer-Verlag, 2002.

4. Howard, Michael and LeBlanc, David, **Writing Secure Code**, Microsoft Press, Second Edition, 2003.

5. Jaquith, Andrew, "The Security of Applications: Not All Are Created Equal", @Stake Research Report, February 2002. Available at website (http://www.securitymanagement.com/library/atstake_tech0502.pdf)

6. Kuperman, Benjamin A., Brodley, Carla E., Ozdoganoglu, Hilmi, Vijaykumar, T. N. and Jalote, Ankit, "Detection and Prevention of Stack Buffer Overflow", *Communications of the ACM*, 48, (November) 51-56, 2005.

7. McGraw, Gary and Felten, Edward W., **Securing Java,** John Wiley and Sons, Inc. 1999.

8. Seacord, Robert C., **Secure Coding in C and C++,** Addison Wesley, 2006.

9. Schneier, Bruce, **Secrets and Lies: Digital Security in a Networked World**, John Wiley and Sons, Inc., 2000.

10. Swiderski, Frank and Snyder, Window, **Threat Modeling**, Microsoft Press, 2004.

11. Trevor, Jim, et al, "Cyclone: A Safe Dialect of C", Proceedings of USENIX Annual Technical Conference, 275-288, Monterey, CA, June 2002.

12. Viega, John and McGraw, Gary, **Building Secure Software: How to Avoid Security Problems the Right Way**, Addison-Wesley, 2002.

13. Wall, Larry, Christiansen, Tom and Orwan, Jon, **Programming Perl**, O'Reilly, 3rd Edition, 2000.