HAL open science

Using testing techniques for vulnerability detection in C programs

Amel Mammar, Ana Rosa Cavalli, Willy Ronald Jimenez Freitez, Wissam Mallouli, Edgardo Montes de Oca

To cite this version:

Amel Mammar, Ana Rosa Cavalli, Willy Ronald Jimenez Freitez, Wissam Mallouli, Edgardo Montes de Oca. Using testing techniques for vulnerability detection in C programs. 23th International Conference on Testing Software and Systems (ICTSS), Nov 2011, Paris, France. pp.80-96, 10.1007/978-3-642-24580-0_7 . hal-01303013

HAL Id: hal-01303013

https://hal.science/hal-01303013v1

Submitted on 15 Apr 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Using testing techniques for vulnerability detection in C programs*

Amel Mammar[1], Ana Cavalli[1], Willy Jimenez[1],
Wissam Mallouli[2] and Edgardo Montes de Oca[2]

[1] Télécom SudParis, SAMOVAR
9 rue Charles Fourier, 91011 Evry Cedex, France
{amel.mammar,ana.cavalli,willy.jimenez}@it-sudparis.eu
[2] Montimage, 39 rue Bobillot Paris 75013, France
{wissam.mallouli,edgardo.montesdeoca}@montimage.com

Abstract. This paper presents a technique for vulnerability detection in C programs. It is based on a vulnerability formal model called "Vulnerability Detection Conditions" (VDCs). This model is used together with passive testing techniques for the automatic detection of vulnerabilities. The proposed technique has been implemented in a dynamic code analysis tool, TestInv-Code, which detects the presence of vulnerabilities on a given code, by checking dynamically the VDCs on the execution traces of the given program. The tool has been applied to several C applications containing some well known vulnerabilities to illustrate its effectiveness. It has also been compared with existing tools in the market, showing promising performances.

Keywords: Dynamic Code Analysis, Vulnerabilities Detection, Passive Testing.

1 Introduction

1.1 Context and motivations

The detection of vulnerabilities[3] in software has become a major concern in the software industry. Although efforts are being made to reduce security vulnerabilities in software, according to published statistics, the number of vulnerabilities and the number of computer security incidents resulting from exploiting these vulnerabilities are growing [7].

One of the reasons for this is that information on known vulnerabilities is not easily available to software developers, or integrated into the tools they use. Thus many activities are designed to support secure software development like security education on vulnerability causes, security goal and vulnerability class identification, goal

* The research leading to these results has received funding from the European ITEA-2 project DIAMONDS.

[3] In this paper, a vulnerability is defined as a specific instance of not intended functionality in a certain product/environment leading to degradation of security properties or violation of the security policy. It can be exploited by malicious code or misuse.

and vulnerability driven inspections etc. Vulnerability cause presence testing is one of the main activities that support the validation of secure software. It is used to detect vulnerabilities in software products in order to remove/mitigate them. Several testing techniques can be used to perform this detection based on different models and approaches (static/dynamic code analysis, fuzz testing, active/passive testing, etc.). In this paper, we present a systematic approach to increase software security by bridging the gap between security experts and software practitioners. Indeed, we provide providing software developers with the means to effectively prevent and remove occurrences of known vulnerabilities when building software. To achieve this goal, we will rely on a formal method for dynamic code analysis technique based on vulnerability detection conditions (VDCs) models.

Currently, there are a large number of techniques and related tools that help developers improve software security quality. Among these techniques, we can cite formal verification and validation (V&V)[11] and also the static and dynamic code analyzers [20,16]. However, existing approaches are often limited and do not present rigorous descriptions of vulnerabilities they deal with [9,12,15]. It is quite difficult for a user to know which vulnerabilities are detected by each tool since they are poorly documented. A more detailed description of the related work is provided in section 2.

1.2 Contribution

Our approach combines a new formalism called *Vulnerability Detection Conditions* (VDCs) and formal passive testing in order to implement a new method to detect vulnerabilities in C programs. These two concepts are detailed respectively in sections 3 and 4.

A VDC allows to formally describe a vulnerability without ambiguity. This task is performed by a security expert that needs to study vulnerabilities then determine its **causes**. Each cause needs to be extracted and translated into a logical predicate on which it becomes possible to reason. In a second step, VDCs descriptions are instantiated by a dynamic analysis tool to allow the automatic detection of this vulnerability in any C program. The tool is based on passive testing technique, which has proven to be very effective for detecting faults in communication protocols [4]. In summary, the main contributions introduced by this paper are:

- A new formalism, called *Vulnerability Detection Conditions* (VDCs), is designed to describe vulnerability causes in a rigorous way without ambiguity. This formalism also constitutes a good way to have a good understanding of each software vulnerability and its causes. It bridges the gap between security experts, developers and testers.
- An editor tool to build new VDCs based on a set of know vulnerability causes described in the the SHIELDS SVRS[4].

