Detecting C Program Vulnerabilities [1]

Anton Ermakov, Natalia Kushik
dept. of Information Technologies
Tomsk State University
Tomsk, Russsia
antonermak@inbox.ru, kushiknatalya@yahoo.com

***Abstract*—C/C++ language is widely used for developing tools in various applications, in particular, software tools for critical systems are often written in C language. Therefore, the security of such software should be thoroughly tested, i.e., the absence of vulnerabilities has to be confirmed. When detecting C program vulnerabilities static source code analysis can be used. In this paper, we present a short survey of existing software tools for such analysis and show that for some kinds of C code vulnerabilities this analysis is insufficient. Thus, we briefly present an approach for *SPIN* based approach for vulnerability detection which may be useful in some cases.**

***Index Terms*—C programming language, software vulnerability, static/dynamic detection method**

I. Introduction

THE problem of computer-aided software testing becomes important as the complexity of software tools increases and programs written in C/C++ programming language are often used in many critical systems. The security of such software should be thoroughly tested, i.e., the absence of vulnerabilities has to be confirmed. There are two different approaches for vulnerability testing: static and dynamic methods. In this paper, we present a short survey of existing tools based on static vulnerability detection methods and show that for detecting some vulnerabilities, for example a buffer overflow vulnerability, *SPIN* [1] based approach may be more appropriate.

The structure of the paper is as follows. Section II contains preliminaries. Section III is devoted to static code analyzers: a short survey of existing tools for static vulnerability detection is presented in this Section. Section IV discusses a *SPIN* based approach for vulnerability detection while Section V concludes the paper.

II. Preliminaries

A program *vulnerability* is a property of the program that allows a user to disturb confidentiality, integrity, and/or availability of this software. Given a set of vulnerabilities (features) of a C program, if the program has none of these features then the program is said to be *safe* w.r.t. the given set of vulnerabilities; otherwise, the program is *unsafe* w.r.t. this set of vulnerabilities. Vulnerability detection methods can be classified as static and dynamic methods [2]. When static detecting methods are applied the source code is analyzed without running the program while dynamic detection methods require the program of interest to be executed.

Given a C program, in this paper, when illustrating the approaches, we consider the following types of possible vulnerabilities: type overflow, type conversion overflow, array overflow (incorrect array index), string overflow which can be considered as different types of a buffer overflow vulnerability and double free vulnerability. All these types of vulnerabilities are specified in details in [3]. Type overflow occurs in a C code when a variable v is defined as a variable of type t and the value e of this variable when executing the code can exceed the maximal value for type t. It can occur when a given expression e is assigned to a variable v, i.e., the C code has an instruction $v = e$, and in general, the maximal value for type t might be different for different platforms and operating systems. An array overflow takes place when a programmer deals with an array a that has *size_a* items while using a variable $a[i]$ for $i >=$ *size_a*. When analyzing student software tools implementing well-known array algorithms such as different sorts and/or search of minimal/maximal array item, we noticed that many those programs are unsafe w.r.t. type overflow and array overflow (incorrect array index) vulnerabilities. In order to estimate whether existing static methods can detect type overflow and array overflow vulnerabilities we consider three student implementations of array algorithms and run existing tools for detection of such vulnerabilities. In the next section, we present a short survey of existing tools for static code analysis and their outputs for several vulnerable student programs. We then show that some of such vulnerabilities can be detected using *SPIN* based approach.

III. Static code analyzers

When estimating the security of student implementations of array algorithms we considered the following tasks: calculating the average value of integer array items, the bubble sort, the insertion sort. C implementations of these programs are specified in the Table 1 which is divided into three

[1] This work is partly supported by RFBR-NSC grant № 10-08-92003

sections. Table 1.1 contains a C implementation of calculating the average value of integer array items (Program 1), Table 1.2 contains a C implementation of the bubble sort (Program 2) while Table 1.3 contains a C implementation of the insertion sort (Program 3). Program 1 has a type overflow vulnerability in the line

sred+=a[i];

There is no check in Program 1 if *sred* variable value does not exceed the maximal value of the type *unsigned short*; in this paper, the maximal value equals 65536 and each *unsigned short* variable occupies two memory bytes. Programs 2 and 3 have an array overflow vulnerability, since array indexes of arrays *a* and *arr* are not checked whether they exceed the number of array items.

