Received: 16 September 2019 | Revised: 21 May 2020 | Accepted: 16 June 2020

DOI: 10.4218/etrij.2019-0427

ORIGINAL ARTICLE

ETRI Journal WILEY

A method for preventing online games hacking using memory monitoring

Chang Seon Lee[1] | **Huy Kang Kim**[2] | **Hey Rin Won**[3] | **Kyounggon Kim**[2]

[1]Trust & Safety of Cyber Security Center, LINE Corporation, Tokyo, Japan

[2]School of Cybersecurity, Korea University, Seoul, Rep. of Korea

[3]Department of Information Security, Seoul Women's University, Seoul, Rep. of Korea

Correspondence
Kyounggon Kim, School of Cybersecurity, Korea University, Seoul, Rep. of Korea.
Email: anesra@korea.ac.kr

Abstract

Several methods exist for detecting hacking programs operating within online games. However, a significant amount of computational power is required to detect the illegal access of a hacking program in game clients. In this study, we propose a novel detection method that analyzes the protected memory area and the hacking program's process in real time. Our proposed method is composed of a three-step process: the collection of information from each PC, separation of the collected information according to OS and version, and analysis of the separated memory information. As a result, we successfully detect malicious injected dynamic link libraries in the normal memory space.

KEYWORDS

abuse monitoring, game abuse, injection detection, malware detection and damage recovery, memory map

1 | INTRODUCTION

Online game hacking is typically used to level up a player's character faster than in normal game play. Additionally, gold-farming groups, which are malicious groups that use game hacking programs to gain illegal financial profits, are formed to monopolize in-game items and money [1]. A gold farmer in an online game aims to harvest virtual money using automated programs (eg, game bots and macros) or hired low-cost laborers [2]. Real Money Trading (RMT) in online games is often associated with other criminal activities, such as money laundering and identity theft [3,4]. Online game hacking can be classified into three types: automatic play using a macro, memory modification, and denial of service. In this study, we focus on detecting memory modification programs. To modify the memory of an online game client, hackers often employ security solution bypassing, dynamic-link library (DLL) injection, and memory modification techniques. As in an arms race, hackers continuously identify new methods to bypass the security measures used by online game companies, while the companies rely only on signature-based detection and heuristics.

The most widely adopted methods to detect game hacking tool are signature-based or heuristic-based detection, whereas in this study, we propose a method to detect users who use memory injection techniques. Our proposed method is lightweight and can achieve high accuracy. The organization of the rest of this paper is as follows. In Section 2, we describe well-known techniques to attack online games, and review the literature. In Section 3, we present the main algorithm of the proposed detection method. In Section 4, we present the experimental result. Finally, in Section 5, we conclude this paper and suggest directions for future research.

2 | BACKGROUNDS

Online game hacking can be performed in many ways, such as auto mouse, macro, map, and speed hacks [5]. In particular, this section explains attack methods using security solution

bypassing, memory forgery and alternation, and DLL injection, which is a representative memory hacking technique. The most common way to block a DLL injection is to use a game security solution. However, because game security solutions operate in the same PC environment, it is possible to bypass the attacker by analyzing the security solution; thus, the defense using only the security solution in the user's PC is limited.

2.1 | Security solution bypassing

Some of the current security solutions for online games include GameGuard and XignCode. These security solutions feature the detection of client forgeries, as presented in Table 1. However, a PC owner can become an attacker and can be circumvented through analysis.

Therefore, security solutions are not the best way to safeguard games. When a security solution is bypassed, DLL injection and code injection are possible using several techniques.

2.2 | Memory forgery and alteration

If the security solution is bypassed, the attacker can then analyze the online game client and attempt to attack based on the analysis. One of the methods of attack is memory forgery and alteration. Memory forgery and alteration changes some options and values used by the game client on the operating system to make the game easier and faster. Memory forgery and alteration uses the process illustrated in Figure 1.

The attack method depicted in Figure 1 uses the Cheat Engine program to modify the memory, and the modified value is reflected in the Minesweeper game. By simply changing these values, the attacker can achieve the desired result in the game. In online games, this simple method of attack is not possible; however, a simple online game client running on a user's PC is vulnerable to a forgery attack.

2.3 | DLL injection

A "DLL" refers to modules and programs with functionalities that can be shared by other applications [6]. Because a DLL is a module developed program, it can be updated and reused more easily than application programs. When a DLL is loaded into the memory address space, the application can access it at any time to take advantage of its functionality, thereby reducing memory overhead.

Among the modularized DLLs, there are some DLLs essential for use in the Windows environment. KERNEL32.DLL, USER32.DLL, and GDI32.DLL are among the DLLs that must be included in the application development process.

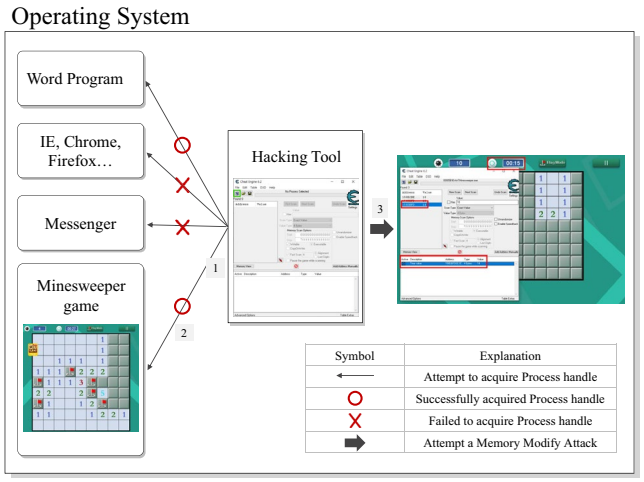

FIGURE 1 Memory forgery and alteration process flow: (1) Handle permission of the target process is obtained, (2) properties and values used in the game are examined, and (3) properties and values are changed

KERNEL32.DLL is a DLL that provides functions to control memory, processes, and threads. USER32.DLL is a DLL used to control user interfaces (UIs). If a UI exists in an application, USER32.DLL must always be used. Finally, GDI32.DLL is used to draw graphic images and display text. Similar to USER32.DLL, if a UI, text, or image is used, GDI32.DLL must be included in the application. DLLs are categorized into import functions and export functions. Import functions are used only inside a DLL and cannot be used in external applications, whereas export functions can be used externally.

