International Journal of Grid and Distributed Computing
Vol. 11, No. 3 (2018), pp.79-88
http://dx.doi.org/10.14257/ijgdc.2018.11.3.07

A Dynamically Linked Library Based Indirect Call Function Analysis for Detecting Banned API Usage in Binary Code

Junho Jeong[1] and Yunsik Son[2*]

[1]*Electronic Commerce Institute, Dongguk University*
[2]*Department of Computer Science and Engineering, Dongguk University*
yanyenli@dongguk.edu, sonbug@dongguk.edu

Abstract

The use of Inherently Dangerous Function could cause vulnerabilities in a program which makes it disadvantageous. If the source code exits, this problem can easily be solved by simply removing the use of dangerous functions based on the list of prohibited functions. However, if only the binary code exits, it is difficult to analyze the list of functions used in the code. Furthermore, it is challenging to understand the information of functions used in analysis, such as reverse engineering, because a lot of the information in library functions that are linked dynamically in a typical binary file has been removed. In this paper, we propose a method to find indirectly called function information by using the information when calling a function in binary code based on indirect calling method used in the windows environment.

***Keywords**: Code Analysis; Indirect Call; Vulnerable function; Secure Software*

1. Introduction

Today, the impact of software on our society continues to increase. It is critical to analyze the use of vulnerable functions in software using trusted software [1, 2]. It has become more difficult to find devices that are not inherent in software, from military, medical, to everyday life. Furthermore, various security incidents have occurred due to security vulnerabilities inherent in software [3-6]. Therefore, development of secure software is a crucial issue.

Various studies have been conducted to remove security weaknesses during software development in order to overcome these vulnerability issues [7-9]. Most of these studies are about analyzing security weaknesses in software where the source code exists.

However, there is not much research on binary code, such as libraries without source code, and it is difficult to analyze the indirect calls inherent in the binary code [10-13]. This is because there is no instance of the code being called in the binary code. Also, at the operating system level, function calls in the source code can be changed to indirect calls even though the developer did not make an indirect call.

An indirect call is an instruction that calls a function using an address stored in memory rather than using the address of the function, for example, calling the Call function [0x41905c]. Such indirect calls make it difficult to analyze software safety through reverse engineering.

In order to solve this problem, in this paper, we analyze the indirectly called functions by directly analyzing the dynamic linking when the operating system executes the program. In Section 2, we analyze the use of inherently dangerous functions in binary code, and in Section 3, we introduce how to find indirectly called functions by analyzing

Received (December 27, 2018), Review Result (February 22, 2018), Accepted (February 26, 2018)
* Corresponding Author

ISSN: 2005-4262 IJGDC

the indirect call of the Windows PE (Portable Executable) format. Section 4 analyzes the experimental results and concludes in Section 5.

2. Analysis of Banned API Usage on Binary Code

The use of Inherently Dangerous Function is disadvantageous therefore desirable. For example, in the case of code using the gets() function as shown in Figure 2, the input buffer is flooded because the function does not check the input size limit. Using these functions can be a software vulnerability. Therefore, it is possible to prevent the occurrence of a vulnerability in the software by prohibiting the use of inherently dangerous functions and using a safe function in advance. This problem can be solved by using the fgets() function instead of the gets() function, which is vulnerable to buffer overflow attacks.

```
#include <stdio.h>
#include <stdlib.h>
#define BUFSIZE  100

void requestString() {
    char buf[BUFSIZE];
    gets(buf);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define BUFSIZE  100

void requestString() {
    char buf[BUFSIZE];
    fgets(buf, BUFSIZE,  stdin);
}
```

Figure 2. Improved Code for Code that Use of Inherently Dangerous Function

In other words, this weakness can be solved easily by removing and changing the use of functions known to be dangerous based on the list of prohibited functions when the source code of the program exists. However, for libraries that do not have source code, there is no guarantee that this problem has already been solved. Since the library is developed without any consideration of software weaknesses, the use of the library may be a vulnerability. Therefore, it is very important to find a known dangerous function used in already developed binary code. The process for solving this is as follows.

