

The Life Cycle of Vulnerabilities in the Representations of Software for Telecommunication Devices

Mikhail Buinevich*, Konstantin Izrailov*, Andrei Vladyko*

**The Bonch-Bruевич Saint-Petersburg State University of Telecommunications, Russian Federation, Saint-Petersburg, 22-1 Prospekt Bolshevikov*

bmv1958@yandex.ru, konstantin.izrailov@mail.ru, vladyko@bk.ru

Abstract— This article investigates the problem of search for software vulnerabilities in terms of their life cycle. Each point life of vulnerability determined by the state of software with form and content – software representation. A typical sequence of representations in the field of telecommunication devices is given. Description prototype of software tool for Methodological Support System for Development of Safe Software is given.

Keywords— information safety, machine code, software representation, telecommunication devices, vulnerability, methodology

I. INTRODUCTION

One of the most actual problems in the field of information security is an insufficient amount of satisfactory methods of search for software (SW) vulnerabilities, the existence of which may compromise the safety of critical information. The available methods may be deemed to be convenient only for certain types of vulnerabilities or application conditions (for instance, monitoring of software exceptions by means of data fuzzing in the memory of the tested program [1]). To address this issue, we must first explore the problematic object itself – the vulnerability, both its static and dynamic properties that determine its life cycle. A subsequent analysis of dynamic properties will allow for a wider coverage of all possible methods of search, as they directly depend on which type of representation has a vulnerability. As for static properties, the latter will only provide convenient classifications of vulnerabilities and partly form additional knowledge of the same.

A special place within this issue belongs to telecommunication devices (hereinafter referred to as TCD) for the following reasons. Firstly, their intended purpose is exactly the transmission of information, the need for security of which just creates the first part of a problematic contradiction – the existence of requirements. Secondly, by reason of working with huge volumes of information, some restrictions are imposed on the SW performance of such devices (for instance, processing large volumes of data within a short time). This leads to the need for developing SW in the languages with a low level of abstraction (in fact, only in C), the final form of which represents a machine code for a physical CPU. As a result, the vulnerability search methods in such SW are complicated by several times, which determines the second part of the problematic contradiction –

the lack of opportunities. The modern features of using TCD, such as their application for critical governmental functions and supplying without source code (hereinafter, the SC) only aggravate the problematic contradiction. Therefore, this paper will consider the software code of these particular devices.

The dynamic properties of a vulnerability determine its entire life cycle – from emergence to detection (whereupon its neutralization will be merely a technical problem). As a vulnerability exists only inside the SW, its form shall be determined by the SW state at the corresponding point of its life cycle. Hereinafter we will designate the SW state with given form and content as the SW representation. For example, the SW state at the moment of its assembling will be referred to as the representation of the assembler code having a textual form and including instruction for the CPU. Accordingly, in the process of SW development, the latter proceeds from some representations to another ones. Hence, vulnerabilities may occur in some SW representations, mutate when proceeding to another ones and may be detected in the third.

The paper presents the results of the analysis of all possible TCD SWs, their properties and mutual transitions, with overlaying the vulnerabilities life cycle. Additionally, a static property of vulnerability is taken into consideration, such as its types according to the structural level. Such a generalizing scheme is referred to as the life cycle of vulnerabilities in SW representations.

II. SOFTWARE REPRESENTATIONS

For each SW representation having a real practical application, the following properties can be distinguished. Firstly, each representation has its own purpose. Secondly, each representation features its unique content that corresponds to its purpose. Thirdly, the type of the representation content has a definite form that is the most suitable for a particular purpose. Fourthly, each representation is derived from the previous one in a certain way. Fifthly, each representation is potentially recoverable from the next one in a certain way. Sixthly, when obtaining a new representation, we may receive some vulnerabilities, and following its analysis we can detect another ones. And, seventhly, for each representation, there are only certain ways to search vulnerabilities, depending on its form and content. Possible values of some of these properties are given below.

Each representation may take the following forms (or their combination): verbal (V_F) – using a natural language; graphical (G_F) – in the form of flowcharts, diagrams, graph images; software language (SL_F) – using a formal-sign text in programming languages; binary (B_F) – in the form of a set of bytes (less often, bits).

