An Intrusion Detection System which can Restore Altered Data

Fumiaki NAGANO† Kohei TATARA† Toshihiro TABATA‡ Kouichi SAKURAI†
†Kyushu University, 744 Motooka, Nishi-ku Fukuoka, Japan, 812-8581
‡Okayama University, 3-1-1 Tsushima-naka, Okayama, Japan, 700-8530
{nagano, tatara}@itslab.csce.kyushu-u.ac.jp, tabata@cs.okayama-u.ac.jp, sakurai@csce.kyushu-u.ac.jp

Abstract – We propose an intrusion detection system. Our system can detect the alteration of data in memory and also can restore altered data. This type of intrusion detection system has been proposed variously so far. But many of them can detect only a part of attacks. And as far as we know, few of them can restore altered data. Our system can detect attacks which can not be detected by existing systems and also can restore altered data. Our system protects data in the kernel area using hash functions. The overhead of accessing the kernel area and using a hash function is high. But our system reduces the frequency of accessing the kernel area and using a hash function in safety.

Index Terms – Buffer Overflow, Format String, Integer Overflow

1 Introduction

A lot of attacks against computers using the alteration of data have been reported in recent years. One of the most common attacks use buffer overflow vulnerability, according to the Symantec Internet Security Threat report [1]. If malicious attackers use buffer overflow vulnerability, attackers could execute arbitrary code [2].

Besides attacks using buffer overflow vulnerability, there are many attacks against computers using the alteration of data: e.g., Format String Bug, Integer Overflow vulnerability. Moreover, attacks using buffer overflow vulnerability have became crafty recently; heap overflow attack is one of the crafty attacks.

A detection system for detecting the alteration of data is therefore indispensable to defend these attacks.

1.1 Organization of the paper

The rest of this paper is organized as follows. In Section 2, we describe related works. In Section 3, we describe our proposed method. In Section 4, we discuss our implementation approach and our micro benchmark. Finally, we provide a summary and discuss future works.

2 Related works

Bound checking
A bound checking in [3] checks the bounds of buffers to detect the alteration of data. This method provide full bounds for C code while maintaining sizeof(void *) == sizeof(int). This compiler does so by using associative lookup on each pointer reference to an array descriptor that stores base and bounds. However, it was reported that this method can not detect the alteration of data using Format String Bug [4]. Moreover, this method can not restore altered data.

Non-Executable Buffers
Non-executable stack segments [5] prohibits executing a code on stack area. PAX project prohibits executing a code on heap area.

However, it was reported that these methods can not detect the alteration of data using Format String Bug and can not apply to code with recursive programs [4]. Moreover, this method can not restore altered data.

Variable protection

These methods, however, are bypassed by notorious attackers because the attackers use other variables which are not protected by these methods. The attackers can also guess canaries by using vulnerability that enable attackers to read data in memory [10]. Moreover, this method can not restore altered data.
Random address space

A method in [11] randomizes the location of key memory objects. Attackers can not understand the location for attacks.

However, it was reported that randomizing address space was not so efficient in many cases [12]. Moreover, this method can not restore altered data.

3 Proposed method

Our system can detect attacks which can not be detected by existing systems and also can restore altered data.

Our proposed system uses “Verifier” to detect the alteration of data and to restore altered data. Verifier consists of the arrays of “Verifier Structures” and the lists of “Verifier Structures”. A Verifier Structure is a structure which is used for detecting the alteration of data and restoring altered data. Verifier Structures exist in the kernel area and user memory. The work of Verifier Structures in kernel area is to detect the alteration in Verifier Structure in user memory. The work of Verifier Structures in user memory is to detect the alteration of vulnerable data and to restore altered data. Our proposed system assume that the kernel area is the area which attackers can not access. And from now we abbreviate Verifier Structure to VS.

