Taint-based Analysis Techniques against Evasive Malware

解析回避機能を持つマルウェアの解析手法：テイント伝播によるアプローチ

February, 2019

Yuhei KAWAKOYA

川古谷 裕平

Taint-based Analysis Techniques against Evasive Malware

解析回避機能を持つマルウェアの解析手法：テイント伝播によるアプローチ

February, 2019

Waseda University

Graduate School of Fundamental Science and Engineering

Yuhei KAWAKOYA

川古谷 裕平

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Over the past decade, malware threats have become a serious Internet problem. Malicious Internet activities such as massive spam emailing and denial-of-service attacks have arisen from botnets comprising a large number of malware-infected machines. Botnets are used as an infrastructure for malicious activities and can be rented by the hour in black markets to anyone who wants to perform these kinds of activities. In addition, crypto-mining malware has recently appeared. It implicitly abuses computer resources such as memory or CPU power for cryptocurrency mining without being explicitly permitted by the user of the computer. Owners of crypto-mining malware can make money directly through this cryptocurrency mining. This money becomes a source of revenue for criminal organizations. Similar to the above, there is an ecosystem on the Internet between malware authors, malicious service providers, and service users with various black markets to connect them. Malware plays the central role in this unwanted Internet ecosystem. Due to this, many malware-defense experts spend their resources on fighting malware. A very effective and straight-forward approach is analyzing malware. By analyzing the malware, these defenders can clarify the hidden behaviors or intentions of malware and then develop effective countermeasures against it. Thus, malware analysis techniques are a key factor in fighting threats on the Internet.

There are mainly two types of malware analysis techniques: dynamic analysis and static analysis. Dynamic analysis is a technique that executes malware in an isolated environment in which tools for monitoring have been installed in advance. More specifically, there are several types of dynamic analysis techniques such as system call monitoring, user-land application programming interface (API) monitoring, or instruction tracing. Among these, API monitoring is a very popular technique because APIs have high-context, and are sufficiently well-documented and human-readable for analysts to understand quickly malware activities and intentions.

The other line of research for malware analysis is static analysis. Static analysis is a technique that analyzes the code of malware without executing it. Since static analysis does not execute malicious code of malware, there is no risk of damaging the environment. It also has an advantage in its completeness, which is a serious weakness of dynamic analysis. That is, we can theoretically analyze all codes of malware and extract all potential behaviors including those that may be unseen using dynamic analysis. However, static analysis has a problem in its scalability. That is, it is time-consuming and may not be practical when analyzing a long code. An approach to mitigating this weakness is to focus on APIs imported by malware. Since APIs contain much information, as mentioned above, they become an essential data source in the efficient progress of static analysis. In short, APIs are a key factor in both dynamic and static malware analysis techniques. We call these dynamic and static analyses focusing on APIs *API-oriented* analyses in this thesis.

Since malware developers are also familiar with malware analysis techniques and recognize the importance of APIs in the analysis process, they embed anti-analysis functions into their malware to evade API monitoring in dynamic analysis or hide its imported APIs in static analysis [1][2][3][4][5][6][7][8][9]. Various techniques that evade API-oriented analyses have currently been adopted in real-world malware. These techniques are mainly classified into four types: target evasion, hook evasion, code obfuscation, and resolution evasion. Target evasion is used to obfuscate the API caller instance by, for example, invoking APIs from the code injected into a benign process. Hook evasion is a technique that evades hooks added to the entry of APIs for monitoring. Code obfuscation is a technique that encodes the code and

data of a malware executable to avoid pattern-matching based detections. Resolution evasion is a technique that obfuscates the relationship between code or data, and their symbol names.

