Development of a Source Code Analysis Aid Tool Focusing on Security

L.O. Duarte[1] and A. Montes[2]
[1]Laboratory for Computing and Applied Mathematics - LAC
Brazilian National Institute for Space Research - INPE
C. Postal 515 – 12245-970 – São José dos Campos - SP
BRAZIL

[2]Information Systems Security Division - DSSI
Renato Archer Research Center – CENPRA
13069-901 – Campinas - SP
BRAZIL

E-mail: duarte@lac.inpe.br, antonio.montes@cenpra.gov.br

This work presents a proposal of a source code analysis aid tool focused on security. Its main goal is to help developers to find vulnerabilities in their own software. The proposed tool analyzes a software source code to find buffer overflow vulnerabilities through a preventive and software-dependent approach, in a syntactic level. To achieve it, the environment tries to supply the limitations found in other tools.

Keywords: secure programming, syntactic analysis, secure coding, weak functions, security.

1. INTRODUCTION

In the past years, the expressive increase in the development of computer technologies has been followed by an increasing number of systems being probed, infected, compromised and used to launch attacks on other systems. Successful attacks against computer systems are due to the exploitation of some kind of vulnerability. Poor programming techniques are the main cause of vulnerabilities. The more common vulnerabilities are:

- **Buffer Overflow:** It is the most common vulnerability present in software. It often occurs when more data is inserted in a buffer than it can handle. (Pincus, 2004), (Aleph One, 1996)
- **Race Condition:** This vulnerability occurs when two or more processes try to access the same resource simultaneously. One process can intentionally change a resource in such a way that a second process behaves unexpectedly. (Cowan *et* al, 2001a)
- **Injections:** It is a vulnerability associated with poor software input validation causing situations such as unexpected instruction or SQL command execution.
- **Format String Vulnerability:** It is a vulnerability in the format string of functions like printf() and syslog() that can allow information leak, unauthorized accesses to memory locations or access to the system by an attacker. (Cowan *et* al, 2001b)

Statistics of NVD/NIST (National Vulnerability Database / National Institute of Standards and Technologies) show that the number of discovered vulnerabilities in software is increasing. The number of indexed vulnerabilities from January to September of 2006 was bigger than all 2005 year vulnerabilities (Figure 1). Input validation errors are the main cause of those vulnerabilities. The buffer overflow vulnerability is related to validation input error and is the focus of this work.

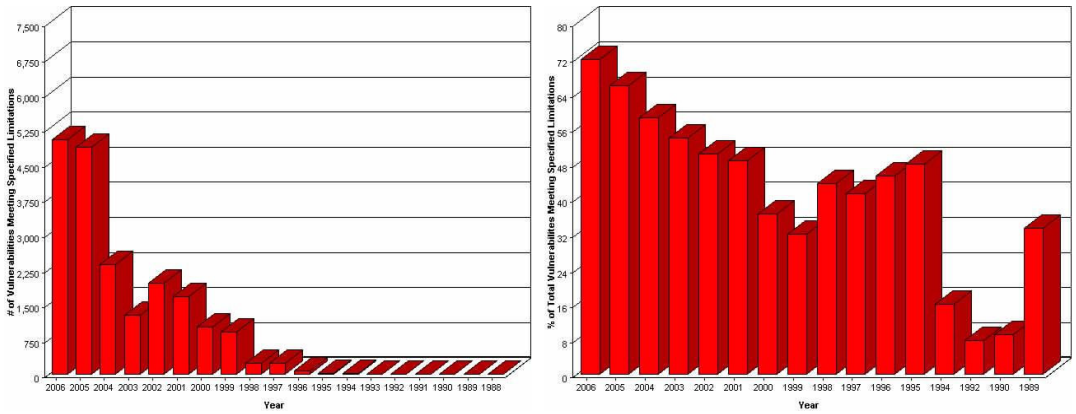

Figure 1: The left graphic represents the number of indexed vulnerabilities. The percentage of indexed vulnerabilities due to input validation errors is shown in the right graphic.

