2-4 Buffer-Overflow Detection in C Program by Static Detection

NAKAMURA Goichi, MAKINO Kyoko, and MURASE Ichiro

Buffer_overflow is the most dangerous vulnerability implicit in C programs. whether a Buffer_oveflow emerges or not in the program runtime, is depend upon inputs of the C program. We developed algorithms and tools to detect buffer_overflows, not accidentally partial detection by monitoring program runtime situation, but exhaustively detection by static-analysis of C source code before the program run. The exhaustively detection is the detection of, not only the source code position where a Buffer_overflow emerges, but also the essential source code structure makes the Buffer_overflow vulnerability.

Keywords
Buffer_overflow, Static analysis, Vulnerability detection

1 Introduction

Although various factors contribute to breaches of computer security, one of the major causes is found in latent vulnerabilities within the programs themselves. These vulnerabilities often lie in program structures that cause the phenomenon referred to as "buffer overflow". So-called "security holes" caused by buffer overflow often lead to serious damage, such as the execution of arbitrary code by attackers.

Simply put, in buffer overflow, data overflows into (i.e., is written to) an area not intended by the developer (programmer) to house this data. This phenomenon mainly occurs during execution of C programs.

The most popular software description languages today are the C (including C++) language and Java. With Java, type safety has been taken into consideration since the design stage. Issues such as detection and prevention of buffer overflow are addressed through the design and implementation of virtual machines (bytecode verifiers, in particular)[1]. In fact, Java programs rarely cause security holes due to buffer overflow. In contrast, C programs are likely to cause buffer overflow due to their runtime data configurations, which may lead to serious security problems.

Even if a C program contains a latent element of code that can cause buffer overflow, actual buffer overflow occurs depending on input to the program during execution. In this study, we developed a static analysis algorithm and tool to detect program structures that may cause buffer overflow, as opposed to dynamic monitoring of program execution to detect buffer overflow that happens to occur at that time.

2 About buffer overflow

A variety of information-security issues have recently grown more pressing, from hacking to computer viruses to information leakage and webpage tampering. Factors contributing to these problems are mainly classified into those related to organizational management (such as lack of security policies or management); factors related to system management (such as improper system/network

settings or operations); and latent vulnerabilities within individual software programs (such as buffer overflow and memory leakage). The issue of software vulnerabilities is the most significant factor, as these vulnerabilities are difficult to detect and may cause information-security problems despite measures taken in organizational and system management.

Buffer overflow is the latent software vulnerability of greatest concern today. A software program with potential for buffer overflow may allow an attacker to interfere with normal operation, or, in the worst case, to take over the program's processes. In fact, buffer overflow is the number-one cause of reported problems.

2.1 Buffer overflow occurrences found in vulnerability reports

According to websites that list reports on software vulnerabilities, such as the CVE (http://cve.mitre.org/cve/downloads/full-cve.html) and ICAT (http://icat.nist.gov/icat.cfm) sites, buffer overflow makes up a considerable percentage of causes of reported vulnerability incidents. For example, the following table shows the data available to the public at these websites as of May 2004, illustrating total number of reported vulnerability incidents and the number of incidents caused by buffer overflow.

Table 1 *Number of reported buffer overflow incidents*

Organization	Total number of incidents	Number of buffer overflow incidents
ICAT	1131	386
CVE	6449	1482

These vulnerability reports show the following trends:

- Reported incidents are mainly classified into two categories: vulnerabilities caused by internal program defects such as buffer overflow, and vulnerabilities caused by improper program settings or operations.
- Buffer overflow is by far the most frequent cause of reported incidents, accounting for about 30% of the total. This has been the case for the past few years.
- Other major internal program defects include format string bugs, memory leakage, and cross-site scripting. However, reports of these incidents are fewer than those involving buffer overflow.
- Since 2000, the total number of reported incidents has been increasing along with the proportion of buffer overflow, with additional reports on incidents caused by new types of vulnerabilities such as cross-site scripting.

Judging from these trends, it is clear that buffer overflow represents a serious internal program defect.

2.2 Types of buffer overflow

The term "buffer overflow" is used mainly in two senses. In this R&D project, we have treated these concepts separately as "stack overflow" and "buffer overflow (general)".

- Stack overflow: This type of buffer overflow may allow attacks in a process known as "stack smashing". Memory features an area for critical data generated during execution, and user programs must not write data to this area. However, due to a certain program structure, data may be written to this critical area under instructions to write to an ordinary data area. In this situation, attackers could alter the critical data to an arbitrary data in order to take over the program process. We refer to this program structure as "stack overflow vulnerability".
- Buffer overflow (general): In a broader sense, buffer overflow is a phenomenon in which, under instructions from a user program to write to or refer to an ordinary data area, data is written to (or the reference is made to) this area and another area at the same time. The effects of such an incident range from an inability to run the program properly to process takeover.

