atsec information security corporation
9130 Jollyville Road, Suite 260
Austin, TX 78759
Tel: 512-615-7300
Fax: 512-615-7301
www.atsec.com

Guidelines for Secure Coding

by Trupti Shiralkar and Brenda Grove
January 2009

Guidelines for Secure Coding

Writing secure code is an essential part of secure software development. Generally speaking, the main objective of a developer is to write code that will work. Unfortunately, neither a novice nor an experienced programmer necessarily knows how to write code securely. A developer's unintentional ignorance of known vulnerabilities and insecure coding practices can generate a security flaw in a program. Such a flaw could be responsible for crashing a program or enabling a denial of service attack. In any case, an effective approach must be adopted to detect and eliminate such flaws.

Practicing secure coding techniques helps avoid most of the software defects responsible for causing vulnerabilities and improves the quality of the software.

Programming languages such as C and C++ were designed for building both low- and high-level applications in the early 1970s. These languages were intended for high performance and ease of use; security was not a concern at that time. However, the growing number of published vulnerabilities and history of attacks points out various security flaws in C/C++ applications. Therefore, the main purpose of this paper is to provide a practical and effective set of secure coding guidelines.

Java was developed 1995 and is believed to be more secure than C/C++. Unfortunately, by practicing insecure coding constructs, a Java developer can still introduce security flaws in a program. Hence, this paper also covers secure coding guidelines for Java..

In order to develop a secure application, practicing secure coding techniques alone is not sufficient. Security should be incorporated into every phase of the software development life cycle. Therefore, this paper also provides an overview of a secure software development life cycle (secure SDLC).[1]

The intended audience for this paper is security professionals, as well as developers who are keen to understand various security aspects of code development.

Impact of vulnerabilities and associated cost

According to the National Institute of Standard and Technology (NIST), more than 15,000 vulnerabilities have been reported and added to the Common Vulnerabilities and Exposures (CVE) database to date. The growing number of reported vulnerabilities is causing increasing concern.

Poor development practice is significantly responsible for causing defects and creating vulnerabilities. Many times, the programmer fails to detect these vulnerabilities at an early stage in the development cycle. When the product is launched into the market, such vulnerabilities are discovered and reported by researchers. Vulnerabilities that are discovered and exploited by hackers/attackers can remain unnoticed until a considerable amount of damage has been caused. It then takes time to develop and release a patch to address a vulnerability. Once a patch is available, it must be configured on existing systems/applications to fix the security problem. However, this type of patch management system is not very effective for several reasons. First, it is a time-consuming process to develop and release the patch. Second, there is a possibility that a patch will not be compatible with an existing application/system and, hence, it cannot be deployed in a specific environment without testing. Failure to configure the new patch correctly can cause a critical security-related issue.

Overall, the impact of vulnerability and patch management includes the costs of:

- evaluation of the vulnerability
- patch development
- patch retrieval, assembly, and testing
- patch notification
- support for patch management issues
- downtime while applying the patch

[1] Note that atsec does not advocate the specific secure SDLC methodology described.

In fact, when all factors are considered, the cost to resolve a security issue by development and application of a patch is approximately 60 times the cost of fixing the security bug in an early stage of the SDLC [01].

There is an additional potential cost to a company that releases software containing security vulnerabilities. A pattern of such activity can result in damage to the company's reputation, to the detriment of the company's stock price and other measures of its value.

Known Vulnerabilities Introduced By Coding

In computer security, the term 'vulnerability' implies a weakness or a fault in design, development, or configuration which, upon exploitation, can violate an organization's security policy and allow unauthorized access to the attacker.

It is a good idea to have a working knowledge of software vulnerabilities which have caused significant damage in the last decade. This approach helps programmers to pinpoint vulnerabilities in existing code.

The following is a brief overview of some common vulnerabilities.

Buffer overflow

A buffer overflow occurs when a program allows input to write data beyond allocated memory. The attacker can gain control over an entire application or crash a program by exploitation via buffer overflow. The most commonly-affected languages are C and C++. Some languages, like Java, C#, and Visual Basic, have an array bound checking mechanism and native string types, which generally prohibit direct memory access. Hence, these languages are less prone to buffer overflows.

```
/** Example of Buffer Overflow  **/

int main (int argc, char const *argv[])
   {
      char buffer1[5] = "VXYZ";
      char buffer2[5] = "PQRS";
      strcpy(buffer2, argv[1]);
      printf("buffer1: %s, bufffer2: %s\n", buffer1, buffer2);
      return 0;
   }
```

In the example, the argument is copied into buffer 2 without checking its size. This flaw introduces a buffer overflow vulnerability.