[4] The SHIELDS SVRS is a centralized repository that allows the storage and sharing security models in order to reduce known security vulnerabilities during software development

- A model-based dynamic analysis tool *TestInv-Code*[5] that automatically detects vulnerabilities in C programs based on VDCs;
- An end-to-end methodology that allows to detect vulnerabilities and provides for each detection information about the vulnerability, the different ways to avoid it and the C code line where the vulnerability occurs (if the code is available).
- Application of the approach and the obtained results on an open source application XINE that contains a known vulnerability.

The approach proposed in this paper is original since it covers all the steps of vulnerability detection, from the modelling phase relying on VDCs, to their automatic detection on the executable traces using the *TestInv-Code* tool.

The rest of the paper is organized as follows. The section 2 presents different approaches used in literature for dynamic detection of vulnerabilities. Section 3 introduces the VDC model, its basics and use. Section 4 introduces the dynamic code analysis technique based on these models and its tool TestInv-Code. Section 5 introduces the experimentation and results and Section 6 summarizes our work and describes future work.

2 Related work

Different techniques have been proposed to perform dynamic detection of vulnerabilities [3]. *Fuzz testing* is an approach that has been proposed to improve the security and reliability of system implementations [14]. Fuzz testing consists in stimulating the system under test, using random inputs or mutated ones, in order to detect unwanted behavior as crashing or confidentiality violation. *Penetration testing* is another technique that consists in executing a predefined test scenario with the objective to detect design vulnerabilities or implementation vulnerabilities [22]. *Fault injection* is a similar technique that injects different types of faults in order to test the behavior of the system [10]. Following a fault injection the system behavior is observed. The failure to tolerate faults is an indicator of a potential security flaw in the system. These techniques have been applied in industry and shown to be useful. However, most of the current detection techniques based on these approaches are ad hoc and require a previous knowledge of the target systems or existing exploits.

Model checking techniques have also been revisited for vulnerability detection. Hadjidj et al.[13] present a security verification framework that uses a conventional push down system model checker for reachability properties to verify software security properties. Wang et al. [23] have developed a constraint analysis combined with model checking in order to detect buffer overflow vulnerabilities. The memory size of buffer-related variables is traced and the code is instrumented with constraints assertions before the potential vulnerable points. The vulnerability is then detected with the reachability of the assertion using model checking. All model checking works are

[5] TestInv-Code testing tool is one of Montimage tools (http://www.montimage.com). It is a dynamic code analysis tool that aims at detecting vulnerabilities by analyzing the traces of the code while it is executing.

based on the design of a model of the system, which can be complex and subject to the combinatorial explosion of the number of states.

In the *dynamic taint approach* proposed by Chess and West [8], tainted data are monitored during the execution of the program to determine its proper validation before entering sensitive functions. It enables the discovery of possible input validation problems which are reported as vulnerabilities. The sanitization technique to detect vulnerabilities due to the use of user supplied data is based on the implementation of new functions or custom routines. The main idea is to validate or sanitize any input from the users before using it inside a function. Balzarotti et al. [2] present an approach using static and dynamic analysis to detect the correctness of sanitization process in web applications that could be bypassed by an attacker.

3 Vulnerability Modelling

In order to describe the presence of a vulnerability in a program, we rely in this paper on Vulnerability Detection Conditions (VDCs) formalism. VDCs basically indicate that the execution of an action under certain conditions could be dangerous or risky for the program. They permit to express in a logical predicate the different causes that lead to the considered vulnerability. The main idea behind the definition of the VDC formalism is to point out the use of a dangerous action under some particular conditions, for instance "it is dangerous to use unallocated memory". Thus, if we evaluate a piece of code where we find such VDC we know that it is vulnerable.

3.1 Definitions

Definition 1. *(Vulnerability Detection Condition). Let Act be a set of action names, Var be a set of variables, and P be a set of predicates on (Var $\cup$ Act). We say that Vdc is a vulnerability detection condition if Vdc is of the form (long brackets denote an optional element):*

$$Vdc ::= a/P(Var,Act)|a[/P(Var,Act)];P'(Var,Act)$$

where a denotes an action, $P(Var,Act)$ and $P'(Var,Act)$ represent any predicates on variables Var and actions Act. A vulnerability detection condition $a/P(Var,Act)$ means that action a occurs when specific conditions denoted by predicate $P(Var,Act)$ hold.