First, it lists the inherently dangerous function, such as analyzing the source code. We have listed the functions using the 150 banned APIs defined in Microsoft Security Develop Lifecycle (SDL) as shown in Figure 3. We used this list as input for the analysis of the use of inherently dangerous function and implemented the analyzer so that future functions could be easily added.

The second is to list the functions used in the binary code. Finally, we compare the function used in the binary with the list of banned APIs to analyze the use of inherently dangerous functions in binary code.

However, it is very difficult to list the functions used in the binary code due to reverse engineering and analysis in the second phase. Even functions in dynamically linked libraries are indirect calls rather than direct calls, so binary code simply records specific addresses. Therefore, even if the binary code is disassembled and assembled, it is difficult to analyze what kind of function is called by simply looking at a specific address, rather than revealing the kind of function.

To solve this problem, we analyzed the binary code of the PE format of the Windows environment. Based on the analysis results, the binary code is disassembled and parsed into function unit codes, and the assembly code is divided into basic block which is a unit without branching of the control flow. And we made a list by collecting information about the names and the start address of the functions that are imported and exported at the block unit

strcpy	strcpyA	strcpyW	wcscpy	_tcscpy
strcat	strcatA	strcatW	wcscat	_tcscat
sprintfW	sprintfA	wsprintf	wsprintfW	wsprintfA
wvsprintf	wvsprintfA	wvsprintfW	vsprintf	_vstprintf
strncpy	wcsncpy	_tcsncpy	_mbsncpy	_mbsnbcpy
strncat	wcsncat	_tcsncat	_mbsncat	_mbsnbcat
gets	_getts	_gettws	StrCatChainW	_tccat
IsBadWritePtr	IsBadHugeWritePtr	IsBadReadPtr	IsBadHugeReadPtr	IsBadCodePtr
memcpy	RtlCopyMemory	CopyMemory	wmemcpy	_ultow
wnsprintf	wnsprintfA	wnsprintfW	_snwprintf	_snprintf
_vsnprintf	vsnprintf	_vsnwprintf	_vsntprintf	wvnsprintf
strtok	_tcstok	wcstok	_mbstok	lstrcpyA
makepath	_tmakepath	_makepath	_wmakepath	wcslen
_splitpath	_tsplitpath	_wsplitpath	lstrcpy	strlen
scanf	wscanf	_tscanf	sscanf	swscanf
_itoa	_itow	_i64toa	_i64tow	_ui64toa
CharToOem	CharToOemA	CharToOemW	OemToChar	OemToCharA
_mbscpy	StrCpy	StrCpyA	StrCpyW	alloca
_mbscat	StrCat	StrCatA	StrCatW	_stscanf
sprintf	swprintf	_stprintf	StrNCatA	_ui64tot
vswprintf	strcpynA	StrNCpyA	StrNCpyW	OemToCharW
StrCpyN	StrCpyNA	StrCpyNW	StrNCpy	_alloca
StrCatN	StrCatNA	StrCatNW	StrNCat	snscanf
_mbccat	_ftcscat)	StrNCatW	lstrncat	_ui64tow
IsBadStringPtr	lstrcpyn	lstrcpynA	lstrcpynW	CharToOemBuf
StrCatBuffW	lstrcatnA	lstrcatnW	lstrcatn	snwscanf
_sntprintf	lstrcatW	StrCatBuff	StrCatBuffA	_ultoa
wvnsprintfA	wvnsprintfW	lstrcat	lstrcatA	CharToOemBuf
lstrcpyW	_tccpy	_mbccpy	_ftcscpy	_sntscanf
_mbslen	_mbstrlen	StrLen	lstrlen	_ultot

Figure 3. Banned API LIST

3. Analysis Windows PE Format Structure for Indirect Call

3.1. Analysis of PE Format Structure

In this paper, we analyze the PE format, which is the executable file structure of Windows, to find indirectly called functions in binary code in the Windows environment. The PE format consists mainly of DOS Header, PE Header, and Section. Data Directories exist in the Image Optional Header of the PE Header, and 16 data tables are used during program execution. The second table in the data tables is an Import Directory Table (Image Import Descriptor) that can be used for indirect call analysis.

