Requirements of a Better Secure Program Coding

Marius POPA
Department of Economic Informatics and Cybernetics,
Academy of Economic Studies, Bucharest, Romania
marius.popa@ase.ro

Secure program coding refers to how manage the risks determined by the security breaches because of the program source code. The papers reviews the best practices must be doing during the software development life cycle for secure software assurance, the methods and techniques used for a secure coding assurance, the most known and common vulnerabilities determined by a bad coding process and how the security risks are managed and mitigated. As a tool of the better secure program coding, the code review process is presented, together with objective measures for code review assurance and estimation of the effort for the code improvement.

Keywords: *Secure Coding, Code Vulnerabilities, Code Review*

1 Requirements of the Secure Software Development Process

Reducing the software vulnerabilities and increasing the software protection against the cyber-attacks are critical to increase the confidence in software products. Details and issues regarding information and computer systems, the security and objective assessments of these systems are presented in [1], [6], [8], [10], [13], [16] and [17].

To achieve the above goals, specific activities have to perform during the software development life cycle. The software developers, integrators, operators and end users are factors that lead to improved software products.

There are many software development methodologies. According to [21], the common elements of these methodologies and their characteristics are:

- *concept* – the software goals, interactions with the users and other components of the IT infrastructures are defined;
- *requirements* – they are established in a measurable, observable and testable form; the software functionalities are defined together with the impact on the next stages of the software development life cycle;
- *design and Documentation* – it is a critical stage for an efficient programming; also, a very detailed documentation is the main source for the documentation of the released product;
- *programming* – design and specifications are translated into code; best practices and according to standards are required for an effective program coding;
- *testing, Integration and Internal Evaluation* – coding process is verified and validated as completeness, covering of the requirements, test plans and documentation;
- *release* – potential clients can use the available software product; the software vendor assures the software marketing and distribution;
- *maintenance, Sustaining Engineering and Incident Response* – they aim the support for the software product as bug fixes, user interface enhancements, modifications regarding the usability and performance; the support elements are used for new versions of the software product.

In [21], best practices for secure software application during its development life cycle are highlighted and defined as it follows:

- *security Training* – includes the activities regarding the security and privacy issues addressed to the software development team; it is a requirement given by the complex tools used in software development allowing the security vulnerabilities to creep in the software product;
- *defining Security Requirements* – is made in the same time with the software development; security requirements address architecture and design, software development and programming best practices, and assurance, testing and serviceability; for a better effectiveness, the security requirements are integrated into a framework to implement the security requirements traceability into the software development life cycle;
- *secure Design* – permits the identification and addressing the potential threats and the ways to reduce the risk to the acceptable or manageable level; the goal is to develop a

software product designed to be secure; for a better effectiveness, abstract designs are built as secure design patterns as descriptors used in many different situations; using the secure design patterns reduces or eliminates or mitigates the effects of the accidental insertion of vulnerabilities into code;

- *secure Coding* – implies the using of the best programming practices to build a secure software product; it is performed by source code review as combination of manual methods and/or automated analysis tools to identify the potential security defects that are exploited as software vulnerabilities; software vulnerabilities are reduced or eliminated by identifying the coding errors and defects before the software deployment;
- *secure Source Code Handling* – aims all measures to assure the access control to the source code, tracking and confidentiality protection of the source code; without a management of the source code handling, malicious code can be intentionally or unintentionally inserted in the software product; also, the development team manager has to know the traceability of the source code written by software developers;
- *security Testing* – represents a specialized validation of the security requirements, design and coding best practices are covered by the software product; at implementation time, the security testing implies vulnerability analysis, penetration testing, and use of testing techniques; the following concepts have to be met by a secure software: confidentiality, integrity, availability, authentication, authorization and non-repudiation;
- *security Documentation* – helps the software customers to configure the security controls; the configuration options have to be established in such a way to prevent explosion of the potential security vulnerabilities of the software product;
- *security Readiness* – is the final check of the software product made by developer; security issues are evaluated, documented and assessed by developer before to proceed with software product releasing;
- *security Response* – manages the vulnerabilities exploited by attackers or identified by customer in released software product; the software vendor or developer communicates with the customer and investigates the vulnerability to mitigate the risk; as result, a patch for the released software can be provided to the customer; also, the risk mitigation is made by a automatic patch policy for released software product;
- *integrity Verification* – is used to verify the legal and correct software utilized by customer; in such a way, a compromised or corrupted software product can be easily identified and the risks of using this kind of software are mitigated;
- *security Research* – helps to identify new attack ways of the existing technologies, new risk mitigation methods and techniques, and to adapt the new technology for an improved software security;
- *security Evangelism* – aims the promoting of the best practices by software assurance specialists in open forums, articles, papers and books.