Program 1 –
C implementation of calculating the average value of integer array items
	**int main(){
	unsigned short n=0, a[10];
	printf("Enter size of array, please:");
	scanf("%d",&n);
	for (int i=0; i<n; i++)
	{printf("%d. ",i);
	scanf("%d",&a[i]);
	}
	unsigned short sred=0;
	for (int i=0; i<n; i++)
	{
	sred+=a[i];
	}
	sred/=n;
	printf("Sred:%d",sred);
	system("pause"); return sred;
	}**

Table 1.1 C implementation of array algorithms (Program 1)

Program 2 –
C implementation of the bubble sort
	**int main()
	{
	unsigned short j=0,i=0,n, a[10];
	cout<<"Enter integer, please:";
	cin>>n;
	for (i=0; i<n; i++)
	{
	cout<<i<<" = ";
	cin>>a[i];
	}
	unsigned short temp;
	bool t = true;
	while (t==true)
	{
	t = false;
	for (j=0;j<n-1; j++)
	{
	if (a[j]>a[j+1])
	{
	temp=a[j];
	a[j]=a[j+1];
	a[j+1]=temp;
	t=true;
	}
	}
	}
	for (i=0;i<n; i++)
	{
	cout<<i<<"="<<a[i]<<endl;
	}
	system("pause");return1;
	}**

Table 1.2 C implementation of array algorithms (Program 2)

Below we describe the outputs of several static source code analyzers that have been run against C implementations in the Table 1.

A. ***ITS4*** is a static code analyzer that has been developed in USA by the *Cigital* company in 1992 [4]. The *ITS4* is a tool for static detection vulnerabilities in C/C++ programs. The tool can be executed under Windows or Linux operating systems.

Program 3 –
C implementation of the insertion sort
	**int main()
	{
	unsigned short length, key, arr[10];
	int i=0, j=0, tmp=0;
	cout<<"length:";
	cin>>length;
	for (i=0; i<length; i++)
	{
	cout<<i<<" = ";
	cin>>arr[i];
	}
	for (i=0; i < length; i++)
	{
	tmp = arr[i];
	for (j=i-1;j>=0 && arr[j]>tmp;j--)
	arr[j+1] = arr[j];
	arr[j+1] = tmp;
	}
	for (i=0;i<length; i++)
	{
	
	cout<<i<<"="<<arr[i]<<endl;
	}
	system("pause"); return 1;
	}**

Table 1.3 C implementation of array algorithms (Program 3)

When analyzing a given C code the *ITS4* relies on its database of potentially dangerous C functions and if there is a call for such dangerous function in the given code the *ITS4* returns a corresponding report with some recommendations about proposes (preferable changes) in the code. The *ITS4* tool is a free software tool that can be easily downloaded from web-site [4]. We executed *ITS4* against Programs 1, 2, 3 (Tables 1.1, 1.2, 1.3) and the *ITS4* has detected two calls for dangerous functions. Those are *scanf*() and *printf*(), in particular, the *ITS4* has reported that *scanf*() is a function of a high risk for a buffer overflow vulnerability.

B. *Flawfinder* is also a static C/C++ code analyzer that has been developed by David A. Wheeler in May, 2004 [5]. *Flawfinder* "scans" a given code and similar to the *ITS4*, has a list of potentially dangerous instructions of a code. Given a code, selected dangerous instructions (if any) are then ordered according to the risks. The *Flawfinder* report for a programmer points out the calls for dangerous functions and proposes a way for changing the code. However for the above Programs 1, 2, 3 the *Flawfinder* report has only one dangerous function – *system*() and the recommendation "*try using a library call that implements the same functionality if available*".

C. *Graudit* is a tool that can also help to statically detect several C code vulnerabilities [6]. In order to run this tool it is necessary to call utility *Grep* under *Unix* operating system. As usual, there can be several options how to run this utility but in the simplest case only the path to *cpp* file has to be specified. As a result, a colorful report will appear where for a given C program, some dangerous instructions are blue colored. One can also manually add more instructions into the database of dangerous functions. For each program in Table 1 the *Graudit* colored functions *scanf*(), *printf*() and stream input/output operators *cin* and *cout*.