Obtaining a subsequent representation from the current ones is effected by means of transformations such as: synthesis (S_T) – the result of human creative activities with development of a new content; direct transformation (DT_T) – the result of operation of special software tools without changing the content; generation (G_T) – the result of operation of special software tools with changing the structure of the content or meta information.

Additionally, the partial or full recovery of the previous representation from the current ones may be effected via transformations such as: analysis (A_T) – extracting information from the representation in respect of the recovered representation, while discarding the non-relevant content (possibly, with some elements of forecast); reverse transformation (RT_T) – the result of operation of special software tools without changing the contents; parsing (P_T) – the result of operation of special software tools with changing the structure of the content or meta information.

The only static property determines the type of vulnerability. Firstly, there are low-level vulnerabilities (LV), such as errors in calculation, data structures, data access, etc. Secondly, there are medium-level vulnerabilities (MV), such as erroneous realization of sub-program algorithms, transmission of input parameters, return from the program, etc. And, thirdly, there are high-level vulnerabilities (HV), such as errors in the software system architecture or concept, for example, violation of the general principles of its operation and security, etc.

Vulnerability search may be both manual (H_S) – subjectively by a human, and automated (A_S) – using special software tools; naturally, it is possible to combine the two modes.

SW representations and their properties are given below.

A. Representation No. 1 – Main Concept

The purpose of the SW main concept is determination of some software design and its functionality with identifying some characters and properties.

The representation form is verbal, as far as there is no opportunity to formalize it.

Representation is deemed to be the start of any SW and is created as a result of a creative process to solve a major task.

Representation may be recovered through analysis from the SW conceptual model.

This representation cannot have a vulnerability, because it is used as the future prototype, i.e. ideal.

B. Representation No. 2 – Conceptual Model

The SW conceptual model is its representation that determines the main concepts, their structures, interrelation

and peculiarities indicating the solution direction and the used approaches. Here, we can identify the basic semantic structure and the corresponding terminological apparatus.

The representation form is verbal and graphical as the most convenient for description of generalized models with associated textual explications.

This representation is created as a result of synthesizing the possible embodiments of the SW main concept by means of primary formalization, in line with the selected solution.

The representation may be recovered by analyzing the SW architecture.

In this representation, HV may occur as a result of erroneous modelling of the main concept. In this representation, HV search may be effected, as the conceptual model operates with high-level concepts, without descending to the level of their immediate execution, i.e. algorithms. Due to an extreme high abstraction, the vulnerabilities search is possible here only in manual mode.

C. Representation No. 3 – Architecture

SW architecture is its most complete high-level representation in the form of a model that is close to the SC and is described using specific features (technologies, programming languages, templates/paradigms, means of implementation, formats, etc.). The architecture determines the structure of model elements, their relationship and properties.

The representation form is graphical (with verbal notes). This representation is created based on the conceptual model as a result of synthesis of requirements and opportunities using the experience of an expert architect.

The representation may be recovered by analyzing the SW algorithms. Recovery is usually feasible only by using the expert method with partial application of special software tools.

HV may occur in this representation as a result of high-level errors in the architecture designing. In this representation, HV may be effected, because it is composed of the same high-level elements (general structure of the SW, physical and logical modules, their interaction). Given an extremely difficult formalization and the subjective nature of evaluation, vulnerabilities may be found in most cases only manually.

D. Representation No. 4 – Algorithms (of the Source Code)

The set of SC algorithms determines the representation that divides the SW into sub-programs with specification of their purpose and a sequence of execution steps. Algorithms are aimed at a specific solution of particular tasks of the SW functionality.

The representation form is graphical by way of flowcharts with verbal explication of its elements.

The representation is created as a results of the SW architecture synthesis, following the purpose of its modules. As each algorithm solves a specific task, often being unique and/or having special requirements, the development process seems to be completely creative.

The representation may be recovered from the SC by means of its manual analysis, by an expert or automatically, using special software tools. The results of the first method are characterized by a higher level of abstraction and structuralization. The second one may be only considered as formal way to recover algorithms, since the latter appear to be too detailed.

