Mitigating Use-After-Free Attack with Application Program Loader

Takamichi Saito Ryota Sugawara Masateru Yokoyama
Dept. of Science and Engineering
Meiji University
1-1-1, Higashi-Mita, Tamaku, Kawasaki-shi,
Kanagawa 214-8571, Japan
saito@cs.meiji.ac.jp

Shuta Kondo Hiroyuki Miyazaki Wang Bing Ryohei Watanabe
Dept. of Science and Engineering
Meiji University
1-1-1, Higashi-Mita, Tamaku, Kawasaki-shi,
Kanagawa 214-8571, Japan

***Abstract*-In the area of software security, use-after-free vulnerabilities have been reported since 2006. When the target vulnerable application is run, the attack exploits a dangling pointer after the heap memory is released. Until today, use-after-free attacks have been frequently reported in popular software such as browsers. This is a serious software security problem because a use-after-free attack allows an attacker to execute an arbitrary code to hijack an application control flow or to force a system crash. Some countermeasures have been proposed to thwart such attacks. However, most of these countermeasures have some problems such as the necessity of a source code or the problem of dependency. In this paper, we propose and evaluate the implementation of an application-level program loader to mitigate the use-after-free attack.**

Keywords-Mitigation; Use-After-Free; Vulnerability; Memory Corruption;

I. INTRODUCTION

Use-after-free vulnerabilities are categorized based on the Common Weakness Enumeration (CWE) [1] as CWE-416 [2]. Use-after-free vulnerabilities first appeared in 2006. According to the National Vulnerability Database (NVD) [3], the use-after-free vulnerabilities started to increase in 2010, and they are still reported quite frequently (Figure 1).

Although use-after-free vulnerabilities are mainly associated with the memory releasing process of C/C++ codes, they have been reported in generally widespread programs, such as Adobe's FlashPlayer, and major browsers such as Internet Explorer (IE) and Google Chrome, which has Javascript [4] engines.

An attacker targets use-after-free vulnerabilities to execute a shellcode by using a dangling pointer (hereafter called as use-after-free attacks). A dangling pointer is a pointer that points to a memory area that has already been freed.

Since use-after-free vulnerabilities are often reported, protection and mitigation techniques against them have been proposed. However, most of existing techniques have some problems. For example, in some cases or some applications, these techniques cannot protect from use-after-free attacks, or they require rewriting the library.

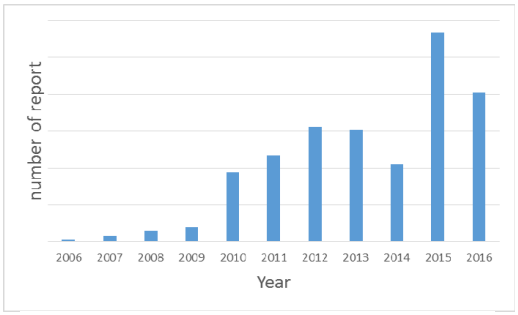

Fig.1. Number of Use-after-free Vulnerabilities in the NVD.

Also, some of countermeasures must be applied at the development phase. On that point, these countermeasures cannot be applied to the applications that are already distributed in the users. In addition, if these countermeasures is not applied at default, a non-expert developer cannot select to apply an application to these countermeasures. Hence, the countermeasure that is applied at run-time is required when the application is running.

Therefore, in this paper, we propose a program loader as an application, not an operating system (OS), replacing free() with secure free() to mitigate use-after-free attacks at run-time. In this paper, we focus on programs written in C and Executable and Linkable Format (ELF) binaries that are compiled using the GNU Compiler Collection (GCC) on a 32-bit Linux OS.

II. RELATED KNOWLEDGE

A. Use-after-free vulnerabilities

Use-after-free vulnerabilities involve the reuse of a pointer referencing memory area after it has been freed. They cause a running program to crash or create a dangling pointer. A use-after-free attack allows an attacker to execute an arbitrary code or to hijack an application control flow. Therefore, use-after-free vulnerabilities are as dangerous as other memory vulnerabilities.

1550-445X/17 $31.00 © 2017 IEEE
DOI 10.1109/AINA.2017.62

B. ELF

ELF is a common file format that is executable in Linux.

ELF manages file information by section and segment (Figure 2). A linker handles the sections, and a loader handles the segments. The loader expands a program to virtual memory in increments of segments by ELF information before a program is executed.

