A Cheating Attack on a Whitelist-based Anti-Ransomware Solution and its Countermeasure*

*A WAR countermeasure against a DLL injection attack

Se-Beom Cheon
Dept. of Information Security
Suwon Univ.
Hawsung-Si, South Korea
chsg03150@suwon.ac.kr

Geun-Yeong Choi
Dept. of Cyber Security
Korea Univ. Graduate School
Seoul, South Korea
geunyeong@korea.ac.kr

DaeYoub Kim
Dept of Information Security
Suwon Univ.
Hawsung-Si, South Korea
daeyoub69@gmail.com

***Abstract*—When proposing a whitelist-based anti-ransomware solution (WAR), it was pointed out that WAR could be vulnerable to a dynamic-link library (DLL) injection attack: An attacker can cheat the WAR using DLL injection techniques so that it can access files for which it does not have access rights. Therefore, if such an argument may be valid, it is essential to improve the vulnerability to utilize WAR securely. This paper first analyzes whether the DLL injection technique causes the vulnerability of WAR. Then, it proposes a solution to prevent a DLL injection attack on WAR.**

Keywords—ransomware, access control, whitelist, Windows OS, DLL injection

I. Introduction

Ransomware is a critically serious threat to various devices connected to a network. As attackers utilize it for economic or political purposes, it causes increasingly severe damage to many organizations and personal users [1]. Many anti-ransomware solutions utilize the code signatures of analyzed ransomwares, similar to traditional anti-malware solutions. However, such solutions cannot detect unknown malware that has never been examined before. Hence, until someone collects and analyzes such a new malware and reports the analyzed result, the new malware can continue to damage systems. Also, the solutions generally compare the analyzed code signatures with many files/procedures of a system. So, the solutions can severely burden the system. Other approaches utilize malware activity monitoring or characteristic analysis to improve such a weakness. However, since they also monitor many processes of a system in real time, they cannot solve the second weakness [2].

Another approach utilizes a whitelist of programs/processes with proper access rights for a system [3][4]. In [3], the proposed ransomware prevention scheme implements both a file access monitor module as a kernel level module and an access control module as a user level module. If a program/process tries to access a file, the monitor module intercepts the I/O request packet (IRP) generated by the I/O manager before a file system driver. The IRP includes various information describing the program/process, the file, and so on. After sending the IRP to the access control module, the monitor module waits for the response of the access control module. If the access control module judges that the IRP is not an acceptable access request based on its whitelist, the monitor module removes the IRP so that the program/procedure cannot access the file. Otherwise, the monitor module sends the IRP to the file system driver, which is the original receiver of the IRP, so that the program/process can access the file. [4] also control suspicious access to the folders and/or files of a protected system.

This research was funded by NFR of Korea, grant number NRF-2021R1F1A1062954.

However, a whitelist-based access control scheme such as [3] and [4] has a weakness: If an unauthorized object can impersonate as an authorized object, the unauthorized object can bypass an access control mechanism of a target system and access the target system. In Windows OS, an attacker abuses a dynamic-link library (DLL) injection skill for such impersonation. Both [3] and [4] may be vulnerable to a DLL injection attack [5][6][7][8]. Using DLL injection, the attack can bypass the access control system and illegally access the resources of the target system. Hence, a whitelist-based access control scheme must efficiently prevent a DLL injection attack on authorized processes of a target system. This paper proposes a detection mechanism for DLL injection attacks and improves a whitelist-based access control scheme [3] using the proposed detection skill.

II. A Whitelist-based Anti-Ransomware Solution

A. *Overview of a Whitelist-based Anti-Ransomware*

A whitelist-based anti-ransomware solution (WAR) presented in [3] proposes controlling processes' file access activities on a system operated with Windows OS using a whitelist. WAR implements a file access monitoring module with a Windows kernel level to monitor file access requests of processes [9]. WAR also implements an access control module with a Windows user level. The main roles of the access control module are as follows: First, it manages the whitelist consisting of a pair of both a file type and a list of programs having access rights for the file type. Second, it communicates with the file access monitoring module using a socket and analyzes the IRP using the whitelist.