The best practices applied during the software development life cycle contribute to software assurance. The software assurance is a process continued by software integrators, operators and end users after the software product development. Description of these roles is made in [21] as it follows:

- *Integrators* – work with software vendors and developers to identify and mitigate the vulnerabilities resulted from increasingly higher information system environments and integration of the new application with other software products and legacy systems; within integrated information system, the component subsystems have to work together in such a way to mitigate the vulnerabilities;
- *Operators* – configure the system in an optimized manner from security point of view; configuration options aim software patching and defense measures; operators manage security, user access and monitor and perform routine operations in the information system;
- *End Users* – are responsible users of the information system; they have to report potential bugs and vulnerabilities, and to prevent introduction of software from untrusted sources into the system.

The software assurance is implemented by applying the best practices during the software development life cycle. The best practices are separated depending on the role of each actor in the software development life cycle.

2 Methods and Techniques for a Secure Program Coding

The secure code is an artifact meeting the following requirements as it is presented in [3]:

- covers all technical and functional software specifications;
- remains within trust boundaries;
- validates all inputs in the proper way;
- escapes all outputs in the proper way;
- does not hardcode personal or sensitive data;
- does not violate any security standard specification and recommendation;
- cannot be hacked.

The program coding is a stage of the software development life cycle in which the design requirements of the software are implemented into the source code written in programming language. The design requirements include security issues to address the software product security.

The secure program coding is a process aiming the increase of the code quality and decreasing the security risk. It is implemented by the following activities as it is shown in [3]:

- strategy and metrics;
- policies and compliance;
- education and guidance;
- threat assessments;
- security requirements;
- secure architecture;
- design reviews;
- code reviews;
- security testing;
- vulnerability management;
- environment hardening;
- operational enablement.

The secure program coding refers the elimination of software defects during the implementation of the design requirements into the source code. This goal is accomplished by using the best programming practices. According to [20], the best programming practices are:

- minimization of the unsafe function use – eliminates the buffer overrun vulnerability from C and C++ code; the vulnerability cause is given by unsafe string- and buffer-copying functions at run-time;
- using of the latest compiler toolset – offers defense for buffer overrun at compile-time and run-time; the defense tools of the compiler are: stack-based buffer overrun support, image and stack randomization, CPU-level No-eXecute support, exception handler protection, warnings for insecure C runtime function detection and removal;
- using of static and dynamic analysis tools – aids source code and binary analysis to find vulnerabilities; they are complementary to manual reviewing; these tools are used on large amount of results containing many false positives;
- manual reviewing of the code – is made on high-risk code; it addresses to the following classes of vulnerabilities: buffer overruns and integer arithmetic issue, web vulnerabilities, database vulnerabilities and cryptographic issues;
- validation of input and output – eliminates the most common vulnerabilities; it establishes whether the format of incoming data is correct applying validation procedures on strings for text and XML data as string comparisons or verifying the data length and field validity for binary data; validation of output data mitigates vulnerabilities as cross-site scripting, HTTP response splitting and cross-site request forgery;
- using of anti-cross site scripting libraries – is very useful to encode the web-based output;
- using of canonical data formats – derives a canonical expression from polymorphic expressions; canonical data format is filtered by the security mechanisms;
- avoiding the string concatenation for dynamic SQL – eliminates building of SQL statements by concatenation of untrusted data with string constants; instead of concatenation, a better way to build SQL statements is to use placeholders or parameters;
- elimination of the weak cryptography – aims the insecure cryptographic entities to be used within software applications; it is better to use cryptographic algorithms and implementation proven to be secure by security standards;
- using of the logging and tracing – implies data recording for successful and failed events, and bug detection in the software application.

The security of software during its development life cycle is improved by establishing the coding guidelines for commonly used programming languages. For example, secure coding standards are:

- ISO/IEC 9899:1999 for C programming language;
- ISO/IEC 14882:2003 for C++ programming language;
- Java Platform Standard Edition 6.

In the above secure programming standards, there are defined recommendations, rules and risk assessment summaries grouped into categories.

For instance, the attempt to modify string literals in C is a rule defined in the category *Characters and Strings*. The rule explains the concept of string literal, when and how a string literal is created, what are the noncompliant and compliant code cases. For example, attempting to modify a string literal has the following noncompliant content in C programming language [24]:

```
char *p = "string literal";
p[0] = 'S';
```

The **char** pointer **p** is initialized to the address of the string literal. The second source code line causes undefined behavior of the program because the location **p[0]** cannot be overwritten by the symbol **S**.