Integer overflow

An integer overflow takes place when the integer variable tries to store a larger value than the valid range as a result of an arithmetic operation [02]. C and C ++ are unsafe languages and are likely to turn an integer overflow into a buffer overflow. Some languages, such as Java and Ada, implement a range-check integer type, which significantly reduces the danger.

```
/** Example of Integer Overflow  **/

short int number = 0;
char buffer[large_value];

while (number < MAX_NUM)
  {
     number += getInput(buffer+number);
  }
```

In the example, the integer variable 'number' may continuously create smaller values than MAX_NUM and would result in an integer overflow. This scenario will also overwrite MAX_Num-1 bytes of buffer.

Guidelines for Secure Coding

Format String Attacks

This attack takes place when user-supplied data of an input String is evaluated as a command by the application in such a way that the attacker can print data from the stack, execute arbitrary code, or disclose information. The malicious input might include format String parameters like %x or %n. The most vulnerable languages are C and C++.

```
/** Example of Format String Attack  **/

int main(int argc, char * argv[])
  {
        printf(argv[1]);
        return 0;
   }
```

In the example, if the input argument contains a string parameter like %x or %n , the program will print unexpected output. Use of printf(argv[1]) instead of printf("%s", argv[1]) introduces a potential format string vulnerability.

Command injection

A command injection takes place when malicious data is embedded into input and is passed to the shell (or a language interpreter). As a result, the input is interpreted as some sort of command. Programs written in any language that fails to perform proper input validation could be vulnerable to these types of attacks.

```
/** Example of Command Injection **/

int main(char* argc, char** argv)
 {
     char command[MAX] = "head ";
     strcat(command, argv[1]);
     system(command);
 }
```

In the example, the program executes the UNIX command '*head filename*' and displays the first 10 lines of a file which is entered as input. The 'strcat' concatenates the command with filename and the 'system' command executes it. Since there is no input validation, in this example the attacker can inject malicious command in input, e.g., ' hello.c; rm welcome.c'. The execution of this program will display the first part of hello.c and delete the welcome.c file.

Cross-site scripting

A cross-site scripting (XSS) vulnerability is generally found in web applications in which an attacker enters malicious data in such a way that it will bypass access control mechanisms. As a result, the malicious data is reflected to the client's web browser (non-persistent XSS) or stored on the server side (persistent XSS), and can be responsible for stealing sensitive information, such as cookies (DOM-based XSS). By exploiting this vulnerability, the attacker can deface websites, perform phishing attacks, inject malicious links into trusted web pages, or send confidential information to other untrusted websites. If a proper means of server-side input validation is not implemented, then any language used for building web applications (for example, PHP, C#, VB.Net, ASP.NET, J2EE, and Active Server pages (ASPs)) is vulnerable to these types of attacks.

```
/** Example of Cross-Site Scripting **/

<? php
   $input_text=$_GET['input_text'];
   echo " $input_text welcome to my page";
?>
```