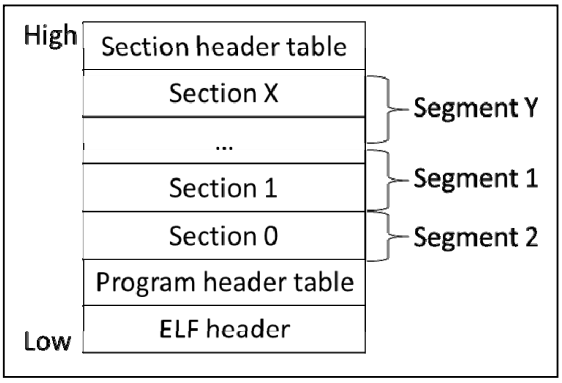

Fig. 2. ELF Structure.

Sections include information required for executing the program. For example, .text contains the instruction code and .rodata has read-only data. On the contrary, segments collectively manage similar sections. For example, the LOAD segment unites the sections related to map information such as .text and .data. Another example, the DYNAMIC segment unites the sections related to dynamic link information such as .dynamic.

In addition, .got, .got.plt, and .plt dynamically link shared libraries at runtime. .got and .got.plt are sections that store the addresses of libraries, which are dynamically linked. .plt runs when these addresses are called.

To illustrate the operation of sections, we examine the case of calling free() via .got.plt (Figure 3). When free() in .text is called, free@plt in .plt is called. After that, referring to .got.plt, if the address of free() is resolved, the corresponding entity of free() is called. However, if the address of free() is not resolved, then the address of free() in the shared library is calculated by .dynamic. Afterward, the address is stored in .got.plt to address the resolved state [6].

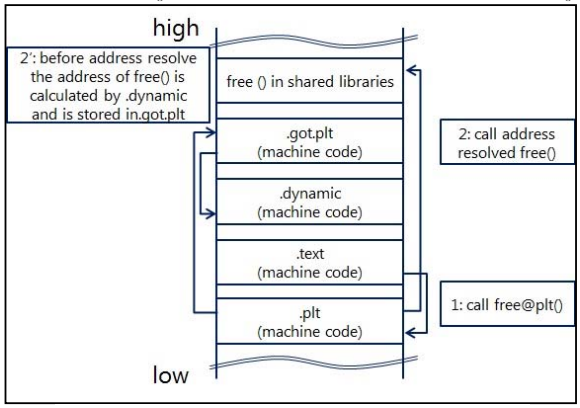

Fig.3. Calling shared libraries by using .got.plt.

III. EXISITING TECHNIQUES

Until now, various techniques have been proposed for protecting against or mitigating the impact of use-after-free attacks. The features and problems of these techniques are thoroughly discussed in [7]. Here, we offer a short description of these techniques based on that review study.

1) Naïve approach

The naïve approach for mitigating the results of a use-after-free attack is never to use the same heap area. This approach can completely protect against a future use-after-free attack, but it is extreme and impractical.

2) Pre-analysis technique

Pre-analysis technique involves detecting dangling pointers that may be used in a use-after-free attack before its running. The specific techniques in this category are tainting tracking, dynamic binary translation, and static binary analysis [8-15]. However, these techniques may not always be able to protect against a use-after-free attack.

3) Compiler technique

In this technique, an additional code for detecting a use-after-free attack is added at the compilation time, in order to detect dangling pointers at run-time [13,16]. This technique prevents a use-after-free attack by detecting the attack at run-time. However, these kinds of countermeasures cannot be applied without a source code, and the program cannot incorporate the newly required countermeasures after it has been compiled.

4) Library technique

Replacing the original library with the modified a run-time library that has a safe release process and a safe memory allocation system is another technique that has been proposed [17-19]. In addition, similar techniques involve replacing malloc() on a page-by-page basis [7,20]. These techniques prevent a use-after-free attack at run-time, but present some problems. For example, they reduce the efficiency of memory use, and they require replacing the library, as a result, it occurs the problem of the library dependency.

5) Security patch

Another approach to prevent a use-after-free attack is to apply a security patch to the target program. For example, there are two such security patches: MS14-035 [21], which modifies the memory corruption vulnerability in IE, CVE-2014-1770; and MS14-037 [22], which modifies the memory corruption vulnerability in IE, CVE-2014-1763, and CVE-2014-1765.