Fig. 1 describes the WAR procedure presented in [3]: (B1) The file access monitoring manager module (FAM) intercepts the IRP when a process tries to access a file in a system. (B2) The access control manager module (ACM) receives the IRP intercepted by the FAM. (B3) The ACM analyzes the IRP and compares the IRP with its whitelist. (B4) The ACM judges the access right requested with the IRP. If the process and file information recorded in the IRP is acceptable enough when judged based on a whitelist, it considers the IRP authorized. Otherwise, it must deny the IRP. (B5) The ACM sends the judgment to the FAM. (B6) The FAM handles the IRP according to the judgment of the ACM. If the ACM notifies that the IRP is acceptable, the FAM sends the IRP to a file system driver, which is the original receiver of the IRP, so that the system can handle the IRP normally. In this case, the process trying to access the file can utilize the file. If it notifies that the IRP must be denied, the FAM deletes the IRP. Then, the process trying to access the file cannot access the file.

B. *DLL Injection Attack*

A DLL injection attack tries to inject third-party code into a running process by loading a third-party dynamic library.

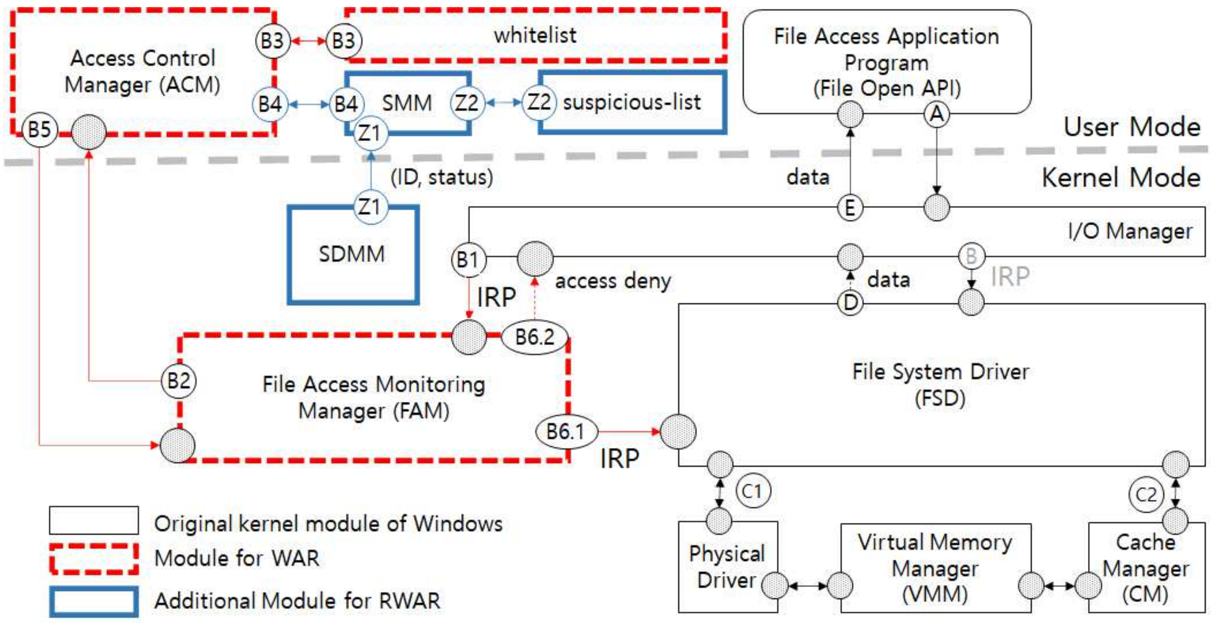


Fig. 1. The procedure of WAR and RWAR. SDMM monitors suspicious threads and reports them to ACM/SMM in order to prevent DLL injection attack.

For that, an attacker generally creates a new thread in an active process of the target application and adds its own DLL to the active process.

