Electronic Preprint for Journal of Information Processing Vol.26

Regular Paper

Stealth Loader: Trace-free Program Loading for Analysis Evasion

Yuhei Kawakoya[1,a)] Eitaro Shioji[1,b)] Yuto Otsuki[1,c)] Makoto Iwamura[1,d)] Jun Miyoshi[1,e)]

Received: March 8, 2018, Accepted: May 10, 2018

Abstract: Understanding how application programming interfaces (APIs) are used in a program plays an important role in malware analysis. This, however, has resulted in an endless battle between malware authors and malware analysts around the development of API [de]obfuscation techniques over the last few decades. Our goal in this paper is to show the limit of existing API de-obfuscation techniques. To do that, we first analyzed existing API [de]obfuscation techniques and clarified that an attack vector commonly exists in these techniques; then, we present *Stealth Loader*, which is a program loader to bypass all existing API de-obfuscation techniques. The core idea of Stealth Loader is to load a dynamic link library (DLL) and resolve its dependency without leaving any traces on memory to be detected. We demonstrated the effectiveness of Stealth Loader by analyzing a set of Windows executables and malware protected with Stealth Loader using major dynamic and static analysis tools. The results indicate that among other obfuscation tools, only Stealth Loader is able to successfully bypass all analysis tools.

Keywords: API obfuscation, Windows, program loader, malware analysis

1. Introduction

Malware analysis is essential for fighting cyber crime. Analysts take advantage of various analysis techniques to effectively reveal the behaviors of malware. Windows userland APIs are important information sources for understanding the behaviors and intentions of malware since a sequence of APIs expresses a significant proportion of the functionalities of malware. That is, APIs are a fundamental factor for malware analysis.

Malware authors understand this situation, so they try to hide APIs used in their malware by managing various obfuscation techniques [16], [27], [29], [33]. One example is API redirection, which is an obfuscation technique that aims to confuse the control flows from call instructions to APIs by inserting junk code in the middle of the flows. Another example is DLL unlinking, which aims to make control flows unreachable from call instructions to the code of any recognized APIs. This is done by hiding loaded DLLs containing API codes, which possibly become the destination of the control flows.

To fight these API obfuscation techniques, many API de-obfuscation techniques have been proposed over the past few decades [10], [14], [22], [23], [26], [33]. For example, one technique aggressively collects traces of loaded DLLs from multiple sources, e.g., the process environment block (PEB), virtual address descriptors (VADs), or callback events, and creates a complete list of loaded DLLs. Another one executes a deep control flow analysis until it finds any API code reachable from call instructions in the original code by taking advantage of various static analysis techniques.

An essential step in these API de-obfuscation techniques is API name resolution, i.e., relating a virtual memory address to the API name. To do that, API de-obfuscation techniques have to identify the positions of loaded DLLs that contain API code. As far as we have investigated, to identify the positions of loaded DLLs, most API de-obfuscation techniques are likely to depend on data structures that the underline operating system (OS) manages. For example, in the case of Windows, many analysis tools are designed to acquire the addresses of loaded DLLs from the PEB or VADs. We argue that, behind this design, it is expected that the Windows OS precisely manages loaded DLLs and keeps track of them by storing the metadata of them in specific data structures. We also argue that this expectation possibly becomes an attack vector for malware authors to evade existing API de-obfuscation techniques.

Our goal in this paper is to show the limitation of existing API de-obfuscation techniques by actually attacking this expectation. To do that, we present *Stealth Loader*, a program loader to evade all existing API de-obfuscation techniques. The design principle of Stealth Loader is that it loads a DLL without leaving any traces in Windows-managed data structures. To achieve this, we take two approaches. The first is that we redesign each phase of program loading to become trace-free. The second is that we add two new features to a program loader; one is for removing some fields of the portable executable (PE) header of a loaded

[1] NTT Secure Platform Laboratories, Musashino, Tokyo 180–8585, Japan
[a)] kawakoya.yuhei@lab.ntt.co.jp
[b)] shioji.eitaro@lab.ntt.co.jp
[c)] otsuki.yuto@lab.ntt.co.jp
[d)] iwamura.makoto@lab.ntt.co.jp
[e)] miyoshi.jun@lab.ntt.co.jp

This is an extended version of a paper published, *Proc. Research in Attacks, Intrusions and Defenses, RAID2017*, Lecture Notes in Computer Science, Vol.10453, Springer, Cham.

DLL from memory after it has been loaded, and the other is for removing the behavioral characteristics of Stealth Loader.

One effect of Stealth Loader is that a *stealth-loaded* DLL [*1] is not recognized as a loaded DLL by analysis tools and even by the Windows OS because there is no evidence in Windows-managed data structures to recognize it. Due to this effect, calls of the functions exported from stealth-loaded Windows-system DLLs, such as kernel32.dll and ntdll.dll, are not recognized as API calls because the DLLs are not recognized as loaded, i.e., analysis tools fail API name resolution.

The main challenge of this paper is to design a trace-free program loader without destroying the runtime environment for running programs. A program loader is one of the core functions of an OS. Therefore, simply changing the behavior of a program loader is likely to affect the runtime environment, and that change sometimes leads to a program crash. In addition, changes excessively specific to a certain runtime environment lose generality as a program loader. We need to carefully redesign each step of the program loading procedure while considering the side effects on runtime environments that our changes may cause.

To demonstrate the effectiveness of Stealth Loader against existing API de-obfuscation techniques, we embedded Stealth Loader into several Windows executables and analyzed them with major malware analysis tools. The results indicated that all of these tools failed to analyze the invoked or imported APIs of stealth-loaded DLLs.

In addition, to show that the current implementation of Stealth Loader is practical enough for hiding malware's fundamental behaviors, we protected five real pieces of malware with Stealth Loader and then analyzed them using a popular dynamic analysis sandbox, Cuckoo Sandbox [19]. The results of this experiment indicated that pieces of malware whose malicious activities were obviously identified before applying Stealth Loader successfully hid most of their malicious activities after Stealth Loader was applied. Consequently, they could make Cuckoo Sandbox produce false negatives.

The contributions of this paper are as follows.

- We analyze existing API [de]obfuscation techniques and reveal a common expectation of API de-obfuscation techniques that may become an attack vector for malware authors to bypass analyses and detections.
- We introduce *Stealth Loader*, a program loader to make a loaded system DLL invisible to evade existing analysis tools by exploiting this attack vector.
- We demonstrated the effectiveness of Stealth Loader by analyzing Windows executables and real malware protected with Stealth Loader. The results indicated that Stealth Loader successfully evaded seven primary analysis tools.
- We discuss possible countermeasures against Stealth Loader and their effectiveness in detail.

2. Problem Analysis

In this section, we explain existing API obfuscation techniques used in real-world malware and API de-obfuscation techniques that are used in both major malware analysis tools and academic studies. Then, we clarify a common expectation shared in API de-obfuscation techniques.

2.1 API Obfuscation

API obfuscation is a technique for hiding imported or invoked APIs from static or dynamic analysis tools, respectively. Malware authors often take advantage of this technique to protect their malware from being detected or analyzed. We first discuss the basics of the PE format and then explain import address table (IAT) obfuscation and DLL unlinking as a technique against static analysis. Finally, we explain API redirection as one technique against both static and dynamic analyses.

A PE executable usually has IATs and import name tables (INTs) to manage external APIs if it depends on them. An IAT is a table that contains function pointers to APIs whose code is located in an external DLL. An INT is also a table that contains the names of external APIs corresponding to the IAT entries. Since these tables are referenced from the header of a PE executable, malware analysts can acquire the list of APIs that a piece of malware depends on from its PE header when they analyze a PE-format piece of malware.

To interfere with static analysis, malware often deletes the INTs and disconnects the reference to the INTs and IATs from its PE header. This is called *IAT Obfuscation*. Even if a piece of malware does not have any references to the tables from its PE header, since it keeps the list of APIs inside and restores it at runtime, it can sustain the feasibility of the original functionality.

DLL unlinking [14] is another technique for interfering with static analysis by obfuscating loaded DLLs. It makes control flows unreachable from call instructions to any APIs by hiding the metadata of loaded DLLs that could possibly become the destination of the flows. Since a control flow of an external function call does not reach any memory area where a Windows-system DLL is mapped, analysis tools fail to recognize this flow as an API call reference. This technique achieves this by removing the registered metadata of the DLL from the lists of the PEB, which is a data structure of Windows for managing loaded DLLs and their status in a process. Since some Windows APIs, e.g., EnumProcessModules, depend on the PEB to extract loaded DLL lists, unlinked DLLs can avoid being listed by these APIs.