3.1 Overview of our proposed system

Figure 1 shows our proposed system’s flow. Our system makes Verifier when a process starts. When Verifier is made, the arrays of VSes and the lists of VSes are empty. After that period, VSes are made for vulnerable data which might be used by attackers. A VS in user memory is made before vulnerable data could be altered by attackers and is stored to Verifier. And a VS in kernel area is made to detect the alteration of the VS in user memory and is stored to Verifier.

When the program uses the vulnerable data (a VS was made for the vulnerable data), our system checks vulnerable data using Verifier. To detect the alteration of the VS in user memory, we use the VS in kernel area. When we can not detect the alteration of the VS in user memory, we use the VS to detect the alteration of the vulnerable data.

3.2 Verifier

Figure 2 shows Verifier [13]. The number of VSes in kernel is $K$. In user memory, there are Verifier Structure Arrays and Verifier Structure Lists. A Verifier Structure Array is the array of VSes. (Verifier Structure Array is abbreviated to VSA from here.) A Verifier Structure Lists is the list of VSes. (Verifier Structure List is abbreviated to VSL from here.) The number of VSes in VSAs is $N \times K$. $N$ is the number of VSes in one VSA. Next we explain each element of a VS.

The elements of a VS are shown at Figure 3. The elements in kernel and the elements in user memory are different. Moreover, the elements of the VA of a VSA and the elements of the VS of a VSL are different. The explanation of each element is as follows.

\{The elements of a VS in kernel\}

- General Verification Data
  This element stores the hash values of VSes in user memory.
- Altered Flag
  The flag is set when the alteration of VSes in user memory are detected.
- cache
  These are the caches of VSes in user memory. We will explain the detail in section 3.4.1.

\{The elements of a VS in user memory\}

Verifier Structure in kernel

Verifier Structure in user memory

Note: Structure Pointer exists only in Verifier Structure List

Figure 3. Verifier Structure

- Verification Data
  This element stores the hash value of vulnerable data or the value of vulnerable data.
- Verification Pointer
  This element stores the address of vulnerable data.
- Verification Length
  This element stores the length of vulnerable data.
- Altered Flag
  The flag is set when alteration of vulnerable data is detected.
- Control Flag
  The flag shows whether this VS is used or not. And this flag also shows whether this Verification Data is a hash value or a real value.
- Structure Pointer
  This element stores the address of a VSL. This element only exists in VSLs.

3.3 Verification method

3.3.1 Initialization

When a process is executed, initialization is done. That is, Verifier shown at Figure 2 is made. But VSLs are not made in this time. The Control Flags in kernel and in user memory are initialized to show the VSes are not in use. From here, we do not think about VSL. VSL is used when a VS can not be stored in VSA. The essence is not different.

3.3.2 How to make a VS in user memory

VSes should be made dynamically because data are made dynamically in a program. For example, return addresses are made dynamically. So in our proposed method, VSes can be made dynamically.

The work of a VS in user memory is to detect the alteration of vulnerable data and to restore altered data. The elements of a VS in user memory are Verification data, Verification Pointer, Verification Length, Altered Flag and Control Flag. The Verification Data stores the hash value of the vulnerable data or the value of the vulnerable data. Our system uses MD5 as a hash function. The Verification Pointer stores the address of the vulnerable data. The Verification Length stores the length of the vulnerable data. The Altered Flag is not used in this period. The Control Flag is updated to show this VS is used and to show this Verification Data stores the hash value or the real data.

3.3.3 How to store a VS in user memory into Verifier

Afterwards, the VS is stored into Verifier. In this period, our system needs to determine which VSA should be used and which VS in the VSA should be used. From now we explain them.

First, we explain where to store the VS in $K$ VSAs. This algorithm uses a hash search to determine where to store the VS. The value of a Verification Pointer is divided by $K$, and that remainders are used to determine where to store the VS. For example, a VS is stored in the top of the VSAs in Figure 2 when the remainder is 1. We recommend that the value of $K$ is a prime number. The reason is that remainders divided by $K$ could be random numbers.