2. MITIGATION EFFORTS FOR BUFFER OVERFLOW

There are several methods to treat problems related to buffer overflow. Those methods could minimize the impact of a vulnerability exploitation or could completely mitigate the vulnerability. These methods can be classified as:

- Compiler dependent approach; (Etoh, 2001) (Vendicator, 2000)
- System dependent approach;
- Software dependent approach.

2.1. Compiler Dependent Approach

This approach inserts some additional information that can be some short checking routines in the program at the compilation step. So, vulnerabilities can be found at execution time. Tools that use those techniques generally are integrated to the system compiler. For this reason they are called compiler dependent. Programs can be normally compiled and be protected at the end of the compilation process.

A huge programmer interaction is not needed in this approach. So, this approach is extremely useful to mitigate vulnerabilities in large programs, with some millions of source-code lines. In the Table 1, four well known tools that use compiler dependent approach, and changes in compiler time to make the application secure in running time, are summarized.

Table 1: Example of compiler dependent tools.

Tool	Effect	License
StackGuard	The program ends when a buffer overflow exploitation attempt occurs.	GPL
ProPolice	The program ends when a buffer overflow exploitation attempt occurs.	GPL
StackShield	The program ends when a buffer overflow exploitation attempt occurs.	GPL
FormatGuard	The program ends when a format string exploitation attempt in the printf function family occurs.	GPL

From the tools showed in the table above, only StackGuard, Propolice and StackShield treat buffer overflow problems. The FormatGuard tool treats only Format String problems.

The StackGuard and Propolice tools use a so called canary or guardians. The goal of these guardians is to not allow the modification of the return address in a stack frame. As it is known, the return address modification is one of the main steps of a buffer overflow exploitation attack.

The technique used in these tools is basically to insert a guardian that could not be modified between the function variables and the return address. In other words, in case of a variable overflow, a guardian exists before the return address and the guardian can not be modified because it is checked at the function epilog. An example of a code stretch and the respective stack frame with a guardian are showed at picture Figure 2 and 3.

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];

   gets(buffer1);
}

int main(void) {
   function(1,2,3);
   return(0);
}
```

Figure 2: The source code that was used to generate the two stack frames of the Figure 3.

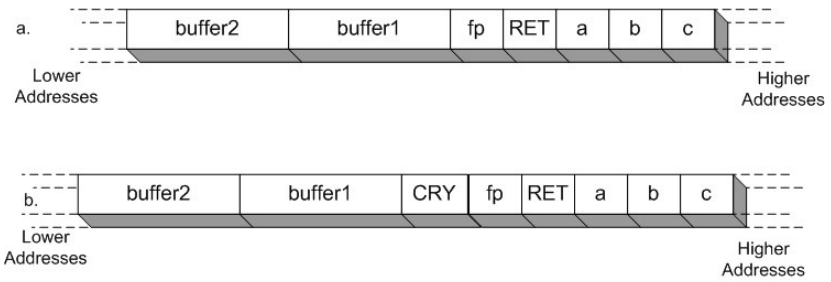

Figure 3: The stack frame showed in a. is related with a system without canary implementations. The stack frame showed as b. is related with a system with a canary implementation.

The StackShield tool does not use guardians, but stores the return address value before the function call in a secure address in memory. In this way, the value stored in memory is checked at the function epilogs. If occurs a modification in the return address content, the program ends.

Some research like the one done in (Bulba e Kil3r, 2000) shows that it is possible to bypass the entire security system generated by those tools, using attacks methods like pointer subterfuge ones. The pointer subterfuge is a buffer overflow exploitation technique that uses a buffer overflow to change the address that a pointer is pointing, in a way that a pointer can point to an arbitrary address in the stack. This address can be, for example, the return address of the function or the secure address used by StackShield to store the original return address value.

Using the pointer subterfuge technique the modification of the return address value is possible without change the guardian value. Although the StackGuard mitigates this problem using random canaries XORed with the original return address value, the problem can occurs in other ways. (Bulba e Kil3r, 2000)

2.2. System Dependent Approach

The system dependent approach describes the protection entirely in the execution time, usually making the hardening of the libraries and the operation system kernel. Among the available tools, the one that treat the buffer overflow problems are included in the behavior blockers category.

The behavior blockers avoid that some behaviors occurs in the system, because these one are well known related to an attack. Some behavior blockers techniques are implemented in open source tools like Openwall, Libsafe, RaceGuard and Systrace.

The Openwall is a Linux Kernel patch that aims to increase the security level of the entire system using behavior blockers. It implements non executable stacks, avoiding shellcode insertion in functions variables that can be exploited. Users that do not have superuser rights could not create links to files that they don't own. Finally, process with superuser privileges should not follow links.

The non executable stack mitigates the standard stack smash attack against buffers. The other features implemented by Openwall are used to minimize other problems with race condition and symbolic link for example. These other problems are not covered in this work.

The Libsafe is a wrapper for the standard C library. The main goal of this wrapper is to check the arguments that are passed to the glibc functions, verifying if those arguments are reasonable and will not cause security problems. The way that Libsafe works can be seen in Figure 4. This approach is able to prevent buffer overflow and format string exploitation in the standard C library functions. An example of the Libsafe implementation, over a glibc function is showed in Figure 5.

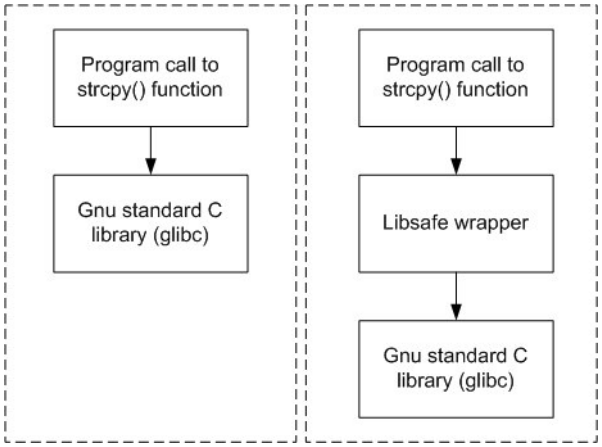

Figure 4: The original strcpy() function call is showed in left part of the figure, the strcpy() function call in a system with Libsafe is showed in the right part of the figure.