The process of analyzing an indirectly called function can be expressed in three steps. The first step is to analyze the PE format to find the offset and size of the Import Directory Table. The second step is to locate the Import Name Table and Import Address Table using the found Import Directory Table. The Import Directory Table is composed of several structures as shown in the following Table 1 [3]. Each structure indicates a piece of DLL to be imported.

The Import Lookup Table RVA represents the offset of the Import Lookup Table of the corresponding DLL. The Name RVA indicates the offset of the name of the imported DLL. The Import Address Table RVA indicates the offset of the memory space to store the address where the function of the DLL is loaded. Therefore, we use the Import Lookup Table RVA, Name RVA, and Import Address Table RVA to find libraries that are dynamically linked.

Table 1. Import Directory Entry Format

Offset	Size	Field	Description
0	4	Import Lookup Table RVA	The RVA of the import lookup table. This table contains a name or ordinal for each import.
4	4	Time/Date Stamp	The stamp is set to zero until the image is bound. After the image is bound, this field is set to the time/data stamp of the DLL.
8	4	Forwarder Chain	The index of the first forwarder reference.
12	4	Name RVA	The address of an ASCII string that contains the name of the DLL. This address is relative to the image base.
16	4	Import Address Table RVA	The RVA of the import address table. The contents of this table are identical to the contents of the import lookup table until the image is bound.

The third step is to find the name of the function from the library in the Import Lookup Table. The Import Lookup Table has an offset of the Hint / Name Table as shown in Table 2. The Hint / Name table contains the offset of the name of the imported function as shown in Table 3.

Therefore, we could find the memory offset that stores the information of the imported DLL, the function, and the address where the function is loaded.

Table 2. Import Lookup Table Entry Format

Bit(s)	Size	Bit field	Description
31/63	1	Ordinal/Name Flag	If this bit is set, import by ordinal. Otherwise, import by name. Bit is masked as 0x80000000 for PE32, 0x8000000000000000 for PE32+.
15-0	16	Ordinal Number	A 16-bit ordinal number. This field is used only if the Ordinal/Name Flag bit field is 1 (import by ordinal). Bits 30-15 or 62-15 must be 0.
30-0	31	Hint/Name Table RVA	A 31-bit RVA of a hint/name table entry. This field is used only if the Ordinal/Name Flag bit field is 0 (import by name). For PE32+ bits 62-31 must be zero.

Table 3. The Information of Hint/Name Table

Offset	Size	Field	Description
0	2	Hint	An index into the export name pointer table. A match is attempted first with this value. If it fails, a binary search is performed on the DLL's export name pointer table.
2	variable	Name	An ASCII string that contains the name to import. This string must be matched to the public name in the DLL. This string is case sensitive and terminated by a null value.
*	0 or 1	Pad	A trailing zero-pad byte that appears after the trailing null byte, if necessary, to align the next entry on an even boundary.

3.2. Analysis of RAW, RVA and Imagebase

We learned about DLL information and functions imported through PE format analysis, and memory offset that stores the address where the function is loaded. However, since these offsets are stored in RVAs, they must be used in files or converted when used in binary code.

In Figure 4, we can see the two offset concepts, RAW and RVA, used in the PE format. RAW is the actual offset used in the program, and when the program is executed and loaded into memory by the process, it is relocated, and the offset is different. The relocated offset of the process is called the RVA. Information about how the RAW and RVA are relocated is contained in the section headers of the section. All three tables we used belong to the idata section or the rdata section.