As far as we have considered, a reason why malware has a chance to evade analyses is the *target-gap* problem, which is a common design problem existing in analysis tools. The target-gap problem is a gap between what we really want to analyze and what we actually analyze. That is, existing analysis techniques for specific target codes are built on indicators of the target codes that express the existence of the target codes, even though they should target the codes themselves. An example is when we analyze malware using dynamic analysis techniques, we mostly identify the executions of the target codes based on the process identifier (PID) or thread identifier (TID) of an instance of the malware, even though what we really want to analyze is the code of the malware. Another example is that when we capture the executions of a specific API, we identify the execution of the API based on that of the address where the API should be loaded, even though what we really want to capture is the executions of the code of the API. In this way, there is a gap between what we really want to analyze and what we actually analyze using existing analysis tools. Malware takes advantage of this gap to evade analyses.

1.2 Thesis Contributions

The goal of this thesis is to establish malware analysis techniques that are sufficiently practical for fighting evasive malware. To this end, this thesis focuses on the target-gap problem and extends the capabilities of existing dynamic and static malware analysis techniques with taint tracking to solve the problem. In particular, this thesis first presents one new evasion technique that shows clearly the target-gap problem by actually exploiting it using a proof-of-concept code. Next, we propose two new analysis techniques for dynamic and static analyses that provide a definitive collection of practical datasets. We argue that the effectiveness of the proposed techniques using real-world large-scale datasets and then discuss issues that should be addressed in the future.

Problem Description is presented in Chapter 2. In this chapter, we first explain basic malware analysis procedures, especially those focusing on APIs, i.e., API-oriented analyses. Next, we classify existing evasion techniques often used in real-world malware into four groups: hook evasion, target evasion, code obfuscation, and resolution evasion. Then, we separately overview the literature on dynamic and static malware analyses. Finally, we discuss the target-gap problem that exists in both dynamic and static analysis techniques.

A New Threat: Trace-Free Program Loader is presented in Chapter 3. The goal of this chapter is to show the target-gap problem. To this end, we present *Stealth Loader*, which is a program loader that bypasses all existing API-oriented analysis techniques. The core idea of Stealth Loader is to load a dynamic link library (DLL) to memory and resolve its dependency without leaving any detectable trace in the memory. We show the effectiveness of Stealth Loader by analyzing a set of Windows executables and malware protected with Stealth Loader using several tools in which major dynamic and static analysis techniques are implemented. The results indicate that among the other considered evasion tools only Stealth Loader successfully bypasses all analysis tools.

Taint-Assisted Dynamic Malware Analysis is discussed in Chapter 4. The goal of this chapter is to present a solution for the target-gap problem in dynamic analysis. To achieve this goal, we introduce the design and implementation of an API monitoring system called *API Chaser*, which is resistant to evasive malware. The core technique in API Chaser is *code tainting*, which enables us to identify precisely the execution of monitored instructions by propagating three types of taint tags added to the codes of API, malware, and benign executables. Additionally, we introduce *taint-based control transfer interception*, which is a technique that captures precisely API calls invoked from evasive malware. We evaluate API Chaser based on several real-world and synthetic malware to show the accuracy of our API hooking technique. We also perform a large-scale malware experiment by analyzing 8,897 malware samples to show the practical capability of API Chaser. These experimental results show that 571 of 8,897 malware samples employ hook evasion techniques to hide specific API calls, while 344 malware samples use target evasion techniques to

hide the source of API calls.

Taint-Assisted Static Malware Analysis is introduced in Chapter 5. The goal of this chapter is to present a solution for the target-gap problem in static analysis. For that purpose, we introduce *taint-based API name resolution*, which is an API name resolution technique in import address table (IAT) reconstruction based on taint analysis to defeat semantic evasion techniques, which are among the group of resolution evasion techniques. The key idea behind the proposed technique is that we add rich semantics information to the machine code using taint tags. Specifically, we make instructions recognizable using taint tags that were set to the instructions of each API before starting the analysis. To accomplish this, we first define taint tags where each tag has a unique value for each API and then apply the taint of the API to each of its instructions. Next, we track the movement of the API instructions by propagating the tags and then resolve API names from the propagated tags for IAT reconstruction after acquiring a memory dump of the process under analysis. Finally, we experimentally show that a system in which taint-based API name resolution has been implemented enables us to identify correctly imported APIs even when malware authors apply various evasion techniques to their malware.