In this representation, MV may occur due to an erroneous solution of the architecture-defined tasks by the algorithms. In this representation, MV may be effected, since it does not determine the SW architecture (having HV) and does not include any specific embodiment of algorithms (having LV), but it is located between these levels of abstraction. The vulnerability search method is manual, with possible application of automated tools.

E. Representation No. 4.1 – Algorithms (of the Assembler Code)

As was the case of the SC, the assembler code also allows for building algorithms, however with the following important differences.

Firstly, the algorithm steps are maximally detailed in this representation.

Secondly, instead of the verbal content of algorithm elements, there are (usually) processor instructions or an equivalent pseudocode.

Thirdly, the algorithm data may be recovered from the assembler code only. Manual recovery features a high labor intensity. On the other hand, automated recovery is performed using some tools that are usually embedded in the visual environment for disassembling, that receive extremely weakly structured algorithms.

Fourthly, the representation, along with MV, includes LV (in the elements of flowcharts), the latter being found not only manually, but also in an automated mode.

And, fifthly, this representation is rather often used by hackers to introduce vulnerabilities.

F. Representation No. 4.2 – Algorithms (of the Assembler Code) – Modified

A wide-spread hackers' practice is modification of the assembler code of devices to introduce LV and MV – i.e. the synthesis of a new representation. Then, such a modified assembler code, through the generation of a machine code, is assembled into an image file and distributed or uploaded to the terminal device. Such image files and devices are considered as infected and pose a significant threat. It is obvious that the vulnerability search in this representation is not effected, as far as it is available only to the hacker. Any recovery of such representation does not make sense as well.

In other respects, this representation is completely analogous to a non-modified one.

G. Representation No. 5 – Source Code (Original)

The SC is the “working” representation of a SW, since it fully determines its logic. Its purpose is developing the algorithms embodiment in the selected programming language with subsequent debugging, support, documentation, etc.

The form of representation is a programming-language one, but sometimes a combination of programming-language and flowchart forms is used.

The representation is created as a result of synthesizing the SW algorithms and corresponds quite accurately to their steps, since it is their specific implementation.

There is no single method to recover the SC from subsequent representation (usually, the assembler code), because a large amount of meta information (names of variables and subprograms, computing methods, comments, etc.) are lost during compilation.

In this representation, both LV and MV may occur due to errors in algorithms implementation and their erroneous interpretation by a programmer. In this representation, a search for LV and MV may be effected, because it is a connecting link between the algorithmic representation of a SW and its final low-level embodiment. There may be applied both manual and automated search methods. The former cover mostly the MV area, the latter, the LV one.

H. Representation No. 5.1 – Source Code (Pseudo)

Although it is impossible to obtain an unambiguous SC representation from the subsequent ones, however, it may be partially recovered into a pseudocode using analyzer tools known as decompilers. Actually, there are only a few of them; even fewer may be deemed to be working for a very limited number of CPUs.

The representation form is a programming-language one. Such a language may be both suitable for an automated processing using software tools and designed only for manual analysis by a man.

This representation may be also recovered manually from the assembler code, however this method is extremely labor intensive.

The vulnerabilities search methods (for LV and MV) are completely analogous to their equivalents in the original SC.

I. Representation No. 6 – Assembler Code

The assembler code is a SW representation that determined the final variables and instructions to be executed by the CPU. It also includes user names and subprograms, although the algorithms of the latter have a several times lesser structured view than the original one. This representation is considered as intermediate.

The representation form is a programming-language one, with the use of the processor assembler syntax.

This representation is generated based on the SC using special tools – compilers, partially rejecting the meta information (for instance, the comments to the SC, structures, types, etc.). This representation may also be transformed from the recovered and modified algorithms of the assembler code using a trivial method.

This representation may be almost completely recovered from the machine code using parsing programs – disassemblers, since the assembler code is just a more readable form of the machine code. Thus, all the methods of recovery based on the assembler code are applicable to the machine one through the disassembling module.

In this representation, LV may occur due to erroneous operation of SC compilers; however, this probability is very low. The search of LV in this representation is possibly only using a labor-intensive manual method.

J. Representation No. 7 – Machine Code

The machine code is a SW representation that is the most close to the execution at the terminal device. It represents unreadable data that set the execution of the CPU instructions.