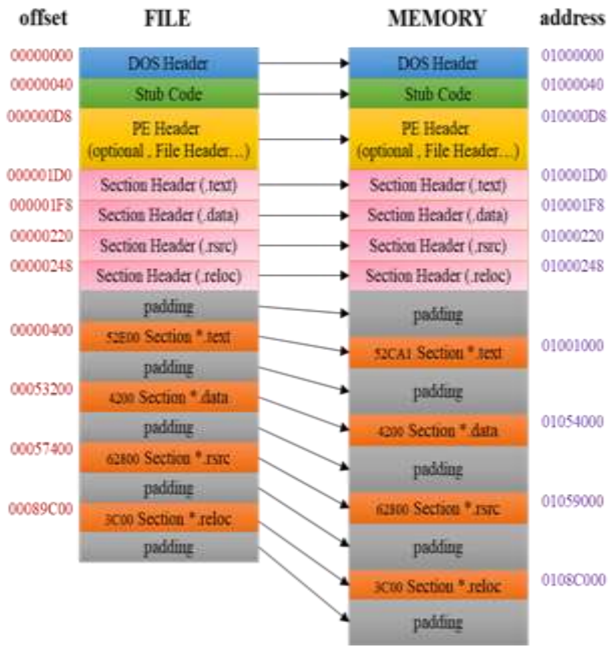

Figure 4. PE Format Structre (RAW, RVA)

In addition, the PE format consists of several sections. The structure of PE format includes a dynamic library reference for linking, an API export address table, an import address table, resource management data, TLS data, and so on. The rdata section contains an IMAGE_EXPORT_DIRECTORY structure that stores the number of functions exported by the DLL file and the name and address of each function so that the function exported to the binary code can be analyzed.

Since the address of each function is represented by RVA, the RVA Offset must be converted to RAW Offset in order to calculate the exact address of the function. In the process of converting RVA to RAW, the RVA Offset is used to find the section containing the RVA Offset, and the Virtual Address and PointerToRawData can be found through the Section Header structure and the RAW Offset can be calculated using Equation (1).

$$\text{RAW Offset} = \text{RVA Offset} - \text{VirtualAddress} + \text{PointerToRawData} \qquad (1)$$

Imagebase points to the start address when the PE file is loaded into the memory. By default, 0x400000 is specified for EXE files and 0x1000000 for DLL files. This Imagebase is the basis of RV. That is, this Imagebase and the RVA address of the memory that stores the loaded address must be combined to get the actual destination address.

4. Experimental Result

In this paper, we used various test programs to test the function to be imported. Figure 5 is one of them, it has been converted into a binary file and tested.

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char small_buffer[0x10];
   char big_buffer[0x100];

   scanf("%s", buf_buffer);
   strcpy(small_buffer, big_buffer);

   return 0;
}
```

Figure 5. Example of the Program Using Indirect Function Call

```
00000000 8b ec                    mov ebp, esp
00000002 81 ec 18 01 00 00        sub esp, 0x118
00000008 a1 00 30 40 00           mov eax, [0x403000]
0000000d 33 c5                    xor eax, ebp
0000000f 89 45 fc                 mov [ebp-0x4], eax
00000012 8d 85 f8 fe ff ff        lea eax, [ebp-0x108]
00000018 50                       push eax
00000019 68 f4 20 40 00           push 0x4020f4
0000001e ff 15 a4 20 40 00        call dword [0x4020a4]
00000024 83 c4 08                 add esp, 0x8
00000027 8d 8d f8 fe ff ff        lea ecx, [ebp-0x108]
0000002d 51                       push ecx
0000002e 8d 95 e8 fe ff ff        lea edx, [ebp-0x118]
00000034 52                       push edx
00000035 ff 15 9c 20 40 00        call dword [0x40209c]
0000003b 83 c4 08                 add esp, 0x8
0000003e 33 c0                    xor eax, eax
00000040 8b 4d fc                 mov ecx, [ebp-0x4]
00000043 33 cd                    xor ecx, ebp
00000045 e8 04 00 00 00           call 0x4e
0000004a 8b e5                    mov esp, ebp
0000004c 5d                       pop ebp
0000004d c3                       ret
```

Figure 6. The Assembly Code Generated by Reverse Engineering from Binary Code

Figure 6 shows the result of converting the analysis program into a binary file and reverse-engineering it to assembly code. The scanf() and strcpy() functions are indirectly called in the form of dword [0x4020a4] and dword [0x40209c].

Applying the proposed analysis model to the target binary file yields the results shown in Figure 7. That is, call dword [0x4020a4] calls the scanf() function of MSVCR100.dll call dword [0x40209c] calls the strcpy() function of the same dll.