Similarly, a vulnerability detection condition $a[/P(Var,Act)];P'(Var,Act))$ means that action a used under the optional conditions $P(Var,Act)$ is followed by a statement whose execution satisfies $P'(Var,Act)$. Naturally, if action a is not followed by an action, the predicate $P'(Var,Act)$ is assumed to be true.

More complex vulnerability detection conditions can be built inductively using the different logical operators according to the following definition.

Definition 2. *(General Vulnerability Detection Conditions). If Vdc_1 and Vdc_2 are vulnerability detection conditions, then $(Vdc_1 \vee Vdc_2)$ and $(Vdc_1 \wedge Vdc_2)$ are also vulnerability detection conditions.*

3.2 Some examples

Let us define a vulnerability detection condition Vdc_1 that can be used to detect possible accesses to a free or an unallocated memory. If we denote by $Assign(x,y)$ the assignment of value y to the memory variable x and *IsNot_Allocated* a condition to check if memory x is unallocated then the VDC is given by the expression:

$$Vdc_1 = Assign(x,y)/IsNot_Allocated(x)$$

In programming languages like C/C++, there are some functions that might lead to a vulnerability if they are applied on out-of-bounds arguments. The use of a tainted variable as an argument to a memory allocation function (e.g. `malloc`) is a well-known example of such a vulnerability, which is expressed by the vulnerability detection condition Vdc_2 below. A variable is tainted if its value is obtained from a non-secure source, or in other words, produced by reading from a file, getting input from a user or the network, etc.

$$Vdc_2 = memoryAllocation(S)/tainted(S)$$

3.3 Describing vulnerabilities with formal Vulnerability Detection Conditions

An informal description of a vulnerability states the conditions under which the execution of a dangerous action leads to a possible security breach. So, it should include the following elements:

1. *A master action*: an action denotes a particular point in a program where a task or an instruction that modifies the value of a given object is executed. Some examples of actions are variable assignments, copying memory or opening a file. A master action *Act_Master* is a particular action that produces the related vulnerability.
2. A set of conditions: a condition denotes a particular state of a program defined by the value and the status of each variable. For a buffer, for instance, we can find out if it has been allocated or not. Once the master action is identified for a scenario, all the other facts are conditions $\{C_1, \ldots, C_n\}$ under which the master action is executed. Among these conditions, a particular condition C_k may exist, called *missing condition*, which must be satisfied by an action following *Act_Master*.

Let $\{P_1, \ldots, P_k, \ldots, P_n\}$ be the predicates describing conditions $\{C_1, \ldots, C_k, \ldots, C_n\}$. The formal vulnerability detection condition expressing this dangerous scenario is defined by:

$$Act/(P_1 \wedge \ldots \wedge P_{k-1} \wedge P_{k+1} \ldots \wedge P_n); P_k$$

Finally, the vulnerability detection condition representing the entire vulnerability is defined as the disjunction of the all sub-vulnerability detection conditions for each scenario (Vdc_i denotes the VDC associated with each scenario i):

$$Vdc_1 \vee \ldots \vee Vdc_n$$

For example, consider the vulnerability CVE-2009-1274, a buffer overflow in *XINE* media player. According to the description, the vulnerability is the result of computing a buffer size by multiplying two user-supplied integers without previously checking the operands or without checking the result of the allocation. An attacker may cause the execution of arbitrary code by providing a specially crafted media file to the user running the *XINE* application. A scenario associated to this vulnerability can be expressed as:

1. An allocation function is used to allocate a buffer
2. The allocated buffer is not adaptive
3. The size used for that allocation is calculated using tainted data (data read from the media file)
4. The result returned by the allocation function is not checked

To define the VDC associated with this scenario, we have to express each of these conditions with a predicate:

Use of malloc/calloc/realloc the program uses C-style memory management functions, such as `malloc`, `calloc` or `realloc` to allocate memory. For each memory function allocation f, applied on value V to allocate a buffer B, the following predicate holds:

$$memoryAllocation(f, B, V)$$

Use of nonadaptive buffers the program uses buffers whose sizes are fixed when they are allocated (allocation may take place at run-time, e.g. `malloc`, or at compile-time). Non-adaptive buffers can only hold a specific amount of data; attempting to write beyond their capacity results in a buffer overflow. Adaptive buffers, in contrast, can adapt themselves to the amount of data written to them. For each declared nonadaptive buffer B, the following predicate holds:

$$nonAdaptiveBuffer(B)$$

User supplied data influences buffer size the size of a dynamically allocated buffer is computed, at least in part, from user-supplied data. This allows external manipulation of the buffer size. If a buffer is made too large, this may result in a denial of service condition; if it is too small, then it may later result in a buffer overflow. For each variable V whose value is produced from an insecure source, the following predicate holds:

$$tainted(V)$$

Note that a tainted variable will be untainted if it is bound checked by the program.