In MS14-035, functions that use the heap allocate the memory block in the heap only for IE instead of the original heap. In MS14-037, this delays memory-block release. Unfortunately, these measures, which provides a

patch for fixing the vulnerability in every case, protect only IE and cannot be applied to other programs.

IV. Overview

In a use-after-free attack, [7][22] indicated that an attacker often reuses memory blocks immediately after they have been released. Therefore, in this paper, we propose an original program loader (hereafter called as SafeTransLoader) that loads the target protected program into a memory at run-time and simultaneously applies the proposed countermeasure to the target protected program. SafeTransLoader replaces the target program's original free() with SafeTransLoader's free() (hereafter called as Hfree), delaying the memory-release process. These measures prevent and mitigate use-after-free attacks without rewriting the binary, recompiling the source code, and replacing the library.

SafeTransLoader is a modification of the program loader [23]. The target program is ELF, which is used commonly in Linux. Due to space limitations, in this paper, we only replace *free*(), even though SafeTransLoader can also replace other unsafe functions [25] of C/C++ such as strcpy().

A. SafeTransLoader

When SafeTransLoader executes the target program, it modifies the target program.

First, SafeTransLoader, which is located in from 0x20000000, is expanded to the heap area of the virtual memory. Second, the target program, which is located in from 0x08048000, is expanded to the heap area of the virtual memory on SafeTransLoader.

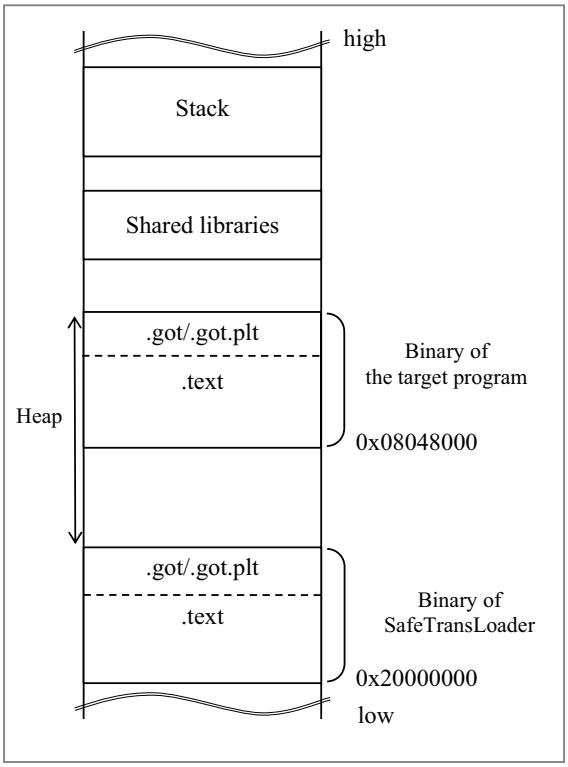

Fig.4. Memory Map Image of SafeTransLoader

SafeTransLoader analyzes the ELF header, the program header, and the section header of the target program before SafeTransLoader maps the target program. As a result, SafeTransLoader obtains information on the offset, size, and attributes of all sections and segments of the target.

The segments information of the target program are important for SafeTransLoader. The shared library addresses contained in the DYNAMIC segment of the target program are relocated properly, after the DYNAMIC segment of the target program is mapped in virtual memory. In particular, after SafeTransLoader map the DYNAMIC segment containing .got and .got.plt of the target program in virtual memory, SafeTransLoader replaces the address of the original *free*() code with the address of *Hfree*() defined in SafeTransLoader. However, if the shared library address has already been relocated, the address of the target program is replaced with it.

SafeTransLoader replaces the original *free*() in the section .got or .got.plt in the binary of the target program, as shown in Figure 4. SafeTransLoader achieves to replace the unsafe original *free*() to the safer *Hfree*() with rewriting the shared library address when the target program is loaded. In our proposal, *Hfree*() is defined in SafeTransLoader as a global function.

SafeTransLoader maps only the LOAD and DYNAMIC segments of the target program, and then the others segments are shared between SafeTransLoader and the target program.

After the target program has been loaded to be modified, the program counter of CPU is set to the beginning of the .text section to transfer the program flow from SafeTransLoader to the target program.