2.3.1 | Load-time dynamic linking

One of the important processes when an application runs is to import a list of DLLs for use. Because the application does not contain all the code in the DLL, the DLL must exist in the specified path. Otherwise, the application will not run. When Windows loads a DLL, it allocates a virtual memory space and maps the DLL to the allocated memory.

The section part of an application program shows the DLL information that is mapped to the memory. When the memory mapping process is completed, the application's name and memory address can be verified inside the application. Additionally, if an application error occurs, an environment can be used to analyze the source of the error.

2.3.2 | Run-time dynamic linking

This process explicitly loads the required DLL and then calls it, if there is a desired symbol, when the application runs. A

TABLE 1 Online game security solutions and security features

Solution	Security features
GameGuard	Server Authentication: prevents bypass by game server and GameGuard mutual authentication. Hack Tool Blocker: blocks hacking tools using game auto macros, mouse, and graphics. File forgery and manipulation prevention: file comparison analysis to prevent game file forgery and alteration, malicious debugging, and disassembly prevention. Client Reporting: real-time detection and blocking of process memory tampering. File protection with self-encryption algorithm. Antivirus and Spyware: detects malware using signature algorithms and file format analysis and diagnoses new and modified malware that is difficult to detect with existing antivirus engines. Updates: real-time, regular, and urgent updates. Reports: Real-time errors, hacking tools, and statistical analysis reports
XignCode	Non-client bot detection using "one-time executable code" patent. Block general purpose hacking tool and variant hacking tool using "Win32 API call pattern and frequency" patent. Block illegal users (control game operators). Emergency pattern generation tool provided (game operator control available). Detect and block VPN access (control game operator). DirectX tampering and illegal call detection. Speed nuclear detection. DLL injection detection. Nuking hack and drop nucleus detection. Software and hardware macro detection. Multi-running detection. Game resource tampering detection. Stealth process/module/driver detection. Kernel and user mode debugging detection. Virtual environment execution detection

thread inside the process decides whether to call a function inside the DLL, and the thread loads the DLL into the process' address space and enables the function inside the DLL to be called.

The thread can load the DLL using the following function (LoadLibrary() or LoadLibraryEx()):

```
LoadLibrary() and LoadLibraryEx():
HMODULE LoadLibrary(PCTSTR pszDLLPathName);
HMODULE LoadLibraryEx(PCTSTR pszDLLPathName,
        HANDLE hFile, DWORD dwFlags);
```

The LoadLibrary and LoadLibraryEx functions use a specific search order to identify DLL files on the system. These functions are then used to map the file image of the identified DLL to the address space of the calling process. The virtual memory address to which the file image is mapped is identified when returned from the HMODULE function. When injection occurs, it affects the behavior of the application in memory in a way that the user does not expect or intend. There are two modes of dynamic DLL injection, each of which is performed by an attacker in a series of steps.

A. Remote thread injection

Remote thread injection is used to create a thread in the target process to load a malicious DLL and then call the LoadLibrary function. In this way, LoadLibrary can be used to load malicious DLLs. An attacker cannot easily control a thread from a process that is not created at first; thus, it must create a new thread to access the target process. As a result, creating a thread gives the attacker control over the application. For this purpose, the CreateRemoteThread function is used. Listing 1 is the declaration for the CreateRemoteThread function in Windows [7].

Listing 1 CreateRemoteThread function.

```
HANDLE
CreateRemoteThread(
    HANDLE hProcess,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD fdwCreate,
    PDWORD pdwThreadId);
```

Figure 2 depicts the method that can be used by malware to inject a malicious DLL into other processes. First, the malware opens the process using the OpenProcess function, which returns an open handle that is responsible for checking the process privileges; this handle is used to grant the right access to the target process. Second, the malware allocates memory using the VirtualAllocEx function to specify the correct path for the malicious DLL. Third, it writes the DLL path using the WriteProcessMemory function. Once the path has been created, the malware initiates the CreateRemoteThread function to create a thread on the target process, instructing the thread to load the malicious DLL remotely. As a result, the malware attaches the malicious DLLs on the target process and can compromise critical data on the victim's machine.