API redirection [33] is a technique for attacking both static and dynamic analyses by obfuscating API references. As **Fig. 1** (a) shows, it modifies call instructions in the original code. Otherwise, as Fig. 1 (b) shows, it modifies the IAT entry. With these modifications, it forces control flows to APIs to detour a stub, which executes junk instructions and finally jumps to APIs. By inserting a stub between an IAT entry or call instruction; and API code, malware breaks the direct connection between the caller and callee of an API. Since API call instructions are expected to directly refer to API code or at least via an IAT entry in many analysis tools, this technique can confuse their expectations regarding the relationship between an API caller and callee.

Additionally, advanced API redirection, shown in Fig. 1 (c), is involved with stolen code [33]. At the same time, when API redirection is applied, it copies some instructions at the entry of an

[*1] a DLL loaded by Stealth Loader

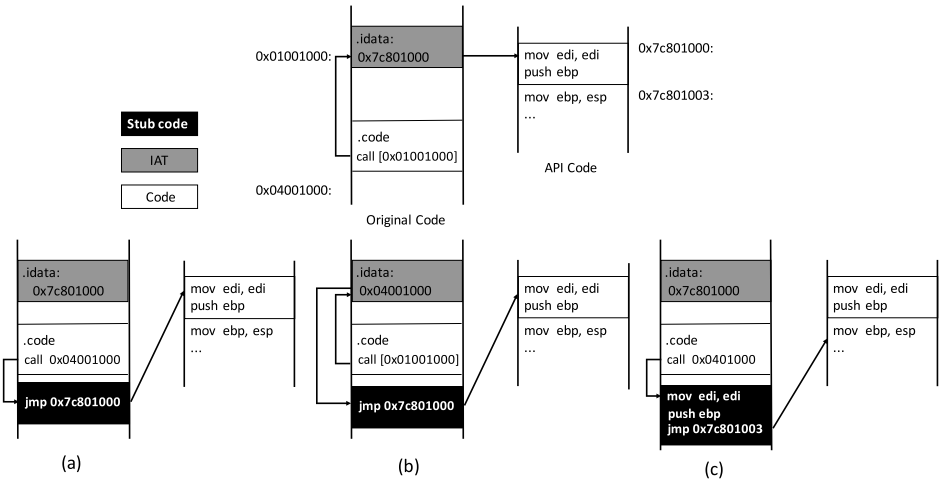


Fig. 1 Three patterns of API redirection. The top is the case of a normal Windows executable before applying API redirection. (a) Pattern in which the reference of the call instruction is modified, (b) that in which the entry of the IAT is modified, and (c) that in which API redirection is conducted with stolen code.

API, i.e., `mov edi, edi` and `push ebp` to the position before the `jmp` instruction in the allocated buffer for a stub. An execution performed after running these instructions in the buffer is transferred to the instruction after the copied ones in the API code, i.e., `mov ebp, esp`. By doing this, malware can avoid analyses that monitor the executions of an API at the entry instruction of an API, i.e., `mov edi, edi`.

2.2 API De-obfuscation

Malware analysts take advantage of API de-obfuscation techniques to clarify imported APIs or invoked APIs for static or dynamic analysis, respectively.

Regarding IAT Obfuscation, it is necessary to reconstruct obfuscated IATs and deleted INTs. To reconstruct them, most existing IAT reconstruction tools, such as impscan (a plugin of The Volatility Framework [14]) and Scylla [18], follow four steps: acquiring a memory dump, finding the IATs, resolving the API names, and restoring the PE header.

(1) Run a target program until it resolves imported APIs and fills in the IATs with the resolved addresses and then acquire a memory dump of it.
(2) Find the original IATs by analyzing code sections of a target program, e.g., collecting memory addresses often referred by indirect call instructions such as `call [0x01001000]`.
(3) Resolve the API names from each entry of the found IATs by identifying the loaded addresses of each DLL and then make a list of the imported APIs.
(4) Rebuild the INTs with the resolved API names and then update the pointers in the PE header to point to the found IATs and rebuilt INTs.

To defeat DLL unlinking, even if a loaded DLL is not listed on the PEB, we can find the existence of an unlinked DLL by parsing VADs if we use ldrmodules, which is a plugin of The Volatility Framework [14]. Quist et al. also proposed a technique of parsing VADs to identify mapped DLLs [22]. In addition, Rekall [24] identifies loaded DLLs in memory dumps on the basis of the debug section included in the PE header of each loaded DLL. In a PE header, a globally unique identifier (GUID) can be contained, and Rekall sends a query to a Microsoft symbol server to find the DLL related to the GUID.

Some dynamic analysis tools, such as Cuckoo Sandbox [19], make the correspondence between addresses and API names by monitoring APIs or events. For example, by monitoring LoadLibrary, we can obtain both the loaded address of a DLL and its file name at the same time since the address is returned from this API and the file name is passed to this API. Raber et al. proposed a technique of focusing on API hookings [23]. It hooks API calls, collects the call sites of the API calls, analyzes the call sites to identify the address of the IATs, which are referenced from the call site instructions, and then resolve each entry of the IATs with the monitored API names.

To fight API redirection, Sharif et al. [26] proposed a technique of statically analyzing control flows from call instructions until the flows reach any API code. Even if there is a stub between them, their technique can get over it by continuously analyzing flows to the end of a stub.

To overcome stolen code, as shown in Fig. 1 (c), Kawakoya et al. [9] proposed a technique of tracking the movement of API code with taint analysis. Their technique sets taint tags on API code and tracks them by propagating the tags to identify the position of copied instructions.

2.3 Analysis

A common intention of existing API obfuscation techniques is to attack API name resolution, i.e., the intention is to make it difficult to relate a virtual memory address to an API name. If analysis tools fail to establish the relationship between an executed virtual memory address and an API name, they fail to recognize an execution transfer from the virtual address to the API code as an API call.

On the other hand, the strategies that existing API de-obfuscation techniques take to fight API obfuscation techniques are either to complement lacking or hidden DLL information by finding the information from multiple data sources or to perform

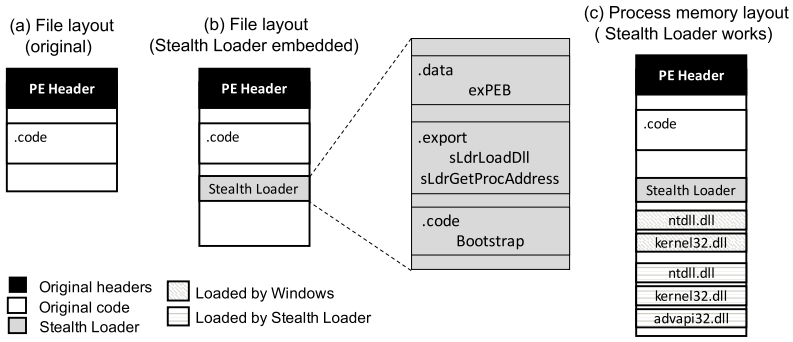


Fig. 2 How Stealth Loader works and its components. (a) The file layout of an executable before Stealth Loader is embedded, (b) that after Stealth Loader is embedded and the components of Stealth Loader are also described, and (c) the process memory layout after Bootstrap resolves the dependencies of an executable and stealth-loaded DLLs.

deeper code analysis until they reach a certain point where DLL information is found. In both cases, they rely on the metadata of DLL, which is stored in some of the data structures the OS manages. In other words, they expect that the OS precisely manages loaded DLLs, keeps track of their loading and unloading, and stores their metadata in certain data structures.

3. Design

In this section, we present *Stealth Loader*, which is a program loader that does not leave any traces of loaded DLLs in Windows-managed data structures. First, we give an overview of Stealth Loader and then introduce its design.

3.1 Overview

Figure 2 shows the components of Stealth Loader and how it works. Stealth Loader is composed of *exPEB*, *sLdrLoadDll*, *sLdrGetProcAddress*, and *Bootstrap*. exPEB is the data structure to manage the metadata of stealth-loaded DLLs, sLdrLoadDll and sLdrGetProcAddress are exported functions and the main components of Stealth Loader, sLdrLoadDll is used for loading a specified DLL in the manner we explain in this Section, and sLdrGetProcAddress is used for retrieving the address of an exported function or variable from a specified stealth-loaded DLL. Bootstrap is a code snippet for resolving the API dependencies of an executable and stealth-loaded DLLs by using the two exported functions.

The workflow for applying Stealth Loader to a PE executable we want to protect, called a *target* executable, is as follows. We first parse the PE header of a target executable for enumerating the imported APIs. We next embed Stealth Loader into a target executable with the information of the enumerated APIs, and then generate a new executable, called a *protected* executable. At that time, we drop the INTs and remove the links from the PE header to the INTs and IATs of a target program for obfuscation.