B. Hfree

We describe how the *Hfree*() code realizes the safe memory-release process in Figure 5.

```
1  void* Hfree(void *ptr){
2
3          /* initialize */
4          static int f_bottom = 0;
5          static int f_count = 0;
6          static void *free_queues[7] = {NULL};
7          static int f_flag = 0;
8
9          /* release */
10         if(f_flag){
11                 free(free_queues[f_bottom]);
12                 f_bottom = (f_bottom+1)%7;
13         }
14
15         /* allocate pointer */
16         free_queus[f_count] = ptr;
17         if(f_count==6){
18                 f_flag=1;
19         }
20         f_count = (f_count+1)%7;
21 }
```

Fig.5. Code of *Hfree*().

In run-time of the target program, *Hfree*() stores the pointer referring to the memory block, that is to be released next in the queue, as a static variable in SafeTransLoader, each time the free function *Hfree*() is called. If *Hfree*() is called the same number of times as the depth of the queue, we set that the depth is 7 in this experiment, the memory block is released. The memory-release process in the *Hfree*() code releases one pointer stored in the queue, free_queues[f_bottom] in Figure 5, in the old order.

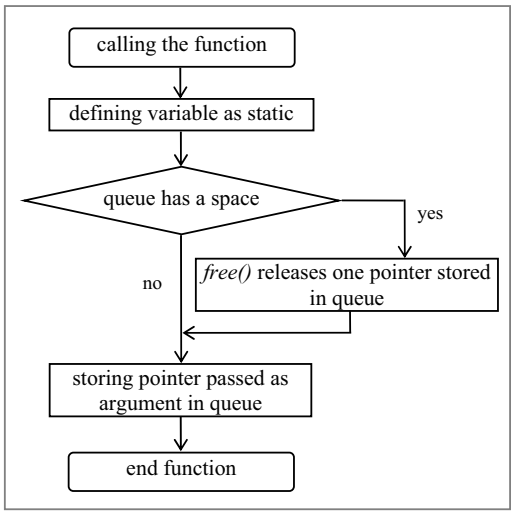

Fig.6. Flow of *Hfree()*

V. EVALUATION

A. Effectiveness against use-after-free attack

For evaluating our proposal, we prepare the program shown in Figure 7 (hereafter called as targetX), which is based on the program of [24] including use-after-free vulnerabilities, and we evaluate its effectiveness of SafeTransLoader. When we execute a use-after-free attack against targetX on SafeTransLoader, we confirm that the use-after-free attack is successfully prevented.

Our measurement environment is as follows:

- Ubuntu 12.04.5 LTS 32 bit
- GCC 4.6.3

In the experiment, for simplicity, Address Space Layout Randomization [25] and Data Execution Prevention [25] are disabled in this evaluation.