Finally, Chapter 6 presents the conclusions of this thesis.

Chapter 2

Problem Description

2.1 Introduction

In this chapter, we introduce the *target-gap* problem, which is a design problem that commonly exists in current dynamic and static analysis techniques. To explain this problem, we first review the procedures of basic API-oriented dynamic and static analysis techniques. Then, we introduce a taxonomy of evasion techniques that malware often use to defeat analysis techniques, and describe the details of each evasion technique. Next, we show literature related to dynamic and static malware analyses and qualitatively evaluate their resistance capabilities against evasion techniques. Finally, we discuss the target-gap problem by giving several examples of situations in which difficulties could arise in analyzing malware using current techniques.

2.2 API-*oriented* Analysis

In this section, we explain basic API-oriented analyses. API-oriented analyses represent an analysis approach that primarily focuses on APIs in dynamic and static analyses to enable these analyses to progress efficiently. APIs are high-context, well-documented, and human-friendly. They become important hints for analysts to help them quickly understand the code under analysis. On the other hand, machine instructions such as x86 are low-context and machine-friendly. It is difficult for

humans to grasp the meaning or intention of the code from only those machine instructions. So, APIs are a key factor in determining the precision and efficiency of malware analyses including when we manually perform analysis. In this section, we first discuss API monitoring as a representative of dynamic analysis techniques and then describe IAT reconstruction as a representative of static analysis techniques.

2.2.1 API Monitoring

API monitoring is a technique that analyzes malware by executing it in an isolated environment and monitoring API calls invoked from it. To perform API monitoring, we need to consider the following three phases: target, hook, and resolution. The target phase distinguishes the target instances from others since many instances may be running in the same analysis environment when performing API monitoring. To accomplish this, we are likely to rely on a process identifier (PID), control register 3 (CR3), which is an equivalent to PID, or thread identifier (TID). Specifically, we first make a list of PIDs as monitoring targets and provide the list to an analysis environment to identify targets. Then, we manage the list by adding or removing the PID of the newly-created process or terminated processes, respectively, at runtime. When we perform an analysis at the process-level, e.g., using a process-level sandbox, the target phase is not required because we have already identified the target process. Therefore, we can skip this phase in such a case.

The hook phase captures executions of specific API instructions. To accomplish this, there are several techniques such as those involving data modifications or code modifications, address matching, and DLL hijacking. The first technique involves data modifications. We hook the API executions by modifying the entry of the API in a function table, which contains pointers to each API. In a Windows environment, an IAT and export address table (EAT) represent two major targets for hooking modifications. These two tables contain virtual addresses or offsets to each imported or exported API, respectively. By replacing the entry to a specific API in one of these tables with a pointer to a prepared handler, we can hook the calls to the API and transfer the executions to the prepared handler. A traditional technique involving code modifications is to set a breakpoint, i.e., software breakpoint, at a

specific memory address, which internally replaces the instruction at the address with `0xcc`, and then capture an exception generated when the `0xcc` instruction is executed at the CPU. Another technique with code modifications is an inline-hook, which replaces the instruction at the address with a `jmp` instruction. This `jmp` instruction transfers the execution to a prepared hook handler. Then, the execution returns to the instruction immediately after the `jmp` instruction after the handler has completed its task.

A technique that does not involve any modification is address matching. With address matching, we capture the executions of target instructions based on a comparison of the instruction addresses to the expected addresses. More specifically, address matching works as follows. It first calculates the addresses where target API instructions are expected to be loaded into memory in advance, monitors the executions of the addresses with one of any instrumentation techniques during dynamic analysis, and compares each executed address to the calculated one. If the addresses match, this fact indicates that one of the target APIs is being executed. Address matching is often used in a malware analysis environment with virtual machine (VM) introspection techniques [10][11][12][13][14] because this technique does not require modification of any malware code or data. These modifications often expose the existence of analysis modules to malware, and results in malware stopping or changing malware behavior. Address matching has a low risk of detection by malware due to modified parts of code or data for hooking.