Failed to check return value from calloc the program does not contain mechanisms to deal with low memory conditions in a safe manner (i.e. deal with NULL return values from `calloc`). Running out of memory in programs that are not written to handle such a situation may result in unpredictable behavior that can possibly be exploited. This cause is detected when the return value B of an allocation function is not followed by a check statement. For each value B returned from an allocation memory function, the following formula is defined:

$$notChecked(B, null)$$

The vulnerability detection condition expressing this scenario is then defined by:

$$memoryAllocation(f, B, V) / \begin{pmatrix} nonAdaptiveBuffer(B) \\ \wedge \\ tainted(V) \end{pmatrix} ; notChecked(B, null)$$

This last vulnerability detection condition expresses a potential vulnerability when a given allocation function f is used with a non-adaptive buffer B whose size V is produced from an insecure source and its return value is not checked with respect to NULL.

3.4 VDC editor

The VDC editor is a GOAT[6] plug-in, which offers security experts the possibility to create vulnerability detection conditions (VDCs). These VDCs will be used to detect the presence of vulnerabilities by checking software execution traces using Montimage TestInv-Code testing tool. The VDC editor user interface includes some features that allow simplifying the construction and composition of VDCs. The VDC editor has the following functionalities:

- The creation of new VDCs corresponding to vulnerability causes from scratch and their storage in an XML format.
- The visualization of already conceived VDCs.
- The editing (modification) of existing VDCs in order to create new ones.

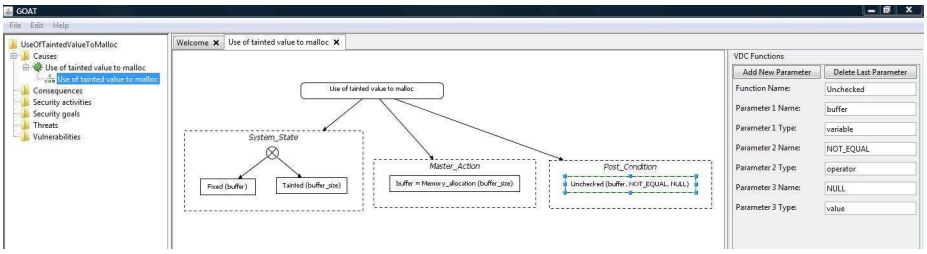

Fig. 1. Vulnerability detection condition for “Use of tainted value to malloc" in GOAT.

The VDCs are stored in an XML file that constitutes one of the inputs for the Montimage TestInv-Code tool. A vulnerability is discovered if a VDC signature is detected on the execution trace. A VDC is composed within the editor of at most 3 parts:

1. *Master condition:* The triggering condition called also master action (denoted a). When analysing the execution trace, if this condition is detected, we should verify if the state and post conditions of the VDC hold as well. If this is the case, then a vulnerability has been detected. The master condition is mandatory in a VDC.

[6] http://www.ida.liu.se/divisions/adit/security/goat/

2. *State condition:* A set of conditions related to the system state (denoted P(Var,Act)). The state condition describes the states of the specified variables at the occurrence of the master action. The state condition is mandatory in a VDC.
3. *Post condition:* A set of conditions related to the system future state (denoted P′(Var,Act)). If a master action is detected in the state condition context, then we should verify if the post condition holds in the execution that follows. If this is the case, a vulnerability has been detected. This post condition is not mandatory in a VDC.

4 Dynamic code analysis for vulnerability detection

4.1 Basics: Passive testing

Our approch for dynamic code analysis is inspired from the classical passive testing technique [1,19,17] designed for telecommunication traffic analysis. Passive testing allows to detect faults and security flaws by examining captured trafic packets (live trafic or log files) according to a set of events-based properties that denote either:

- a set of functional or security rules that the trafic has to fulfill[4,5,18], or
- a set behavioral attacks like those used in classical intrusion and detection systems.

In the case of executable code analysis, events are assimilated to the disassembled instructions that are being executed in the processor. They are produced by executing the program under the control of the *TestInv-Code* tool, similar to what a debugger does.

For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to cover different program behaviours. Use of classical testing techniques for code coverage helps to ensure that an adequate part of the program's set of possible behaviours has been observed. Also, care must be taken to minimize the effect that instrumentation has on the execution (including temporal properties) of the target program.