D. *CppCheck* 1.46 is a tool with the original name *C++check* that has been developed by *Daniel Marjamäki* and *Cppcheck team* from 2007 until 2010 [7]. The *CppCheck* utility is specialized for memory leakage vulnerabilities. As it is mentioned in [7] *CppCheck* has detected 21 errors in the Linux Core and many other errors in free software. The *Cppcheck* is also a free software tool under the conditions of the *GNU General Public License*. We have run the *Cppcheck* against above Programs 1, 2, 3 and the output message "No errors found" has been returned.

E. *AEGIS* is another tool for static detection vulnerabilities in C/C++ programs [8]. The *AEGIS* has been developing in Digitek Labs since 2008. This laboratory is strongly connected with Saint-Petersburg Polytechnic University, Russia. One of the advantages of this tool is that the *AEGIS* supports vulnerability detection for several files simultaneously if they are united in one project. The *AEGIS* detects vulnerabilities that can often occur in C programs, such as memory leakage, incorrect pointers, incorrect array indexes, uninitialized variables, the use of potentially dangerous functions etc. In order to statically detect these vulnerabilities the *AEGIS* derives the abstract model of the program for verification. The free usage of the analyzer is available via the official Digitek Labs web-site [9]. Before running this tool it is necessary to make some transformations of a given C code for further compiling. For example, in the *AEGIS*, it is prohibited to analyze a code where two or more C instructions are located in the same program line. We have correspondingly changed the above Programs 1, 2, 3 and have run the *AEGIS*. For Program 1 of average value calculating the *AEGIS* detected an incorrect array index for the array *arr* while for Programs 2 and 3 of array sorts the *AEGIS* mentioned only the call of unsafe function *system*().

F. There are other static code analyzers that can be used for vulnerability detection in C programs. For example, *Cqual* [10], developed by Dan Wilkerson in 2004, *Eshelon AK-VS* [11] developed in Russia, *Klocwork Truepath* [12] developed by *Klocwork* company and *Coverity Static Analysis* [13] developed by *Coverity* company in USA, *MOPS* [14] and *BOON* [15] are tools for static detection vulnerabilities. We could not execute these tools due to some reasons such as a high price, lack of documentation, absence of demonstrating version etc. However, according to their descriptions [10–15], all these tools are developed for static detection of vulnerabilities and many of them allow static analysis not only for C/C++ code but also for Java or C# programs.

According to the above short survey of static code analyzers, one can conclude that most existing tools only search for dangerous functions and despite of their descriptions do not detect type overflow and incorrect array index vulnerabilities. The latter means that for some kinds of software vulnerabilities static detection is not enough, that is the reason why in the next section we present a brief overview of an approach for dynamic detection vulnerabilities [3].

IV. SPIN BASED APPROACH FOR DETECTING VULNERABILITIES

Most existing tools providing dynamic detection vulnerabilities are based on randomly generated input data for a given program. Thus, it is difficult to guarantee the fault coverage for such security testing. There also exist special tools for distributed programs testing, for example, Helgrind [16] that is designed for multithreaded programs testing. We note that this tool does not support buffer overflow detection technique but it is able to control synchronization between threads.

There are other model checking techniques which are widely used for vulnerability detection. Working together with our French colleagues we proposed a detection technique based on *SPIN* model checker [1][2] and have partially presented the obtained results in the technical report [3]. In this case, a vulnerability is described as a property that has to be verified. However, *SPIN* accepts a program written in PROMELA language and thus, the first question is how to translate a C code into PROMELA instructions when verifying a property of interest. If the program is vulnerable, i.e., possesses a "bad" feature, then *SPIN* produces a counterexample that corresponds to the values of internal variables or of input data of the program. We note that, according to SPIN documentation features might be specified as temporal logic formulas or Buchi automata [17]. In the former case, we propose how to inject such data into the program in order to show a programmer which part of the code is vulnerable. The proposed technique somehow takes into account both static and dynamic vulnerability detection, since PROMELA model is verified statically while counterexample is injected into the program through its run-time. In [3], some discussions can be found how to translate C instructions into PROMELA instructions and how the

[2] The work was done together with French scientific group of Prof. Ana Cavalli (TELECOM & Management Sud Paris)

injection procedure can be implemented. In PROMELA language verified properties are described as assertions and such assertions have to be constructed for each type of vulnerabilities. Unfortunately the translation performed by MODEX tool [18] cannot be applied directly and since we are in the process of developing new automatic tools for such translation, some C codes were manually converted into PROMELA codes and corresponding assertions were added. We have applied a proposed technique to the above Programs 1, 2, 3 and SPIN produced counterexamples for all of them. We injected data according to these counterexamples, found out that the programs return wrong results and no error message about “bad” input data has appeared, i.e., *SPIN* has detected type overflow and array overflow vulnerabilities in the above programs. For example, for Program 1 a counterexample produced by *SPIN* has the value 10005 for each array item value, the returned result when running the program was 3451 while the right value should be 100050, i.e., this C code has a type overflow vulnerability.