A cheating attack on a whitelist-based anti-ransomware solution injects a malicious DLL, including attack code, into an authorized process with access rights for files in a target system. Using CreateRemoteThread and LoadLibraryA (or LoadLibraryW), Win32 API functions provided by Windows, an attacker can inject the malicious DLL into a target process. Assume that an attacker injects a malicious DLL including attack code for encrypting files in a target system into general applications such as internet browsers and word processors, which general users widely utilize to access files. Then, WAR cannot prevent the attack code from encrypting the files because the attack code looks like a part of an authorized application recorded in the whitelist.

III. The Prevention of DLL Injection Attacks

A. *Monitering Suspicious DLL Injection*

This section proposes a DLL injection monitor module with a Windows kernel level. Windows OS provides useful system APIs to monitor active processes/threads as follows.

(1) *PsSetCreateThreadNotifyRoutine* routine registers a driver-supplied callback that is subsequently notified when a new thread is created and when such a thread is deleted. This paper calls it a thread notify callback routine. Its input parameter has a callback pointer to the driver's implementation of such thread notify callback routines. This thread notify callback routine is implemented by a driver to notify the caller when a thread is created or deleted. *ThreadId* is the thread ID of the created/deleted thread. *ProcessId* is the process ID of the process having the thread.

(2) *PsGetCurrentProcessId* and *PsGetCurrentThreadId* routines identify the current process and thread, respectively. Especially if the thread notify callback routine internally calls these routines, they return the creator's process/thread ID of the thread having caused the callback routine.

Using such routines, it is possible to check the IDs of active processes and threads of a target system. Table 1 shows the results to read such IDs according to three cases:

(1) A process (P[a]) creates another process (P[b]). In this case, T[b]* is a main thread of the process P[b].

(2) A process (P[a]) creates a thread (T[a]) internally.

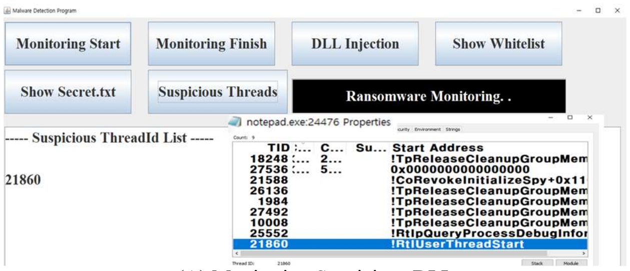
(3) A process (P[a]) creates a thread (T[b]) in the virtual address space of another process (P[b]).

In Table 1, the bPID and the bTID are process and thread IDs that are the input parameters of the callback routine. The cPID and the cTID are the return values of *PsGetCurrentProcessId* and *PsGetCurrentThreadId* when these two routines are called in the thread notify callback routine. $ID_{x[y]}$ denotes the ID of process x[y] (or thread x[y]).

As shown in Case 3 of Table 1, when a process (P[a]) creates a thread (T[b]) of the virtual address space of another process (P[b]), bPID ≠cPID and bTID≠cTID. Calling the *CreateRemoteThread* API function results in the same result as in Case 3. Hence, if the thread notify callback routine includes a subroutine comparing bPID/bTID with cPID/cTID, it can select a suspicious thread suspected of having been created during the DLL injection attack. Although Case 1 looks like causing the same result as Case 3, in Case 1, a process P[b] has only one thread (i.e., the main thread). Hence, the subroutine can distinguish Case 1 from Case 3.

TABLE I. Comparison of Process/Thread IDs

IDs	Case 1	Case 2	Case 3
	P[a]:P[b]	P[a]:T[a]	P[a]:T[b]
bPID	$ID_{P[b]}$	$ID_{P[a]}$	$ID_{P[b]}$
bTID	$ID_{T[b]*}$	$ID_{T[a]}$	$ID_{T[b]}$
cPID	$ID_{P[a]}$	$ID_{P[a]}$	$ID_{P[a]}$
cTID	$ID_{T[a]}$	$ID_{T[a]}$	$ID_{T[a]}$