After generating a protected executable and when it begins to run, the embedded Stealth Loader works as follows. First, Bootstrap code is executed, it identifies necessary DLLs for a target executable, and then loads them using sLdrLoadDll. In this process, it does not rely on Windows-loaded DLLs [*2] at all to resolve the dependency of stealth-loaded DLLs. After loading all necessary DLLs and resolving APIs, the execution is transferred from Bootstrap to the code of a target executable.

Our intention behind Stealth Loader is to attack API name resolution as other API obfuscation techniques do. We achieve this by hiding the existences of loaded Windows-system DLLs. This is the same intention as DLL unlinking, but Stealth Loader is more robust against API de-obfuscations. We tackle this from two different directions. The first is that we redesign the procedure of program loading to be trace-free. The second is that we add two new features to a program loader; one is for removing traces left on memory after completing DLL loading, and the other is for removing the characteristic behaviors of Stealth Loader.

3.2 Program Loader Redesign

We first break the procedure of a program loader into three phases: code mapping, dependency resolution, and initialization & registration. Then, we observe what traces may be left at each phase for loading a DLL. On the basis of observation, we redesign each phase. In addition, we consider that the side effects caused by the redesigns are reasonable as an execution environment.

3.2.1 Code Mapping

3.2.1.1 Observation

The purpose of this phase is to map a system DLL that resides on disk into memory. Windows loader conducts this using a file-map function, such as CreateFileMapping. The content of a mapped file is not loaded immediately. It is loaded when it becomes necessary. This mechanism is called “on-demand page loading.” Thanks to this, the OS is able to consume memory efficiently. That is, it does not always need to keep all the contents of a file on memory. Instead, it needs to manage the correspondence between memory areas allocated for a mapped file and its file path on a disk. Windows manages this correspondence using the VAD data structure. A field in a VAD indicates the path for a mapped file when the corresponding memory area is used for file mapping. This path of a mapped file in a VAD becomes a trace for analysis tools to detect the existence of a loaded system DLL on memory. In fact, ldrmodules [14] acquires the list of loaded DLLs on memory by parsing VADs and extracting the file path of each mapped file.

[*2] DLLs loaded by Windows

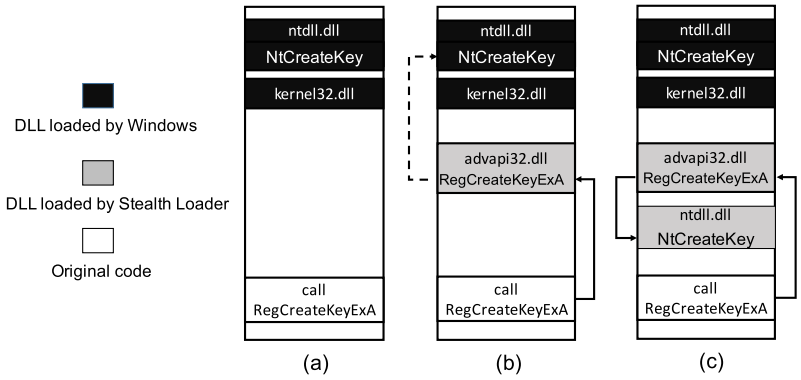


Fig. 3 Example of resolving dependency with Stealth Loader. (a) The layout before Stealth Loader starts, (b) the stealth-loaded advapi32.dll does not create a dependency on the Windows-loaded ntdll.dll, and (c) the stealth-loaded advapi32.dll creates a dependency on the stealth-loaded ntdll.dll.

3.2.1.2 Design

Instead of using file-map functions, we map a system DLL using file and memory operational functions such as CreateFile, ReadFile, and VirtualAlloc, to avoid leaving path information in VADs. The area allocated by VirtualAlloc is not file-mapped memory. Therefore, the VAD for the area does not indicate any relationship to a file. The concrete flow in this phase is as follows.

(1) Open a DLL file with CreateFile and calculate the necessary size for locating it onto memory.
(2) Allocate continuous virtual memory with VirtualAlloc for the DLL on the basis of the size.
(3) Read the content of an opened DLL file with ReadFile and store the headers and each section of it to proper locations in the allocated memory.

3.2.1.3 Side Effect

Avoiding file-map functions for locating a DLL on memory imposes two side effects. The first is that we have to allocate a certain amount of memory immediately for loading all sections of a DLL when we load the DLL. This means that we cannot use on-demand page loading. The second is that we cannot share a part of the code or some of the data of a stealth-loaded DLL with other processes because memory buffers allocated with VirtualAlloc are not shareable, while those where files are mapped are sharable. Regarding these side effects, we argue that they are not significant limitations of Stealth Loader because recent computers have sufficient memory; thus, this does not become a critical issue.

3.2.2 Dependency Resolution

3.2.2.1 Observation

The purpose of this phase is to resolve the dependency of a loading DLL. Most DLLs somehow depend on APIs exported from other DLLs. Therefore, a program loader has to resolve the dependency of a loading DLL to make the DLL ready to execute. When the Windows loader finds a dependency, and if a dependent DLL is already loaded into memory, it is common to use already loaded DLLs to resolve the dependency, as shown in **Fig. 3** (b).

However, this dependency becomes a trace for analysis tools, i.e., behavioral traces. For example, if a stealth-loaded advapi32.dll has a dependency on a Windows-loaded ntdll.dll, the APIs of ntdll.dll indirectly called from advapi32.dll may be monitored by analysis tools. In other words, we can hide a call of RegCreateKeyExA but cannot hide that of NtCreateKey. Analysis tools can obtain similar behavior information from NtCreateKey as that from RegCreateKeyEx since RegCreateKeyEx internally calls NtCreateKey while passing almost the same arguments.

3.2.2.2 Design

To avoid this, Stealth Loader loads dependent DLLs to resolve the dependency of a loading DLL. In the case in Fig. 3, it loads ntdll.dll to resolve the dependency of advapi32.dll. As a result, after advapi32.dll has been loaded and its dependency has been resolved, the memory layout is like that shown in Fig. 3 (c). On the basis of this layout, when an original code calls RegCreateKeyExA, RegCreateKeyExA internally calls the NtCreateKey of stealth-loaded ntdll.dll. Therefore, this call is invisible to analysis tools, even if a Windows-loaded kernel32.dll and ntdll.dll are monitored by them.

3.2.2.3 Side Effect

The side effect caused by this design is reduced memory efficiency. That is, Stealth Loader consumes approximately twice as much memory for DLLs as the Windows loader since it newly loads a dependent DLL even if the DLL is already located on memory. We consider this side effect as not being that significant because recent computers have sufficient memory, as we previously mentioned.

3.2.3 Initialization & Registration

3.2.3.1 Observation

Windows loader initializes a loading DLL by executing the initialize function exported from a DLL, such as DllMain. At the same time, it registers a loaded DLL to the PEB. In the PEB, the metadata of loaded DLLs is managed by linked lists. Many analysis tools often check the PEB to acquire a list of loaded DLLs and their loaded memory addresses.

3.2.3.2 Design

Stealth Loader also initializes a loading DLL in the same way as Windows loader does. However, it does not register the metadata of loaded DLLs to the PEB to avoid being detected by analysis tools through the PEB.

3.2.3.3 Side Effect

The side effect of this design is that stealth-loaded DLLs cannot receive events such as process-creation or process-termination. This is because these events are delivered to DLLs listed in the

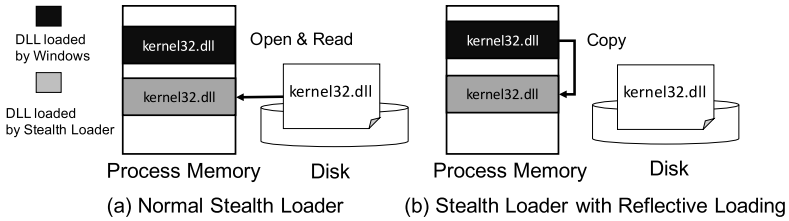


Fig. 4 Behaviors of normal Stealth Loader and Reflective Loading. (a) Stealth Loader loads kernel32.dll from a disk, and (b) Stealth Loader with Reflective Loading loads kernel32.dll from the memory, i.e., the Windows-loaded one.

PEB. We consider this effect as not being very significant because most system DLLs do not depend on these events at all as far as we have investigated. Most are implemented to handle only create-process and -thread events, which are executed mainly when the DLL is first loaded.