The form of representation is a binary one.

The representation is generated based on the assembler code using special tools – assemblers, fully rejecting the meta information that is not required for execution by the processor (for instance, user names).

This representation may be unambiguously recovered using the method of inverse transformation from a SW image file at the terminal device.

The vulnerabilities and the methods for searching the same are completely analogous to their equivalents in the assembler code.

K. Representation No. 8 – Image File

Usually, the image file is uploaded to the terminal device and represents some information structured in some container (the main part of which is occupied by the machine code), including everything required for its unpacking in the device memory and execution. This is a final representation of a SW.

The form of representation is a binary one.

The representation is transformed (by assembling) from the machine code using a trivial method.

The representation may be obtained from the terminal device without any transformation by reading with the use of special software and hardware tools.

The vulnerabilities and the methods for searching the same are completely analogous to their equivalents in the machine code.

III. THE RESULTS OF THE ANALYSIS AND THE VULNERABILITIES LIFE CYCLE

A summary table resulting from the SW representation synthesis, vulnerabilities and their properties, is given in Table I and Table II.

TABLE I. SUMMARY TABLE OF SW REPRESENTATIONS, VULNERABILITIES AND THEIR PROPERTIES (PART II)

Rep No.	Name	Purpose	Content
1	Main concept	SW design	Functionality, characteristics and properties
2	Conceptual model	Basic semantic structure and terminological apparatus	Main concepts, their structure, relationship, peculiarities
3	Architecture	SW high-level	Specifics, elements

		model	of the system and relationship
4	Algorithms (of the source code)	Subprogram algorithms (medium-level)	Subprogram steps
4.1	Algorithms (of the assembler code)	Subprogram algorithms (low-level)	Subprogram steps
4.2	Algorithms (of the assembler code) – modified	Subprogram algorithms (low-level) for introducing vulnerabilities	Subprogram steps
5	Source code (original)	Development of algorithm embodiments	Programming language elements
5.1	Source code (pseudo)	Reverse development of algorithms embodiments	Programming language elements (pseudo)
6	Assembler code	Intermediate low-level SW embodiment	Assembler language elements
7	Machine code	SW final code	Set of CPU instructions
8	Image file	Image to upload to the device	Container with a set of CPU instructions

TABLE II. SUMMARY TABLE OF SW REPRESENTATIONS, VULNERABILITIES AND THEIR PROPERTIES (PART II)

Rep No.	Form	Obtaining from Rep No.	Recovery in Rep No.	Vulnerabilities		
				Occurrence	Detection	Search type
1	V_F	N/A	A_T in 2	N/A	N/A	N/A
2	V_F, V_G	S_T from 1	A_T in 3	HV	HV	H_S
3	V_G (+ notes)	S_T from 2	A_T in 4	HV	HV	H_S
4	V_G	S_T from 3	A_T in 5 or 5,1	MV	MV	H_S
4.1	V_G	N/A	A_T in 6	N/A	LV, MV	H_S
4.2	V_G	S_T from 4.1	N/A	LV, MV	N/A	N/A
5	SL_F	S_T from 4	N/A	LV, MV	LV, MV	H_S, A_S
5.1	SL_F	N/A	A_T in 6	N/A	LV, MV	H_S
6	SL_F	G_T from	P_T in 7	LV	LV	H_S

	(assembler)	5, DT_T from 4.2				, A_S
7	B_F	G_T from 6	RT_T in 8	LV	LV	Eq. to 6
8	B_F	DT_T from 7	N/A	LV	LV	Eq. to 7

Transitions between SW representation with overlaying the life cycle of various types of vulnerabilities may be presented in an easily legible graphical form using a vulnerabilities life cycle scheme in the SW representation, as shown in Figure 1.