While static analysis collects information based on source code, dynamic analysis is based on the system execution (binary code), often using instrumentation. The advantages that can be expected from using dynamic analysis are:

- Has the ability to detect dependencies that are not detectable in static analysis. Ex.: dynamic dependencies using reflection, dependency injection etc.
- Allows the collection of temporal information.
- Allows the possibility of dealing with runtime values.
- Allows the identification of vulnerabilities in a runtime environment.
- Allows the use of automated tools to provide flexibility on what to scan for.
- Allows the analysis of applications for which you do not have access to the actual code.
- Allows identifying vulnerabilities that might be false negatives in the static code analysis.
- Permits validating static code analysis findings.
- It can be conducted on any application.

4.2 Using VDCs in *TestInv-Code*

In order to use the *TestInv-Code* tool, the main step consists in defining the vulnerabilities causes that are of interest. Starting from informal descriptions of the vulnerabilities and VDCs models, a set of conditions that lead to a vulnerability are derived. These conditions are formally specified as regular expressions that constitute the first input for *TestInv-Code* tool.

Thus, end-to-end code analysis using *TestInv-Code* proceeds along the following steps:

1. *Informal definition of vulnerable scenarios.* A security expert describes the different scenarios under which a vulnerability may appear. A scenario denotes a set of causes that produces the vulnerability.
2. *Definition of VDC.* A VDC, expressing formally the occurrence of the related vulnerability, is created for each possible situation that leads to the vulnerability using the VDC editor.
3. *Vulnerability checking.* Finally, *TestInv-Code* checks for evidence of the vulnerabilities during the execution of the program. Using the VDCs, it will analyze the execution traces to produce messages identifying the vulnerabilities found, if any, indicating where they are located in the code.

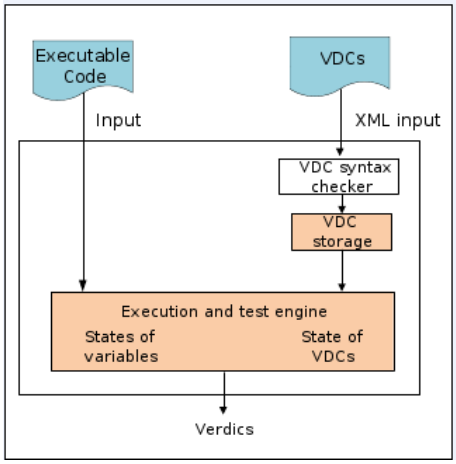

Fig. 2. Passive testing for vulnerability detection.

Figure 2 depicts the passive testing architecture for vulnerability detection. As shown, the *TestInv-Code* tool takes as input:

1. *The vulnerability causes.* The file containing the vulnerabilities causes formally specified using VDCs.

2. *The executable.* The Executable Linked Format (ELF) file for the application that is to be tested. This file contains the binary code of the application and it should include debug information if we want the tool to be able to determine the line of code where the vulnerability occurs and provide this information in the final verdicts.

In order to detect the presence of a VDC in an execution trace, it needs to be processed in such a way that it is detected when and if it occurs during the execution of the program. In the case of *TestInv-Code*, predicates and actions in the VDCs correspond to functions that allow analysing the executed instructions and determining it they are satisfied. The tool keeps information on the state of all the variables used by the program, heap or stack memory addresses and registers. The states, are for instance, tainted or not, bound checked or not, allocated or not etc. It also maintains information on the potential VDCs. The tool is able to detect when a system call is made, the controls that are made on variables or return values from function calls, when buffer allocations are made, etc. Thus it can verify all the conditions that are used in the VDCs and generate messages if the VDCs are satisfied. The symbolic tables are necessary to be able to determine the line of code that provokes the vulnerability that is detected.

It must be noted that the functions used to detect the VDC conditions could vary depending on the execution environment, the compiler and the compilation options used. In this work we assume that the execution environment is Linux version 2.6, the compiler is gcc version 4.3.3 and that the compilation was performed for debugging (including symbolic tables) and without any optimisations. Other variants could work but this has not yet been tested on other platforms. Certain optimizations performed by the compiler could make it necessary to adapt the algorithms of the functions to certain particularities introduced by the compiler.

5 Experiment and Results

5.1 XINE application

We demonstrate the application of our vulnerability detection method to an open source application and free multimedia player that plays back audio and video, XINE[7] written in C. This application was selected as an example since it is a real world application, open source (so the source files are available free of copyright), and contains a number of known vulnerabilities which can be used to demonstrate the effectiveness of our approach.

The application contains a set of modules and librairies. The one we are concentrated on is *xine-lib*[8] (xine core). This is a module developed in C language and which has several vulnerabilities inside its files. We will select an obsolete version of xine-lib so we can use the vulnerabilities found in them.

[7] http://www.xine-project.org
[8] Xine-lib source code can be downloaded from: http://sourceforge.net/projects/xine.

5.2 Xine selected vulnerability

The xine v1.1.15 application has a number of vulnerabilities. The one that we will deal with is CVE-2009-1274.