We use the phrase "buffer overflow" to

refer to the latter category.

2.3 About languages

Although various program description languages are available, today the C language, Java, and other script languages are the most common, especially within the kinds of network software that are most often targets of attack. The C language has the following characteristics:

- Since this language has been used for many years, C programs account for a large proportion of programs developed worldwide to date. Vulnerabilities latent in these existing programs pose a serious threat of buffer overflow.
- C is still used as a major programming language to develop C programs. These programs may also have the potential for buffer overflow.
- Since the C language was developed based on the assignment of highest priority to execution speed, it is inadequate in many aspects of security. C programs are highly likely to cause buffer overflow due to their runtime memory configurations and execution control methods. In other words, simply programming in the C language is likely to create the potential for buffer overflow.

In contrast, with Java, type safety has been taken into consideration since the design stage. Issues of buffer overflow are addressed through type verification of virtual machine bytecode[1]. In fact, Java programs rarely lead to security holes simply due to buffer overflow. Even if there are latent vulnerabilities in Java programs, the threat of such vulnerabilities is negligible, in part because these account for a much smaller proportion of accumulated programs relative to existing C programs.

2.4 Subjects of this R&D

As described above, it is clear that when conducting R&D to address software security holes, it is important to focus on C programs and buffer overflow.

We have already presented a method for static detection of stack overflow[2][3]. In this paper, we will describe a method for static detection of buffer overflow (general).

3 Existing methods for detecting buffer overflow

3.1 Methods for replacing standard library functions

Some standard library functions (such as strcpy) are found to cause buffer overflow. Avaya Labs' LibSafe[4] and several similar tools search source code for these functions and replace them with safer ones. This method has one disadvantage, as follows:

- Inability to deal with user-programmed segments: These tools are designed to find certain standard library functions that through repeated use may cause buffer overflow, but not to detect buffer overflow attributed to segments programmed by the user.

Therefore, if it is necessary to detect buffer overflow in programs including user-programmed segments, these tools are far from adequate. However, replacement of dangerous functions in standard libraries is the minimum requirement for elimination of vulnerabilities in programs.

3.2 Dynamic detection

A method/tool that dynamically detects buffer overflow performs the following: (1) modification of a processing or compiled program such that dummy data of a given value is placed at pivotal points (such as those close to an area allocated for program execution) in memory; (2) constant monitoring to determine whether this dummy data is altered during program execution; and (3) if this data is altered, the user is notified of the possibility that buffer overflow has occurred. StackGuard is a common tool of this type [5]. This type of dynamic detection tool is the most popular among measures against buffer overflow. The world's largest software vendor has announced that it will incorporate this dynamic detection capability into the next version of

its program development environment.

Since this dummy data is generally called "canary" data, we refer to the method of inserting this data as a "canary method" below. Although dynamic detection methods do not have the disadvantage of an inability to deal with user-programmed segments, they do feature the following disadvantages:

- Decrease in execution speed: Program execution speed is significantly reduced by the constant monitoring to determine whether the canary has been altered. This contradicts the design philosophy of the C language, which places a higher priority on processing speed than on program security. Even though this tool can be used at the debugging stage, the decrease in execution speed is undesirable in actual operation.
- Incomplete detection: Dynamic detection of buffer overflow entails the detection of buffer overflow that has already occurred due to specific input to a program. This method is not designed for comprehensive detection of buffer overflow points and conditions (such as during program input). It is virtually impossible to prevent buffer overflow completely using a canary method/tool in dynamic analysis. Even with repeated debugging of a program using this tool, eliminating buffer overflow case by case, there is no theoretical guarantee of improved program security.
- Insufficiency of information: This tool can detect the occurrence of buffer overflow, but cannot detect the program structure that caused it. Since information on the program structure is essential for modification of the program, the absence of this information puts programmers at a serious disadvantage.
- Problem caused by insertion of the canary itself: In the case of detection of stack overflow (a type of buffer overflow described above), it is important to determine whether data is written to a critical area in memory. However, the position of this critical area is shifted due to the insertion of the canary into the program. As a result, it is likely that this tool cannot detect stack overflow that would occur in the same program in the absence of a canary.

These disadvantages are inherent in dynamic analysis. To avoid them, static analysis is required. As a side note, the typical "Purify" memory-check tool can also be used in dynamic detection of buffer overflow, though no canary data is used.