For Program 2 *SPIN* produced a counterexample as well as for the array dimension as for array item value. In this case when detecting array overflow vulnerability the counterexample was n = 11 when each array item equals 11 too. When detecting type overflow vulnerability *SPIN* produced the value 70035 that was then assigned to each array item. After applying these input data to Program 2 incorrect result has been obtained when running the C program while no error occurred. According to the incorrect result that can easily be checked, one can conclude that *SPIN* has detected type and array overflow in Program 2. For Program 3 (Table 1) SPIN has produced the same counterexample n = 11 for an array overflow while in the counterexample for a type overflow vulnerability, each array item was assigned to 80040.

In order to compare *SPIN* based vulnerability detection technique with other tools providing dynamic vulnerability detection we have run the *Memcheck* utility of Valgrind software [15] against Programs 1, 2, 3. *Memcheck* is designed to detect memory leakages in C/C++ programs and incorrect use of uninitialized values. Valgrind allows a programmer to assign desirable values to input variables and by use of a virtual machine the *Memcheck* utility checks whether memory leakage occurs during the program execution. We have run *Memcheck* against Programs 1, 2, 3 with counterexamples produced by *SPIN* and neither type overflow nor array overflow vulnerability has been mentioned.

Based on the obtained experimental results, we can conclude that *SPIN* based detection techniques could be useful when analyzing the C code safety.

V. Conclusions

In this paper, we have presented a short survey of existing tools providing vulnerability detection in C/C++ programs. Several tools have been executed against student implementations of array algorithms. The experimental results clearly show that for some kinds of C code vulnerabilities static analysis can be insufficient and we have presented a brief overview of a *SPIN*-based approach for vulnerability detection. The obtained preliminary results clearly show that *SPIN* based detection techniques could be useful when analyzing the C code safety. In this paper, we did not discuss vulnerability detection techniques based on other model checkers; such a comparison is a part of our future work.

References

[1] G. Holzmann. Spin Model Checker. Primer and Reference Manual. Addison Wesley, 2003.

[2] Willy Jimenez, Amel Mammar, and Ana R. Cavalli. Software Vulnerabilities, Prevention and Detection Methods. A Review, SEC-MDA workshop.– Enschede, The Netherlands, June, 24, 2009.

[3] Technical report of the joint FCP Russian-French grant № 02.514.12.4002, Step 4.

[4] Cigital [Electronic resource] – http://www.cigital.com/its4/

[5] Flawfinder home page [Electronic resource] – http://www.dwheeler.com/flawfinder

[6] Just Another Hacker [Electronic resource] – http://www.justanotherhacker.com/projects/graudit/download.html

[7] Sound Forge [Electronic resource] – http://sourceforge.net/apps/mediawiki/cppcheck/

[8] Digitek Labs [Electronic resource] – http://www.digiteklabs.ru/aegis/

[9] Digitek Labs [Electronic resource] – http://aegis-demo.digiteklabs.ru/s2a.webserver/

[10] Department of computer science. University of Maryland [Electronic resource] – http://www.cs.umd.edu/~jfoster/cqual/

[11] Soft Line [Electronic resource] – http://soft.softline.ru/NPO-Echelon/eshelon-ak-vs/

[12] Klocwork [Electronic resource] – http://www.klocwork.com/products/insight/klocwork-truepath/

[13] Coverity [Electronic resource] – http://www.coverity.com/products/static-analysis.html

[14] Electrical engineering and computer sciences [Electronic resource] – http://www.cs.berkeley.edu/~daw/mops/

[15] Electrical engineering and computer sciences [Electronic resource] – http://www.cs.berkeley.edu/~daw/boon/

[16] Valgrind [Electronic resource] – http://valgrind.org/info/tools.html

[17] SPIN [Electronic resource] – http://spinroot.com/

[18] Modex [Electronic resource] – http://cm.bell-labs.com/cm/cs/what/modex/index.html