3.3 Stealthiness Enhancement

Apart from finding traces in Windows-managed data structures, there are other techniques of identifying the existence of a loaded DLL. In this subsection, we present the possibility of detecting loaded DLLs from characteristic strings in the PE header of a certain DLL or behaviors of Stealth Loader. Then, we introduce our techniques to hiding the string patterns and behaviors.

3.3.1 PE Header Removal

Stealth Loader deletes some fields of the PE header on memory after it has loaded a DLL and resolved its dependency. This is because some of the fields may become a hint for analysis tools to infer a DLL loaded on memory. For example, GUID may be included in the debug section of the PE header of a system DLL and becomes an identifier of a specific DLL. Another example is that the tables of exported and imported API names of a system DLL, which are pointed from the PE header, also provide sufficient information for analysis tools to identify a DLL. Like these examples, the PE header contains a large amount of information for identifying a DLL.

To avoid being identified through characteristic fields in the PE header, we delete the debug section, timestamp, version information, INTs, and export name table (ENT) in the PE header. Basically, the debug section, timestamp, and version header, are not used by the original code in a process under normal behavior; they are only used for debugging purposes or providing extra information of a DLL. Thus, we can delete them without any concern as this deletion degrades the feasibility of execution. However, we need to pay attention to the timing of deleting INTs. An INT is necessary to resolve dependencies only when a DLL is being loaded. After it is completed, this table is not referenced from the code and data. Therefore, we can delete them after a DLL has been loaded.

Unlike the above-mentioned fields, we cannot simply delete the ENT since it is accessed after a DLL has been loaded to retrieve the address of an exported API of the loaded DLL at runtime. This is called "dynamic API resolution". Therefore, we prepared an interface, sLdrGetProcAddress, to resolve APIs exported from stealth-loaded DLLs. We also prepared a data structure, exPEB, in Stealth Loader to manage the exported API names and corresponding addresses of each stealth-loaded DLL. Therefore, we can also delete the ENT without losing the dynamic API resolution capability in a protected executable.

There are publically available tools for removing fields unnecessary for execution from the PE header after compilation, such as PE explorer [31] or `strip` command. However, they basically do not remove fields necessary for running programs, such as INTs or ENT. On the other hand, Stealth Loader can do this because it runs inside of a protected executable and performs deletion by determining the context of the execution.

3.3.2 Reflective Loading

Reflective Loading is used for hiding the API calls invoked from Stealth Loader. While the calls invoked from original code are successfully hidden by Stealth Loader, API calls invoked from Stealth Loader are still visible to analysis tools because Stealth Loader basically uses APIs exported from Windows-loaded DLLs (**Fig. 4** (a)). These exposed API calls enable analysis tools to detect the existence of Stealth Loader because some of the behaviors of Stealth Loader are not often seen in normal programs. For example, `CreateFile(‘‘kernel32.dll’’)` is very characteristic since programs normally load a DLL with `LoadLibrary(‘‘kernel32.dll’’)` and do not open a Windows-system DLL as a file with CreateFile.

To avoid this, we use Reflective Loading. The core idea of Reflective Loading is to copy all sections of an already loaded DLL to allocated buffers during the code mapping phase instead of opening a file and reading data from it (Fig. 4 (b)). This idea is inspired by Reflective DLL injection, introduced by Fewer [5], as a technique of stealthily injecting a DLL into another process. We leveraged this to load a DLL as a part of Stealth Loader without opening the file of each DLL. If a target DLL is not loaded at that time, we use the APIs of the stealth-loaded kernel32.dll to open a file, allocate memory, and conduct the other steps. kernel32.dll and ntdll.dll are always loaded before Stealth Loader because these DLLs are loaded by Windows as a part of process initialization. Thus, we can completely hide all API calls invoked by Stealth Loader from analysis tools monitoring API calls.

4. Implementation

We have implemented Stealth Loader on Windows 7 Service Pack 1. In this section, we explain the dynamic API resolution of Stealth Loader, stealth-loadable APIs, and Console Subsystem Cheating.

4.1 Dynamic API Resolution

Stealth Loader supports dynamic API resolution with sLdrLoadDll and sLdrGetProcAddress. When Stealth Loader loads a

DLL depending on the LdrLoadDll or LdrGetProcedureAddress of ntdll.dll, e.g., kernel32.dll, it replaces the entries in the IAT for ntdll.dll to the two functions in the loading DLL with pointers to sLdrLoadDll or sLdrGetProcAddress, respectively. In this situation, when the original code attempts to dynamically load a DLL, for example, using LoadLibrary, which internally calls LdrLoadDll, the API call to LoadLibrary redirects to sLdrLoadDll, and then Stealth Loader loads a specified DLL.

4.2 Stealth-loadable APIs

In Stealth Loader, we support 12 DLLs: ntdll.dll, kernel32.dll, kernelbase.dll, gdi32.dll, user32.dll, shell32.dll, shlwapi.dll, ws2_32.dll, wininet.dll, winsock.dll, crypt32.dll, and msvcrt.dll. This means that we support in total 7,764 APIs exported from these 12 DLLs. The number of unsupported APIs is 1,633. The reasons we cannot support them are described in Appendix A.1. Since these reasons are very detailed and specific to the Windows 7 environment, we put them into this appendix. We can support more DLLs with no or at least little cost. However, we consider the current number of supported APIs to be enough for the purpose of this paper because we have already covered 99% (1018/1026) of APIs on which IDAScope, a popular static malware analysis tool [21], focuses as important APIs. We also covered 75% (273/364) of the APIs on which Cuckoo Sandbox, a popular sandbox whose target APIs are selected by malware analysts [19], sets hooks for dynamic analysis. Regarding the remaining 25% of APIs, they separately reside in several DLLs in a small group.

4.3 Console Subsystem Cheating

A console application with stealth-loaded kernel32.dll does not work properly or sometimes crashes on Windows 7 or later environments. The reason for the crash is as follows. To begin with, a Windows console application must establish a connection to a console server, i.e., conhost.exe, to create a console window and activate the standard output, input, and error. This connection is established while kernel32.dll is being initialized, i.e., while DllMain is being executed. A console application with stealth-loaded kernel32.dll fails to establish the connection since the Windows-loaded kernel32.dll has already established the connection when it was initialized. This connection failure causes the program to crash.

To overcome this, we introduce *Console Subsystem Cheating* to properly run a console application with Stealth Loader. Console Subsystem Cheating makes Windows recognize a console application with Stealth Loader as a GUI application while the real kernel32.dll is being initialized. Additionally, it also make Windows recognize a command line interface (CUI) one while the stealth-loaded kernel32.dll is being initialized. More concretely, before executing a protected executable, we modify the subsystem entry of the PE header with value "2", which indicates a Windows GUI application. After starting its execution, while the real kernel32.dll is being initialized, the connection to a console server is not established because the Windows loader recognizes this application as a GUI. Then, before initializing the stealth-loaded kernel32.dll, we replace the value with value "3", which means a Windows CUI application. The stealth-loaded kernel32.dll can successfully connect to a console server because this is the first time a request for connection to the server is made in the process. With this trick, we successfully apply Stealth Loader to console applications as well as GUI.

5. Experiment

To show the feasibility of Stealth Loader, we conducted three experiments: one for comparing its resistance capability against current analysis tools to other API obfuscation techniques, another for confirming its effectiveness with real malware, and the other for measuring the impact of the increase in memory consumption caused by Stealth Loader.

5.1 Resistance

To show the resistance capability of Stealth Loader against current API de-obfuscation tools, we prepared test executables and analyzed them with seven major static and dynamic analysis tools that are primarily used in the practical malware analysis field. These tools are publicly available and cover the various techniques we mentioned in Section 2.2. Regarding the other techniques which are not covered by these tools, we qualitatively discuss the resistance capability of Stealth Loader against them in Section 7.3 because they are not publicly available.

The test executables were prepared by applying Stealth Loader for eight Windows executables, calc.exe, winmine.exe, notepad.exe, cmd.exe, wmplayer.exe, taskmgr.exe, wscript.exe, and ftp.exe. After applying Stealth Loader to them, we verified if the executables were runnable without any disruptions and as functional as they had been before applying Stealth Loader by interacting with running test executables, such as clicking buttons, inputting keystrokes, writing and reading files, and connecting to the Internet. For clarification, we refer to an executable after applying Stealth Loader as a *protected* executable and an executable before applying Stealth Loader as a *vanilla* executable.

For comparison, we prepared tools using different API obfuscation techniques, i.e., IAT obfuscation, API redirection, which is the pattern explained in Fig. 1 (c), and DLL unlinking. Using these tools, we applied these techniques to the same eight Windows executables. We analyzed them with the same analysis tools and compared the results.