The compliant solution is to use an array instead of pointer. A copy of the string literal is stored in the space allocated to the character array. The string stored in **a** is safely modified in the second line of the below source code.

```
char a[] = "string literal";
a[0] = 'S';
```

Another common example of insecure program coding aims the buffer overflow. Buffer overflow occurs when a program allows writing the allocated memory by data having longer length than the allocated memory one. Thereby, an attacker gains the control or crashes a program. The most affected programming languages are C and C++. In other languages, the array length checking is performed and native string types are used [15].

In the below example, the argument is copied into buffer without checking its length. This is the buffer overflow vulnerability.

```
int main (int argc, char const *argv[])
{
        char buffer[4] = "ABC";
        strcpy(buffer, argv[1]);
        printf("bufffer: %s\n", buffer);
        return 0;
}
```

Integer overflow is a coding vulnerability causes by the limited range of the values for program variables defined on standard data types of the programming language. Integer overflow occurs when the developer tries to store in the memory area a value outside the range.

In the below example, the variable **v** is defined by the **int** data type and it is initialized with the maximum possible positive value for a long integer.

```
void main(){
        int v;
        v = 0x7fffffff;
        printf("val = %d \n", v);
        v = v + 1;
        printf("v + 1 = %d \n", v);
}
```

The initializing value of variable **v** is given in hexadecimal which means **2147483647** in base 10. The **int** data type has the **long** specifier which means 4 bytes reserved in stack area at compiling time. The first figure of the values stored by **v** is **7**, having the following binary representation: **0111**. The **long int** is a signed data type which means that the most significant bit is the sign bit. For previous binary representation, the most significant bit has the value 0, the value stored by **v** being a positive one.

When the value stored by **v** is incremented by value **1**, the result has the hexadecimal representation **0x80000000**. The first figure of the representation is **8** which means the binary representation **1111**. As result, the variable **v** permits accessing of an negative integer in contradiction with the developer's expectations.

Format strings are used by an attacker to print data from the stack memory allocated for the process, execute arbitrary code, or disclose information [15]. The format strings control the behavior of the **printf()** family functions. The main problem is that the compiler does not detect the lacks of the format for **printf()** family functions. As result, an attacker removes the format strings, or arguments that have to be associated with the format strings.

Inappropriate uses of **printf()** family functions are highlighted in the below program code:

```
void main(int argc, char * argv[])
```

```
{
        char text[] = " string literal ";
        printf(argv[1]);
        printf(text);
        printf ("%s%s%s%s");
}
```

A program containing the above coding vulnerability crashes at run-time.
Command injection occurs when an application accepts untrusted or insecure inputs. Programs without validation or a proper escaping of the inputs are vulnerable to these types of attacks [19].
The following program code is injectable:

```
int main(char* argc, char** argv) {
    char cmd[CMD_MAX] = "/usr/bin/cat ";
    strcat(cmd, argv[1]);
    system(cmd);
    return 0;
}
```

The above C code is exploitable in computers using UNIX operating systems. The program attaches a filename passed as command line argument to the string stored by variable **cmd**. The program has root privileges and **system()** function executes with root privileges. The application runs appropriately when in **argv[1]** a filename is passed. However, other strings can be passed to the application in such way that the program causes damages in the computer.
The same coding vulnerability is exploitable in cross-platform programming languages like Java which implements linking ways to the native code. Such methods are **java.lang.Runtime.exec** and **java.lang.Runtime.getRuntime**.
Sending malicious code from web application to end-user using a form in a browser side script is cross-site scripting coding vulnerability. The vulnerability has the following types of behavior [15]:

- Non-persistent cross-site scripting – malicious code is reflected to the client's web browser;
- Persistent cross-site scripting – malicious code is stored on server side;
- Document Object Model based cross-site scripting – the client side code executes in a different manner due to modifications in the DOM environment.