The targetX allocates the structure *str* and the member *buf* of the structure as a memory block in the heap. At this time, the size of the structure *str* is 8 bytes because 4 bytes of the flag member of the structure plus 4 bytes of the pointer referring to *buf*, and then *buf* refers to 100 bytes in the memory block. After that, it stores the shell code given by argv[1] into *buf* and releases the memory block referred to by *buf* and *str*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct Str {
6          int flag;
7          char *buf;
8  };
9
10 int main(int argc, char *argv[])
11 {
12         struct Str *str;
13         int size;
14         char *buf2;
15
16
17         str = malloc(sizeof(struct Str));
18         str->flag = 1;
19         str->buf = malloc(100);
20
21         strncpy(str->buf, argv[1], 100);
22
23         free(str->buf);
24         free(str);
25
26         size = atoi(argv[2]);
27         buf2 = malloc(size);
28
29         strncpy(buf2, argv[3], size);
30
31         strncpy(str->buf, argv[4], 100);
32
33         free(buf2);
34         return 0;
35 }
```

Fig.7. Code including use-after-free vulnerabilities

Next, the variable *buf2* allocates the memory block in the heap. At this time, to let *buf2* reuse the memory block referred to by *str*, the required memory block size is 8 bytes given by argv[2]. After that, the address referring to

TABLE1. EXECUTION TIME WHEN WE EXECUTE targetX 1,000,000 TIMES.

	Standard loader in Linux	SafeTransLoader
real (rate of increase from Standard loader)	75 min 50.449 s (0%)	78 min 50.993 s (3.968%)
user (rate of increase from Standard loader)	1 min 2.524 s (0%)	1 min 9.344 s (10.908%)
sys (rate of increase from Standard loader)	34 min 44.128 s (0%)	35 min 15.248 s (1.493%)

free() in .got and .got.plt (given by argv[3]) is written into the released memory block storing the pointer referring to *buf*.

In line 31 in Figure 7, the sentence has use-after-free vulnerabilities. Writing is performed from the address of the pointer referring to *buf*, which has already been released. At this time, since the writing destination is the address referring to *free*() in .got and .got.plt, the address to *free*() is replaced in the address of the pointer referring to *buf* storing the shell code given by argv[4].

Therefore, when *free*() is called in line 33, *free*() is transferred to the shell code, and the shell is started.

B. Overhead

Here, we executed targetX 1,000,000 times and then measured the execution time both the standard loader of Linux and SafeTransLoader. We used Ubuntu 12.04.5 LTS 32 bit and an Intel Core i5-3450 Processor. We used the compiler GCC 4.6.3. We used the time command to measure the execution time. The command measures time in three ways: the real time from program start to end (real), the execution time of the program itself (user), and the OS execution time (sys).

As a result, the execution time of SafeTransLoader is 4% slower than the execution time of the standard loader.

VI. Discussion

In this paper, we demonstrate the use of SafeTransLoader for preventing a use-after-free attack. In this section, we compare SafeTransLoader with the other techniques and show its effectiveness.

A. Comparison with the pre-analysis technique

In the pre-analysis technique, a dangling pointer is detected before the program starts. However, use-after-free vulnerabilities are difficult to detect before running because, in a given execution, the unsafe dangling pointer may not be created before the program starts, or it may be created but not used [9]. Hence, there is a possibility that the pre-analysis technique misses use-after-free vulnerabilities. On the other hand, SafeTransLoader can counter a use-after-free attack at run-time.

B. Comparison with the compile technique

According to the compile technique, the detecting use-after-free attack code is inserted into the program during compilation. In general, a compile countermeasure is strong, because it has a source code containing important information and can insert an additional code [26]. However, the compile technique can only be applied to a source code. Hence, the compiled program cannot be modified afterward. We consider SafeTransLoader to be better than the compile technique with regard to portability.

C. Comparison with the library technique

Here, SafeTransLoader is not implemented in the library technique. In general, it is impossible to apply a library countermeasure to the program if the library depends on the library of the old version (DLL Hell or Dependency Hell). Hence, SafeTransLoader is considered better than the library technique because the countermeasure can be applied without these dependencies.

D. Superiority program loader

In general, OS countermeasures are robust and perform well [26]. Nevertheless, for example, in the embedded OS or in the legacy environment, changing OS affects the application of countermeasures. In addition, the user, in this case, is required to switch out the countermeasures when switching OS. Specifically, in the embedded OS, a firmware is switched out to introduce and apply the countermeasures. On the other hand, since SafeTransLoader runs as a user application independent of the OS, the countermeasures can be applied without switching OS. It is possible to insert the SafeTransLoader function into the OS loader, but due to the above reasons, we propose using SafeTransLoader as a program loader that has greater portability.

E. Software lifecycle