```
char *strcpy(char *dest, const char *src)
{
    static strcpy_t real_strcpy = NULL;
    size_t max_size, len;

    if (!real_memcpy)
        real_memcpy = (memcpy_t)
getLibraryFunction("memcpy");
    if (!real_strcpy)
        real_strcpy = (strcpy_t)
getLibraryFunction("strcpy");

   if ((max_size = _libsafe_stackVariableP(dest)) == 0) {
        LOG(5, "strcpy(<heap var> , <src>)\n");
        return real_strcpy(dest, src);
   }

 -- CUT --

  if ((len = strnlen(src, max_size)) == max_size)
        _libsafe_die("Overflow caused by strcpy()");
    real_memcpy(dest, src, len + 1);
    return dest;
}
```

Figure 5: The strcpy reimplementation made by Libsafe. Note that all needed checks are made before the real_memcpy() function call.

The RaceGuard was developed to avoid race condition exploitation in the temporary file creation. Generally, an time interval exists among the check, creation and open of those type of archive. This interval can permit a race condition attack. The RaceGuard approach is to make the action of check, creation and open to be atomic.

The Systrace is a hybrid implementation of access control and behavior blocker that allows the system administrator to specify which resources a given process can accesses. The Systrace do not mitigate or take any actions when an exploitation occurs. However it give less access privileges to an attacker that has effectively exploited the system.

2.3. Software Dependent Approach

The Software Dependent Approaches aims to prevent the vulnerabilities before the software release. A so called software auditing must be done in this approach. The software audit processes look for problems inserted in the coding stage. This audit can be done in a static or dynamic way. The software dynamic analysis aims to discover problems during the software execution, often without analysing the source-code. This technique demands that all the execution paths of the software to be covered and exhaustively tested with all types off possible inputs. Although effective, analysing all possible paths is not always feasible. Therefore, other techniques like fuzzying(Miller *et* al., 1990) are used.

The static analysis aims to determine the software properties by inspecting its source-code. The software is not executed in this method. The supposed advantage of this method is that it detects errors that could be too difficult to find using others methods. The tools that help the static analysis process vary among functions spotters, that are not more sophisticated than the grep tool, and precompilers. New approaches consist of vulnerability detection using constraint optimization.

This work is based in a source code static analysis technique. Some auxiliary tools that are currently available are: Flawfinder, ITS4, PScan, RATS, Splint and BOON.

The Flawfinder is a python tool that examines a C source-code in a lexical level. This tool has a database of insecure functions that store functions like strcpy(), strcat(), the scanf() family and etc. Format String problems are also identified in printf, snprintf and syslog functions family.

The ITS4 tool aims to identify flaws in C and C++ source-code parts. The tool generates a set of tokens obtained from the lexical analysis of the program source-code. This tool was designed to be used during the software code step and can be used with GNU Emacs. Another quality of this tool is the capability to understand some predefined comments, like "/* ITS4: ignore */", that make the ITS4 to not examine the previous function.