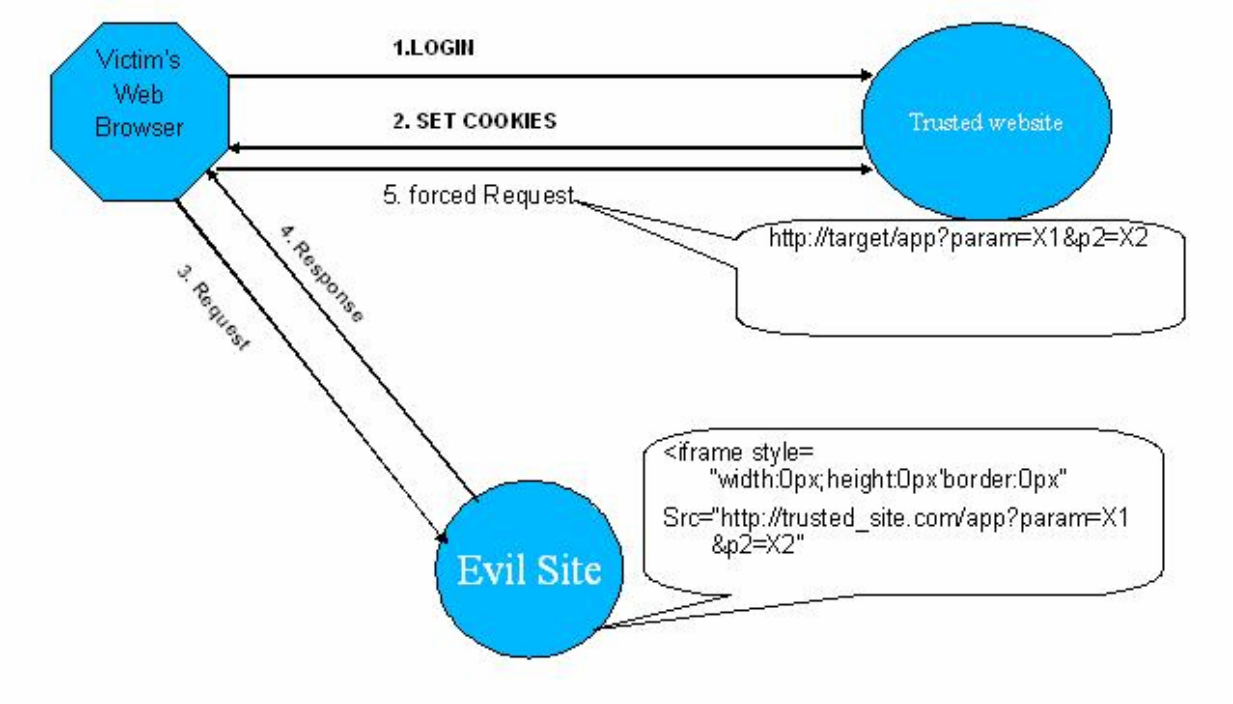
In the example, the code expects the name of a visitor to the web page as input and echoes it without input/output validation. This type of mistake could result in a cross-site scripting vulnerability.

Cross-site request forgery

A cross-site request forgery (CSRF) occurs when an end user is forced to execute unwanted actions in a web application to which he is authenticated. All web applications that use persistent authentication only (a cookie or a session identifier) could be vulnerable to such an attack. By conducting this type of attack, the attacker can execute all types of actions that would benefit him, for example transferring funds from a victim's account to the attacker's own account.

The following figure illustrates an example of a CSRF. In this scenario, the victim first sends login credentials to the Trusted website. Upon successful authentication, the Victim's Web Browser stores cookies sent by the Trusted website. Now, if the victim happens to visit the Evil Site, then the attacking site can launch a CSRF attack against the victim. Basically, it uses the victim's credential stored in the form of cookies and sends HTTP requests to the Trusted website. The website performs the requested action, as this request is authenticated based on the stored cookie.

CSRF attacks are based on the assumption that the Trusted website authenticates the user's browser and not the user [03].



SQL injections

SQL injections are SQL commands/queries which are embedded in user-entered data in such a way that they bypass access control mechanisms and can then display, add, delete, or manipulate data in the backend database. Upon exploitation, the attacker can steal any data in the database including personal identification data such as credit card numbers, social security numbers, names, addresses, and other sensitive information. Any programming language that is connected to a database is affected, for example, C#, PHP, ASP.NET, and JSP.

```
/** Example of SQL Injection **/

<form method="post" action="Login_Account.php">
        <input type="text" name="username">
        <input type="password" name="password">
</form>
```

The HTML snippet shown is an example of basic authentication. User-supplied credentials are sent to Login_Account.php. In the absence of input validation, it is possible to exploit this type of vulnerability by crafting a malicious input SQL statement, for example:

```
Select * from LOGIN where username='john_smith' and password = ' ' or 1=1;
```

In the example, the SQL interpreter will check only for username, as the second half of the query (or 1=1) is always true.