3.3 Detection through static analysis

LCLint[6], a static analysis tool, uses source-code comments to detect points of buffer overflow occurrences. This tool has the following disadvantage:

- Difficulty of creating additional information: To perform static analysis with this tool, the programmer has to add information other than source code to various sections. This method is effective if the added information is correct. However, it is an onerous task to insert accurate information in this manner, especially in the case of a large program.

Since LCLint is designed to analyze specific functions, its ability to detect points of buffer overflow occurrences is limited.

Under one proposed method of detecting points of buffer overflow occurrences, operational meaning is assigned to a machine language and the extracted machine language code is then interpreted[7]. However, to determine behavior or functionality given certain meanings within a machine language, static analysis must be executed on a large scale. This is unrealistic in light of detection accuracy and amount of time required for analysis. Moreover, this method has the disadvantage of providing insufficient information: even if this tool can detect cases of buffer overflow, it is extremely difficult to isolate a program structure that may cause buffer overflow.

3.4 Detection through pattern matching

Pattern matching is commonly used to detect virus code. A buffer overflow occurrence is simply an assignment statement or reference. Whether this assignment statement or reference causes buffer overflow depends intricately on the portion of code located before the statement or reference. Therefore, it is difficult to detect points of occurrence and program structures comprehensively using pattern matching or rule-based analysis. These methods thus present the disadvantage of incomplete detection.

4 Detection through static analysis of C source code

4.1 Requirements of detection methods

Based on the above discussion, we can summarize the requirements of measures against buffer overflow as follows:

(1) Detection of buffer overflow in user-programmed segments (segments programmed by the user, aside from library functions)
(2) Detection of buffer overflow by static analysis of a program before executing, instead of detection during execution
(3) Thorough detection of elements that may cause buffer overflow—i.e., ensuring complete detection with no omissions
(4) To allow detection results to be used in program modifications, establishment of a buffer overflow structure that consists of the following:
  • A data area in which buffer overflow is likely to occur
  • Elements within this data area (statements) that are likely to cause buffer overflow
  • Conditions leading to actual buffer overflow
(5) Direct analysis of source code, without required addition of comments or other descriptions
(6) Use of a detection algorithm, not pattern matching or rule-based analysis

4.2 Overview of detection algorithm

The occurrence of buffer overflow depends on the size of data arrays or pointer areas allocated by “malloc” memory allocation functions or other functions, as well as on the indices for access to these arrays and to the pointer areas in the body of the program (e.g., given expressions such as “a[i]” and “*(p+i)”, what is the value of variable “i” and which of these arrays and pointer areas are indicated by variables “a” and “p”?). To identify these items, we have modified a method of definition-use analysis used by ordinary compilers.

It is most important to ensure complete detection, as mentioned in Section **4.1**(3), in addition to verifying validity (all detected elements must have the potential to cause buffer overflow—no false detection is tolerable). Although it is possible to identify all elements that have the potential for buffer overflow, buffer overflow actually takes place only if certain conditions are met; overflow does not happen at many points. Therefore, even though it is relatively easy to ensure complete detection, it is also necessary to ensure the validity of detection by determining the relevant points of occurrence as well as the conditions of actual buffer overflow at each of these points. To help ensure the validity of detection, this algorithm analyzes not only potential buffer overflow points but also the conditions of actual occurrence based on the structures of the transitions between blocks. Figure 1 shows an overview of this detection algorithm, which meets all of the established requirements of detection methods.

4.3 Detection tool

We implemented this detection algorithm as a tool. Detection results were obtained in the form of a buffer overflow structure, as mentioned in Section **4.1**(4). C language macros and included files must be expanded through preprocessing before static analysis can be performed. Therefore, this detection tool uses C source code that has been preprocessed.