5.1.1 Static Analysis

In this experiment, we analyzed each protected executable with four major static analysis tools, IDA [7], Scylla [18], impscan (The Volatility Framework [14]), and ldrmodules (The Volatility Framework [14]). IDA is a de-facto standard dis-assembler for reverse engineering. Scylla is a tool that reconstructs the destroyed IATs of an obfuscated executable. impscan and ldrmodules are plugins of The Volatility Framework for reconstructing IATs and making a list of all loaded modules on memory, respectively.

We explain how each analysis tool, except for IDA, resolves APIs. Scylla acquires the base addresses of loaded DLLs from the EnumProcessModules API, which internally references the PEB and resolves API addresses with GetProcAddress. In addition, it heuristically overcomes API redirection. impscan also acquires the base addresses from the PEB and resolves API ad-

Table 1 Static and dynamic analysis resistance results.

API Obfuscations	Static Analysis				Dynamic Analysis		
	IDA	Scylla	impscan	ldrmodules	Cuckoo	traceapi	mapitracer
Stealth Loader	✓	✓	✓	✓	✓	✓	✓
IAT Obfuscation	✓			N/A [1]			
API Redirection	✓	[2]	✓	N/A [1]	✓	✓	✓
DLL Unlinking		✓	✓				

✓indicates that the obfuscation technique successfully evaded the tool. Stealth Loader evaded all the tools.

[1] IAT Obfuscation and API Redirection are techniques for API obfuscation while ldrmodules is a tool for extracting loaded DLLs.

[2] When we manually gave the correct original entry point of a protected executable to Scylla, it could identify imported APIs correctly. When we did not, it failed.

dresses from the export address table (EAT) of each loaded DLL. ldrmodules acquires the base addresses from VADs.

5.1.1.1 Procedure

We first statically analyzed each protected executable using each analysis tool and then identified imported APIs. In the case of ldrmodules, we identified loaded DLLs. We then manually compared the identified imported APIs or loaded DLLs with those we had acquired from the same vanilla executables.

5.1.1.2 Results

The left part of **Table 1** shows the results of this experiment. Stealth Loader successfully defeated all static analysis tools, while the others were analyzed with some of them. This is because there were no hints for the analysis tools to acquiring the base addresses of loaded DLLs. IAT obfuscation failed to defeat Scylla and impscan because these two tools were originally designed for reconstructing IATs in the manner we explained in Section 2.2. API redirection failed to evade Scylla because Scylla is designed for heuristically overcoming API redirection. DLL unlinking failed to evade ldrmodules because ldrmodules identified loaded DLLs through VADs, not the PEB.

5.1.2 Dynamic Analysis

In this experiment, we analyzed each protected executable with three dynamic analysis tools, Cuckoo Sandbox [19], traceapi [8], and mini_apitracer [30]. All are designed to monitor API calls. Cuckoo Sandbox is an open-source, dynamic malware analysis sandbox. traceapi is a sample tool of Detours, which is a library released from Microsoft Research for hooking API calls. mini_apitracer, shown as mapitracer in Table 1, is a plugin of DECAF [6], which is a binary analysis framework built on QEMU [3].

Each analysis tool relates API names and memory addresses as follows. Cuckoo acquires the base address of loaded DLLs from callback functions registered with the LdrRegisterDllNotification API and resolves API addresses with GetProcAddress. traceapi acquires the base address of loaded DLLs with LoadLibrary and resolves API addresses with GetProcAddress. mini_apitracer acquires the base addresses of loaded DLLs from the PEB and resolves API addresses by parsing the EAT of each DLL.

5.1.2.1 Procedure

We first ran each protected executable on each dynamic analysis environment and monitored the API calls. We then compared the monitored API calls with those we had collected from the same vanilla executable.

5.1.2.2 Results

The right part of Table 1 shows the results of this experiment. Stealth Loader successfully evaded all dynamic analysis tools, while the others were captured by some of them. IAT obfuscation totally failed because the dynamic analysis tools did not depend on the IATs to identify the locations of APIs. API redirection successfully defeated all of them. This is because even though the dynamic analysis tools set hooks on the first instruction of each API for API monitoring, API redirection avoided executing them. As we explained in Section 2.1, when an API is called API redirection transfers an execution to the code at a few instructions after the entry of the API. DLL unlinking also failed because the analysis tools calculated the locations of each API from the addresses of loaded DLLs and set hooks on each API before DLL unlinking had hidden DLLs.

5.2 Real-world Malware Experiment

The purpose of this experiment was to demonstrate that the current Stealth Loader implementation is practical enough for hiding the major characteristic behaviors of malware even though it has unsupported APIs.

5.2.1 Procedure

First, we collected 117 pieces of malware from VirusTotal [32] that were detected by several anti-virus products. At that time, we selected four (DownloadAdmin, Win32.ZBot, Eorezo, and CheatEngine) because they were not obfuscated. We also selected one piece of malware (ICLoader) from 113 obfuscated ones as a representative case of obfuscated ones. Next, we applied Stealth Loader to the five pieces of malware. Then, using Cuckoo Sandbox, we analyzed both the malware before and after Stealth Loader was applied. Finally, we compared the results of the analyses in terms of the malicious score, number of detected events, hit signatures, and monitored API calls. The malicious scores were calculated from observed behaviors matched with pre-defined malicious behavioral signatures [19].

To achieve the purpose of this experiment, we believe that the variety of malware's behaviors is more important than the number of malware. We also consider that the behaviors of the four pieces of malware (DownloadAdmin, Win32.ZBot, Eorezo, and CheatEngine) can cover the majority of behaviors, such as modifying a specific registry key or injecting code into another process, exhibited in all of the pieces of malware we collected for this experiment. This is because the signatures hit by analyzing those four contributed to detecting 637 out of 792 events generated by

Table 2 Real malware experiment results.

	without Stealth Loader				with Stealth Loader			
Malware Name	Score	Signatures	Events	# of Calls	Score	Signatures	Events	# of Calls
DownloadAdmin	3.6	11	16	9,581	1.8	5	12	224
Win32.ZBot	5.0	11	46	1,350	1.4	4	10	183
Eorezo	5.6	15	192	20,661	0.8	3	10	64
CheatEngine	4.8	12	209	126,086	1.6	5	10	120
ICLoader	4.0	11	33	3,321	4.0	11	38	1,661

Score is calculated as hit signatures, which are scored depending on the severity of each behavior; score of less than 1.0 is benign, 1.0–2.0 is warning, 2.0–5.0 is malicious, and higher than 5.0 means danger. Signatures means number of hit signatures, Events indicates number of captured events, and # of Calls is the number of API calls captured by Cuckoo Sandbox.

analyzing the 117 pieces of malware.

To ensure that the protected pieces of malware actually ran and conducted malicious activities, we configured Cuckoo Sandbox to write a memory dump file after each analysis had been done and then manually analyzed it with The Volatility Framework. This is for confirming the traces that had been seen before applying Stealth Loader, such as created files or modified registries, were actually found.

5.2.2 Results

Table 2 shows the results of this experiment. Regarding DownloadAdmin, Win32.ZBot, Eorezo, and CheatEngine, Stealth Loader successfully hid the malicious behaviors, then the scores dropped from malicious or danger levels to warning or benign levels. Regarding ICLoader, Stealth Loader did not obfuscate its malicious behaviors, and the scores before and after applying Stealth Loader were the same.

In the cases of DownloadAdmin, Win32.ZBot, Eorezo, and CheatEngine, the scores dropped, but did not become zero. The reason of this was that some signatures were likely to increase the score with non-standard file format. For example, if a malware has a section in its PE header, which does not look complier-generated one, the score is increased. Stealth Loader is the case. That is, since it embeds itself into an executable by adding a section, the score of an executable protected with Stealth Loader does not become zero. We do not consider that this is a significant issue of Stealth Loader because we can easily avoid this detection by embedding Stealth Loader into an existing section or giving a name like compiler-generated to the section where Stealth Loader is stored when we add a new section.

DownloadAdmin is a type of information-stealing malware. It accesses a registry to steal or check a browser configuration. Also, it checks foreground windows constantly to check if it is running on any analysis environment because analysis environments tend to have no windows during analysis. These two behaviors were the main reasons for increasing the score of this malware when we analyzed it before Stealth Loader was applied to it. After applying Stealth Loader, it successfully hid these two behaviors; consequently, Cuckoo Sandbox failed to identify these behaviors. As a result, the score dropped to 1.8 (warning) from 3.6 (malicious).