One more technique that should be mentioned here is DLL hijacking [15]. DLL hijacking injects a fake DLL into a target process by manipulating the order of the search strategy for a DLL to be loaded by a Windows program loader. The fake DLL should export the APIs in the same manner as those of the legitimate DLL. The APIs exported from the legitimate DLL are called via the injected fake DLL. This technique does not directly modify the code of a target process, i.e., malware. However, a suspicious DLL, i.e., a fake DLL, must be located in the same process memory space as running malware. So, the existence of the fake DLL may become a trigger for malware to find analysis instances.

The resolution phase adds semantics, i.e., symbols, to a memory address that is

captured in the hook phase. To accomplish this, we first generate a correspondent table of API names and their loaded addresses. To calculate the loaded addresses of each API, we add the base address of a loaded DLL with the relative virtual addresses (RVAs) of each API exported from the DLL. The base address of a loaded DLL is acquired from OS-managed data areas such as the process environment block (PEB) or virtual address descriptor (VAD). Another approach for acquiring the base address of loaded DLLs is to make use of OS-provided functions with dynamic analysis. For example, Windows OSs have a callback mechanism that invokes registered handlers whenever a specific type of event happens such as DLL loading or unloading. This callback mechanism allows us to collect the base address of a loaded DLL. One more example is to monitor API calls related to DLL management such as LoadLibrary or LoadLibraryEx. By hooking the calls of these two APIs and investigating their arguments before and after the invocations, we can collect the name of a loading DLL, which is passed to an API call as an argument, and its loaded address, which is returned as a result of the API call. Regarding the RVAs of each API, they are mostly acquired from the portable executable (PE) header of a DLL. After completing the corresponding table, we can search the table for the memory address captured in the hook phase. If the entry is found in the table, it reruns the correspondent API name as the symbol of the address.

2.2.2 IAT Reconstruction

IAT reconstruction is a technique that clarifies the APIs imported by a specific target instance as a preprocess for static analysis. A PE executable usually has IATs and import name tables (INTs) to manage the imported APIs from each DLL; these tables are pointed to from the PE header. Malware often removes the list of imported APIs from its PE header to disturb analyses. Specifically, it deletes INTs and de-links these tables from the PE header. Therefore, we must repair the destroyed parts of the PE header to know the imported APIs. IAT reconstruction is a necessary step for efficient static analysis because imported APIs add semantics to inorganic machine codes or byte sequences, which are difficult to read as they are. The procedure for IAT reconstruction also comprises three phases: target, find table, and resolution.

The target phase is necessary only when we analyze the code that exists in a memory dump such as a raw memory dump, which is a snapshot of the physical memory at a particular execution point during dynamic analysis. When we examine a memory dump, we first need to identify the target code regions in the memory dump because a memory dump is likely to contain the processes of many executable files that are not directly related to the target one. To identify the code regions, we often use the `process name` or `PID` to identify the specific instances to analyze. For example, if we provide a process name or PID to a forensics tool such as The Volatility Framework (Volatility) [16], we could know where the code regions of the process are in a memory dump. On the other hand, when we analyze the code of an executable file, we have already identified our targets, i.e., the code regions of the executable file. So, we can skip this phase.

The find-table phase identifies function tables, i.e., IATs, in the target code or data regions. These tables are used to import APIs exported from other parts of regions, i.e., DLLs mapped to other memory regions. To find tables, a basic technique that is adopted by many tools such as impscan (a plugin for Volatility) and Scylla [17] is described hereafter. First the code regions of a target process are disassembled to search for indirect call instructions such as `call [0x1001000]`. Next memory addresses that are referred to by the found indirect call instructions are collected. In the above case, `0x1001000` is the address to collect. If the collected addresses are gathered into a cluster within a specific memory address range, we define the memory area within the range as a function table.

The resolution phase in IAT reconstruction is the same as that in dynamic analysis. We resolve the destination address of a pointer in a found IAT to the corresponding API name by following the same process as described in the previous subsection.