Insecure direct object reference

A 'direct object reference' vulnerability occurs when the programmer does not use authorization and exposes a reference to an internal implementation object such as a URL, form parameter, file, directory, or database record [03]. An attacker could manipulate these references to gain access to sensitive information. All languages used to develop web applications are vulnerable to this type of attack.

```
/** Example of Insecure Direct Object Reference **/

http://www.abc.com/resources/accounts/information/getinfo.jsp?padeId=help.html
```

This URL shows HTTP request made to view help page. If the developer of site didn't implement proper authorization, then attacker can jump to any location in directory tree by performing directory traversal. For example:

```
http://www.abc.com/resources/accounts/information/getinfo.jsp?pageId=.. /.. /.. /imp_file
```

An attacker can use '.. / ' to jump from the current location to the target directory to access some important file/resource which he is not authorized to access.

Improper error handling and information leakage

When a programmer fails to implement proper error handling, the application might leak information about its internal state, system configuration, or resources (for example, Apache Web Server 2.2.8). This information could be used to attack the system. Improper error handling can crash, abort, terminate, or restart the program, all of which could result in potential denial of service. If improper error handling occurs, programming languages such as C#, VB.NET, and Java, which use exceptions; languages like C,

C++, ASP, and PHP, which use function error return values; and all other programming languages are vulnerable to information leakage.

```
/** Example of Improper Error handling and information Leakage **/

404 Not Found
Not Found
The requested URL /abc/xyz_help/ was not found on this server
Apache/ 2.2.3(Debian) PHP/5.2.0-8+etch13 mod_ssl/2.2.3 OpenSSL/0.9.8c server at abc.pqr.de port 80
```

In the example, the error message reveals important information about the web server, operating system, port number, and other products used.

Insecure storage and improper use of cryptography

Storing passwords, keys, certificates, and other confidential information in clear text could lead to insecure storage. Use of short keys(DES uses 56-bit keys), weak algorithms (for example, RC4 and MD5), non-standard algorithms, cryptographically-weak random numbers in a pseudo random number generator (PRNG), improper key management (for example, keys stored at an insecure location), and hard coding of keys could cause a breach of sensitive information. Some common protocols like HTTPS include negotiation of cryptographic algorithms and might allow use of insecure algorithms.

Time of check vs. time of use (race condition)

A race condition occurs when the resource is checked for a particular value, the value is changed, and then the resource is used assuming that the value is still the same as it was at the time of check. An attacker can use a race condition to bypass the security check or corrupt the resource in an undesirable way.

Secure Software Development

'Secure software development' is a proactive process of designing, building, and testing software that incorporates security into each phase. It aims to reduce security risks introduced at each stage of the secure software development life cycle (secure SDLC). The following section gives an overview of the phases of secure SDLC [04, 05].

Secure architecture and design

The main goal of the 'Security architecture and design' phase is to reduce the number of vulnerabilities before development starts. It is an iterative review process to analyze the architecture and design of an application from a security perspective. The review includes an analysis of critical areas of the application/product responsible for integrating basic security principles; for example, authentication, authorization, confidentiality, and integrity. It seeks to understand the security needs of a user, as well as the product.

Remember that it is easier and more cost-effective to eliminate security flaws at the design level than in any other phase of the secure SDLC.

Threat modeling

Threat modeling seeks to describe and develop the pertinent threats. The first step is to identify threats that might damage the application or product, and then to identify the vulnerabilities responsible for the threats. The next step includes analyzing the potential attacks caused by the threats and the planned control measures taken to mitigate the likelihood of such attacks.

Secure design consideration

This phase integrates solutions to all possible potential threats into the design itself. The basic concept behind each vulnerability and attack needs to be understood in order to create a general secure design. Then, the design can be constructed keeping in mind all the security prerequisites of the application.

Developers can also make use of 'Security Design Patterns [06]' to deal with security-related issues and solve known security problems.

Secure coding practices

The major aim of the 'Secure coding practices' phase is to augment awareness about software security among the developer community. A program written by following secure coding standards is much more secure than a program that does not follow any such standard. It might sound like extra work, but following secure coding standards not only reduces the probability of introducing vulnerabilities, but also reduces accidental inclusion of other flaws.

Please refer to the "Secure Coding Guidelines" section for a detailed discussion of secure coding techniques.

Software security testing

Software security testing is an indispensable phase of a secure SDLC as it performs sanity checking before release of the code. It is a process of discovering security flaws in given code. The following are approaches to conducting security testing.

Guidelines for Secure Coding

Code review

Code review is a process of software security testing in which the developer of the program, peer reviewers, and a team of quality assurance testers perform review of code together. Code review can be manual or automatic. In case of peer review, software developers can check each other's code to find security bugs (along with other types of bugs) before handing the code to the QA team.