Win32.ZBot registers itself as a startup process by modifying certain registries and injects a part of its code into a child process. Stealth Loader made them invisible to Cuckoo Sandbox, even though they were visible to Cuckoo Sandbox before applying Stealth Loader to this malware. Consequently, the score dropped to 1.4 (warning) from 5.0 (danger).

Eorezo writes down an executable file from its body and executes it as a child process. This created child process conducts malicious activities, such as checking the existence of antivirus products or creating suspicious power shell scripts. On the other hand, the process of Eorezo, i.e., the process that created the child process, did not conduct malicious activities except for creating a child process. Stealth Loader hid the API calls invoked from Eorezo including those related to process creation. Therefore, Cuckoo Sandbox failed to make the parent and child relationship of Eorezo and its child process and it did not recognize the child process as a to-be-analyzed process. As a result, it failed to capture all API calls invoked from the child process even though the activities of the child process mostly contributed to the increase of the score. Also, this was the reason the number of captured APIs was significantly different before and after Stealth Loader was applied. Consequently, the score dropped to 0.8 (benign) from 5.6 (danger).

Regarding CheatEngine, Cuckoo Sandbox mainly detected four behaviors: creating a new process, searching for a web-browser process, creating mutex, which a Banker Trojan is known to use, and creating known malicious files. This malware also created some child processes, which mainly performed malicious activities. Like the Eorezo case, Cuckoo Sandbox failed to track the child processes as to-be-analyzed because Stealth Loader hid the behavior of the CheatEngine process for child process creation. As a result, Cuckoo Sandbox missed capturing the malicious activities done by the child processes and gave this malware a lower score, i.e., 1.6 (warning), than it should be.

Regarding ICLoader, the score was the same before and after applying Stealth Loader because the same behaviors were observed. The reason is that this piece of malware acquires the base address of kernel32.dll without depending on Windows APIs. That is, it directly accesses the PEB, parses a list in the PEB to find an entry of kernel32.dll, then acquires the base address of kernel32.dll from the entry. From this base address, the malware acquires the addresses of LoadLibrary and GetProcAddress of the Windows-loaded kernel32.dll and then resolves the dependencies of the other APIs by using these two APIs. Since this malware did not use LoadLibrary or the equivalent APIs of the stealth-loaded kernel32.dll for dynamic API resolution, Stealth Loader did not have a chance to obfuscate the calls of dynamically resolved APIs invoked from this malware. We consider this as not being a limitation because our expected use case of Stealth Loader is to directly obfuscate compiler-generated executables, not already-obfuscated executables.

Table 3 Memory Consumption Comparison Results.

	without Stealth Loader				with Stealth Loader				% of Increase
Program	Image	Private	Others	Total Size	Image	Private	Others	Total Size	
calc	24,252	596	32,744	57,592	12,920	18,424	41,104	72,448	125.80
winmine	24,200	604	22,108	46,912	12,220	18,432	30,492	61,144	130.34
notepad	25,024	596	22,404	48,024	25,640	18,424	30,828	74,892	155.95
cmd	7,640	92	20,080	27,812	8,900	4,336	26,232	39,468	141.91
wmplayer	71,420	13,876	74,008	159,304	71,860	19,132	75,756	166,748	104.67
wscript	24,748	662	34,754	60,164	26,008	5,552	41,672	73,232	121.72
taskmgr	28,872	784	60,568	90,224	30,140	18,612	70,268	119,020	131.92
ftp	8,320	88	22,608	31,016	8,936	5,428	30,736	45,100	145.41

The unit of `Image`, `Private`, `Others`, and `Total Size` is kilobyte (KB). We measured the memory consumption of each executable with VMMap [25]. The standard Windows program loader locates DLLs in `Image` memory, while Stealth Loader does it in `Private` memory. `Others` includes `Mapped File`, `Shareable`, `Heap`, `Managed Heap`, `Stack`, `Page Table` and `Unusable`. % of Increase is calculated from (`Total Size` of `with Stealth Loader` / `Total Size` of `without Stealth Loader`) * 100.

5.3 Memory Consumption

As we mentioned in Section 3, Stealth Loader affects the efficiency of memory consumption, but we argue that this does not become a significant problem. To demonstrate this, in this experiment, we measured how much Stealth Loader affects memory usage on its running environment before and after being applied.

5.3.1 Procedure

We used the same dataset for this experiment as for the first experiment, i.e., protected and vanilla Windows executables. We measured the memory consumptions of the same executable two times, before and after applying Stealth Loader, and then compared them. We used VMMap [25] for measuring the memory consumption of each executable and focused on `Image` and `Private` memories. This is because a protected Windows executable uses `Private` memory for allocating DLLs, while a vanilla Windows executable uses `Image` memory for mapping DLLs.

5.3.2 Results

Table 3 shows the memory consumption of each executable before or after applying Stealth Loader. Overall, every executable had a tendency of increasing the `Total Size` and `Private` after Stealth Loader was applied. There are two reasons for these increases. The first is that Stealth Loader embeds its code and data into an executable. Therefore, the size of a protected executable becomes larger than that of the vanilla one. The second is that Stealth Loader does not use memory efficiently since it newly loads DLLs even if they are already loaded and existed on memory.

Regarding `Image` memory, the consumptions of calc and winmine decreased after applying Stealth Loader. In the two cases, the total number of Windows-loaded DLLs in protected ones was less than that in the vanilla ones because some DLLs were loaded in `Private` memory with Stealth Loader, instead of mapping them in `Image` memory. On the other hand, for the other executables, the size of `Image` of a protected executable was almost same as that of its vanilla one or slightly increased. The same number of DLLs were mapped on `Image` memory between protected and vanilla executables. This is because when one of the non-target DLLs loaded in a protected executable was dependent on a target DLL, the standard program loader loaded the DLL in `Image` memory even though the same DLL was loaded by Stealth Loader in `Private` memory. As a result, the number of Windows-loaded DLLs becomes the same between a protected and a vanilla executable.

6. Related Work

In this section, we briefly repeat the API obfuscation techniques which we mentioned in Section 2.1 for comparison with Stealth Loader and then explain other types of API obfuscation techniques related to our research.

IAT obfuscation has a different target from Stealth Loader. It disturbs API name resolution by deleting the INTs and IATs and disconnecting them from the PE header, while Stealth Loader focuses on Windows-managed data structures, such as the PEB or VADs. DLL unlinking obfuscates loaded DLLs. Its purpose is the same as Stealth Loader. However, DLL unlinking focuses on only the PEB, not VADs, while Stealth Loader focues on both. API redirection obfuscates the control flow from API call instructions to recognized API code, whereas Stealth Loader attacks API name resolution. That is, Stealth Loader attempts to make API code unrecognizable.

One closely related study is by Abrath et al. [2]. They proposed a technique of linking Windows-system DLLs statically with an executable and deleting imported API information from it to prevent API calls from being monitored. The effect of linking Windows-system DLLs with an executable could be similar to the effect we obtained. However, static linked DLLs may lose the portability of a PE executable since system DLLs tend to depend on specific Windows versions, and the size of a linked executable increases.

Aside from the obfuscation techniques that we explained in Section 2.1, another type of obfuscation approach, called "API sequence obfuscation", has been proposed. Shadow Attack [16] is an API sequence obfuscation technique that works by partitioning one piece of malware into multiple processes. These multiple processes execute some of the original behaviors of the malware. Illusion Attack [27] is another API sequence obfuscation technique that passes requested system call numbers and arguments via ioctl to an underlining kernel driver. From a monitoring tool viewpoint, it looks like a sequence of ioctl. These attacks mainly focus on scrambling executed API calls to avoid detection, while Stealth Loader focuses on hiding each API call to escape from both detection and analysis.

There are some techniques for loading a malicious DLL to a memory used in real-world malware, such as Reflective DLL Injection [5], DLL Side Loading [28], Process Hollowing [12], and

Code Doppelganging [13]. These techniques are mainly designed to hide a malicious DLL, which is a part of malware, not system DLLs, such as kernel32.dll or ntdll.dll. We can also apply Stealth Loader for hiding a malicious DLL and argue that is easier than for Windows-system DLLs because malicious DLLs do not heavily depend on Windows OS, which is the most complicated component to analyze. However, we may have to pay attention to the dependency on the other parts of malware code to maintain consistency of its execution.

7. Discussion

In this section, we discuss the platform dependency of Stealth Loader, other de-obfuscation techniques, and possible countermeasures against Stealth Loader.

7.1 Platform Dependency

As we mentioned in Section 4, the current Stealth Loader is implemented to run on the Windows 7 environment. However, we believe that the design explained in Section 3 is also applicable to other Windows platforms including Windows 8 and 10. Of course, since Windows 8 and 10 have different implementations from Windows 7, we need to make Stealth Loader runnable on these platforms without any issues. More concretely, we have to resolve some corner cases, as we mentioned in Appendix A.1. In other words, the other part of this paper, i.e., all sections except for Appendix A.1 is applicable to other Windows platforms.