B. Windows registry DLL injection

The remote thread injection technique dynamically injects a DLL that contains the attack code for an attacker developing or using an injection program. In contrast, there is a method to force a DLL to be loaded into any program while Windows is running. When using the registry key "AppInit_DLLs" provided by Windows, DLL_PROCESS_ATTACH is called when the User32.dll library is mapped to a newly created process.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\
    Windows NT\CurrentVersion\Windows\
    AppInit_DLLs
```

When the call is processed, User32.dll calls LoadLibrary for each DLL specified in the AppInit_DLLs key. The entire library is loaded, and the DllMain function associated with the library is called with fdwReason set to DLL_PROCESS_ATTACH.

2.4 | Binary code injection

The binary code injection approach is similar to that of DLL injection. In binary code injection, an attacker sets the remote address of the injected binary code segment as the thread routine and creates a remote thread to execute the injected binary code. This method injects into the target process without an additional DLL stored in the system. Considering the steps of binary code injection, this method seems easier than DLL injection. However, the key point is the construction of the binary code.

The injected binary code cannot be initialized as a DLL mapped into the address space of the target process. Although the binary code can be implemented as a function in the malicious process, there are still complications. First, the binary code uses absolute addresses to reference variables or call functions; however, these are the actual addresses in the address space of the malicious process. Thus, the injected binary code cannot retrieve the same data or call the same functions at these addresses in the target process. It would be easier to combine all the subroutines into a large function, unless an injection into each subroutine and a pass of their remote addresses to the injected binary code are desired. A further complication is that the binary code is relocated in the malicious process. The target process does not relocate the injected binary code. Most commonly, operators crash the target process when a remote thread is created to execute the injected binary code.

Inline assembly can be used to construct the binary code. Although this is considerably complicated to implement, there is no need to manage the relocation of the injected binary code. Furthermore, the binary code can be much more easily controlled and is more reliable if assembly language is used to implement the whole malicious process. Binary code injection is much more complicated and riskier than DLL injection. Additionally, it provides the flexibility to inject without an additional DLL.

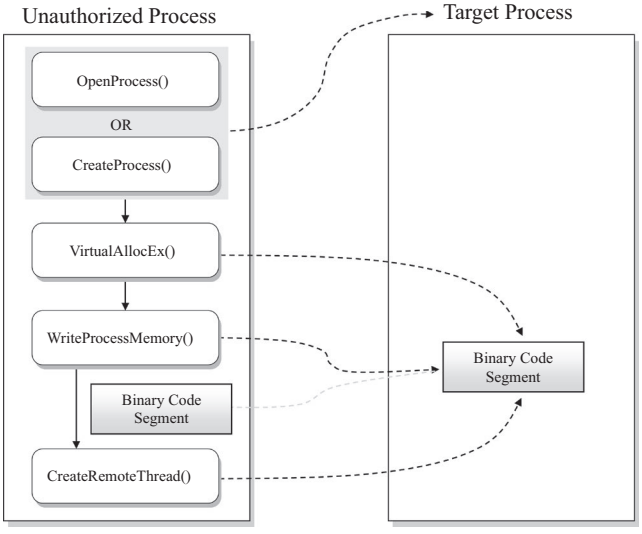

FIGURE 2 Remote thread injection process flow

2.5 | Related work

Only the client's environment was detected by monitoring through hooking in previous studies [8–10]. These detection methods are necessary to enable the monitoring of hooking on the client side and its bypassing by blocking the hooking. Furthermore, the false-positive rate of hooking increases because each PC environment is different. If the client can identify the server-side environment and the user-specific PC environment, detection is expected to be more effective than in previously studied methods.

In [11], Jelena and others categorized detection methods for malware into static and dynamic analysis, and the authors focused on the dynamic method. They identified malware families using memory and CPU usage. Moreover, to detect malware in mobile devices, features extracted from CPU and memory were tested.

Zhixing and others [12] suggested a framework for detecting malware using monitoring and classification of memory access pattern with a machine learning technique. The authors checked for malicious code that attempted to change the control flow and access the program memory. Additionally, they used a machine learning techniques to analyze various rootkits, such as avg-coder and AFkit, which modify the system call table.

Rami and others [13] investigated various memory analysis methods, such as static, dynamic, and hybrid techniques. Memory analysis is a state-of-the-art technique and is widely using in the forensic field [14,15]. In the survey paper, many researchers use API function calls to detect malware. Furthermore, they analyzed memory using machine learning techniques, such as support vector machines and K-nearest neighbors.

Existing papers have proposed static, dynamic, and hybrid methods, as well as the use of machine learning to detect malware in user PCs. However, our study focuses on detecting models for each service, and our method is not a dynamic detection method.

3 | PROPOSED METHOD

We propose the game hack module detection method, as depicted in Figure 3. Our proposed method is a three-step process. The first step is the collection of information from each PC. The second step is the separation of the collected information according to OS and version. The third step is the analysis of the separated memory information.

3.1 | Information collection stage

First, we collect information from online game clients. With the knowledge of the OS and service pack being used, it can be determined as to whether a hacking tool is running on the same base. Information regarding the memory of the environment in which the game is played can be obtained after collecting information about the memory, and the DLL information employed by the game clients can be used to transfer the collected information to the server. Figure 4 depicts a graphical representation of the collection process.

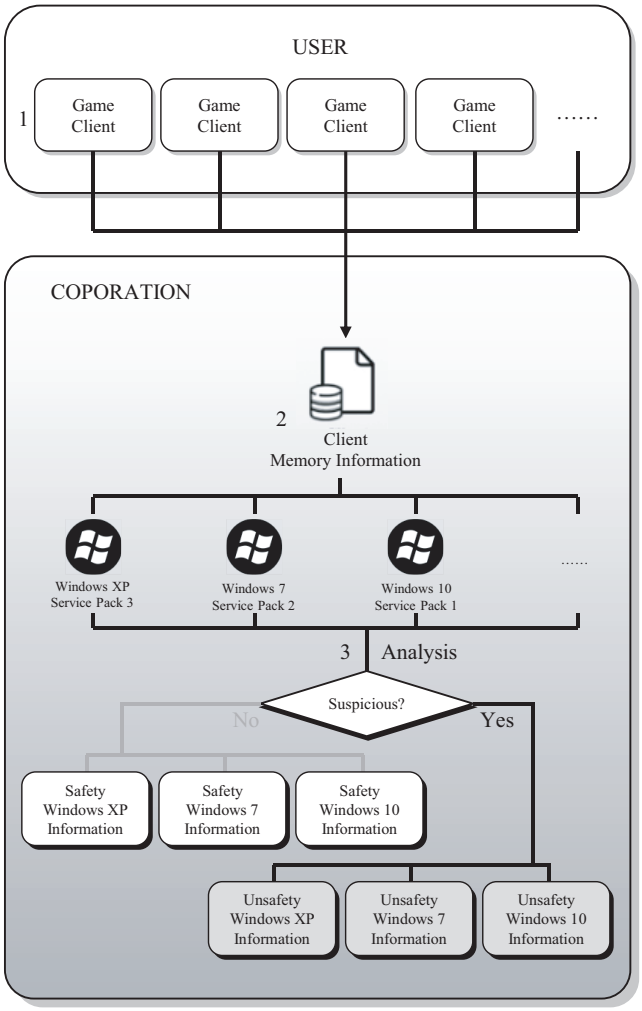

FIGURE 3 Game hack module detection method

Information collected from a user PC is classified based on the OS and OS version through the next step, the separation step. There are two methods for collecting information.

The first method of information collection is to check the registered DLL information on the client and the memory state. This method blocks any DLL that is not in the corresponding loading list. However, a simple file name lookup is easy to bypass; thus, the cyclic redundancy check (CRC) logic must have a vast CRC table of DLLs. This is because if Windows is updated, a patching process occurs, and the DLL list must be managed by the version of Windows, making it inconvenient to manage.

The second method that we propose is the collection of a memory state list. If a program changes into a process environment, various changing areas of memory exist, including available and unavailable areas, readable areas, and writable areas. Monitoring and managing these memory areas can effectively identify an invasive hacking module. Listing 2 presents code that verifies the memory protection constant for memory scanning from a hacking tool or module. Because the attributes of such accessible memory are limited, monitoring is possible.

3.2 | Analysis stage

Malware uses APIs that are different from the APIs used by non-malware, and memory is also used differently [16]. An API

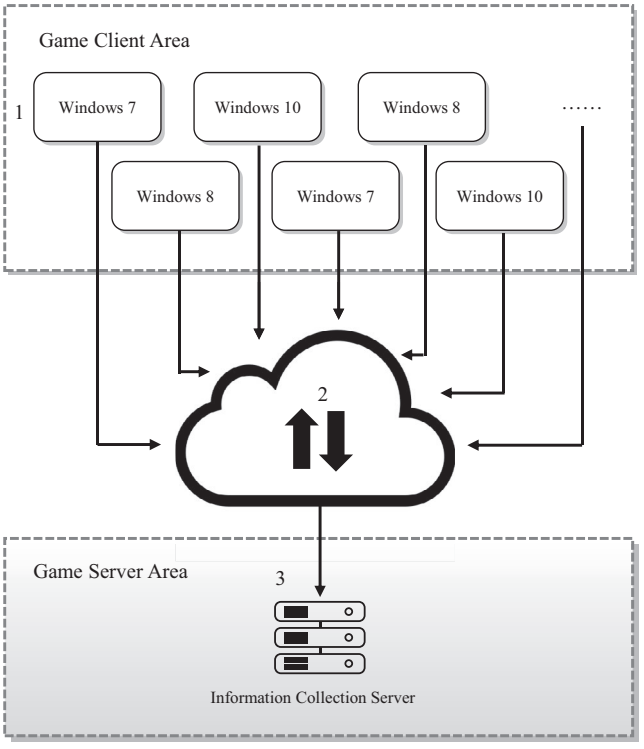

FIGURE 4 Information collection stage: (1) OS and version information for each client is collected, (2) collected information is transferred to the server, and (3) statistical operations on the monitoring server are performed

Listing 2 Checking memory protection constant for memory scanning.