In addition to peer code review, it is a best practice to engage a third-party with special expertise in security to perform source code review before your application ships.

Prepare and maintain documentation of all findings and errors resulting from code review. The following are a few guidelines for code review:

- Look for common vulnerabilities discussed in section 2.
- Detect system calls.

The following table shows some system commands that could be used for hostile purposes [02]:

Language	Construct/Procedure	Comment/Description
C/C++	System(), popen(),execlp(),execvp()	Portable Operating System Interface (POSIX).
C/C++	_wsystem(), the ShellExecute() family of function	Win32 only.
Perl	System, exec	If called as one argument, and if the string has shell meta-characters, it can call the shell.
Perl	Backticks	Can call the shell.
Perl	Open	If the first or last character of the filename is a vertical bar, Perl opens a pipe instead. The rest of the filename is treated as data passed through a shell.
Perl	Vertical bar	Acts like a popen()call.
Perl	Eval	Executes a string argument as Perl code.
Perl	Regular Expression/ e operator	Evaluates a pattern-matched portion of a string argument as Perl code.
Python	Exec, eval	Data gets evaluated as code.
Python	os.system, os.open	These assign to underlying POSIX calls.
Python	Execfile	Takes data from file and gets evaluated as code.
Python	Input	Similar to eval().
Python	Compile	The intent of compiling text into code is to show that it's going to get run.
Java	Class.forName(string), Class.newInstance().	Java bytecode can be dynamically loaded and run.
Java	Runtime.exec()	If called with an argument, it can explicitly invoke a shell.
SQL	xp_cmdshell	This procedure allows execution of any OS command.

- Use source code review tools to examine source code and find security flaws. For example, the PREfast analysis tool uses statistical analysis to find defects in C/C++, and Flawfinder uses a built-in database of C/C++ functions with well-known problems.

Penetration testing

A penetration tester performs black box testing to evaluate the security measures of the application. He has no knowledge of the source code or architecture of the application. A third-party pen-tester can conduct penetration testing. Results of this assessment process should be documented and presented to the concerned party.

Fuzz testing

Fuzz testing means testing the application against malformed data to see how it reacts. If the application fails, then a new bug should be reported and updated in the threat model. Every time the application is changed or updated, fuzz testing should be conducted.

Fuzzing techniques use black box testing. It allows detection of most of the common vulnerabilities, for example, buffer overflow, cross-site scripting, and SQL injections.

Secure Coding Guidelines

Secure coding techniques include both general guidelines that can be used to improve software security no matter what programming language is used for development, and techniques relevant to specific programming languages.

General guidelines

- Efficient input validation is mandatory. As mentioned in section 2, most vulnerabilities are the result of absent or inadequate input validation. Good practice suggests sanitizing every user-entered input. Best practice is to create a 'white list' of expected known good input parameters and formats, rather than relying on a 'black list' of known bad inputs.
- Practicing sound software design and other phases of software engineering facilitates a structured, small, and simple code [08]. In addition, use a secure coding checklist.
- In order to track changes made to the code or document, use version/configuration control; this enables easy rollback to a previous version in case of a serious mistake. Version control facilitates accountability and saves development time.
- Never trust the input to SQL statements. Use parameterized SQL statements. Do not use string concatenation or string replacement to build SQL statements.
- Use libraries (e.g., anti-cross site scripting library) to protect against security bugs during web application development. Test the code with a web application scanner to detect vulnerabilities.
- Good practice also recommends use of the latest compilers, which often include defenses against coding errors; for example, GCC protects code from buffer overflows.
- As mentioned earlier, security design patterns can be used to tackle similar security-related concerns and provide solutions to known problems.
- It is good practice to code with proper error/exception handling. Check the return values of every function, especially security-related functions. Also, check for leakage of sensitive information to untrusted users.
- Make a security policy that prohibits the use of banned functions that make the code weak. Encourage a process of peer code reviews and sound security testing, as explained in the "Software security testing " section.
- Encode HTML input. Attackers use malicious input to conduct XSS types of attacks. Encoding of every user-supplied input can prevent the client's web browser from interpreting these as executable code. Do not store sensitive data in cookies.
- Encrypt all confidential data using strong cryptographic techniques. Handle key management carefully. Use a published and strong cryptographic algorithm with a sufficiently long key. Use of FIPS-approved cryptographic algorithms is encouraged. Do not use security protocols with inherent cryptographic weakness (e.g., SSL V2) and cryptographically weak random numbers.
- Using secure coding practices includes keeping informed about known and new vulnerabilities and software bugs by reading security-related forums, magazines, research papers, and newsletters.
- Every organization must educate its developers on how to write secure code, for example by offering a seminar or training session. The training should cover strategies and best practices to mitigate common threats. Additional emphasis should be given to the security features of programming languages and how to implement those features to build a secure application.