Document Object Model is represented by objects provided by browser to the JavaScript code when the JavaScript is executed at the browser.
A simple example of cross-site scripting vulnerability is provided in the below example, according to [23].

```
<% String eid = request.getParameter("eid");
%>
        ...
        Employee ID: <%= eid %>
```

The code is provided for a web application developed in Java Server Page. The JSP code operates correctly when **eid** contains standard alphanumeric text. When **eid** has an inappropriate content as meta-characters or source code, the application is vulnerable.
The web applications developed in different technologies are vulnerable to this type of attack when input validation mechanisms are not implemented.
Cross-site request forgery consists of forcing an end-user to execute unwanted actions on a web application in which the end-user is authenticated [19]. Thus, an attacker compromises user data and operation. The web applications using the authentication on cookies or session identifier are vulnerable to cross-site request forgery. The attacker obtains and uses the login credential to force requests to the trusted site where the victim has a login account and operations can be made by attacker on behalf of victim.
An example of cross-site request forgery, in [19] is provided a HTTP POST request.

```
POST
http://TicketMeister.com/Buy_ticket.htm
HTTP/1.1
Host: ticketmeister
User-Agent: Mozilla/5.0 (Macintosh; U; PPC
Mac OS X Mach-O;) Firefox/1.4.1
Cookie:
JSPSESSIONID=34JHURHD894LOP04957H
R49I3JE383940123K
ticketId=ATHX1138&to=PO BOX 1198
DUBLIN 2&amount=10&date=11042008
```

Other example of cross-site request forgery aims to build an URL in the same manner which a browser sends the sensitive data to the server. The built URL is hidden under a link and attacker must convince the end-user to click it. After clicking, the URL with embedded data regarding the malicious intent is sent to the server and it is

operated as end-user's action, but with results for attacker's intent.
SQL injection occurs when SQL command is embedded in data sent by the user from a web form to a page hosted by server in such way that the control mechanisms implemented in web application are passed by. The embedded data introduce operation like displaying, adding, deletion, or manipulation in backend database.
A classic malicious input as SQL statement is:

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

The SQL statement is provided in an input control of a web form. It checks if the username **john_smith** is stored in the table **LOGIN**. If the user **john_smith** is found, then the attacker has the login credentials of that user.
During program coding, direct object references are exposed. When exposing is made in an insecure manner, the application becomes vulnerable. The direct object references have the following forms: URL, form parameter, file, directory, or database record [15].
As examples, the following methods are used for URL references:

- a malicious URL is loaded as part of another, and the obtained URL is considered valid;
- a malicious URL is used as parameter of a function which redirect the end-user to malicious web sites;
- a malicious URL is used in search scripts to redirect the end-user to malicious web sites.

Improper error handling and information leakage can introduce vulnerabilities in an application because the feedback information can leak the internal state, system configuration, or hardware and software resources used to operate the application [15]. A possible attacker can use information about internal algorithms, functionality, database structure, user IDs to plan an attack scenario. The program code must manage errors in such way that the feedback information cannot be used for an attack.
Insecure storage and improper use of cryptography aim the security issues regarding management of confidential information as passwords, keys and certificates. Also, using of weak cryptographic algorithms and hard coding of keys cause security breaches of an application [15].
The mistakes in program coding are the main cause of security breaches. During the implementation stage of the software development life cycle, the developers must consider the cryptography of the application server or runtime environment, operating system and hardware in addition to cryptographic mechanisms implemented by them in the software product.
Time of check vs. time of use aims the time difference between the two moments and the value of a resource considered at one of the two moments. This type of vulnerability is related to multithreading codding.
For each programming language, the best practices to eliminate the software vulnerabilities introduced by insecure coding are developed.

3 Processes of Secure Code Review

The code review is a systematic examination of the program source code. The security code review aims the security issues of the program source code like security requirements of the software product or secure development of the application.
Security code review is different than code security audit. Security code review aims to find the known security vulnerabilities during implementation stage of the software development life cycle. Once a vulnerability being identified, the development team must implement solutions to eliminate or mitigate it. Code security audit consists of all examinations performed respecting compliance with specifications, standards, contractual agreements, or other criteria, and expresses a neutral opinion regarding the code security. Also, the code security audit is made on formal or documented procedures and it is included in security system audit process.
The code review is a process more or less formal. In [18], the review processes are listed:

- ad hoc review – it is made by a temporary group of experts selected on their expertise and experience;
- passaround – it is used to select the expert participant in the code review process with no specific roles assigned to them; reviewers examine the program code from specific perspectives, and the program code is distributed among the reviewers' team;
- pair programming – it is an agile software development technique in which a computer is shared by two programmers; one writes the program code while the other reviews the code lines;

- walkthrough – the reviewers ask questions and make comments on program source code; there are the following roles in the reviewers' team: the author who presents the program source code, the walkthrough leader who leads the meeting, and the recorder who notes the potential defects;
- team review – it is an assessment process of the team members to be up to date to the new threats for the software products and new coding methods and techniques introduced by new software development technologies;
- inspection – it uses a well-defined process to find the program code defects; the program code is approved by reviewers before its use in the software project; the inspection has the following stages: planning, overview meeting, preparation, inspection meeting, rework and follow-up.