Regarding applying Stealth Loader to Linux, we consider that the designs of Stealth Loader are applicable to Linux platforms. Since Linux OS and libraries are less dependent on each other than Windows libraries, an implementation of Stealth Loader for Linux may become simpler than that of Windows. We argue that Stealth Loader on Linux could make library calls invisible to library-call-monitoring tools such as ltrace.

7.2 Other De-obfuscation Techniques

Eureka [26] relates the base address of a loaded DLL with a DLL file by monitoring NtMapViewOfSection API calls and extracting the specified file name and return address. Since Stealth Loader does not use file-map functions, this API is not called when Stealth Loader loads a DLL. As a result, Eureka fails API name resolution, even though it overcomes stolen code or API redirection by performing deep program analyses.

API Chaser [9] relates code with the API name before starting an analysis by setting taint tags containing the API name on the code. It then keeps track of its relationship by propagating the tags during its analysis. Since it establishes the relationship before Stealth Loader work, it may not be affected by Stealth Loader. However, it is widely known that tag propagation is disconnected at implicit flow code [4]; therefore, attackers are able to evade taint propagation by simply processing code with implicit flow without changing its value.

7.3 Countermeasures

7.3.1 Monitoring at Kernel Layer

One countermeasure against Stealth Loader is monitoring at the kernel layer. Stealth Loader has to depend on Windows-system-service calls, while it is independent of userland API code. Even though much useful information has already been lost when the executions of some APIs, e.g., network-related APIs, reach the kernel layer, a series of service system calls possibly provides a part of the whole picture regarding the behaviors of the executable protected with Stealth Loader.

7.3.2 Specialized Analysis Environment

Another countermeasure is to install hooks on system DLLs in an analysis environment before starting an analysis by modifying a file of each DLL on disk. This type of modification is likely to be detected and warned by Windows. However, since modified DLLs are loaded by not only benign processes but also processes protected with Stealth Loader, analysis tools probably identify the executions of APIs by the installed hooks when they are executed.

Instrumentation tools, such as Intel PIN [15], could possibly become a solution against Stealth Loader because they may be able to identify the locations of stealth-loaded DLLs by tracking all memory reads and writes related to the DLLs. However, a major drawback of these tools is that they are easily detectable by malware. Therefore, if malware analysts use these tools for analyzing protected malware in practice, they need to further consider hiding these tools from malware.

7.3.3 Detecting System DLLs from Memory Patterns

Scanning memory and finding specific patterns for a DLL may be effective. By preparing the patterns of each DLL in advance and scanning memory with these patterns, it could be possible to identify the modules loaded on memory. Also, comparing binaries using a different tool, such as BinDiff [34], is also effective. By comparing the control flow of a Windows-system DLL with that on memory, we could be able to identify the existence of specific DLLs. However, since there are several binary- or assembly-level obfuscation techniques, such as that proposed by Moser et al. [11], we need different counter-approaches to solve this type of problem.

7.3.4 Inferring DLLs from Visible Traces

Since the current Stealth Loader avoids supporting some APIs, as we explain in the Appendix A.1, this fact may give static analysis tools a hint to infer a DLL. For example, if analysis tools identify the position of the IATs of a stealth-loaded DLL using the approach we explained in Section 2.1, they can probably specify the DLL from only visible imported APIs in the IATs. To solve this, we could take advantage of API redirection explained in Fig. 1 (c) in Section 2.1. This type of API redirection modifies indirect API call instructions in the original code with direct instructions that make the execution jump to a stub for each API. Therefore, since there are no indirect API call instructions in the original code, analysis tools are likely to fail in identifying the IATs.

7.3.5 Detecting Stealth Loader Itself

Detecting Stealth Loader may become another direction to fight against it. One approach is detecting specific byte patterns of Stealth Loader. While Stealth Loader hides its behaviors, as we explained in Section 3.3.2, its code or data may likely have specific patterns available to be detected. However, as we discussed above, several techniques, such as that proposed by Moser et al. [11], have been proposed to avoid byte-pattern-based detec-

tion. If we apply one of them to Stealth Loader, we can avoid being detected.

Focusing on the increase in private-memory consumption is one possibility for detecting the existence of Stealth Loader. As Table 3 shows, when we apply Stealth Loader to an executable, the private-memory consumption of the executable increases. However, we argue that while this side effect of Stealth Loader may provide some information to detect Stealth Loader, it is difficult to have confidence with only this information. This is because the amount of memory usage is totally dependent on programs, and it is difficult to predict it before executing the programs. Without knowledge of the normal amount of private-memory usage of a target program, we cannot determine if the private memory of a running program is larger than or same as normal.

7.3.6 Restricting Untrusted Code

One more direction is to prevent Stealth Loader from working at each phase. Policy enforcement, which is mentioned in safe loading [20], may be partially effective for that purpose. If there is a policy to restrict opening a system DLL for reading, Stealth Loader cannot map the code of a DLL on memory if it is not loaded by Windows yet. On the other hand, if the DLLs are already loaded by Windows, Reflective loading allows us to load them with Stealth Loader.

In addition, safe loading has a restriction to giving executable permissions. No other instances, except for the trusted components of safe loading, give executable permission to a certain memory area. Safe loader supports only the Linux platform; however, if it would support Windows, safe loading may be able to prevent Stealth Loader from providing the executable permission to the code read from a DLL file.

Another line of research in this category is to restrict jumping to untrusted functions, such as control flow integrity (CFI) [1]. In case of Control Flow Guard [17], which is an implementation of CFI in Windows, trusted functions are managed and registered in ntdll.dll. Since Stealth Loader has its own ntdll.dll, stealth-loaded ntdll.dll, it may be able to register the functions in stealth-loaded DLLs in its ntdll.dll and make stealth-loaded DLLs query the ntdll.dll if the jump destination is trustable. As a result, the functions in stealth-loaded DLLs become trustable with the ntdll.dll and executable.

8. Conclusion

We analyzed existing API [de]obfuscation techniques and clarified that API name resolution becomes an attack vector for malware authors to evade malware analyses and detections depending on the API de-obfuscation techniques. We presented Stealth Loader as a proof-of-concept implementation to exploit the attack vector. We then demonstrated that Stealth Loader actually evaded all major analysis tools. We also qualitatively showed that Stealth Loader can evade previously proposed API de-obfuscation techniques in academic studies.

We are not arguing that Stealth Loader is perfect. However, we argue that defeating Stealth Loader is not easy because none of the countermeasures discussed in Section 7.3 can become a direct solution against Stealth Loader. We also argue that most current malware analysis tools depend more or less on some of the API de-obfuscation techniques mentioned in this paper, implying that Stealth Loader can pose a serious real-world threat in the future.

References

[1] Abadi, M., Budiu, M., Erlingsson, U. and Ligatti, J.: Control-flow Integrity, *Proc. 12th ACM Conference on Computer and Communications Security*, *CCS '05*, pp.340–353, ACM (online), DOI: 10.1145/1102120.1102165 (2005).

[2] Abrath, B., Coppens, B., Volckaert, S. and De Sutter, B.: Obfuscating Windows DLLs, *2015 IEEE/ACM 1st International Workshop on Software Protection* (*SPRO*), pp.24–30, IEEE (2015).

[3] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *USENIX Annual Technical Conference, FREENIX Track*, pp.41–46, USENIX (2005).

[4] Cavallaro, L., Saxena, P. and Sekar, R.: On the Limits of Information Flow Techniques for Malware Analysis and Containment, *Proc. 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, *DIMVA '08*, pp.143–163, Springer-Verlag (2008).

[5] Fewer, S.: Reflective DLL Injection, available from ⟨http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf⟩ (accessed 2016-05-16).

[6] Henderson, A., Prakash, A., Yan, L.K., Hu, X., Wang, X., Zhou, R. and Yin, H.: Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform, *Proc. 2014 International Symposium on Software Testing and Analysis*, *ISSTA 2014*, pp.248–258, ACM (online), DOI: 10.1145/2610384.2610407 (2014).

[7] Hex-Rays: Hey-Rays (online), available from ⟨https://www.hex-rays.com/⟩ (accessed 2017-08-17).

[8] Hunt, G. and Brubacher, D.: Detours: Binary Interception of Win32 Functions, *3rd USENIX Windows NT Symposium*, p.8, USENIX (online), available from ⟨http://research.microsoft.com/apps/pubs/default.aspx?id=68568⟩ (1999).