Guidelines for Secure Coding

Programming language-specific guidelines

This section explains secure coding rules and recommendation specific to Java and C/C++. These programming languages were selected because they are used to build many critical infrastructures.

Secure coding practices in Java

Java has implemented many security features, for example, type safety, automatic memory management, array bound checking, byte code verification, and signed applets to mitigate many vulnerabilities, such as buffer overflows and integer overflows. To reduce the probability of security vulnerabilities introduced by a Java programmer, the following guidelines are recommended:

- Validate input: To avoid command injections, overflow conditions, and format string attacks, always make a safe copy of input and validate it.
- Limit accessibility and extensibility: Initially declare all classes, methods, interfaces, and fields as private. Change the access type to protected or public only if necessary. Declare the classes, methods, and fields final. In case of inheritance, change the declaration to non-final. To limit access to a public, non-final class, impose a SecurityManager check for each subclass.
- Understand the effect of a superclass on a subclass: Changes made to a superclass can affect the behavior of a subclass and introduce vulnerabilities [07]. These changes need to be well understood.
- Create a copy of instances of a mutable class, mutable inputs, and outputs: Create a copy of instances of a mutable class to ensure that instances are securely passed to (and returned from) methods in other classes [07]. If the method is not meant to deal with mutable inputs and outputs directly, create a copy of it. Otherwise, an unsafe caller can modify the output to invoke a race condition.
- Define Java wrapper methods around native methods: Unlike Java code, native methods are not safe from buffer overflows and are not supervised by the SecurityManager. Hence, declare the native method as private and wrap it in a Java-based public method which can be used to conduct input validation and a SecurityManager check [07].
- Access internal modifiable state via a wrapper method: A publicly accessible internal modifiable state should be declared as private. Define a public wrapper method around it to enforce a SecurityManager check. Define a protected wrapper method if the internal state is accessed only by subclasses [07].
- Use public static fields for defining a constant: Use a public static field for storing an immutable value and declare it ‘final’ so that its value cannot be altered.
- Sanitize exceptions: Check for exceptions that leak internal sensitive information. Catch and throw such exceptions with a sanitized message.
- Avoid using security-sensitive classes: A security-sensitive class, for example, ClassLoader, can circumvent SecurityManager access control. Therefore, enforce a SecurityManager check at all points where the class is instantiated.
- Make sure that an instance of a non-final class is fully initialized: Ensure that an instance of a non-final class is not partially-initialized, because an attacker can access it. The developer can set an initialized flag (Boolean type) at the end of a constructor before returning successfully.
- Prevent constructors from calling a method that can be overridden: Do not call methods that can be overridden from a constructor, since it gives reference to the object being constructed before the object is fully initialized [07].

Guidelines for Secure Coding

- Be careful about serialization and deserialization: Attackers can access private fields of a serialized class (Java access control cannot be enforced); hence, do not serialize sensitive data in a serialized class [07]. In case of default serialization, declare the sensitive data as transient.

  Check input validation; assign default values that are the same as those assigned in the constructor during deserialization, because it creates a new instance of a class without calling the constructor.

  Enforce SecurityManager checks in serialized classes and during deserialization.

- Stay alert for standard APIs and methods that can bypass access control or security checks: Do not call AccessController.doPrivileged using tainted input, because it allows the code to apply its own permissions during a SecurityManager check.