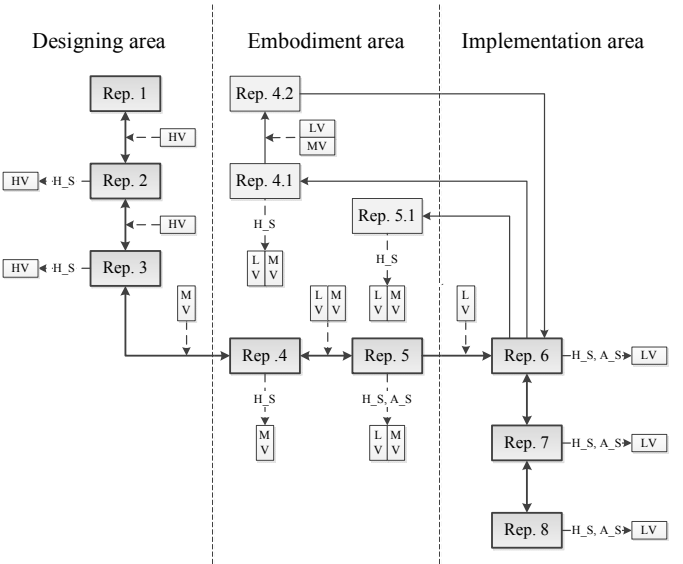


Figure 1. Types of vulnerabilities according to SW build-up level

Note. The following designations are used in the diagram in Figure 1. A large rectangle stands for a SW representation. A small square stands for the vulnerability type. A solid array connecting rectangles means transformation between representations, in direct and reverse directions. A dotted arrow, exiting from a square and entering a solid arrow, means the occurrence of vulnerabilities. A dotted arrow, exiting from a rectangle and entering a square, means the detection of vulnerabilities; the characters on an arrow indicate the vulnerability search method.

It should be noted that, while in the case of a separate SW project for a TCD the scheme of representations has a rather linear form, for complex projects, several architectures may be developed for different components of a SW, even for one conceptual model. This, in turn, will need developing a whole set of programs in different languages, only a part of which having a machine code at the final stage. Thus, in the general case, the scheme of vulnerabilities may have a complex topology, which complicates many times the search for vulnerabilities.

IV. THE METHODOLOGICAL SUPPORT SYSTEM FOR DEVELOPMENT OF SAFE SOFTWARE

Based on the study of theoretical capacities of the synthesized scheme and the practical needs within the methodology of vulnerabilities search, the following software prototype for a TCD SW has been created: the Methodological Support System for Development of Safe Software (MSS-DSS).

This software tool is designed to account for the possible vulnerabilities in each SW representation according to its characteristics. In the process, vulnerabilities from the previous representation remain in the subsequent ones. As a result, a list is created including the required search methods for each type of representation. The prototype also takes into consideration the results of the realized methods, as far as they affect the existence of vulnerabilities in all representations. It should be noted that in later representations some vulnerabilities cannot be found in principle, which is also mentioned. Thus, in each state of a SW we track the information about the potential vulnerability, availability or absence of search methods, and a general assessment of the SW security is carried out.

The prototype uses a local, minimally required DB of vulnerabilities and methods, that in the future is planned to expand to international ones (such as NVD). To date, the prototype is realized in the form of a Web application, the screenshot of which is as follows (Figure 2).

METHODOLOGICAL SUPPORT SYSTEM FOR DEVELOPMENT OF SAFE SOFTWARE [Log In]

File Design Help

CURRENT SOFTWARE REPRESENTATION:
Source Code

PROPERTYS OF SOFTWARE REPRESENTATION (CHECKED, IF REPRESENTATION HAS ENABLED PROPERTY):
☒ Property 2: Description of property 2
☒ Property 3: Description of property 3
☐ Property 4: Description of property 4

REQUIRED AND APPLIED METHODS FOR SEARCH VULNERABILITIES (CHECKED, IF METHOD WAS APPLIED AND VULNERABILITIES WAS FOUND):
☐ Method 1: Description of method 1
☒ Method 2: Description of method 2

POSSIBLE VULNERABILITY (AND IF IT AVAILABLE FOR SEARCH BY METHODS ABOVE IN CURRENT SOFTWARE REPRESENTATION):

Vulnerability description	Findable?
Vulnerability 1: Description of vulnerability 1	Yes
Vulnerability 2: Description of vulnerability 1	No

METRICA SECURITY OF SOFTWARE REPRESENTATION: 0.75%

Figure 2. An example of operation of the MSS-DSS prototype

The prototype has been developed in the environment of Microsoft Visual Studio 2010 based on ASP.NET 4. A typical usage scenario is as follows. After the start of work, the user selects the required SW representation and sets its properties. According to this, and based on the previous representations (using the DB), the program marks possible vulnerabilities and suggests available search methods. The users performs the vulnerability search by sampling methods, registering the results in the program. The list of vulnerabilities and the security metrics for the current representation are updated. The main purpose of the prototype is working out a methodology to ensure the safety of SW for telecommunication devices, a logical extension whereof will

be supporting the projects of any complexity and sets of architectures, technologies, programming languages, etc.