Next we should decide which VS of the VSA is used. To decide it, Control Flags are referred. The Control Flags of unused VSes have been initialized to show the VSes are not in use. Therefore, it is possible to search an unused VS by using Control Flags. If our system finds the same Verifier Pointer and Verifier Length in used VSes, when our system store a new VS, our system stores the new VS on that used VS to save memory.

3.3.4 How to make a VS in kernel area

Next, we explain VSes in kernel. These VSes have two purposes. One is to detect the alteration of VSes in user memory. That is to say, the alteration of VSAs can be detected even if attackers alter them. And the other purpose is to preserve the caches of VSes in user memory to reduce overhead when verification. The latter will be explained in section 3.4.1.

The elements of a VS in user memory are General Verification data, cache and Altered Flag. The General Verification Data stores the XORed values of the hash values of used VSes (refer Fig 5). Our system uses MD5 as a hash function.

We will explain cache in section 3.4.1. The Altered Flag is not used in this period.
3.3.5 How to store a VS in kernel area into Verifier

Afterwards, the VS is stored into Verifier. One VS in kernel verifies one VS in user memory. So the VS is stored into corresponded VS.

At this period, our system finishes making Verifier. Verifier has become like Figure 5.

3.3.6 How to verify vulnerable data

From here, we explain how to verify vulnerable data. Verifying vulnerable data has two processes. (1), Checking the VS in user memory : (2), Checking the vulnerable data.

(1), Checking the VS in user memory

First of all, our proposed system verifies the alteration of the VSA which stored the Verification Data of the vulnerable data. The General Verification Data for the VSA which stored the Verification Data is calculated again using the method shown at Figure 4. Then our proposed system compares the calculated result and corresponding General Verification Data of the VS in kernel (refer to Figure 6).

Because kernel is not accessed by attackers under our assumption, General Verification Data in kernel can not be altered. Therefore, when the calculated result and corresponding General Verification Data are equal, our system can verify that the VSA has not altered. When these two values are not equal, the Altered Flag in the VS is updated to show that alteration has been detected.

(2), Checking the vulnerable data.

When alteration of the VSA has been undetectable, the vulnerable data can be verified with the Verification Data in the VS. The concrete method is as follows. The Verification Data for the vulnerable data is calculated again, and our proposed system compares the calculated result and corresponding Verification Data which is convinced of being unaltered. If both values are equal, our system can verify that the vulnerable data has not been altered. When these two values are not equal, the Altered Flag in the VS is updated to show that alteration has detected. If the Control Flag shows that this VS stores real value of the vulnerable data, our system can restore the altered data.

3.4 Consideration

3.4.1 Cache

The method above is inefficient in a case. An inefficient case arises when a lot of VSes are stored in user memory. The method has to verify all VSes in user memory even when only one VS is verified. This is inefficient clearly. Therefore, to solve this problem, our system uses caches.

Caches are the copies of VSes in user memory. As a result, if there is a cache of the VS which contains Verification data, the VSA need not be verified. This is because kernel is safe according to our assumption. It might not be impossible to store all VSes in kernel. However, kernel memory is usually a precious resource. Therefore we should not store all VSes.

The hit rate may be high because VSes have the feature of “first in first out”. For example, a variable which is used in a local function is not used in another local function. We will sophisticate the cache using this feature.
Table 1. CERT Advisories Classification (1999-2002)

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Number</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer overflow</td>
<td>49</td>
<td>44.95%</td>
</tr>
<tr>
<td>format string</td>
<td>9</td>
<td>8.26%</td>
</tr>
<tr>
<td>double free</td>
<td>2</td>
<td>1.83%</td>
</tr>
<tr>
<td>integer overflow</td>
<td>3</td>
<td>2.75%</td>
</tr>
<tr>
<td>backdoor/Trojan horse</td>
<td>8</td>
<td>7.35%</td>
</tr>
<tr>
<td>denial of service</td>
<td>5</td>
<td>4.59%</td>
</tr>
<tr>
<td>others</td>
<td>33</td>
<td>30.28%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>109</strong></td>
<td><strong>100.00%</strong></td>
</tr>
</tbody>
</table>