We discuss that a countermeasure against use-after-free attack should be applied at the development phase or the operational phase.

According to Security by Design [27], security measures should be applied from the planning phase or the design phase. However, if an appropriate measures is applied at the development phase, vulnerability is often discovered at the operational phase. Hence, although contrary to the policy of Security by Design, we propose the approach that security measures is able to be applied at the operational phase. Of course, It is important to ensure the security. However, as example of the application of the security at the operational phase, there is Web Application Firewall (WAF) [28]. WAF does not apply a web application to countermeasures at the development phase but at the operational phase.

F. Other issues

As mentioned above, use-after-free vulnerabilities have been reported in various programs such as major browsers (IE and Google Chrome). However, since SafeTransLoader is yet in a prototype state, it cannot execute these browsers. Hence, we cannot evaluate, at present, its validity against use-after-free attack in browsers. Therefore, this should be part of the tasks of our next work. Also, another issue is remained. In this paper, we evaluate only targetX. We cannot say that it is enough. Hence, we need to use the well known benchmarks such as SPEC CPU2006 [29] and to show effectiveness by preventing more use-after-free attacks.

VII. Conclusion

In this paper, we show that SafeTransLoader can mitigate use-after-free attacks.

In our implementation, we replace only one unsafe function, *free*(), with a safe function defined in SafeTransLoader. However, the proposed program can

replace multiple unsafe functions with safe functions as defined by C11 or ISO/IECTR24731-2:2010 to prevent use-after-free attacks.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 26330162. We are deeply grateful to Y. Hori for this study.

References

[1] CWE:Common Weakness Enumeration, http://cwe.mitre.org/index.html

[2] CWE-416: Use After Free, https://cwe.mitre.org/data/definitions/416.html

[3] NVD: National Vulnerability Database, https://nvd.nist.gov/

[4] http://www.microsoft.com/ja-JP/download/details.aspx?id=42646

[5] John R. Levine (1999), Linker & Loader, Morgan Kaufmann

[6] http://7shi.hateblo.jp/entry/2013/05/25/103050

[7] T. Yamauchi and Y. Ikeuchi: Memori sairiyou kinshi ni yoru use-after-free zeijakusei kougeki boushishuhou no teian to jissou, Information Processing Society of Japan 2015.

[8] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," Proc. 21st USENIX Conference on Annual Technical Conference, pp. 309-318, 2012.

[9] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities," Proc. 21st International Symposium on Software Testing and Analysis, pp. 133-143, 2012.

[10] N. Nethercote, and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," Proc 11th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 89-100, 2007.

[11] D. Bruening, and Q. Zhao, "Practical Memory Checking with Dr. Memory," Proc. 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 213-223, 2011.

[12] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing Use-after-free with Dangling Pointers Nullification," Proc. 22nd Annual Network and Distributed System Security Symposium, NDSS, 2015.

[13] D. Bruening, and Q. Zhao, "Safedispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," Proc. 21st Network and Distributed System Security Symposium, pp.1-15, 2014.

[14] M.L. Potet, J. Feist, and L. Mounier, "Statically Detecting Use After Free on Binary Code," J. Comp. Virol. Hack. Tech, Vol. 10, No. 3, pp. 211-217, 2014.

[15] D. Dewey, and T. Giffin, "Static Detection of C++ Vtable Escape Vulnerabilities in Binary Code," Proc. 19th Network and Distributed System Security Symposium, pp.1-14, 2012.

[16] C.F. Eigler, "Mudflap: Pointer Use Checking for C/C++," http://gcc.fyxm.net/summit/2003/mudflap.pdf

[17] Pageheap http://technet.microsoft.com/ja-jp/library/cc835607.aspx

[18] Electric fence, http://elinux.org/Electric_Fence

[19] DUMA, http://duma.sourceforge.net/

[20] P. Akritidis, "Cling: A Memory Allocatorto Mitigate Dangling Pointers," Proc. 19th USENIX Conference on Security, pp. 177-192, 2010.

[21] J. Tang, "Isolated heap for inter-net explorer helpsmitigate uaf exploits," http://blog.trendmicro.com/trendlabs-security-intelligence/isolated-heap-for-internet-explorer-helps-mitigate-uaf-exploits/

[22] J. Tang, "Mitigating uaf exploits with delay freefor internet explorer," http://blog.trendmicro.com/trendlabs-security-intelligence/mitigating-uaf-exploits-with-delay-free-for-internet-explorer/

[23] T. Saito, T. Uehara, Y. Kaneko, M. Suzuki, Y. Sumida, H. Yosuke, T. Baba, and H. Miyazaki, "A Prototyping and Evaluation of Prevention/Mitigation of Released Binaries against Stack-based Buffer Overflow Attacks, Symposium on Cryptography and Information Security 2015"

[24] http://inaz2.hatenablog.com/entry/2014/06/18/215452

[25] T. Saito, Mastering TCP/IP the book of information security, Ohmsha, 2013.

[26] T. Saito, Y. Sumida, Y. Hori, T. Baba, H. Miyazaki, B. Wang, R. Watanabe, and S. Kondo, Safe Trans Loader: Mitigation and Prevention of Memory Corruption Attacks, Symposium on Cryptography and Information Security, 2016.

[27] http://www.nisc.go.jp/active/general/sbd_sakutei.html

[28] Web application Firewall (WAF) http://www.ipa.go.jp/security/vuln/waf.html

[29] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmark suite, 2006. http://www.spec.org/osg/cpu2006/