  Methods such as Class.newInstance could bypass the SecurityManager check, depending on the immediate caller's class loader. Do not invoke the following methods on Class, ClassLoader, or Thread instances provided by untrusted code.

```
java.lang.Class.newInstance
java.lang.Class.getClassLoader
java.lang.Class.getClasses
java.lang.Class.getField(s)
java.lang.Class.getMethod(s)
java.lang.Class.getConstructor(s)
java.lang.Class.getDeclaredClasses
java.lang.Class.getDeclaredField(s)
java.lang.Class.getDeclaredMethod(s)
java.lang.Class.getDeclaredConstructor(s)
java.lang.ClassLoader.getParent
java.lang.ClassLoader.getSystemClassLoader
java.lang.Thread.getContextClassLoader
```

  Do not invoke any of the following methods to execute using the immediate caller's ClassLoader instance.

```
java.lang.Class.forName
java.lang.Package.getPackage(s)
java.lang.Runtime.load
java.lang.Runtime.loadLibrary
java.lang.System.load
java.lang.System.loadLibrary
java.sql.DriverManager.getConnection
java.sql.DriverManager.getDriver(s)
java.sql.DriverManager.deregisterDriver
java.util.ResourceBundle.getBundle
```

  The Java Virtual Machine performs language checks whenever an object accesses methods and fields. Using standard APIs (such as reflection API), access control rules can be bypassed. Be careful when using these APIs.

- Use char array or byte array instead of String() to store passwords: String() is immutable. Even if it is deleted, its object keeps floating in memory. Therefore, use char array or byte array instead of String and at the end of the method, use the following method to clear passwords and sensitive information.

```
/** Clear password **/

        private void deletePassword(char[] password)
        {
                for (int i=0; i<password.length; i++)
                password[i]=' ';
                }
```

- Be cautious when dealing with multiple threads [09]: If the code is using multithreading without proper synchronization, it is possible that all threads will manipulate the same data simultaneously, resulting in an unpredictable behavior of the program.

  Make sure that the code is synchronized to handle multiple processes. Organize code in such a way that one piece of data is manipulated by one thread at a time.

  Use semaphore to indicate data is ready for another thread's consumption.

  Use the paint method for all your drawings, or use the paint method to set flags that indicate objects to be painted, and then use run loop to draw.

Secure coding practices in C/ C++

C/C++ is widely used because it is easy to understand, fast and flexible. However, 'C 'is a fundamentally unsafe language. Many of its library functions are vulnerable to buffer overflows and format string type attacks. The following are recommended guidelines to develop secure C/C++ programs:

- Perform input validation: Never trust user-entered data/input. Perform input validation to avoid security flaws like buffer overflows, integer overflows, and format strings.
- Use pointers safely: Developers need to be careful about proper use of pointers to avoid the following security flaws:
  - Pointers forging: Occurs when a language's liberal casting policy is exploited and pointers are misused to access, duplicate, or modify object members that are protected by the language's access control mechanisms.
  - Pointers arithmetic: Allows dynamic memory allocation to an array. However, this provision can be misused to read into surrounding memory locations, with unintended consequences.
  - Dangling pointers: When the object is deleted (or de-allocated) without changing the value of the pointer associated to it, the pointer continues to point to the location of de-allocated memory and is known as a dangling pointer. If the program writes data to the de-allocated memory pointed to by a dangling pointer, it might corrupt the unrelated data and lead to a segmentation fault. An attacker could remotely exploit this vulnerability to crash the system.
  - Wild pointers: Occurs when a pointer is used without first initializing to some state. These pointers are difficult to detect and result in the same unpredictable behavior as that of dangling pointers.

  Enforce initialization of pointers to remove wild pointers.

  Once the memory is free, set the pointer to a null pointer or an invalid address. The developer can use tools mentioned in the following section to detect pointer-related dangers.

The best way to avoid runaway pointers is (no surprise) to be very careful when using pointers. Instead of iterating through an array with pointer arithmetic, use a separate index variable, and assert that the index is never larger than the size of the array.

- Watch out for memory leaks: Memory leaks are a primary source of C++ application crashes and core dumps. This can happen when object instances created during runtime are not de-referenced by setting them to null after the program terminates. This causes chunks of allocated object pointers to remain in memory, causing tiny memory leaks that can accumulate over a sustained period of time and lead to unexpected crashes and termination of the program.

  Run a 'Garbage Collector' to free the memory used by objects that are no longer referenced, for example, system.gc (in C), and delete (in C++).

- Type confusion: Type confusion occurs when an operation defined on a superclass is called on an instance of a subclass. This generally occurs due to non-strict type checking within the language. In this example, object ORDER is an instance of class ORDER. However, it is forcibly typecast to a char* pointer, which confuses the compiler as to the type of object named ORDER.