The quality and improvement basis of the review process are highlighted by the following metrics as it is shown in [22]:

- total labor hours for planning the review process (TP);
- total labor hours for overview meeting (TOM);
- total labor hours preparing for the review process (TPR);
- total labor hours for correcting the defects (TC);
- total hours of the review meeting (TM);
- total number of major defects found by the review team (DFM);
- total number of minor defects found by the review team (DFm);
- total number of corrected major defects (DCM);
- total number of corrected minor defects (DCm);
- total physical lines of code to be reviewed (SP);
- total physical lines of code reviewed actually (SA);
- number of active participants in the review meeting (NR);
- decisions about disposition of the reviewed program source code (IA).

The above metrics highlight the total effort made for a code review process implementation. They offer an objective picture of the inputs and outputs for a review process. Also, they are used to define indicators as it is shown in [22]:

- total labor hours of the review process (TR):

$$TR = TP + TOM + TPR + TC + TM$$

- total number of found defects (DFT):

$$DFT = DFM + DFm$$

- defect density (DD):

$$DD = \frac{DFT}{SA}$$

- total number of corrected defects (DCT):

$$DCT = DCM + DCm$$

- number of labor hours per defect (ED):

$$ED = \frac{TR}{DFT}$$

- number of labor hours per unit size (EUS):

$$EUS = \frac{TR}{SA}$$

- percent of reviewed code (PRC):

$$PRC = \frac{SA}{SP} \cdot 100$$

- percent of major defects (PMD):

$$PMD = \frac{DFM}{DFT} \cdot 100$$

- rate of review (RR):

$$RR = \frac{SA}{TM}$$

- rate of preparation (RP):

$$RP = \frac{SP}{\frac{TPR}{NR}}$$

- number of rework hours per defect (RD):

$$RD = \frac{TC}{DCT}$$

The above metrics and indicators are used to analyze the accuracy and performance of code review process or to improve the activities carried out during implementation stage of the software development cycle. The improvements of the implementation activities aim vulnerabilities detected and mechanisms avoiding their exploitation, and secure development practices applied during software development.

The relation between code review process and using the metrics and indicators during code review processes are depicted in [19] and they are presented in Figure 1.

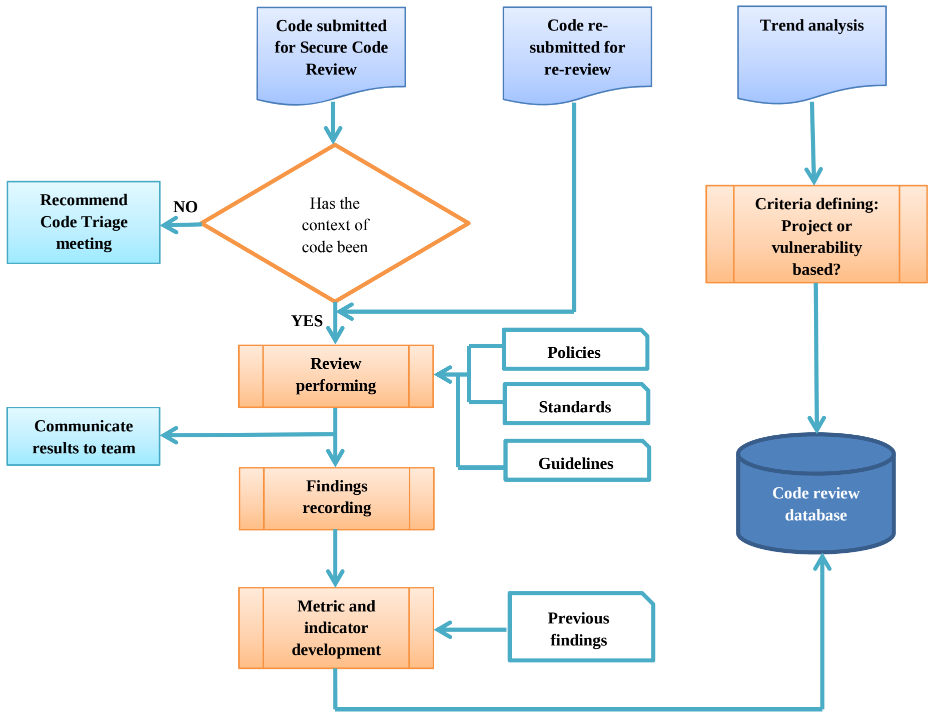
Code submitted for Secure Code Review
Code re-submitted for re-review
Trend analysis
Recommend Code Triage meeting
NO
Has the context of code been
Criteria defining: Project or vulnerability based?
YES
Review performing
Policies
Standards
Guidelines
Communicate results to team
Findings recording
Code review database
Metric and indicator development
Previous findings