- Summary: Integer overflow in the qt_error parse_trak_atom function in de-muxers/demux_qt.c in xine-lib 1.1.16.2 and earlier allows remote attackers to execute arbitrary code via a Quicktime movie file with a large count value in an STTS atom, which triggers a heap-based buffer overflow.
- Published: 04/08/2009
- CVSS Severity: 5.0 (MEDIUM)

The exploitation occurs when someone is trying to play with xine a Quicktime encod-ed video that an attacker has modified to make one of its building blocks (the “time to sample" or STTS atom) have an incorrect value. The malformed STTS atom processing by xine leads to an integer overflow that triggers a heap-based buffer over-flow probably resulting in arbitrary code execution. The patch to this Vulnerability is in v1.1.16.1 that is also included in the v1.1.16.3.

CVE-2009-1274 is a vulnerability instance and can be considered as part of the family or class of vulnerabilities named “Integer Overflow" has the ID CWE 190 in the Common Weakness Enumeration database. The CWE 190 description is summarised as follows “The software performs a calcu-lation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control" [21].

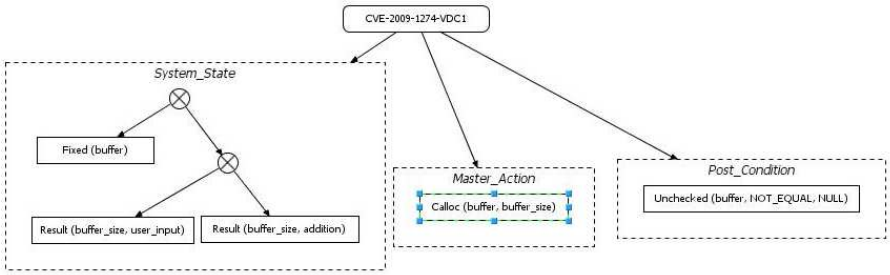

Fig. 3. VDC model of CVE-2009-1274 vulnerability.

5.3 Vulnerability modelling

Starting from the informal description of CVE-2009-1274 vulnerability, we have de-signed the 3 VDCs and the corresponding regular expressions to be used for input to the *TestInv-Code* tool.

1. Calloc(buffer, buffer_size) / Fixed(buffer) ∧ Result(buffer_size, user_input)∧ Result(buffer_size, addition); Unchecked(buffer, NULL)
2. Calloc(buffer, buffer_size) / Fixed(buffer) ∧ Result(buffer_size, user_input) ∧ Result(buffer_size, addition) ∧ Unchecked(buffer_size, buffer_bounds)
3. CopyVar(loop_counter, user_input) / Fixed(buffer) ∧ Unchecked(loop_counter, counter_bounds); CopyData(buffer, user_input, loop_counter)

Using the VDC editor, we can build the VDC models for each cause scenario. Figure 3 illustrates the VDC model for the first scenario.

5.4 Application of TestInv-Code

The created VDCs are one of the inputs needed by the TestInv-C testing tool. In order to analyse the xine-lib it is necessary to use it. To be able to reach the plug-in that contains the error (the quicktime file demuxer), the muxine application was run on a quicktime file. The TestInv-Code tool allows performing the analysis on all the application's functions (including those of the library and the plug-ins). The user can also identify a given function or set of functions that he wants to analyse. Using this feature is necessary to avoid performance issues, particularly in applications that perform intensive data manipulations (like video players). The complete list of available functions can be obtained automatically. Another feature that helps improve the performance of the tool is the possibility of limiting the number of times a piece of code in a loop is analysed. The following *XINE* code is executed:

```
Code fragment from demux_qt.c
...
1907 trak->time_to_sample_table = calloc(
1908  trak->time_to_sample_count+1,
         sizeof(time_to_sample_table_t));
1909 if (!trak->time_to_sample_table) {
1910   last_error = QT_NO_MEMORY;
1911   goto free_trak;
1912 }
1913
1914 /* load the time to sample table */
1915 for(j=0;j<trak->time_to_sample_count;j++)
...
```

where `trak->time_to_sample_table` is tainted since it is set from information taken from the external QuickTime file.

The tool will detect the particular vulnerability used here (CVE-2009-1274) when it is launched on the muxine application using a quicktime video file. This needs to be done using the option to analyse all the functions (of the application, the library and the plug-ins) or just the function parse_trak_atom in the quicktime plug-in. The result of the vulnerability cause presence testing activity provided by TestInv-Code is shown in figure 4.