- Check for access violation: C++ is only partially object-oriented, and hence, access control mechanisms are not as strong as in Java. The access protection defined by keywords 'public' and 'private' is easily overridden and objects of class can easily reference their private variables.

  As mentioned earlier, another type of access violation occurs when a freed block of memory is referenced, causing the program to throw an error such as "Access violation thrown ERROR 0x00005434". This is called a memory access violation and occurs when a pointer attempts to reference a location that has already been freed.

- Avoid using dangerous functions: Avoid using dangerous functions which do not perform bound checking, such as strcpy(), strcat(), sprintf(), vsprintf(), and gets(). These functions need to be replaced by strncpy(), strncat(), snprintf(), and fgets() .

  Functions scanf(), scanf(), fscanf(), sscanf(), vscanf(), vsscanf(), and vfscanf() are also dangerous, but can be used safely by controlling the length.

  Functions which may allow buffer overflow are: realpath(3), getopt(3), getpass(3), streadd(3), strecpy(3), and strtrns(3).

- Manage String dynamically: To make sure that String operations will not result in a buffer overflow, allocate memory dynamically. Use calloc() instead of malloc().

- Implement range checking to protect from integer vulnerability: Lack of range checking for integers can result in overflow, truncation error, or sign error. To avoid this vulnerability, implement range checking as part of input validation.

- Securely delete sensitive data from memory: In order to erase data securely, declare the variable as volatile.

- Prevent sensitive memory regions (storing key for example) from being swapped to the disk: Make use of system call (mlock() in UNIX systems and VirtualLock() in Windows systems), which prevent locked memory from being swapped to the disk.

Summary

Security vulnerabilities in software products are of increasing concern to consumers and to NIST, which monitors and publicly reports such vulnerabilities. The cost to an organization that releases software containing security vulnerabilities is very high. Avoiding introduction of such vulnerabilities requires awareness and commitment on the part of an organization that develops software applications. This paper offers some effective strategies in pursuit of this goal.

It is important that developers are familiar with known security vulnerabilities so that they can avoid writing code that is exploitable by already-discovered vulnerabilities. This paper offers an overview of known security vulnerabilities attributable to faulty coding practices.

We know that the sooner a vulnerability is discovered, the easier and cheaper it is to fix it. Adopting a secure software development life cycle (secure SDLC) that considers security at every stage of development contributes to early identification of potential vulnerabilities. This paper offers an overview of secure SDLC, including discussion of activities at each phase of the development cycle.

Developers need to have a sound understanding of secure coding techniques in order to minimize security bugs and efficiently develop secure applications. This paper offers guidelines for secure coding, including both general guidelines and techniques specific to Java and to C/C++.

If you have questions regarding secure coding, source code review or analysis, security training, or any other IT security-related topic, we'd be happy to assist you. atsec's consultants are experienced IT security professionals with wide-ranging experience across many industry sectors, and we work particularly hard to ensure that our knowledge of software weaknesses, loopholes and vulnerabilities is always up-to-date. You can rely on atsec's expertise in the IT security business as the right resource to help you understand the issues and implement an effective strategy to improve the security of the software applications you develop.

References

01. Kevin Soo Hoo, Andrew W. Sudbury and Andrew R. Jaquith, 'Tangible ROI through Secure Software Engineering', 2006.
02. Michael Howard, David LeBlanc and John Viega, '19 Deadly Sins of Software Security', 2005.
03. Andrew van der Stock, Jeff Williams, Dave Wichers OWASP top 10: The 10 most critical web application security vulnerabilities, 2007.
04. Noopur Davis, 'Secure Software Development Life Cycle Processes: A technology Scouting Report',2006.
05. Michael Howard, Steve Lipner , 'The Security Development Life Cycle', 2006.
06. Erich Gamma, John Vlissides ' Design Patterns CD', 1995.
07. Sun Microsystems, Inc., 'Secure Coding Guidelines for the Java Programming Language, version 2.0', 2007 <http://java.sun.com/security/seccodeguide.html>.
08. Mark G. Graff, Kenneth R. van Wyk, 'Secure Coding Principles, and Practices', 2003.
09. Dave Dyer, 'Can Assure save Java from the perils of multithreading', Java World <http://www.javaworld.com/javaworld/jw-10-1998/jw-10-assure.html>.