Fig. 1. Relation between code review process and review metrics or indicators [19]

In [19], the metrics and indicators used for a secure software development are presented. The metrics and indicators are:

- defect density – it is defined as indicator DD presented above; the indicator is not able to isolate the major defects, and all defects have the same importance; so, only DD does not highlight the program code security;
- lines of code (LOC) – they can see like the metric SP or SA; the metric attempts to quantify the size of the code; it is not relevant for security vulnerabilities that can be found in the program code;
- function point (FP) – other way to quantify the program code, taking into account the functionality; it measured by number of statements for implementation of a specific task;
- risk density – it is similar to defect density, but it is rated by risk and reported to lines of code or function points;
- cyclomatic complexity (CC) – shows the complexity of the flow of control through the program code; it may be used as a confidence measure for a program code, because it is calculated as a number of independent path through the program code; a bigger value of CC means a higher complexity of the program code, so a higher risk of defects because it is difficult to understand, test and maintain it; the risk intervals depending on CC are presented in Table 1;

Table 1. Risk intervals depending on CC [19]

Interval of values for CC	Type of risk	Complexity of program code
0 – 10	Low	Acceptable complexity
11 – 15	Medium	Program code more complex
16 – 20	High	Too many independent paths for the program code

- inspection rate – it is similar to indicator RR; it highlight the code coverage per unit of time during the review process; the calculated value may show the quality of the code review process;
- defect detection rate (DDR) – it measures the number of defects per unit of time; like RR, it is an indicator may be used to highlight the quality of the code review process; a smaller value of RR may lead to a higher value of DDR;

$$DDR = \frac{DFT}{TR}$$

- code coverage – it is similar to indicator PRC; it shows the proportion of the reviewed code;
- defect correction rate – it is similar to ED; knowing the time needed to correct a defect, the management team of the software development project optimizes the project plan;
- re-inspection defect rate (RDR) – it is defined as the number of defects detected after the review process, given by the defects remained after review and new defects generated by the correction process (DAR), reported to the initial total number of defects;

$$RDR = \frac{DFT - DCT + DAR}{DFT}$$

A value below 1 of indicator RDR means that the number of defects detected after the review process is less than before. A value above 1 of indicator RDR means that after the review process the number of defects is higher than before.

Depending on the type of software product and the target clients' group, more organizations have elaborated recommendation and best practices to be followed during the secure program coding process. The secure code review must take into account the best practices used in program coding to increase the quality of the software product.

In order to code the program specifications with high level of quality, including the handling of the security issues, the software developers must use the best methods and techniques provided by the development environment, their expertise and skills, and coding procedures standardized in documents agreed by the best specialists in software development life cycle.

The secure code review has not the scope of an informatics audit process. But, if the review process is closed to the audit procedure, then the developed software product has the all characteristics like a certified product on audit. Also, the requirements specified in audit standards may be easily accomplished therefore the software product is in accordance with the best quality standards. More details about the informatics audit process are presented in [2], [4], [5], [9], [11] and [12].

At Stanford University, the National Accelerator Laboratory, SLAC Computer Security, has established the following top 10 best practices in secure coding process [25]:

- input validation – data coming from external sources must be considered as untrusted; a proper input validation mechanism eliminates the most part of vulnerabilities from data sources as command line arguments, network interfaces, environmental variables, and user controlled files;
- heeding at compiler warnings – it recommends elimination of compiler warnings by modifications of the program code; if the code modification is not needed, it recommends insertion of a comment with reasons to keep the code unaltered;
- building software architecture and design for security policies – the implementation of the security policy is forced by the software architecture and design defined by developers' team;
- keeping a simple design – a complex software design leads to complex

mechanisms to implement quality assurance and system security; the simplicity reduces the likelihood of error making during implementation, configuration and use of the software product;

- default denying – there are defined conditions and rules when the access is permitted;
- adhering to the principle of least privilege – the elevated privileges are temporarily granted to the process; thereby, an attacker has not time to prepare and execute malicious code with elevated privileges in a software system;
- data sanitizing – depending on the context at runtime, a process has to determine the data sainting before of the subsystem invoking; if the process fails to do that, then malicious data may reach to the subsystem and the attacker obtains information to prepare and execute an attack on the subsystem;
- defensive in depth – it refers to the defensive strategies of the risk management; a system of layers is built to cover many cases as possible regarding the risk management;
- using of effective quality assurance techniques – quality assurance techniques lead to effective identifying and management of the vulnerabilities; as quality assurance techniques the following are included: code and functionality testing, secure code reviews and audits, independent and external security reviews;
- secure coding standard – a secure coding standard is developed or applied to coding process in accordance with the programming language and development environment.