MyProjects-->Xine Demo Testing Project -->TestLabs-->Xine Test Lab 1

Testlab Execution Results Report

Project Name:	Xine Demo Testing Project
Project Description:	This is a TestInv-C Demo testing project for Xine application.
Testlab Name:	Xine Test Lab 1
Testlab Description:	This is a test lab that includes the VDCs for vulnerability CVE-2009-1274.
Execution start time:	2010-05-17 20:31:52

Testlab coverage

VDC Model
CVE-2009-1274-VDC1
CVE-2009-1274-VDC2
CVE-2009-1274-VDC3

VDCs coverage: 60%(3/5)

Test results summary

TestLab	Xine Test Lab 1	Passed VDCs		0/3
		Failed VDCs		3/3

Test results

VDC	Defect	Priority	Verdict
CVE-2009-1274-VDC1	Defect: CVE-2009-1274-VDC1	Essential	
CVE-2009-1274-VDC2	Defect: CVE-2009-1274-VDC2	Essential	
CVE-2009-1274-VDC3	Defect: CVE-2009-1274-VDC3	Essential	

Fig. 4. Screenshot of TestInv-Code result for xine vulnerability.

5.5 Analysis

If we apply the same VDCs to other code under the same programming environment, we will be able to detect the same types of vulnerabilities. For instance, we applied the same

VDCs on ppmunbox, a program developed by Linköpings university to remove borders from portable pixmap image files (ppm) and we detected the same vulnerabilities.

This vulnerability is located in the ppmunbox.c file specifically in the following:

```
Code fragment from ppmunbox.c
...
76:/* Read the dimensions */
77:if(fscanf(fp_in,"%d%d%d",&cols,&rows &maxval)<3){
78: printf("unable to read dimensions from PPM file");
79: exit(1);
80 }
81:
82:/* Calculate some sizes */
83:pixBytes = (maxval > 255) ? 6 : 3;
84:rowBytes = pixBytes * cols;
85:rasterBytes=rows;rasterBytes=rowBytes*rows;
86:
87:/* Allocate the image */
88:img = malloc(sizeof(*img));
89:img->rows = rows;
90:img->cols = cols;
91:img->depth = (maxval > 255)?2:1;
92:p = (void*)malloc(rasterBytes);
93:img->raster = p;
94:
95:/* Read pixels into the buffer */
96:while (rows--) {
...
```

To illustrate the applicability and scalability of TestInv-Code, it has been applied to six different open source programs to determine if known vulnerabilities can be detected using a single model. The following paragraphs describe the vulnerabilities and give a short explanation of the results obtained. The results are summarized in table 1.

Table 1. Summary of results running TestInv-Code with VDC codes

Vulnerability	Software	Detected ?
CVE-2009-1274	Xine	Yes
Buffer overflow	ppmunbox	Yes
CVE-2004-0548	aspell	Yes (two)
CVE-2004-0557	SoX	Yes
CVE-2004-0559	libpng	Yes
CVE-2008-0411	Ghostscript	Yes

Besides, the application of the tool to the case study gave good performances. We did some experiments in order to check the scalability of the tool by the application of a high number of VDCs (more than 100) to a software data intensive (as in the case of video decoders). The tool performance remains good. We compared the performance of our tool according to known

dynamic code analysis tools in the market like Dmalloc[9], DynInst [10], and Valgrind[11] and the results were comparable. Indeed, the detection based on our tool does not insert a big overhead (the execution time is almost equal to the programm execution time).

To optimize our analysis, the tool is being modified so that the user can select specific functions to check in the program. But in this case all the input parameters for this function are marked as tainted even if they are not. Another solution that is being studied is to only check the first iteration of loops in the program, thus avoiding to check the same code that is executed more than once.

At present, we have checked applications written in C, which do not have a complex architecture. We are now starting to experiment more complex applications with architectures that integrate different modules, plugins, pointers to function, variable number of parameters or mixing different programming languages.

6 Conclusions and future work

Security has become a critical part of nearly every software project, and the use of automated testing tools is recommended by best practices and guidelines. Our interest lies in defining a formalism, called *Vulnerability Detection Conditions*, to describe vulnerabilities so we can detect them using automated testing.

In this paper, we have also shown how a model-based dynamic code analysis tool, *TestInv-Code*, is used to analyze execution traces and determine if they show evidence of a vulnerability or not. VDCs can be very precise, we believe making it possible to detect vulnerabilities with a low rate of false positives. This is planned to be studied and demonstrated in future work.

Since the vulnerability models are separate from the tool, it is possible for any security expert to keep them up-to-date and to add new models or variants. It also becomes possible for the tool user to add e.g. product-specific vulnerabilities and using the tool to detect them. This is very different from the normal state of affairs, where users have no choice but to rely on the tool vendor to provide timely updates. Nevertheless, it should be noted that if new predicates or actions are required, the function that will allow to detect them needs to be added to the tool.