```
MSVCR100.dll
0x402044  :  ?terminate@@YAXXZ
0x402048  :  _unlock
0x40204c  :  __dllonexit
0x402050  :  _crt_debugger_hook
0x402054  :  _onexit
0x402058  :  _except_handler4_common
0x40205c  :  _invoke_watson
0x402060  :  _controlfp_s
0x402064  :  __set_app_type
0x402068  :  _fmode
0x40206c  :  _commode
0x402070  :  __setusermatherr
0x402074  :  _configthreadlocale
0x402078  :  _initterm_e
0x40207c  :  _initterm
0x402080  :  __initenv
0x402084  :  exit
0x402088  :  _XcptFilter
0x40208c  :  _exit
0x402090  :  _cexit
0x402094  :  __getmainargs
0x402098  :  _amsg_exit
0x40209c  :  strcpy
0x4020a0  :  _lock
0x4020a4  :  scanf
```

Figure 7. The Result of Indirect Call Function Analysis from Binary Code

We analyzed weaknesses of the use of inherently dangerous functions by using the analysis result of the indirect call function. Figure 8 shows the source code that calls functions that are known to be vulnerable to stack buffer overflow and functions that can cause actual stack buffer overflows. The testMain function calls the strcpy() function twice, which is known to be dangerous. The second strcpy() call is a stack buffer overflow that can occur.

```
int testMain(int num, int num2) {
    char buf1[256];
    char buf2[16];

    scanf("%s", buf1);
    scanf("%s", buf2);
    strcpy(buf1, buf2);
    strcpy(buf2, buf1);