2.3 Taxonomy of Evasion Techniques

The main purpose of this section is to classify existing evasion techniques into four categories by considering the target phase of each technique. However, before diving into individual evasion techniques, we first consider an outline for anti-analysis

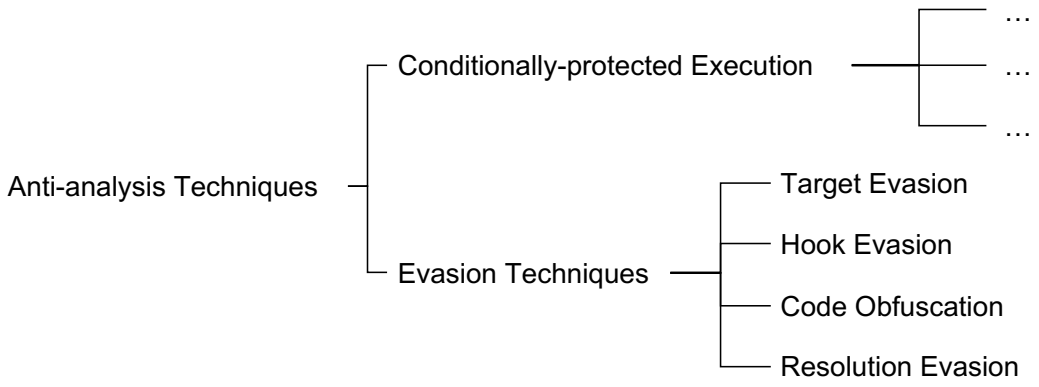

Fig. 2.1 Anti-analysis Overview

techniques.

2.3.1 Overview of Anti-Analysis

Anti-analysis techniques are used to hinder efficient analyses or reverse engineering. Malware often employs some of these techniques to bypass detection and analysis. In this thesis, we first classify them into the following two categories as shown in Fig. 2.1: conditionally-protected execution and evasion techniques. Conditionally-protected execution imposes several conditions on environments where malware is running for the malware to behave normally. Examples of techniques belonging to this category are VM detections [18][19], debugger detections [2], or time bombs [20]. Additionally, conditions are not limited to those that are computer related. They include those related to external environments such as the connectivity to a specific server on the Internet, or those related to humans such as the frequency of clicks on the screen. There is a large volume of research on techniques in this category and many great results have been obtained. Therefore, we simply use these research results in this thesis and consider that this type of anti-analysis technique is outside the scope of this thesis as a research subject.

Evasion techniques are used to hide the execution of instructions of an API from analysis instances even though they were actually executed. With these techniques,

Table 2.1 Taxonomy of Evasion Techniques

Phase		API Monitoring	IAT Reconstruction
Target		Code Injection, File Infection, (Name Confusion)	
Hook		Stolen Code, Sliding Call, Copied API Obfuscation	N/A
Find-Table		N/A	Code Obfuscation
Resolution	Semantics	Name Confusion, DLL Unlinking, Static Linking	
	Control-flow	N/A	API Redirection, (Stolen Code, Sliding Call, Copied API Obfuscation)

malware could conduct malicious activities without being detected and analyzed on an infected computer. Evasion techniques do not stop the execution of malware, while conditionally-protected execution possibly stops malware execution when the conditions are not satisfied. That is, users of analysis tools are likely not to recognize that they fail to analyze the malware and may miss capturing a part of a malicious behavior from the malware. This oversight may lead to more severe damage than simply failing to analyze malware.

Table 2.1 shows a taxonomy of evasion techniques. We classify evasion techniques into four groups based on the phase of each API-oriented analyses. There are three groups to mimic API monitoring, target evasion, hook evasion, and resolution evasion, and three groups to bypass IAT reconstruction, target evasion, code obfuscation, and resolution evasion. Moreover, we divide resolution evasion techniques into two sub-groups based on the steps for resolution: semantic evasion and control-flow obfuscation.

2.3.2 Target Evasion

Target evasion is a technique that enables an instance, such as process or thread, to evade being the target of analysis. We describe two target evasion techniques: code injection and file infection.