```
if(hProc)
{
    while (1)
    {
        if(VirtualQueryEx(hProc, addr, &
            meminfo, sizeof(meminfo)) == 0)
        {
            break;
        }
#define WRITABLE (PAGE_READWRITE
    PAGE_WRITECOPY | PAGE_EXECUTE_READWRITE
    | PAGE_EXECUTE_WRITECOPY)
        if ((meminfo.State & MEM_COMMIT) &&
            (meminfo.Protect & WRITABLE))
        {
            MEMBLOCK *mb = create_memblock(
                hProc, &meminfo);
            if (mb)
            {
                mb->next = mb_list;
                mb_list = mb;
            }
        }
        addr = (unsigned char*)meminfo.
            BaseAddress + meminfo.RegionSize
            ;
    }
}
else
    printf("Failed to open process - error -
        %d\r\rn", error);
```

used by malware has a different model from an API used by non-malware. Therefore, when analyzing APIs that are not used by non-malware, it is necessary to load the DLLs that provide APIs in memory. Similarly, game hacking programs must use the DLLs and APIs that are not used by the online game client, which adds unused memory areas, resulting in a structure that is different from the normal online game client memory map.

In the collection and separation of information steps, a check is performed to confirm that the memory map structure is in place according to each OS and version. As presented in Table 2, by using each address's count of the memory, a different user is selected from the existing client environment. Assuming that 10 users play the game, it is highly likely that hack module has penetrated if 1-4 new memory areas are added among those 10 users. Therefore, users of unusual environments are judged using the hack module through a game history check. However, it is problematic if previously unused memory areas exist, because of the definition of memory usage provided by Windows.

A detailed description of the memory access and usage is provided by Microsoft MSDN. As presented in Table 3, according to the memory protection constant, various rights are granted, including accessible memory addresses, inaccessible addresses, accessible but only readable addresses, and readable but not writable addresses. Therefore, a hacking tool and hacking module first check the memory where PAGE_READWRITE, PAGE_WRITECOPY, PAGE_EXECUTE_READWRITE, and PAGE_EXECUTE_WRITECOPY privileges exist. Therefore, we monitor the constants used in the hacking tool. Figure 5 depicts the result, based on a study of the state by memory addresses in the actual process.

The availability in the protection constant state is defined in Table 4. As presented in Table 4, the "MEM_FREE" state denotes an unused free area, and "MEM_RESERVE" denotes an area where only addresses are assigned and where the "MEM_COMMIT" virtual memory and physical memory are used. Accordingly, hacking tools can inject code in the "MEM_FREE" and "MEM_COMMIT" states. Thus, monitoring mainly proceeds for "MEM_FREE" and "MEM_COMMIT."

There is a memory area set to "MEM_PRIVATE" although it is set to "MEM_COMMIT" in Figure 6. Monitoring needs to proceed without the "MEM_PRIVATE" area because the "MEM_PRIVATE" area is blocked from the access of other processes. If the above conditions are met, the memory map can be verified, as depicted in Figure 7.

3.3 | Memory hacking protection through monitoring

The entire monitoring process is illustrated in Figure 8.

We collect the memory addresses for each country, OS, and service version. If there is a suspicious user based on the monitored memory status, an investigation of that user is initiated. Each investigated memory address

TABLE 2 Analysis table for collected data (Windows 7 Enterprise Service Pack 1)

Address	State	Constant	Count
00000000-00010000	MEM_FREE	-	10
00010000-00020000	MEM_COMMIT	PAGE_READWRITE	10
…	…	…	
00121000-0013000	MEM_FREE	PAGE_READWRITE	9
777D000-777D1000	MEM_COMMIT	PAGE_EXECUTE_READ	1
…	…	…	
7EFD8000-7EFDB000	MEM_COMMIT	PAGE_READWRITE	10

TABLE 3 Memory protection constants [17]

Constant	Value	Monitoring	Description
PAGE_EXECUTE	0 × 10	X	Enables execute access to the committed region of pages. An attempt to write to the committed region results in an access violation. This flag is not supported by the CreateFileMapping function
PAGE_EXECUTE_READ	0 × 20	X	Enables execute or read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation. Windows Server 2003 and Windows XP: This attribute is not supported by the CreateFileMapping function until Windows XP with SP2 and Windows Server 2003 with SP1
PAGE_EXECUTE_READWRTIE	0 × 40	O	Enables execute, read-only, or read/write access to the committed region of pages. Windows Server 2003 and Windows XP: This attribute is not supported by the CreateFileMapping function until Windows XP with SP2 and Windows Server 2003 with SP1
PAGE_EXECUTE_WRITECOPY	0 × 80	O	Enables execute, read-only, or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being made for the process. The private page is marked as PAGE_EXECUTE_READWRITE, and the change is written to the new page. This flag is not supported by the VirtualAlloc or VirtualAllocEx functions. Windows Vista, Windows Server 2003, and Windows XP: This attribute is not supported by the CreateFileMapping function until Windows Vista with SP1 and Windows Server 2008
PAGE_NOACCESS	0 × 01	X	Disables all access to the committed region of pages. An attempt to read from, write to, or execute the committed region results in an access violation. This flag is not supported by the CreateFileMapping function
PAGE_READONLY	0 × 02	X	Enables read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation. If Data Execution Prevention is enabled, an attempt to execute code in the committed region results in an access violation
PAGE_READWRITE	0 × 04	O	Enables read-only or read/write access to the committed region of pages. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation
PAGE_WRITECOPY	0 × 08	O	Enables read-only or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being made for the process. The private page is marked as PAGE_READWRITE, and the change is written to the new page. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation. This flag is not supported by the VirtualAlloc or VirtualAllocEx functions

is transferred to the server and stored. Then, we proceed with statistics regarding the memory addresses that have been recorded, and memory addresses with low counts are investigated.

The reason for selecting the monitoring item is that the WinAPI used in a DLL and code injection process calls the VirtualAllocEx API internally to reserve and confirm memory usage, as shown in Listing 3.

Listing 3 VirtualAllocEx function.

