A Tool for Visualizing Buffer Overflow with Detecting Return Address Overwriting

Isao Sasano
Shibaura Institute of Technology, Tokyo, Japan
sasano@sic.shibaura-it.ac.jp

ABSTRACT

Buffer overflow is a serious problem when the software is run as a server on the internet. Especially when the return addresses are overwritten intentionally, the control flow may be changed as the attacker intends. Although there have been proposed several ways to protect attacks that utilize the buffer overflow, the number of the errors owing to the buffer overflow have been increasing gradually. This paper presents a tool that visualizes the buffer overflow when executing programs in C language, especially when the return addresses are overwritten. The functionality is mainly targeted at beginners of C programming who do not recognize the attacks, which we expect makes the number of errors decrease in the future.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms

Security, Languages

Keywords

integrated development environment, debuggers, buffer overflow, visualization, GUI, C Language

1. INTRODUCTION

In programming in the language C the memory management partly depends on programmers. In the standard of the language C in ISO [2], the behavior of a program is not defined when it accesses some location that is outside the boundaries of arrays.[1] The out-of-bound array accesses originate in the fact that C allows addition and subtraction between pointers and integers, which result in pointers that may point some location in the middle of an object. In contrast, in most of the memory-safe languages like Java or ML-family, references to an object, such as an array, only point to a fixed location inside the object, which makes it easy to access meta-data like the size of the object at runtime. Most of the C compilers produce codes that do not check the boundaries of arrays. So many programs written in C are used without boundary checking, which may result in buffer overflow.

Buffer overflow is a serious problem especially when the program is run as a server on the internet. When the return addresses are overwritten intentionally by some attacker, the control flow may be changed as the attacker intends. Until now there have been proposed several ways to protect attacks that utilize buffer overflow. We show the number of buffer errors (CWE-119) reported in the national vulnerability database statistics [1] in Fig. 1. Despite the efforts devoted to the buffer overflow, the number gradually increases until 2014. Considering the fact that some of the tools like StackGuard [3] is used in the commercial product, there remain many programs run as servers on the internet without using the tools or with the security holes being not covered in such tools.

This paper presents a tool that visualizes the buffer overflow when executing programs written in C language, especially when the return addresses are overwritten. The functionality is mainly targeted at beginners of C programming who do not recognize the attacks. We believe that it is also useful for proficient programmers who can use CUI debuggers like gdb. The tool presented in this paper partly bridges the gap between the current real-world status about the buffer overflow and the tools for protecting the buffer overflow developed until now. The tool helps C programmers to recognize the occurrences of the buffer overflow.

The rest of the paper is organized as follows. Section 2 shows our basic ideas and outline of our solution. Section 3 specifies the GUI and describes our implementation. Section 4 discusses related work. Section 5 describes future work and concludes the paper.

[1]More precisely, when the addition or subtraction between a pointer and an integer does not result in the array bounds (or one past the last element), the behavior is not defined [2, Page 93].

.

BICT 2015, December 03-05, New York City, United States

DOI 10.4108/eai.3-12-2015.2262495

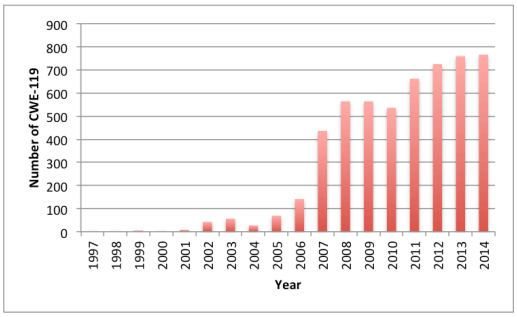

Figure 1: The number of CWE-119 [1]

2. BASIC IDEAS AND THE STRUCTURE OF THE TOOL

Here we show our idea to develop a tool for visualizing the return address overwriting. We construct the tool by communicating with gdb through TCP/IP on text basis.

2.1 Structure of the tool

We show the structure of the tool in Fig. 2.

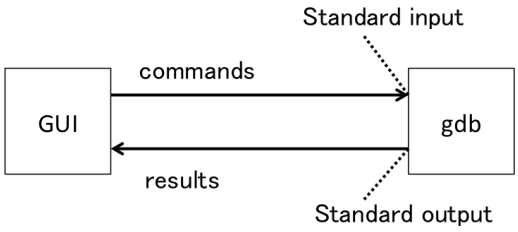

Figure 2: The structure of the tool

Some of the buttons on the tool correspond to gdb commands and some other correspond to the functionalities for controlling the visualization such as changing the font size. When the user of the tool pushes a button that corresponds to some gdb command, the command is executed on gdb.