The SANS Institute considers the following fundamental practices for secure software development as it is depicted in [14]:

- minimize use of unsafe string and buffer functions – depending of programming language, there are string- and buffer-copying function families which introduce vulnerabilities in the software code; over time, safer function families have been developed to mitigate the vulnerabilities;
- input and output validation – the most common vulnerabilities are mitigated by mechanisms of input and output validation; input and output data are managed by program variables those content is validated according to the following guidelines:
    - the input variable has to exist and to be in accordance with the data type;
    - data stored by program variable is normalized or it has a simple or short representation;
    - data has to respect the data type and to be in accordance to the output recipient;
    - data has to respect the value range of the data type or required by program specifications;
    - input limitation to allowed values and types.
- use robust integer operations – they are used for dynamic memory allocations and array offsets; the best practices aim use unsigned integers for array indexes, pointer offsets, buffer size, increment and decrement operation within loop structures;
- use XSS libraries – the XSS libraries are specific to the web-applications; they are used to mitigate the vulnerabilities of Cross Site Scripting family; the anti-XSS techniques include: constrained input, normalized input, input validation at server side, encoded outputs and client side protection by limitation of cookie use or non-availability, anti-virus software;
- use canonical data formats – data are converted to a canonical, standard or normal form; therefore, an expression do not pass any security filter mechanisms;
- avoidance of string concatenation – the concatenation is used for dynamic SQL statements; operation permits to build malicious SQL statements injected to the target software system; it recommends use the functions built within programming languages, libraries or frameworks, a proper configuration of the SQL engine to avoid SQL statements out of the rules defined by developer;
- use strong cryptography – it has an important impact on software application security; standardized protocols instead of low-level cryptographic algorithm, standardized cryptographic algorithms (if necessary), secure management of access to the cryptographic keys are used or implemented in developed software application;
- use logging and tracing – the aim is to identify and mitigate the software vulnerabilities exploited by an attacker, using information stored in the system about

the access in the system and operations performed by the system users; therefore, it can be established what happened to implement measures for vulnerability mitigation.

The improvements regarding the use best practices in coding process are established on review metrics and indicators. Depending on the calculated metrics and indicators, the development team chooses the proper security coding best practices.

4 Conclusions

Software development improvements are necessary for a better software security. Using of best practices during the software development life cycle is a critical requirement to achieve an improved security.

As stage of software development life cycle, the program coding contributes to software assurance using the best practices and standards in program coding. The goal is to eliminate or mitigate the software security risks resulted from a wrong implementation of the design and specifications in a programming language.

Minimization of the code improvement costs is made considering the best practices applied in the coding stage of the software development life cycle. The code improvement cost has the following components:

- cost of the code review process;
- cost of the code correction;
- cost of the re-review process after the correction stage.

The re-review process is necessary because the code correction stage may lead to keeping of some initial vulnerabilities or appearance of new ones. It is possible that the corrected code to be worst that the code before the correction stages. All these elements must be considered when a code review program is implemented in the software development life cycle together with the component of risk management.

Acknowledgement

Parts of this paper were presented by the author at “11th International Conference on Informatics in Economy – Education, Research & Business Technologies”, Bucharest, Romania, 10 – 11 May 2012.