```
LPVOID VirtualAllocEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

In this process, malware sets the state value for those marked "O" in the monitoring columns, as presented in Table 3 and Table 4. It is an API that is used to inject code

```
Enter the pid: 14600
Address  Size(bytes)  Status
00000000 00010000 MEM_FREE
00010000 00010000 MEM_COMMIT
00020000 00010000 MEM_COMMIT
00030000 00001000 MEM_COMMIT
00031000 00001000 MEM_COMMIT
00032000 00001000 MEM_COMMIT
00033000 0000d000 MEM_FREE
00040000 00001000 MEM_COMMIT
00041000 0000f000 MEM_FREE
00050000 00004000 MEM_COMMIT
00054000 0000c000 MEM_FREE
00060000 00001000 MEM_COMMIT
00061000 0000f000 MEM_FREE
00070000 00001000 MEM_COMMIT
00071000 0000f000 MEM_FREE
```

FIGURE 5 Memory state in the actual process

TABLE 4 The state of the pages in the region [18]

State	Value	Monitoring	Description
MEM_COMMIT	0 × 1000	O	Indicates committed pages for which physical storage has been allocated, either in memory or in the paging file on disk.
MEM_FREE	0 × 10 000	O	Indicates free pages not accessible to the calling process and available to be allocated. For free pages, the information in the AllocationBase, AllocationProtect, Protect, and Type members is undefined.
MEM_RESERVE	0 × 2000	O	Indicates reserved pages where a range of the process's virtual address space is reserved without any physical storage being allocated. For reserved pages, the information in the Protect member is undefined.
MEM_IMAGE	0 × 1 000 000	X	Indicates that the memory pages within the region are mapped into the view of an image section.
MEM_MAPPED	0 × 40 000	X	Indicates that the memory pages within the region are mapped into the view of a section.
MEM_PRIVATE	0 × 20 000	X	Indicates that the memory pages within the region are private (ie, not shared by other processes).

```
Enter the pid: 14600
Address  Size(bytes)  Status
00000000 00010000 MEM_FREE
00010000 00010000 MEM_COMMIT MEM_MAPPED
00020000 00010000 MEM_COMMIT MEM_MAPPED
00030000 00001000 MEM_COMMIT MEM_PRIVATE
00031000 0000f000 MEM_FREE
00040000 00001000 MEM_COMMIT MEM_IMAGE
00041000 0000f000 MEM_FREE
00050000 00004000 MEM_COMMIT MEM_MAPPED
00054000 0000c000 MEM_FREE
00060000 00039000 MEM_RESERVE MEM_PRIVATE
00099000 00003000 MEM_COMMIT MEM_PRIVATE
0009C000 00004000 MEM_COMMIT MEM_PRIVATE
000A0000 00001000 MEM_COMMIT MEM_MAPPED
000A1000 0000f000 MEM_FREE
000B0000 00001000 MEM_COMMIT MEM_PRIVATE
000B1000 0000f000 MEM_FREE
000C0000 00001000 MEM_COMMIT MEM_PRIVATE
000C1000 0000f000 MEM_FREE
000D0000 0000c000 MEM_COMMIT MEM_PRIVATE
000DC000 00004000 MEM_FREE
```