2.2 Buttons

We made several buttons corresponding to the gdb commands. We show the screenshot of the buttons just after launching the tool and pushing the run button in Fig. 3.

The button "run" is for running the program until entering the main function. The button "quit" is for quitting the execution of the program. The button "step" is for executing the program in one step, corresponding to the step command in gdb. The button "finish" is for finishing the function that is currently being executed, corresponding to the finish command in gdb. The buttons "view_up" and "view_down" are for moving up or down the focus on the memory location. The button "place" is for switching on and off whether or not the memory addresses are depicted in the slots of the memory. The button "memory" is for switching on and off whether or not the values stored in the memory are depicted in the slots of the memory. The buttons "zoom_in" and "zoom_out" are for zooming in or out the area for the memory. The buttons "size_up" and "size_down" are for changing the font size in the area for the memory.

Figure 3: Buttons [5]

We show a screenshot of gdb in Fig. 4. The first portion surrounded by the red line shows the gdb command "info frame" and the response. The command is executed when the run button is pushed on the tool, in order to obtain the address of the bottom of the activation record for the first invocation of the main function. The second and the third portion show two gdb commands and the responses. These commands are executed when the step button is pushed on the tool, in order to obtain the contents of the stack area from the location pointed by the stack pointer to the bottom of the activation record for the main function obtained above.

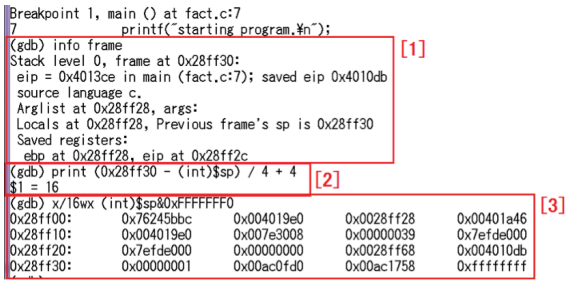

Figure 4: A screenshot of gdb corresponding to some sequence of pushing the buttons run and step on the tool [7]

Fig. 5 shows some other portion of the stack when some functions are called from the main function. The portion surrounded by the blue line is an activation record in the stack and the location surrounded by dotted red line is for holding the return address. Using this kind of commands and understanding the result of the commands require some knowledge of the gdb and the structure of the stack area in the memory.

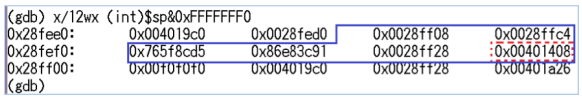

Figure 5: A screenshot of gdb with showing some portion of the stack [7]

3. SPECIFICATION AND IMPLEMENTATION OF THE VISUALIZATION

In this section we specify the visualization functionality and describes our implementation.

3.1 Specification

PROBLEM 1 (VISUALIZATION). *When a return address is changed while running the code, it is displayed on some GUI system and the code execution is suspended.*

Note that when a return address happens to be overwritten with the same address the system does nothing.

3.2 Implementation

Based on the basic ideas described in Section 2 we have implemented a tool for visualizing the buffer overflow. The tool is implemented for gcc on 32bits MinGW (on Windows) on IA-32 architecture firstly [7] and an extended tool for gcc on 64 bits Vine Linux on x86-64 architecture secondly [5]. We plan to make public the implementation available on the internet in the future.

3.2.1 Structure of the tool

The tool is implemented in Java on the platform described above. The buttons on the tool are associated with methods in Java, some of which send commands of gdb, corresponding to the pushed button, to the standard input of gdb. On gdb the command is executed and then the output to the standard output is sent back to the tool. The communication between the tool and gdb is done through TCP/IP.

The current implementation is a tentative one. In particular, the tool and gdb communicate on text basis. When pushing the step button on the tool, the text "step" is sent to gdb, the step command is then executed on gdb, and the tool takes the result from the standard output of the gdb process after waiting a fixed amount of time (say 500ms).

3.2.2 Coloring memory locations

When a return address is overwritten, the location holding the return address is colored with red. Two other colors are used in the tool. Yellow is used for coloring the memory locations for return addresses not yet overwritten. Blue is used for coloring the memory locations that are rewritten but are not for holding the return addresses, although the examples in this paper do not use blue.