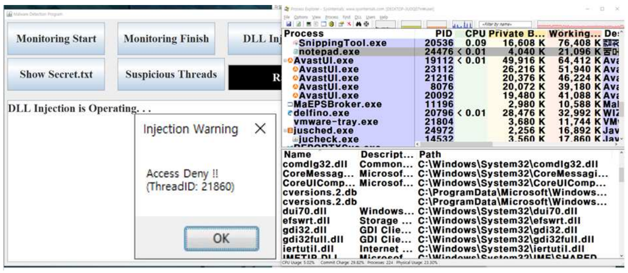


```
----- Suspicious ThreadId List -----

21860
```

```
Count: 9
  TID :...  C...  Su...  Start Address
18248 :...  2...         !TpReleaseCleanupGroupMem
27536 :...  5...         0x0000000000000000
21588                    !CoRevokeInitializeSpy+0x11
26136                    !TpReleaseCleanupGroupMem
 1984                    !TpReleaseCleanupGroupMem
27492                    !TpReleaseCleanupGroupMem
10008                    !TpReleaseCleanupGroupMem
25552                    !RtlpQueryProcessDebugInfor
21860                    !RtlUserThreadStart
Thread ID:   21860
```

(A) Monitoring Suspicious DLL



```
DLL Injection is Operating. . .
```

```
Access Deny !!
(ThreadID: 21860)
```

```
Process              PID    CPU     Private B...  Working...  De:
  SnippingTool.exe   20536  0.09    16,608 K      76,408 K
  notepad.exe        24476  < 0.01   4,040 K      21,096 K
AvastUI.exe          19112  < 0.01  49,916 K      64,412 K    Ava
  AvastUI.exe        23112          26,216 K      51,940 K    Ava
  AvastUI.exe        21216          20,376 K      46,224 K    Ava
  AvastUI.exe         8076          20,072 K      39,180 K    Ava
  AvastUI.exe        20092          19,480 K      41,088 K    Ava
MaEPSBroker.exe      11196           2,980 K      10,588 K    Mal
delfino.exe          20796  < 0.01  28,476 K      32,992 K    WIZ
vmware-tray.exe      21804           3,680 K      11,744 K    VM
jusched.exe          24972           2,256 K      16,892 K    Jav
  jucheck.exe        14532           3,560 K      17,860 K    Jav

Name            Descript...  Path
comdlg32.dll    Common...    C:\Windows\System32\comdlg32.dll
CoreMessag...   Microsof...  C:\Windows\System32\CoreMessagi...
CoreUIComp...   Microsof...  C:\Windows\System32\CoreUIComp...
cversions.2.db               C:\ProgramData\Microsoft\Windows...
cversions.2.db               C:\ProgramData\Microsoft\Windows...
dui70.dll       Windows...   C:\Windows\System32\dui70.dll
efswrt.dll      Storage ...  C:\Windows\System32\efswrt.dll
gdi32.dll       GDI Clie...  C:\Windows\System32\gdi32.dll
gdi32full.dll   GDI Clie...  C:\Windows\System32\gdi32full.dll
iertutil.dll    Internet ... C:\Windows\System32\iertutil.dll
```

(B) Preventing DLL Injection Attack on RWAR

Fig. 3. The implement result of RWAR

B. *Improvement of RWAR*

To prevent a cheating attack using a DLL injection skill on WAR, this paper proposes inserting a suspicious DLL checking routine into WAR. We call the improved scheme a robust whitelist-based anti-ransomware solution (RWAR). As shown in Fig. 1, the RWAR has three new elements over the original WAR.

(1) A suspicious list. It is the list of suspicious threads causing the result of Case 3 in Table 1.

(2) A suspicious-list manager module (SMM). It handles the suspicious list according to reports of a suspicious DLL monitor module (SDMM). If the report is about the creation of a thread, SMM adds the ID of the thread on the suspicious list. If it is about the deletion of a thread, SMM deletes the ID of the thread on the suspicious list. Additionally, SMM responds to the query for a thread of a file access control manager module (ACM).

(3) A suspicious DLL monitor module (SDMM). It registers a thread notify callback routine and manages the result of the callback routine. Callback routine selects suspicious threads which are filtered by following two conditions:

- A thread is classified as Case 3 in Table 1.
- The address (ETHREAD.Win32StartAddress) of the classified thread is the same to the address of LoadLibraryA (or LoadLibraryW) function.