FIGURE 6 Protection constants inside the process

```
Enter the pid: 14600
Address  Size(bytes)  Status
00000000 00010000 MEM_FREE
00010000 00010000 MEM_COMMIT PAGE_READWRITE
00020000 00010000 MEM_COMMIT PAGE_READWRITE
00030000 00001000 MEM_COMMIT PAGE_READWRITE
00031000 0000f000 MEM_FREE
00040000 00001000 MEM_COMMIT
00041000 0000f000 MEM_FREE
00050000 00004000 MEM_COMMIT
00054000 0000c000 MEM_FREE
00060000 00001000 MEM_COMMIT
00061000 0000f000 MEM_FREE
00070000 00001000 MEM_COMMIT PAGE_READWRITE
00071000 0000f000 MEM_FREE
00080000 000fc000
0017C000 00001000 MEM_COMMIT PAGE_READWRITE
0017D000 00003000 MEM_COMMIT PAGE_READWRITE
```

FIGURE 7 Memory to monitor

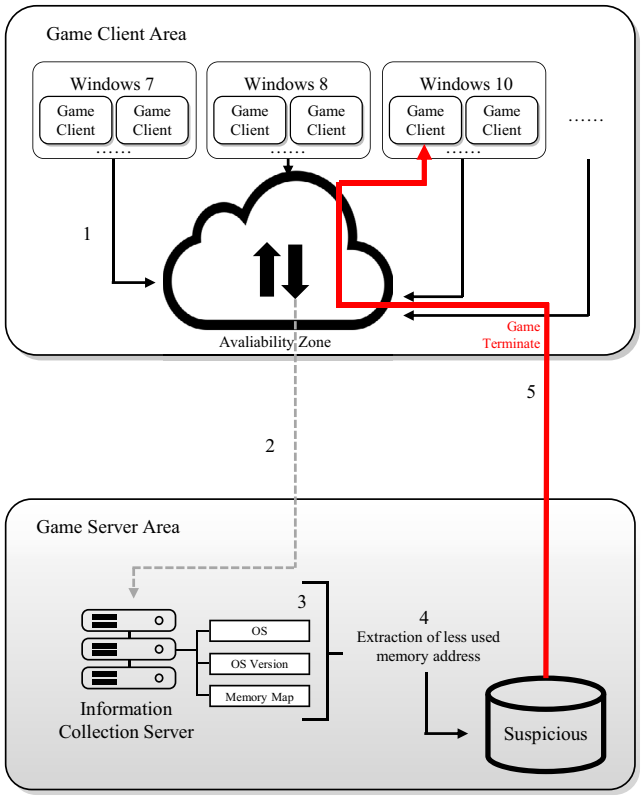

FIGURE 8 Monitoring stage: (1) information regarding the OS, version, and memory map from each client is collected, (2) collected information is delivered to the game servers, (3) collected information is separated, (4) memory map with low frequency is selected from separate information, and (5) the corresponding user is prohibited from using it

into other processes and malware; thus, when monitoring the value of the set argument, external processes can check the code insertion after access. We select memory addresses that have read and write permissions for monitoring. The reason is that online hacking tools and malware need space

to read and write their modules into memory. If malware repeats writing or reading from other memory, it causes the program to crash and either hang or produce the "blue screen of death." Therefore, an online game hacking tool is programmed to secure and operate the weakness to be used in the tool's module.

4 | EXPERIMENTAL RESULT

We strove to identify a structure that can collect information from existing client environments and manage and detect the information detected from a central server. Because the memory structure of each service is known to the game developer, it is possible to check the used and unused areas in the situation wherein the service is running. Additionally, because used and unused modules can also be checked, our experiment is expected to reduce the number of false negatives occurring in the process of detecting existing unknown malware.

The experiment was conducted under the following conditions.

1. The Client.exe program on a Windows PC was run, and the resulting memory map was transferred to the server.
2. The memory map monitored by a DLL injection of the virtual hacking tool was checked.

4.1 | Collecting a memory map on a Windows PC

During the collection of the memory maps, we observed that different addresses were assigned each time a specific Client.exe was running. However, because all DLL lists are assigned under the same conditions, it is possible to detect DLL injection, as depicted in Figure 9. Additionally, when comparing the memory structures of PC A and PC B, differences in the DLL region can be found, as depicted in Figure 10.

In this situation, as depicted in Figure 11, Microsoft was unable to maintain the memory map with the same address and size because the security level of the address space layout randomization was increased so that the memory map could not be used in the same environment. Each PC's area is unique; thus, monitoring is not possible under the same conditions. Therefore, we can monitor based on the user and check whether DLL injection has been added.

4.2 | Collecting the memory map of the user

As the result of Section 4.1 shows, monitoring of the area where a certain DLL list exists can be performed only in the unchanged section. As a result of testing, all the user-defined areas were variable areas used by the application to be protected. The more the variables that are used, the more the likely it is for them to change. Therefore, malicious changes can be effectively judged when monitoring the area that uses DLLs outside the areas used by the game application.