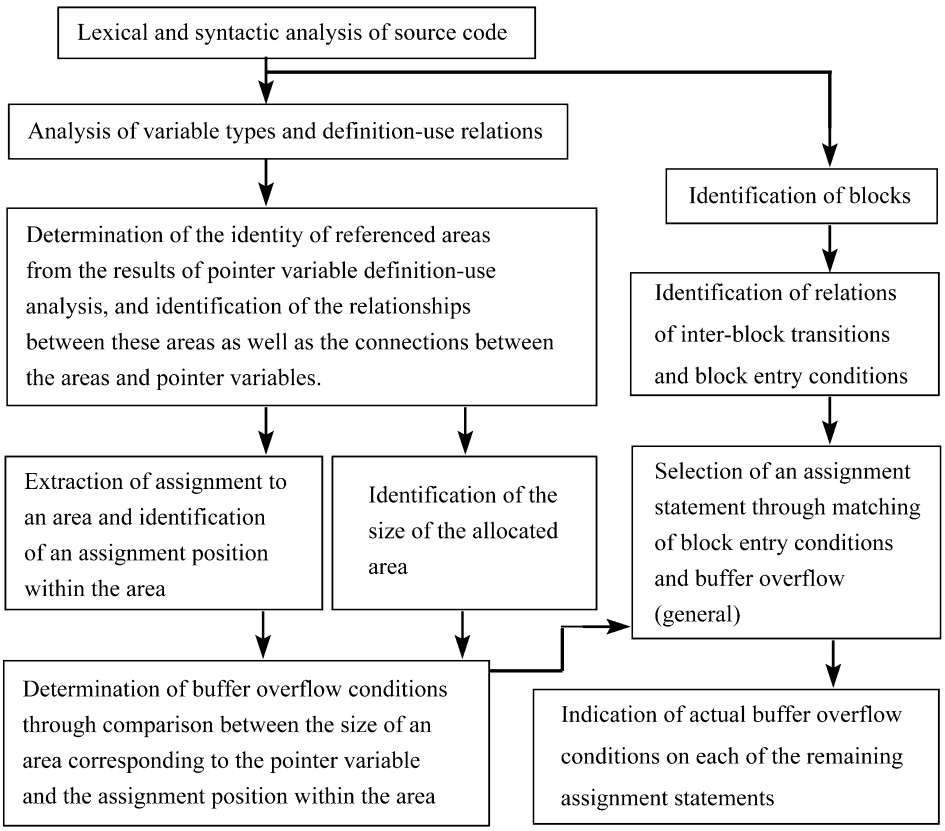

Fig. 1 *Overview of buffer overflow detection algorithm*

4.4 GUI and experiments

Since a buffer overflow structure is fairly complex, it is not appropriate to display it to the user in string format. We thus created a GUI to allow the user (i.e., the programmer) to use this data in easy debugging to eliminate buffer overflow structures. This GUI displays a number of the components of the buffer overflow structure: the "Potential area of occurrence", a data area in which buffer overflow may occur; "Potential points of occurrence", points (statements) within this data area that may cause buffer overflow; and "Conditions of occurrence", conditions relating to actual buffer overflow events.

It is obvious that the relationship between a "Potential area of occurrence" and "Potential points of occurrence" is usually one-to-many. Conditions of actual buffer overflow vary at different locations (within the control sequence) before a "Potential point of occurrence". Therefore, the relationship between a "Potential point of occurrence" and "Conditions of occurrence" is also one-to-many. This GUI allows the user to view these items in an easy-to-read format based on mouse-click operations. We have been conducting experiments on the detection of buffer overflow using available source code from programs with buffer overflow structures reported in books on C language or in vulnerability reports. To date the newly developed tool has been shown to detect buffer overflow within practical analysis times while meeting the necessary requirements of detection methods, including demands of completeness and validity.

5 Conclusions

Buffer overflow is the most dangerous latent software vulnerability. We have categorized buffer overflow into two types: specific phenomena and general buffer overflow. To

address the latter, we developed a static analysis algorithm and tool for the thorough detection of structures in C programs that may cause buffer overflow.

Going forward, we will conduct research on the application of this algorithm and tool to C++ and other languages susceptible to buffer overflow. We will also work on the development of methods of static analysis to detect memory leakage and other potential software vulnerabilities.

References

1 Hagiya, M., "About A New Data_flow Analysis for Java Virtual Machine Procedure", First Workshop on Programing and Apllication_systems, Japan Society for Software Science and Technology, 1998.

2 Nakamura, G., et al., "Buffer_overflow Detection in C Program by Static Analysis", Special Group on Programming in Information Processing Society of Japan, Jun. 2002.

3 Nakamura, G., et al., "About Buffer_overflow Detection by Static Analysis", Special Group on Computer Security in Information Processing Society of Japan, Jul. 2002.

4 AVAYAlabs LibSafe:http://avayalabs.com/project/libsafe/.

5 Eto, H., et al., "propolice-Improvement of detection of Stack-Smashing-Attack", IEICE ISEC2001-42, 2001.

6 Larochelle, D. and Evans, D., "Statically Detecting Likely Buffer Overflow Vulnerabilities", 2001 USENIX security symposium,Washington D.C., Aug. 1996.

7 Xu, Z., Miller,B.P., and Reps, T., "Safety Checking of Machine Code", proceedings of the conference on programming language design and implementation, June 2000.

NAKAMURA Goichi
Mitsubishi Research Institute, Inc.
Information Security, Quantum Computing

MAKINO Kyoko
Mitsubishi Research Institute, Inc.
Information Security

MURASE Ichiro
Mitsubishi Research Institute, Inc.
Information Security