3.4.2 Vulnerable data

User’s policy can decide what data should be detectable and restorable using a VS. There are so many data which are used for attacks. Pointers are most vulnerable data [7]. So we recommend that pointers are detectable and restorable. However, arguments of execve system call [11] and integers using Integer Overflow [14] are also used for attacks. Table 1 shows a classification of vulnerable data from 1999 to 2002 [14]. The other categories in Table 1 include vulnerabilities such as worms, viruses, weak authentication, or insecure default settings. Our targets of this paper are detecting the alteration of data in memory. So our targets are buffer overflow, format string, double free and integer overflow. This table shows that integers (which is not a pointer) is vulnerable data because integers is used for integer overflow attacks.

Our system can make VSes for all vulnerable data to detect alteration and to restore altered data. However the overhead becomes high. This relation is trade off.

3.4.3 When to make/verify VSes

Our system has two policies about when our system makes/verifies VSes. One is anomaly detection. Another is misuse detection.

Anomaly detection makes VSes right after making data and verifies them right before using the data. Misuse detection makes VSes before the fear of attacks and verifies them after the fear of alteration of the data. We show the difference of the two at Figure 7.

Anomaly detection can detect unknown attacks. We now assume that an attack never happens during [⋯] in Figure 7. But we can not conclude that attacks will never happen during [⋯] in Figure 7. Anomaly detection protects data right after making data and verifies them right before using the data. So anomaly detection can detect unknown attacks.

Misuse detection can reduce the overhead. We show our experiment result at Figure 8. We measured time needed for hash calculation. The x-axis shows the length of calculated data. (CPU Pentium(R) III 1GHz, MEMORY 512MB)

We now consider an array. The array has four elements which size is 15 byte. When we need to calculate the hash against each element, we need \(2.64 \times 4 = 10.56 \, \mu s\). But when we calculate the whole elements, we need only \(5.17 \, \mu s\).

As you see from this example, misuse detection can reduce overhead because it can make a hash value for plural data.

Now we have to think about "the fear of attacks" in Figure 7. For the answer, [15] proposed that I/O system calls are dangerous. They define an I/O system call as a system call interacting with I/O objects (files, pipes, sockets, etc.) or with other operating system resources (e.g. send signals to other processes, manage virtual memory attributes, change the ownership of the process, etc.).

According to the data of STACKFENCES [15], when our system makes VSes for return addresses using anomaly detection, our system needs to access to kernel area and hash calculation 437,946 times. But our system only needs 48 times using misuse detection. As you see from this ex-
ample, misuse detection can reduce overhead.

4 Implementation

4.1 How to implement

To implement our proposed system, the following four things are necessary.

1. A system call to store VSes into Verifier in kernel.
2. A system call to compare VSes in user memory and General Verification Data.
3. A function to store VSes into Verifier in user memory.
4. A function to compare vulnerable data and Verification Data.

1 and 2 become possible when new system calls are installed. 3, 4 become possible when a compiler is improved. Moreover, if these four things are implemented, programmers can detect the alteration of any data and restore the altered data at arbitrary positions.

4.2 Micro benchmark

We show the result of our micro benchmark. We made a VS for a word (="test") before a buffer overflow attack and a format string attack, and we checked whether our system can detect the attacks and can restore the altered data after the attacks. The result of our micro benchmark, we could verify that our system can detect the attacks and can restore the altered data.

As you see in this example, if our system can find a cache, our system can reduce the overhead because our system need not a hash calculation.

5 Conclusion

We proposed a system which can detect the alteration of data in memory and also can restore altered data. Our system protects data using a hash function and kernel area. Although the overhead of accessing kernel area and using a hash function is usually high, we proposed a policy that can reduce the overhead.

Future works are further experiments and the evaluations of actual applications. Moreover, we will sophisticate the cache to reduce the overhead.

References