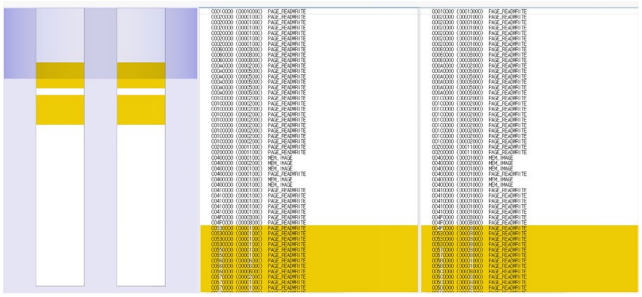

FIGURE 9 Detecting DLL injection

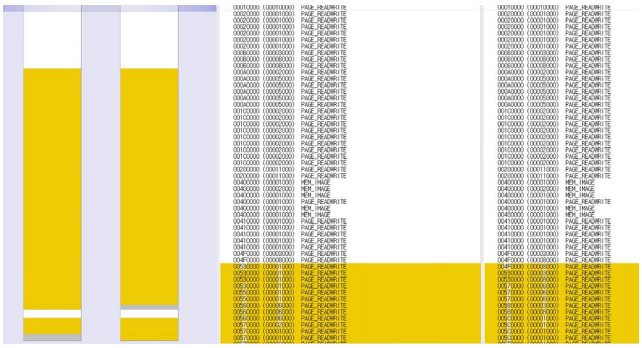

FIGURE 10 Differences in DLL regions

Therefore, we created a memory map for each user and counted the memory addresses used, as presented in Table 5. Then, when a different type of domain was added or changed, it was detected as a DLL injection.

4.3 | Overhead for determining abnormal users

We prepared the following formula to measure the overhead for our proposed methodology.

$$\mathrm{OT} = \frac{J_c}{D_{ac}}.$$

The description of each factor is as follows.

OT = overhead time.

J_c = number of executions required for decision.

D_{ac} = average number of game runs per day.

In order to improve data reliability, 400 cases are needed [19]. Thus, the time it takes for one user to execute 400 cases is 40 days when running 10 times a day on average [20].

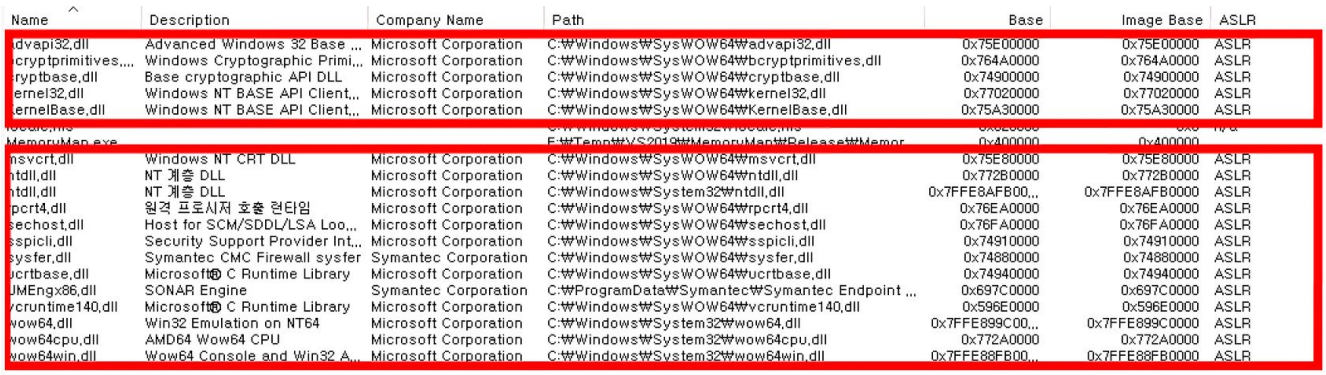

FIGURE 11 Address space layout randomization of DLLs

TABLE 5 Create memory map by user

User	Item	Normal	Include hack module	Description
A	Memory address	00000000-7FFF0000	00000000-7FFF0000	No difference
	Memory count	262	286	Hack module used more
	MEM_COMMIT	172	187	Hack module used more
	MEM_RESERVE	35	41	Hack module used more
	MEM_FREE	54	58	Hack module used more
B	Memory address	00000000-7FFF0000	00000000-7FFF0000	No difference
	Memory Count	262	262	
	MEM_COMMIT	172	172	
	MEM_RESERVE	35	35	
	MEM_FREE	54	54	

Since this is an average value, it is possible to judge an abnormal user, even if it is a high usage user, in 40 days or less.

5 | CONCLUSION

We aimed to obtain the same type of memory map in the same OS and the same country; however, it was confirmed that the injected DLL could be detected when acquiring information by each user monitoring the memory map.

Because the memory area used by an application varies from time to time, it is extremely difficult to monitor the area. Therefore, this approach to securing online gaming requires further study. Additionally, a continuous study of the items and methods that can be monitored for each Windows OS needs to be conducted, rather than focusing on the insurmountable task of monitoring each user.