The RATS tool has the main goal to discover potential vulnerabilities in C/C++, Perl, PHP and Python source-code. The two type of vulnerabilities addressed by this tool is buffer overflow vulnerabilities in well know insecure functions and TOCTOU (Time of Check, Time of Use). The TOCTOU vulnerability is a kind of race condition. Generally occurs when a process checks some proprieties of an archive and tries to access the archive in the subsequence instruction. The problem is that even if the instruction to access the archive is the next function after the proprieties check, exist a possibility that a second process could invalidate the first check in a malicious way.

The RATS tool makes de lexical analysis of the source-code. Besides that the tool uses a XML data base for each language. This approach permits de insertion of new vulnerabilities with low effort, but the operation system needs the Expat library in order to install RATS.

The Splint tool uses a different approach. The programmer must insert several comments during de coding step. The two most important comments are the *requires* and *ensures* ones. The *requires* comments defines the preconditions that must be accomplished and the *ensures* defines, for example, the state of the variables after the function execution.

The BOON tool helps the source-code analysis of C source-codes. As the buffer overflow vulnerabilities are due to the wrong string manipulation, this tool mode each string with two proprieties. The first one is the total allocated space for a given variable; the second is the effective number of bytes actually in use.

All functions that manipulate characters arrays are modeling based on the side effects in this two proprieties. Those side effects are used to generate restrictions. So those restrictions can be solved to determine if a malicious use was occurred. In case of a malicious manipulation alerts are generated.

This work is based on a software dependent approach. In this approach the software must be secure by itself, without the need of a specific operating system or compiler. To assure that a software is secure, one must observe the entire software development process to avoid programming errors.

3. DEVELOPMENT OF A SOURCE CODE ANALYSIS AID TOOL

The main goal related to the development of a new source code analysis aid tool is try to improve the quality of the source code analysis. This is done by adding new characteristics to the analysis process such as the syntactic analysis. There are some main characteristics that the system has:

- New insecure functions specifications could be inserted in the database without recompile all the system;
- The tool must discover a major number of vulnerabilities, even that a huge lost of

performance occurs.

- The insecure function identification is done in a syntactic level. Previously, this identification was done in a lexical level by ITS4, FlawFinder and Rats.
- The constraint analysis, besides not implemented, is supported by this tool.
- The analysis process can be done locally, into modules or in all source code.

To accomplish this characteristics the analysis aid tool is divided into modules. Each module has distinct implementation and features. Besides, the tool receives source-codes, user choices and a database of insecure functions as input. The output is a set of analysis report, that shows possible problems spotted in the source-code. The structure of the source-code aid tool is shown in the Figure 6.