Code injection injects a piece of malicious code into another process and enables that code to be executed in that process. If an API monitor distinguishes its monitoring target based on a PID, which is very common in most existing systems

[21][14][22][23][24], that API monitor must find the behaviors for the injection and add the injected process to the monitoring targets. If it does not find the code injection behaviors, it fails to monitor correctly the API calls invoked from the malicious code in the injected process. That is, API monitoring is evaded.

File infection is another target evasion technique. It adds a piece of code to an executable file and modifies pointers in its PE header to cause the added code to execute after the program begins to run. Similar to code injection, it is difficult to distinguish between API calls from malicious code and those from the original benign code if the API monitor tries to identify its target using PIDs.

2.3.3 Hook Evasion

Hook evasion is a technique that causes executions of specific instructions, which may be monitored by an analysis instance, to be skipped or avoided in order to prevent triggering a hook. A basic hook evasion technique is dynamic API resolution, which is a technique used to evade IAT hooking. When attackers create malware, they drop imported API information from the PE header of the malware executable file and let it resolve its API dependencies at runtime by itself using the LoadLibrary and GetProcAddress APIs. Using this technique, malware is not required to go through any IAT to call external APIs. Thus, malware could bypass monitoring hooks set in the IAT, i.e., IAT hooking, using this technique.

Another basic technique is self API resolution, which is a technique used to evade analysis that monitors the LoadLibrary API calls to install hooks on the loading DLL. This technique acquires the base address of a loaded DLL from a PEB instead of calling the LoadLibrary API and then parses the PE header of the DLL to find the offset of a specific API exported by the DLL without depending on the GetProcAddress API. This technique is often seen in a shellcode [25][26] since a shellcode must resolve API dependencies by itself to run in a situation where it can use neither the LoadLibrary nor GetProcAddress API. Malware applies this technique to avoid analysis. Since malware does not call the LoadLibrary and GetProcAddress APIs, they can bypass analysis instances that monitor the calls of the two APIs that trigger install hooks on newly loading DLLs.

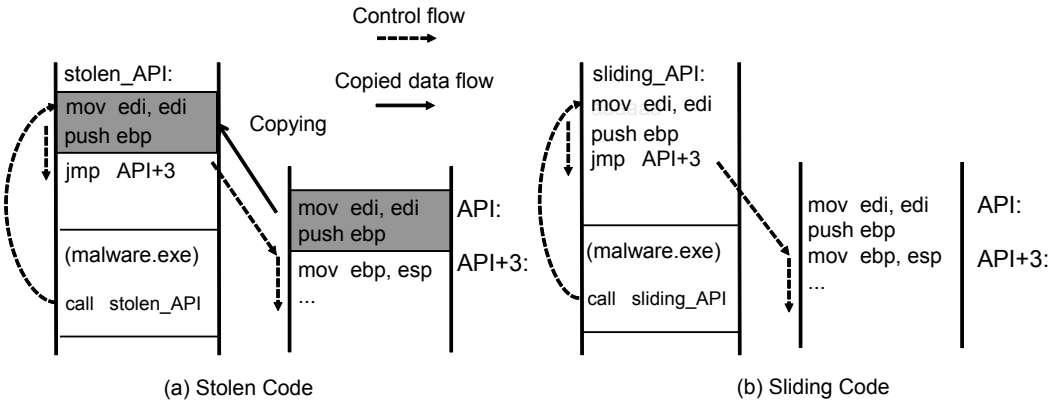

Fig. 2.2 Stolen Code and Sliding Call Mechanism

In addition to the above two basic techniques, we describe three advanced hook evasion techniques: stolen code, sliding call, and copied API obfuscation. Fig. 2.2(a) shows the behavior of stolen code. Stolen code copies some instructions from the entry of an API to allocated memory areas in the malware process at runtime. When malware attempts to call the API, it first executes the copied instructions and then jumps to the address of the instruction in the API following the copied instructions. Some existing API monitors [14][22][21][24] identify their target API calls using the address-matching technique. The expected addresses for matching are computed from the base address of the loaded module containing these APIs and the offsets to them, which are written in the PE header of the module. If the instructions of these APIs are copied to addresses different than the originals, existing API monitors may miss capturing the execution of these APIs because the calculated addresses, i.e., the ones where target APIs should reside, are not executed.