    return 0;
}

int bof() {
    short a = 10;
    int b = 1, c = 2;
    testMain(b, c);
    return b;
}
```

Figure 8. The Example of Use of Inherently Dangerous Function

Figure 9 shows the assembly code of the bof() function, which is disassembled and parsed by the function, and calls the testMain() function in the basic block expressed by the assembly code as a result of analysis of the binary code. Figure 10 also shows that the strcpy() function is used twice in the assembly code of the testMain function.

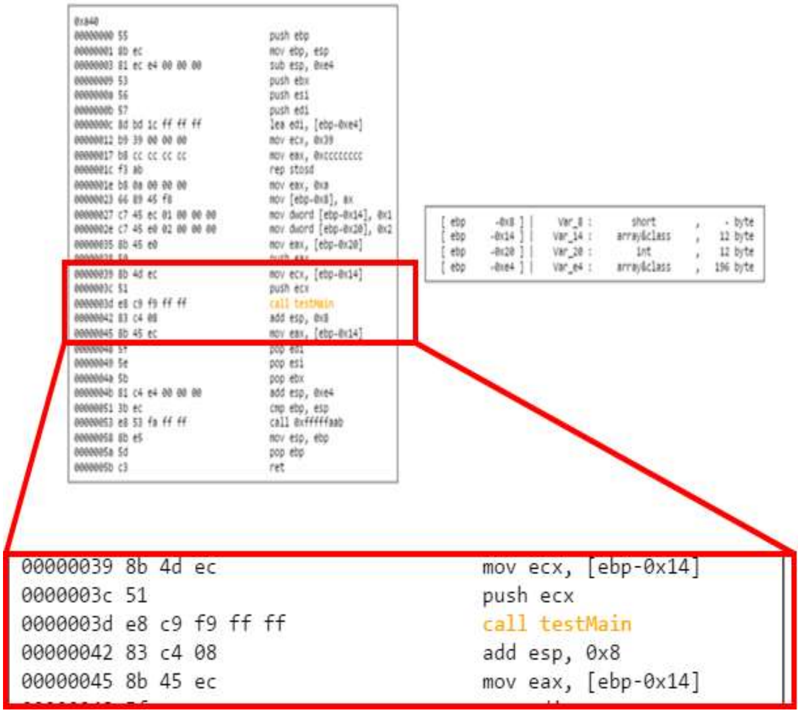

Figure 9. The bof() Function Assembly Code Expressed As A Basic Block

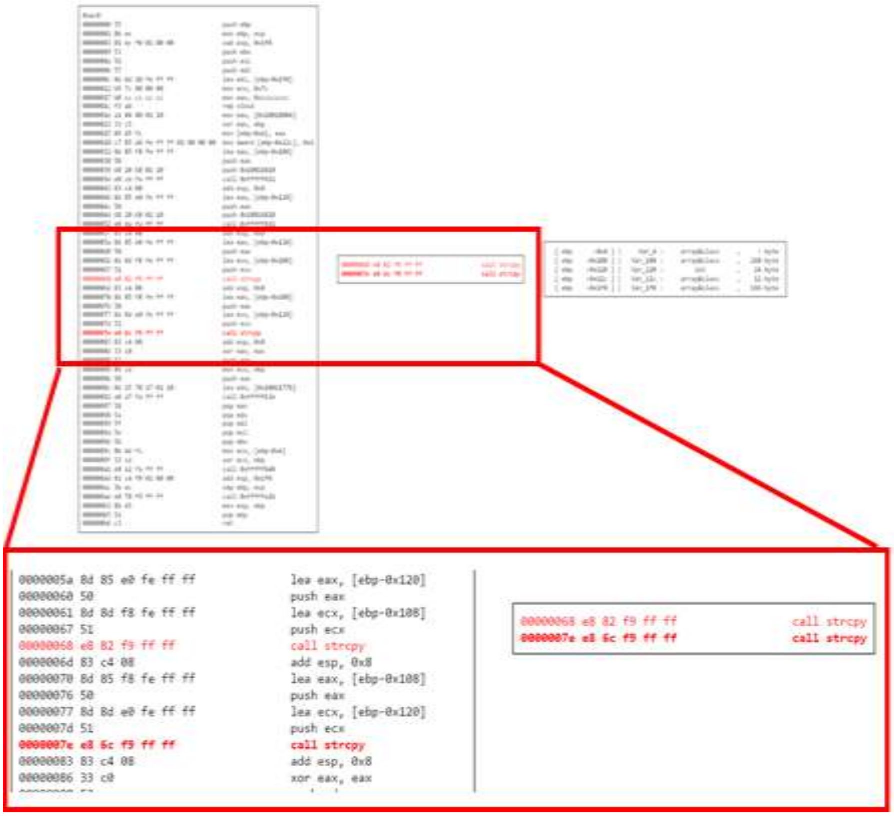

Figure 10. The testMain() Function Assembly Code Expressed As A Basic Block

As shown in the assembly basic block of the bof() function, if it is simply a call to a function, it is displayed in yellow, and when using a function known to be dangerous as shown in Figure 11, it is displayed in red.

5. Conclusion

Binary code generated in the Windows environment have had problems in C / C ++ programs with indirectly calling well-known vulnerable functions such as strcpy(). Therefore, it was difficult to analyze vulnerable functions using static analysis through reverse engineering.

In this paper, we analyzed the indirect call routines in the binary code of the software of the Windows environment and analyzed the information of the used DLL and function. We also analyze the binary code and propose a method to prevent the use of inherently dangerous function by comparing indirectly called functions with the list of banned functions. All of the functions imported and exported from the binary code could be made into lists, which enabled us to detect the use of inherently dangerous functions.

However, in this paper, we have not analyzed the vulnerabilities that can be generated due to the use of inherently dangerous functions. In future work, we will analyze data flow and parameter analysis to determine vulnerability from binary code.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP; Ministry of Science, ICT & Future Planning) (No. 2017R1C1B5018257) and also this research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2017R1D1A3B03029906). And this paper is a revised and expanded version of a paper entitled Indirect Call Functions Analysis of Binary Code in Windows Environment presented at 2017 1st International Workshop on Cultural and Technological Exchange and Mutual Development of The Pacific Rim Countries, Cheonan, Korea, December 17 2017.

References

[1] B. Chess and G. McGraw, "Static analysis for security", IEEE Security & Privacy, vol. 2, no. 6, **(2004)**, pp. 76-79.
[2] M. Shahzad, M. Z. Shafiq and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles", Proceedings of 34th International Conference on Software Engineering, Zurich, Switzerland, **(2012)**.