References

[1] C. Boja and M. Doinea, “Security Assessment of Web Based Distributed Applications”, *Informatica Economică*, vol. 14, no. 1, 2010, pp. 152 – 162

[2] C. Ciurea, “The Informatics Audit – A Collaborative Process”, *Informatica Economică*, vol. 14, no. 1, 2010, pp. 119 – 127

[3] J. Canup, *Secure Coding: Best Practices*, North America CACS, May 8, 2012

[4] I. Pedrosa and C. J. Costa, “Financial Auditing And Surveys: How Are Financial Auditors Using Information Technology?: An Approach Using Expert Interviews”, *Proceedings of the Workshop on Information Systems and Design of Communication (ISDOC 2012)*, ACM New York, NY, USA 2012, pp. 37-43

[5] I. Pedrosa and C. J. Costa, “Computer Assisted Audit Tools and Techniques in Real World: CAATT's Applications and Approaches in Context”, *International Journal of Computer Information Systems and Industrial Management Applications*, Volume 4, 2012, pp. 161-168,

[6] P. Pocatilu and C. Boja, “Quality Characteristics and Metrics Related to M-Learning Process”, *Amfiteatru Economic*, vol. 11, no. 26, 2009, pp. 346 – 354

[7] M. Popa, “Security Characteristics of the Program Coding”, *Conference Proceedings – The 11th International Conference on Informatics in Economy – Section: Audit and Project Management*, 2012, ASE Publishing House, Bucharest, pp. 211 – 215

[8] M. Popa, “Characteristics of Program Code Obfuscation for Reverse Engineering of Software”, *Proceedings of the 4th International Conference on Security for Information Technology and Communications*, ASE Publishing House, Bucharest, 2011, pp. 103 – 112

[9] M. Popa, “Framework for Evaluation of the IT&C Audit Metrics Impact”, *Informatica Economică*, vol. 15, no. 4(60), 2011, pp. 119 – 133

[10] M. Popa, “Techniques of Program Code Obfuscation for Secure Software”, *Journal of Mobile, Embedded and Distributed Systems*, vol. 3, no. 4, 2011, pp. 205 – 219

[11] M. Popa, “Methods and Techniques of Quality Management for ICT Audit Processes”, *Journal of Mobile, Embedded and Distributed Systems*, vol. 3, no. 3, 2011, pp. 100 – 108

[12] M. Popa, “Techniques and Methods to Improve the Audit Process of The Distributed Informatics Systems Based on Metric

System", *Informatica Economică*, vol. 15, no. 2(58), 2011, pp. 69 – 77

[13] G. Sabau, M. Muntean, A. R. Bologa, R. Bologa and T. Surcel, "An Evaluation Framework for Higher Education ERP Systems", *WSEAS Transactions on Computers*, vol. 8, Issue 11, 2009, pp. 1790 – 1799

[14] S. Simpson (editor), *Fundamental Practices for Secure Software Development 2nd Edition*, Software Assurance Forum for Excellence in Code, February 8, 2011

[15] T. Shiralkar and B. Grove, *Guidelines for Secure Coding*, Atsec Information Security, January 2009

[16] C. Toma, "Security Issues for 2D Barcodes Ticketing Systems", *Journal of Mobile, Embedded and Distributed Systems*, vol. 3, no. 1, 2011, pp. 34 – 53

[17] C. Toma and C. Boja, "Survey of Mobile Digital Rights Management Platforms", *Journal of Mobile, Embedded and Distributed Systems*, vol. 1, no. 1, 2009, pp. 32 – 42

[18] K. E. Wigers, *Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2001

[19] Open Web Application Security Project, *Code Review Guide, V1.1*, OWASP Foundation, 2008

[20] Software Assurance Forum for Excellence in Code, *Fundamental Practices for Secure Software Development*, October 2008

[21] Software Assurance Forum for Excellence in Code, *Software Assurance: An Overview of Current Industry Best Practices*, February 2008

[22]http://www.cs.toronto.edu/~sme/CSC444F/handouts/inspection_process_model.pdf (October 2012)

[23] https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (October 2012)

[24]https://www.securecoding.cert.org/confluence/display/seccode/STR30C.+Do+not+attempt +to+modify+string+literals (October 2012)

[25]http://www2.slac.stanford.edu/computing/security/TEA/Programming_Tips_Best_Practicies. htm (October 2012)

Marius POPA has graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 2002. He holds a PhD diploma in Economic Cybernetics and Statistics. He joined the staff of Academy of Economic Studies, teaching assistant in 2002. Currently, he is university lecturer in Economic Informatics field and branches within Department of Economic Informatics and Cybernetics at Faculty of Cybernetics, Statistics and Economic Informatics from Bucharest University of Economic Studies. He is the author and co-author of 9 books and over 120 articles in journals and proceedings of national and international conferences, symposiums, workshops in the fields of data quality, software quality, informatics security, collaborative information systems, IT project management, software engineering. Also, he was involved in 14 national research projects as team member and 2 national research projects as project manager. From 2009, he is a member of the editorial team for the Informatica Economică and between 2003 and 2008 he was a member of the editorial team for the journal Economic Computation and Economic Cybernetics Studies and Research.