The work presented in this paper is part of the SHIELDS EU project [21], in which we have developed a shared security repository through which security experts can share their knowledge with developers by using security models. Models in the SHIELDS repository are available to a variety of development tools; *TestInv-Code* is one such tool.

Looking to the future, we plan on applying the methods presented here to various kinds of vulnerabilities in order to identify which predicates are required, and whether the formalism needs to be extended.

---

[9] Dmalloc is a library for checking memory allocation and leaks. Software must be recompiled, and all files must include the special C header file dmalloc.h.

[10] DynInst is a runtime code-patching library that is useful in developing dynamic program analysis probes and applying them to compiled executables. Dyninst does not require source code or recompilation in general, however non-stripped executables and executables with debugging symbols present are easier to instrument.

[11] Valgrind runs programs on a virtual processor and can detect memory errors (e.g. misuse of malloc and free) and race conditions in multithread programs.

References

1. B. Alcalde, A. R. Cavalli, D. Chen, D. Khuu, and D. Lee. *Network Protocol System Passive Testing for Fault Management: A Backward Checking Approach*, In FORTE, pages 150–166, 2004.
2. D. Balzarotti, M. Cova, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. *Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications*. In IEEE Symposium on Security & Privacy, pages 387–401, 2008.
3. S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, A. Vincent. *The BINCOA Framework for Binary Code Analysis*. CAV conference. pp 165-170. 2011.
4. E. Bayse, A. Cavalli, M. Núnez, and F. Zaidi. *A Passive Testing Approach Based on Invariants: Application to the Wap*. Computer Networks and ISDN Systems, 48(2):247–266, 205.
5. A. R. Cavalli, C. Gervy, and S. Prokopenko. *New Approaches for Passive Testing using an Extended Finite State Machine Specification*. Information & Software Technology, 45(12), pages 837–852, 2003.
6. A. R. Cavalli and D. Vieira. *An Enhanced Passive Testing Approach for Network Protocols*. In ICN,ICONS,MCL,pages 169–169, 2006.
7. CERT Coordination Center. CERT/CC statistics. (accessed October 2007).
8. B. Chess and J. West. *Dynamic Taint Propagation: Finding Vulnerabilities without Attacking*. Information Security Technical Report, 13(1):33–39, 2008.
9. Coverity. *Prevent*. (accessed September 2008).
10. W. Du and A. Mathur. *Vulnerability Testing of Software System using Fault Injection*. In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), Workshop on Dependability Versis Malicious Faults, 2000.
11. Stefan Fenz, Andreas Ekelhart. *Verification, Validation, and Evaluation in Information Security Risk Management*. IEEE Security and Privacy (IEEESP) 9(2):58-65. 2011.
12. Fortify Software. *Fortify SCA*.(accessed September 2008).
13. R. Hadjidj, X. Yang, S. Tlili, and M. Debbabi. *Model Checking for Software Vulnerabilities Detection with Multi-Language Support*. In Sixth Annual Conference on Privacy, Security and Trust, pages 133–142, 2008.
14. M. Howard. *Inside the Windows Security Push*. In IEEE Symposium on Security & Privacy, pages 57–61, 2003.
15. Klocwork. *K7*. (accessed September 2008).
16. C. Kuang, Q. Miao, and H. Chen. *Analysis of Software Vulnerability*. In ISP 06: Proceedings of the 5th WSEAS International Conference on Information Security and Privacy, pages 218–223, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).
17. D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John. *Passive Testing and Applications to Network Management*. In ICNP 97: Proceedings of the 1997 International Conference on Network Protocols (ICNP 97), Washington, DC, USA, 1997. IEEE Computer Society.
18. W. Mallouli, F. Bessayah, A. Cavalli, and A. Benameur. *Security Rules Specification and Analysis Based on Passive Testing*. In The IEEE Global Communications Conference (GLOBECOM 2008), 2008.
19. R. E. Miller and K. A. Arisha. *Fault Identification in Networks by Passive Testing*. In Advanced Simulation Technologies Conference (ASTC), pages 277–284. IEEE Computer Society, 2001.
20. S. Redwine and N. Davis. *Processes to Produce Secure Software*. 2004. Task Force on Security Across the Software Development Lifecycle, Appendix A.
21. SHIELDS: *Detecting Known Security Vulnerabilities from within Design and Development Tools*. “D1.4 Final SHIELDS approach guide".

22. H. Thompson. *Application of Penetration Testing*. In IEEE Symposium on Security & Privacy, pages 66–69, 2005.
23. L. Wang, Q. Zhang, and P. Zhao. *Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking*. In Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, pages 165–173, 2008.