We have to store the values in the stack area in order to color each memory slot as described above. This is done just by memorizing all the values in the stack area when the step button is pushed and compare the values with the stored values one step before. We assume the return address is stored in the bottom of each activation record.

3.2.3 Various functionalities

The tool provides a functionality for stepping into a function. When we need to step into library functions, we need the compiled code of the library functions with the debugging information. We implement this functionality just by searching the source code of the library functions.

The tool also provides a functionality for highlighting the location where the control flow is now. The tool supports programs that consist of multiple files.

3.3 An example

We show a screenshot of the overwriting of a return address in Fig. 6. The screenshot is taken just after a return address is overwritten, when the execution of the program is suspended.

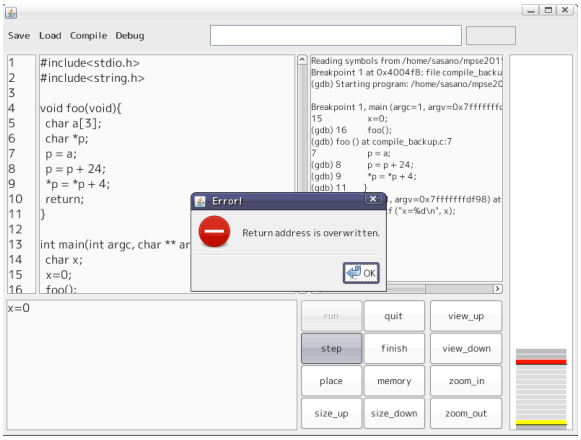

Figure 6: A screen shot of detecting the return address overwriting [5]

4. RELATED WORK

In order to protect software systems from the attacks utilizing the buffer overflow, there have been presented some systems. A system used in the real-world is StackGuard [3], which is a C compiler that attaches some number, which is called Canary word, next to the return address with a little performance penalties. When the control is about to return to the caller of the function, it is checked whether or not the Canary word is changed. If it is the case, the program is forced to be halted. Stack Shield [8] is a tool for protecting software systems from the attacks utilizing the buffer overflow and it supports gcc on Linux. In both of StackGuard and Stack Shield no source code changes are required.

Fail-safe C [6] is a completely memory-safe compiler for the ANSI C standard, which is comparable to memory-safe languages like Java or ML-like languages. It increases memory usage at runtime.

As for programming environment with visualization, a system called AZUR [4] was developed. It visualizes block structures and shows the control flow real time by animation. The system is implemented in Java, communicating with gdb.

As we mentioned in Section 1, the number of buffer overflow increases until 2014 despite the efforts like the above. The present work aims at helping C programmers, especially beginners, to recognize the occurrences of the buffer overflow. It may reduce the number of buffer overflow that might occur in the future.

5. CONCLUSIONS AND FUTURE WORK

We presented a tool which partly bridges the gap between the current real-world status about the buffer overflow and the tools for protecting the buffer overflow developed until now. The tool helps C programmers to recognize the occurrences of the buffer overflow. In the future we plan to

experiment with around 100 students in the class of learning programming in C by using the tool presented in the present paper.

6. ACKNOWLEDGMENTS

We would like to thank the anonymous referees for many helpful comments. The idea of visualizing buffer overflow was presented and a tool was implemented based on the idea in Bachelor's thesis of Koji Okada [7] and Takahiro Ogawa [5] under the supervision of the author. This work was partially supported by JSPS KAKENHI Grant Number 25730047.

7. REFERENCES

[1] National vulnerability database statistics. `https://web.nvd.nist.gov/view/vuln/statistics`.

[2] ISO/IEC 9899, Information technology — Programming languages — C, 2011.

[3] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[4] Toshiyuki Imaizumi, Hiroaki Hashiura, Saeko Matsuura, and Seiichi Komiya. A programming learning environment "AZUR" : Visualizing block structures and program function behavior. In *IEICE Techinical Report KBSE2010-45*, pages 61–66, 2011. in Japanese.

[5] Takahiro Ogawa. Bachelor's thesis, Shibaura Institute of Technology, Japan, 2015. in Japanese.

[6] Yutaka Oiwa. Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2009.

[7] Koji Okada. Bachelor's thesis, Shibaura Institute of Technology, Japan, 2012. in Japanese.

[8] Vendicator. Stack shield: A "stack smashing" technique protection tool for Linux. `http://www.angelfire.com/sk/stackshield/`.