[9] Kawakoya, Y., Iwamura, M., Shioji, E. and Hariu, T.: API Chaser: Anti-analysis Resistant Malware Analyzer, *Proc. 16th International Symposium Research in Attacks, Intrusions, and Defenses, RAID 2013*, pp.123–143 (2013).

[10] Korczynski, D.: RePEconstruct: Reconstructing binaries with self-modifying code and import address table destruction, *11th International Conference on Malicious and Unwanted Software, MALWARE 2016*, pp.31–38 (online), DOI: 10.1109/MALWARE.2016.7888727 (2016).

[11] Kruegel, C., Kirda, E. and Moser, A.: Limits of Static Analysis for Malware Detection, *Proc. 23rd Annual Computer Security Applications Conference* (*ACSAC 2007*) (2007).

[12] Leitch, J.: Process Hollowing (online), available from ⟨https://github.com/m0n0ph1/Process-Hollowing⟩.

[13] Liberman, T. and Kogan, E.: Lost in Transaction: Process Doppelganging, Black Hat EU Briefings (2017).

[14] Ligh, M.H., Case, A., Levy, J. and Walters, A.: *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, Wiley Publishing, 1st edition (2014).

[15] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J. and Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '05*, pp.190–200, ACM (online), DOI: 10.1145/1065010.1065034 (2005).

[16] Ma, W., Duan, P., Liu, S., Gu, G. and Liu, J.-C.: Shadow attacks: Automatically evading system-call-behavior based malware detection, *Journal in Computer Virology*, Vol.8, No.1, pp.1–13 (online), DOI: 10.1007/s11416-011-0157-5 (2012).

[17] Microsoft: Control Flow Guard, available from ⟨https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx⟩ (accessed 2018-02-06).

[18] NtQuery, available from ⟨https://github.com/NtQuery/Scylla⟩ (accessed 2017-08-17).

[19] Oktavianto, D. and Muhardianto, I.: *Cuckoo Malware Analysis*, Packt Publishing (2013).

[20] Payer, M., Hartmann, T. and Gross, T.R.: Safe Loading - A Foundation for Secure Execution of Untrusted Programs, *Proc. 2012 IEEE Symposium on Security and Privacy*, *SP '12*, pp.18–32, IEEE Computer Society (online), DOI: 10.1109/SP.2012.11 (2012).

[21] Plohmann, D. and Hanel, A.: simpliFiRE.IDAScope, Hacklu (2012).

[22] Quist, D., Liebrock, L. and Neil, J.: Improving antivirus accuracy with

hypervisor assisted analysis, *Journal in Computer Virology*, Vol.7, No.2, pp.121–131 (online), DOI: 10.1007/s11416-010-0142-4 (2011).
[23] Raber, J. and Krumheuer, B.: QuietRIATT: Rebuilding the Import Address Table Using Hooked DLL Calls, Black Hat DC Briefings (2009).
[24] Rekall, available from ⟨http://www.rekall-forensic.com/⟩ (accessed 2017-12-19).
[25] Russinovich, M.: Sysinternals (online), available from ⟨https://docs.microsoft.com/en-us/sysinternals/downloads/vmmap⟩ (accessed 2018-01-29).
[26] Sharif, M.I., Yegneswaran, V., Saidi, H., Porras, P.A. and Lee, W.: Eureka: A Framework for Enabling Static Malware Analysis, *ESORICS*, Jajodia, S. and Lopez, J. (Eds.), Lecture Notes in Computer Science, Vol.5283, pp.481–500, Springer (2008).
[27] Srivastava, A., Lanzi, A., Giffin, J. and Balzarotti, D.: Operating system interface obfuscation and the revealing of hidden operations, *DIMVA 2011, 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Lecture Notes in Computer Science*, Vol.6739/2011, pp.214–233 (online), available from ⟨http://www.eurecom.fr/publication/3459⟩ (2011).
[28] Stewart, A.: DLL SIDE-LOADING: A Thorn in the Side of the Anti-Virus Industry (2014).
[29] Suenaga, M.: A Museum of API Obfuscation on Win32, Symantec Security Response (2009).
[30] Sycurelab, available from ⟨https://github.com/sycurelab/DECAF⟩ (accessed 2016-05-16).
[31] Tools, H.: Heaven Tools (online), available from ⟨http://www.heaventools.com/remove-debug-information.htm⟩ (accessed 2018-01-29).
[32] VirusTotal, available from ⟨https://www.virustotal.com/⟩ (accessed 2016-11-17).
[33] Yason, M.V.: The Art of Unpacking, Black Hat USA Briefings (2007).
[34] Zynamics, available from ⟨https://www.zynamics.com/bindiff.html⟩ (accessed 2016-05-17).

Appendix

A.1 The Reasons for Unsupported API

In this Appendix, we explain the reasons we cannot support several APIs with Stealth Loader on the Windows 7 platform.

A.1.1 ntdll Initialization

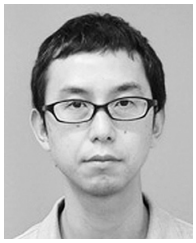
ntdll.dll does not export the initialize function, i.e., DllMain does not exist in ntdll.dll, and LdrInitializeThunk, which is the entry point of ntdll.dll for a newly created thread, is also not exported. This inability of initialization leads to many uninitialized global variables, causing a program crash. As a workaround to this, we classified the APIs of ntdll.dll as whether they are dependent on global variables by using static analysis. We then defined the APIs dependent on global variables as unsupported. As a result, the number of supported APIs for ntdll.dll is 776, while that of unsupported APIs is 1,992.

A.1.2 Callback

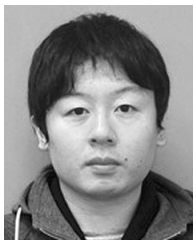
APIs triggering callback are difficult to apply Stealth Loader to because these APIs do not work properly unless we register callback handlers in the PEB. Therefore, we exclude some of the APIs of user32.dll and gdi32.dll, which become a trigger callback from our supported APIs. To distinguish whether APIs are related to callbacks, we developed an IDA script to make a call flow graph and analyzed win32k.sys, user32.dll, and gdi32.dll using the script. We then identified 203 APIs out of 839 exported from user32.dll and 202 out of 728 exported from gdi32.dll.

A.1.3 Local Heap Memory

Supporting APIs to operate local heap objects is difficult because these objects are possibly shared between DLLs. The reason is as follows. When a local heap object is assigned, this object is managed under the stealth-loaded kernelbase.dll. However, when the object is used, the object is checked under the Windows-loaded kernelbase.dll. This inconsistency leads to failure in the execution of some APIs related to the local heap object operation. To avoid this situation, we exclude the APIs for operating local heap objects from our supported API. As a result of static analysis, we found that local heap objects are managed in BaseHeapHandleTable, located in the data section of kernelbase.dll. Therefore, we do not support six APIs depending on this table in the current Stealth Loader.



Yuhei Kawakoya received his B.E. and M.S. in science and engineering from Waseda University in 2003 and 2005, respectively. He has been engaged in R&D since 2005 on computer security. From 2013 to 2016, he was engaged in R&D of NTT Innovation Institute, Inc. as a software engineer. He is a member of IPSJ and IEICE.



Eitaro Shioji received his B.E. in computer science and M.E. in communications and integrated systems from Tokyo Institute of Technology in 2008 and 2010, respectively. Since joining NTT in 2010, he has been engaged in R&D on computer security. He is a member of IEICE.

Yuto Otsuki received his B.E. degree from College of Information Science and Engineering, Ritsumeikan University in 2011, and his M.E. degree from Graduate School of Science and Engineering, Ritsumeikan University in 2013. He also received his D.Eng. degree from Graduate School of Information Science and Engineering, Ritsumeikan University in 2016. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2016, he has been engaged in research of malware analysis and digital forensics. He is now with the Cyber Security Project of NTT Secure Platform Laboratories.

Makoto Iwamura received his B.E., M.E., and D.Eng. in science and engineering from Waseda University, Tokyo, in 2000, 2002, and 2012, respectively. He joined NTT in 2002. He is currently with NTT Secure Platform Laboratories, where he is engaged in the Cyber Security Project. His research interests include reverse engineering, vulnerability discovery, and malware analysis.

Jun Miyoshi received his B.E. and M.E. degrees in system science from Kyoto University in 1993 and 1995, respectively. Since joining NTT in 1995, he has been researching and developing network security technologies. From 2011 to 2016, he was engaged in R&D strategy management of NTT Secure Platform Laboratories. Now he is a research group leader of Cyber Security Project in the Laboratories. He is a member of IEICE.