Then, it notifies both the ID and status (either create or delete) of suspicious threads to the SDMM. The SDMM sends the notification to the SMM.

The RWAR improves the ACM of WAR so that the ACM can control suspicious threads. Fig. 1 shows the RWAR procedure.

(A) An application program tries to access a target file in a system.

(B) The I/O manager generates the IRP for the request. Originally, the IRP is transmitted to a file system driver (FSD).

(B1) However, the FAM of RWAR(WAR) intercepts the IRP before the FSD.

(B2) The FAM obtains the information of the target file, the application accessing the target file, and the thread accessing the target file. Then, the FAM queries the access request of the information to the ACM.

(B3) The ACM checks the whitelist to see if the application has permission to access the file. If it does not, the ACM notifies the result (access-deny) to the FAM as step B5.

(B4) Otherwise, the ACM queries the thread accessing the target file to the SMM. The SMM checks the suspicious list to determine whether the thread was created by a process other than the application.

(B5) The ACM notifies the result (either access or access-deny) of checking both the whitelist and the suspicious list to the FAM.

(B6) According to the notification of the ACM, if it is an access-deny, the FAM notifies the access-deny to the I/O manager and deletes the IRP. Otherwise, the FAM sends the IRP to the FSD.

(C) Using the IRP, the FSD reads data of the file either from a physical driver or from a cache manager.

(D) The FSD sends the data to the I/O manager.

(E) The I/O manager sends the data to the application.

IV. Implementation and Evaluation

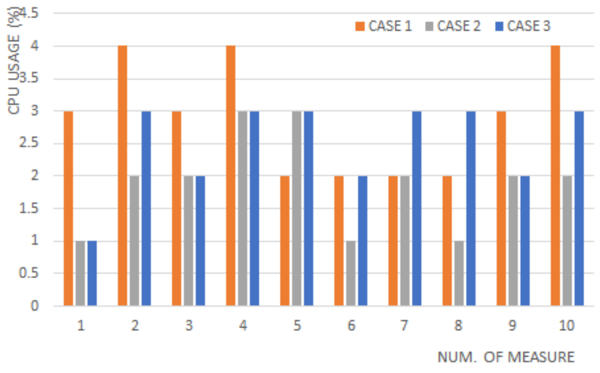
A. *RWAR Implementation*

Fig. 2 shows how RWAR prevents the DLL injection attack: (A) RWAR judges a thread (TID: 21860) as being suspicious. Then it adds the thread to the suspicious thread list. (B) When an attacker injects a malicious DLL into a notepad process listed in a whitelist, it shows that the suspicious thread (TID: 21860) cannot access a target file. That is, the malicious code of an injected DLL fails to encrypt a target file.

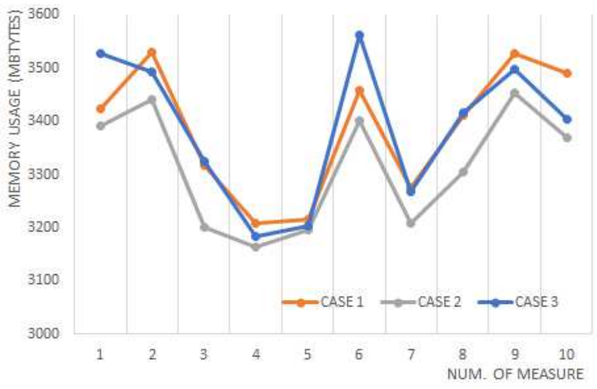
B. *RWAR Performance Evaluation*

Fig. 3 describes the performance evaluation result of RWAR. To evaluate the degree of device performance degradation, this paper measures the system performance of a device (CPU: Intel i7-10700F, RAM: 32 GB, OS: Windows 10 (64 bit)) among three cases. The first case is that two software programs (a word processor and an anti-virus program) are operating but RWAR is not yet installed. The second case is that both a word processor and RWAR are running. However, RWAR does not monitor suspicious threads. The third case is that RWAR monitors suspicious threads, and an attacker tries to inject an attack DLL.