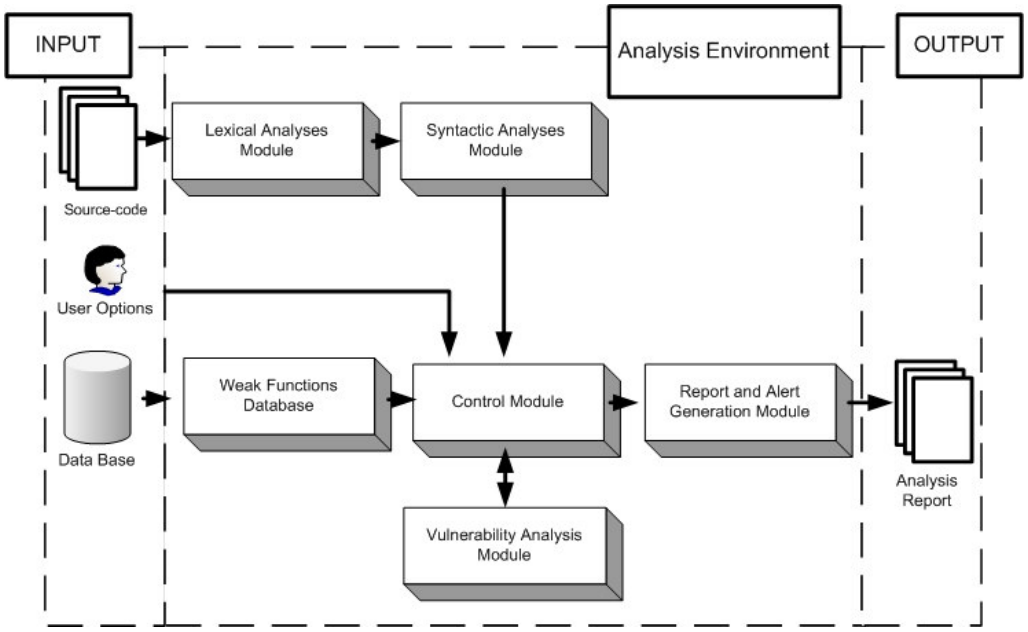

Figure 6: The tool proposed.

The tool is started through the user interaction that establishes the way that the given source-code must be analysed. After receive the source-code and the user choices, the tool start the lexical analysis step. This lexical analysis step generates tokens that will be used by the syntactic analysis module.

The syntactic analyses will fill a pre-defined set of structures that will help to build the execution flow of the source-code. Those structures are indexed by the function name, so it is possible to identify the set of a call made by a particular function, for example.

The control module receives a subset of the pre-filled structures by the syntactic analysis and a set of pre-filled structures with insecure functions and starts the analyses. The structures with insecure functions is filled in the insecure functions database module.

The weak functions database module reads an XML database with several well known insecure functions. The goal of this module is to fill a set of predefined structures. Those structures are stored in live memory, to speed up the analysis.

The analysis done by the vulnerability analysis module is the search for insecure functions. This search is similar with the one done by other applications such as ITS4, FlawFinder and etc. The results are then organized in a report handle by the report and alert generation module.

Several tests have been made against known testbeds examples, like the one made in (Wilander & Kamkar, 2003). Those tests will show the effectiveness and the applicability of this tool.

4. REFERENCES

Bulba; Kil3r. *Bypassing Strackguard and Stackshield*. Phrack Magazine, v. 56, n. 5, May 2000. Online at: <http://www.phrack.org/phrack/56/p56-0x05>.

Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G.,Frantzen, M. and Lokier, J.(2001) *FormatGuard: Atomatic Protection From printf Format String Vulnerabilities.* Paper presented to Conference: Usenix Security Symposium, 10th., Washington, United States of America.

Cowan, C.; Beattie, S., Wright, C. and Kroah-Hartman, G.(2001) *RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities.* Paper presented to Conference: USENIX Security Symposium, 10th., Washington DC, United States of America.

Etoh, H.(2001) *GCC extension for protecting applications from stack-smashing attacks*. Online at: <http://www.trl.ibm.com/projects/security/ssp>

One, A.(1996) *Smashing The Stack For Fun And Profit.* Phrack Magazine, v. 49, n. 14. Online at: <http://www.phrack.org/phrack/49/P49-14>

Pincus, J.; B., B.(2004) *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns*. IEEE Security & Privacy, v. 2, n. 4.

Vendicator(2000) *Stack Shield: A stack smashing technique protection tool for Linux*, 2000. Online at: <http://www.angelfire.com/sk/stackshield/>.

Wilander, J., Kamkar, M.(2003) *A Comparison of Publicly Avilable Tools for Static Intrusion Prevent.* Paper presented to Symposium: Network and Distributed System Security Symposium, 10th., San Diego, California, United States of America.

Miller, P. B., Fredriksen, L., So, B., (1990) *An empirical study of the reliability of UNIX utilities.* Comuninications of the Association for Computing Machinery.