ORCID

Kyounggon Kim https://orcid.org/0000-0002-5675-4253

REFERENCES

1. H. Kwon et al., *Crime scene reconstruction: Online gold farming network analysis*, IEEE Trans. Inf. Forensics Secur. **12** (2016), 544–556.
2. E. Lee et al., *You are a game bot!: Uncovering game bots in MMORPGs via self-similarity in the wild*, in Proc. NDSS (San Diego, CA, USA), Feb. 2016, doi: 10.14722/ndss.2016.23436
3. H. Kim, S. Yang, and H. K. Kim, *Crime scene re-investigation: A postmortem analysis of game account stealers' behaviors*, in Proc. Annu. Workshop Netw. Syst. Support Game (Taipei, Taiwan), June 2017, pp. 1–6.
4. J. Woo, H. J. Choi, and H. K. Kim, *An automatic and proactive identity theft detection model in mmorpgs*, Appl. Math **6** (2012), 291–302.
5. H. B. Jang, K. G. Kim, and S. J. Lee, *A study on technical countermeasures according to game service breach types*, Inf. Systems Rev. **9** (2007), 83–98.
6. Microsoft, *Dynamic-link libraries*, 2011, Available from: https://docs.microsoft.com/ko-kr/windows/win32/dlls/dynamic-link-libraries?redirectedfrom=MSDN [last accessed March 2020]
7. Microsoft, *Createremotethread function*, 2018, Available from: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread [last accessed March 2020]
8. M. Jang, H. Kim, and Y. Yun, *Detection of DLL inserted by windows malicious code*, in Proc. Int. Conf. Convergence Inf. Technol. (Gyeongju, Rep. of Korea), Nov. 2007, pp. 1059–1064.
9. W.-C. Feng, E. Kaiser, and T. Schluessler, *Stealth measurements for cheat detection in on-line games*, in Proc. ACM SIGCOMM Workshop Netw. Syst. Support Games (Worcester, MA, USA), Oct. 2008, pp. 15–20.
10. F. Desheng, S. Zhou, and C. Cao, *A windows rootkit detection method based on cross-view*, in Proc. Int. Conf. E-Product E-Service E-Entertainment (Henan, Chian), 2010, pp. 1–3.

11. J. Milosevic, A. Ferrante, and M. Malek, *What does the memory say? Towards the most indicative features for efficient malware detection*, in Proc. IEEE Annu. Consumer Commun. Netw. Conf. (Las Vegas, NV, USA), 2016, pp. 759–764.
12. X. Zhixing et al., *Malware detection using machine learning based analysis of virtual memory access patterns*, Design, Autom. Test Eur. Conf. Exhibition (Lausanne, Switzerland), 2017, pp. 169–174.
13. R. Sihwail, K. Omar, and K. A. Z. Ariffin, *A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis*, Int. J. Adv. Sci., Eng. Inf. Technol. **8** (2018), no. 4-2, 1662–1671.
14. R. Mosli et al., *Automated malware detection using artifacts in forensic memory images*, in Proc. IEEE Symp. Technol. Homeland Security (Waltham, MA, USA), May 2016, pp. 1–6.
15. R. Mosli, *A behavior-based approach for malware detection*, in Proc. IFIP Int. Conf. Digital Forensics (Orlando, FL, USA), 2017, pp. 187–201.
16. M. Wagner et al., *A survey of visualization systems for malware analysis*, in Proc. Eurographics Conf. Visualization (Cagliari, Italy), 2015, pp. 105–125.
17. Microsoft, *Memory protection constants*, 2018, Available from: https://docs.microsoft.com/en-us/windows/desktop/memory/memory-protection-constants [last accessed January 2020].
18. Microsoft, *Memory_basic_information structure*, 2018, Available from: https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-memory_basic_information [last accessed January 2020].
19. L. Al-Haddad, C. W. Morris, and L. Boddy, *Training radial basis function neural networks: Effects of training set size and imbalanced training sets*, J. Microbiol. Methods **43** (2000), 33–44.
20. M. Hong and H.-M. Lee, *A study on characteristics of serious game user through implementation of mobile sequence game*, The KIPS Transactions: Part A **19** (2012), no. 3, 155–160.

AUTHOR BIOGRAPHIES

Chang Seon Lee received his BS degree in computer science from Academic Credit Bank System in 2009, MS degree in information security from Chonnam National University in 2013, and PhD at the Department of Business Administration, Sangmyung University, in 2018. He began studying security in 2000 and has won a number of CTFs. After serving in the Navy Cyber Investigation Service, he became interested in client security. Since 2006, he has offered consulting services through a company, and since 2008, he has worked on client security and abuser detection. He is currently working on system design, analysis, and monitoring to detect signs of abuse and fraud for message, mobile game, payment, virtual currency, and financial services at LINE Corporation. He has also published introduction to college textbooks on computer security and forensics. His research interests include money laundering detection and analysis, monitoring system design, solving security problems in financial computing based on user behavior analysis and data mining, and malware analysis.

Huy Kang Kim received his BS degree in industrial management and MS degree and PhD in industrial and systems engineering from KAIST in 1998, 2000, and 2009, respectively. He founded A3 Security Consulting, the first information security consulting company in South Korea in 1999. He was Technical Director and Head of the Information Security Department of NCSOFT from 2004 to 2010. He is currently Associate Professor at the Graduate School of Information Security, Korea University. His research interests include solving security problems in online games based on user behavior analysis and data mining.

Hey Rin Won received her BS degree in Information Security from Seoul Women's University in 2020. She was awarded the Best of the Best—Next Generation Security Leader Award, which was sponsored by Korea Information Technology Research Institute. Her research interests include vulnerability analysis and security systems using artificial intelligence.

Kyounggon Kim received his BS degree in computer science from Soongsil University in 2008, and MS degree and PhD in information security from Korea University in 2015 and 2020, respectively. He is currently an Assistant Professor at the Department of Forensic Sciences, Naif Arab University for Security and Sciences (NAUSS). He has performed penetration testing for over 130 clients in various industries when he worked for Deloitte, PwC, and boutique consulting firms during over 15 years. He was awarded 6th place at DefCon CTF in 2007 and a first prize at the First Hacking Defense Contest hosted by the Korea Information Security Agency. He has authored a book on Internet hacking and security and has translated numerous security books. His research interests include vulnerability analysis, smart city security, and CPS and IoT security.