This paper measures the system performance (both CPU and memory usages) for one hour each time and then computes the average of the results. Additionally, we performed such a measurement ten times.



(A) CPU Consumption Rate



(B) Memory Consumption

Fig. 3. Performance Evaluation

When comparing Case 1 with Case 3, RWAR still does not cause performance degradation of the device. That is, the average CPU usages are 2.9% and 2.5% for Case 1 and Case 3, respectively. Additionally, when comparing Case 2 with Case 3, the average CPU usages are 1.9% and 2.5% respectively, which is generally not a significant difference. In Cases 2 and 3, RWAR/WAR does not make "the usage rate of CPU/memory" prominently higher than other programs.

V. Conclusion

This article presents three points. First, when [3] and [4] proposed anti-ransomware solutions, some security experts pointed out the possibility of cheating such solutions using DLL injection techniques. Because such solutions confirm whether processes trying to access files are authorized, if an attacker makes the attack code look like a part of an authorized process using DLL injection skills, the attacker can easily cheat such solutions. This paper confirmed the validity/practicality of such a point through implementation. Practically, it shows that an attacker can cheat WAR using the DLL injection technique.

Second, to countermeasure against such vulnerability of WAR, this paper proposes a method to prevent a DLL injection attack against WAR. It monitors threads suspected of being created during the DLL injection attack. Then, it utilizes the suspicious thread information as an additional condition to check the access condition of processes accessing files.

Finally, RWAR is easy to use. Like WAR, users neither need to regularly check every file on their devices nor remember some safe folders monitored/controlled by a malware solution. Additionally, RWAR does not require user intervention to prevent DLL injection attacks. Therefore, RWAR can more completely and efficiently prevent various ransomware and minimize damage to users' files/devices.

Acknowledgment

This research was funded by NFR of Korea, grant number NRF-2021R1F1A106295.

References

[1] S. Chakkaravarthy, D. Sangeetha, and V. Vaidehi, "A Survey on malware analysis and mitigation techniques," Computer Science Review, vol. 32, pp. 1-23, May 2019.

[2] N. Rani, S. Dhavale, A. Singh, and A. Mehra, "A Survey on Machine Learning-Based Ransomware Detection," Proceedings of the Seventh International Conference on Mathematics and Computing, ICMC 2021, Advances in Intelligent Systems and Computing, vol. 1412, Springer, 2021, pp. 171-186.

[3] D. Kim and J. Lee, "Blacklist vs. Whitelist-Based Ransomware Solutions," IEEE Consumer Electronics Magazine, vol. 9, no. 3, pp. 22-28, May 2020.

[4] Microsoft Docs, "Enable controlled folder access," Online: https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/enable-controlled-folders?view=o365-worldwide

[5] B. Ko, W. Choi, and D. Jeong, "A Study on the Tracking and Blocking of Malicious Actors through Thread-Based Monitoring," Journal of the Korea Institute of Information Security & Cryptology, vol. 30, no. 1, pp. 75-86, Feb. 2020.

[6] L. Abrams, "Windows 10 Ransomware Protection Bypassed Using DLL Injection," Online: https://www.bleepingcomputer.com/news/security/windows-10-ransomware-protection-bypassed-using-dll-injection/

[7] A. Klein and I. Kotler, "Windows Process Injection in 2019", Black Hat USA 2019, 2019. Online: https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-InjectionTechniques-Gotta-Catch-Them-All-wp.pdf

[8] A. Alasiri, M. Alzaidi, D. Lindskog, P. Zavarsky, R. Ruhl, and S. Alassmi, "Comparative Analysis of Operational Malware Dynamic Link Library (DLL) Injection Live Response vs. Memory Image," International Conference on Computing, Communication System and Informatics Management (ICCCSIM), Bur Dubai, UAE, 29-30 July, 2012.

[9] Microsoft Docs, "Filter Manager and Minifilter Driver Architecture," 7 Jan., 2020, Online: https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-and-minifilter-driver–architecture/.