V. CONCLUSION

The vulnerability life cycle scheme, synthesized based on the analysis of possible SW representations, allows for a deeper insight into the vulnerability occurrence conditions. At the same time, this scheme may be extended without any significant modification to any modern SW, intended not only for TCD. In addition, according to the scheme, the search for all the 3 types of vulnerabilities, when only the machine code is available, may be effected as follows. Firstly, by developing an automatic method and a tool for a high-speed LV search in the machine code, the theoretical justification of which is given in the author's article [2]. And, secondly, using the automated method and tool for algorithmization of the machine code for an expert MV and HV search by means of SW algorithms representation in a pseudo language, the practical substantiation of which is also given in the author's article [3].

ACKNOWLEDGMENT

The authors are grateful to their parents and relatives both for their support in the scientific activities and for their motivation by personal examples, in particular, to Irina Mikhailovna Izrailova for her repeatedly manifested will and unbroken spirit in the face of difficulties.

REFERENCES

- [1] Blagodarenko A.V., "Some approach of interacted vulnerabilities research in binary software without sources". Highly available systems. Vol. 7, Iss. 2, 2011. pp. 65-69
- [2] Buinevich M.V. and Izrailov K.E., "Method and Utility for Recovering Code Algorithms of Telecommunication Devices for Vulnerability Search", Proceedings of The 16th International Conference on Advanced Communications Technology (ICACT-2014). IEEE, 2014. pp. 172-176
- [3] Buinevich M.V., Izrailov K.E. and Vladko A.G., "Method for partial recovering source code of telecommunication devices for vulnerability search", Proceedings of The 17th International Conference on Advanced Communications Technology (ICACT-2015). IEEE, 2015. pp. 76-80



Mikhail Buinevich was born in 1958 in the USSR. He received education of the military engineer of electronic engineering.

He served in the naval fleet and government agencies for information security. He held classes at various universities.

His research interests include methods of information security.

He has more than 100 scientific works. His primary publications are as follows:

1. M.V. Buinevich and others. Safety provision

of high-security objects of the naval fleet in relation to damage effects in crisis and emergency situations in peacetime./ Under the editorship of the admiral V.S. Vysotskii.– Saint Petersburg: Publishing house ELMOR, 2008.– 300 p.

2. M.V. Buinevich and others. Provision of organizational and technical support of stability of function and safety of general communications network./ Under the general editorship of S.M. Dotsenko.– Saint Petersburg: Publishing house SPbSUT, 2013.– 142 p.

At the present time Dr. Prof. Buinevich is the professor of the Protected Communications System Chair of Saint Petersburg State University of Telecommunications (SPbSUT).



Konstantin Izrailov was born in 1979 in the city of Saint Petersburg (Russia). In 1996 he graduated from Saint Petersburg State Polytechnic University, Physical and Mechanical Department.

At the moment he is a postgraduate student of the Protected Communications System Chair of Saint Petersburg State University of Telecommunications (SPbSUT). He has about 20

published articles; he is an author of 3 scientific and research works and has a patent on the software tool. His scientific interests include information security, search of vulnerabilities in machine code, reverse engineering and telecommunication devices.

Mr. Izrailov has the title of the best postgraduate student of SPbSUT in 2012 and is the presidential scholar in 2013.



Andrei Vladko (IEEE member (M'14)) acquired his Degree of the Candidate of Sciences at Komsomolsk-on-Amur State Technical University, Russia in 2001.

At present he is a head of the Scientific Work Organization and Researchers Training Administration of Bonch-Bruевич Saint-Petersburg State University of Telecommunications, Saint-Petersburg,

Russia.

His major interests include control systems, soft computing, communication networks, network security management.