[3] N. Mehta, "The Heartbleed Bug", **(2014)**.
[4] S. Chazelas, "The Shellshock vulnerability", **(2014)**.
[5] B. Möller, T. Duong and K. Kotowicz, "This POODLE bites: exploiting the SSL 3.0 fallback", **(2014)**.
[6] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube and E. Käsper, "DROWN: Breaking TLS using SSLv2", Proceedings of 25th USENIX Security Symposium, Austin, TX, USA, **(2016)**.
[7] B. Martin, M. Brown, A. Paller and D. Kirby, "CWE/SANS Top 25 Most Dangerous Software Errors", **(2011)**.
[8] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: A taxonomy of software security errors", IEEE Security & Privacy, vol. 3, no. 6, **(2005)**, pp. 81-84.
[9] Apple, "Third-Party Software Security Guidelines".
[10] J. Viega, J.T. Bloch, Y. Kohno and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code", Proceedings of IEEE 16th Annual Conference on Computer Security Applications", New Orleans, USA, **(2000)**.
[11] S. Rawat and L. Mounier, "Finding buffer overflow inducing loops in binary executables", Proceedings of IEEE 6th International Conference on Software Security and Reliability (SERE), Gaithersburg, MD, USA, **(2012)**.
[12] J.M. Lee, H.W. Kim and W.H. Ahn, "BinaryReviser: A Study of Detecting Buffer Overflow Vulnerabilities using Binary Code Patching", Proceedings of the Korean Institute of Information Scientist and Engineers Conference, Gyeongju, South Korea, **(2011)**.

[13] T. Wang, T. Wei, Z. Lin and W. Zou, "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution", Proceedings of 16th Network and IT Security Symposium, San Diego, CA, USA, **(2009)**.
[14] Microsoft, "Windows PE Format", https://msdn.microsoft.com/ko-kr/library/windows/desktop/ms680547(v=vs.85).aspx/.

Authors

Junho Jeong, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 2007, and M.S. and Ph.D. degrees from the Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 2009 and 2015, respectively. Currently, he is a Researcher of the Electronic Commerce Institute, Dongguk University, Gyeongju, Korea. His research areas include Privacy Preserving, Distributed System, Network Security and Secure Software.

Yunsik Son, he received the B.S. degree from the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea, in 2004, and M.S. and Ph.D. degrees from the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea in 2006 and 2009, respectively. He was a research professor of Det. of Brain and Cognitive Engineering, Korea University, Seoul, Korea from 2015-2017. Currently, he is an Assistant Professor of Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. His research areas include secure software, programming languages, compiler construction, mobile/embedded systems, and u-Healthcare.