Fig. 2.2(b) shows the behavior of sliding call. Sliding call behaves almost in the same manner as stolen code. The difference is that malware originally has a few instructions of the entry of a specific API in its body and calls the API after executing those instructions. Almost all existing API monitors focus on the entry of each API [27][21][14][22][24] as a place to add a hook. This approach toward hooking can be evaded because the instruction at the head of the API is not executed

nor even touched by malware using sliding call.

Copied API obfuscation [1] is an evolved version of stolen code. It copies all instructions of an API to an allocated memory area in the malware process at runtime. Unlike stolen code, copied API obfuscation does not transfer the execution to the instructions of the copied API, i.e., it does not execute them at all. So, if analysis tools add a hook to the entry of an API, they fail to capture the API calls since no instruction of the API is executed at all.

2.3.4 Code Obfuscation

Code obfuscation is a technique that hides the existence of specific types of instruction and data from static analysis by encoding them [28]. It destroys the appearances and patterns of code and data by encoding those of a malware executable. When the encoded executable begins to run, it decodes the encoded code and data, and then writes them to a buffer to execute and reference, respectively.

We do not consider this evasion technique in this thesis because of the following two reasons. First, the primary purpose of code obfuscation is to evade only static detections such as pattern matching. So, dynamic analysis approaches, e.g., API monitoring, are not affected by this type of evasion. Second, there is much-related research on this technique such as unpacking or de-obfuscation [29][30][31][32]. Therefore, we use the existing research findings and results to extract the decoded code and data of malware for this research in static analysis, i.e., for IAT reconstruction.

2.3.5 Resolution Evasion

Resolution evasion is a technique that breaks the relationship between bytes in a computer, e.g., data in the memory, and their symbol names for hiding imported or invoked APIs from static or dynamic analysis tools, respectively. There are two types of resolution evasion techniques: control-flow obfuscation and semantic evasion.

Control-flow obfuscation is a technique that inserts junk code between an API call site, i.e., an indirect call instruction, and the API code. Usually, the control flow

of an API call site directly points to the code of the API or it is directed via an IAT. However, malware authors inject junk code between them and modify the destination of the API call sites to point to the junk code. Since the call site results in pointing to the junk code, it does not appear as an API call site. It simply appears as a local function call site. API monitoring and IAT reconstruction are not concerned with local function calls. Therefore, these analyses could be evaded using this technique. We present API redirection as a technique belonging to this type. Also, we explain hook evasion techniques here because they could indirectly affect control flow as a result of taking actions for hook evasion.

API redirection [2] is a technique that attacks static analyses by obfuscating API references. As Fig. 2.3(a) shows, it modifies call instructions in the original code. Otherwise, as Fig. 2.3(b) shows, it modifies the IAT entry. With these modifications, it forces control flows to APIs to circumvent a stub, which executes junk instructions and finally jumps to the 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 refer directly to the API code or at least be directed via an IAT entry in many analysis tools, this technique can confuse their expectations regarding the relationship between the API caller and callee. Additionally, advanced API redirection, as shown in Fig. 2.3(c), is involved with stolen code [2]. At the same time, when API redirection is applied, it copies some instructions at the entry of an API, i.e., `mov edi, edi`, and `push ebp`, to a 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`.

Hook evasion techniques such as stolen code, sliding call, and copied API obfuscation also disturb control flow recovery in static analysis for resolution. The primary effect of these techniques is to avoid the monitored instruction, while a secondary effect is to change the control flow, which is started at an API call site and reaches to the corresponding API code. As a result of this change, the call destination of the API call site becomes a local buffer and not the memory area where the API code resides. Under this memory layout, when we perform address

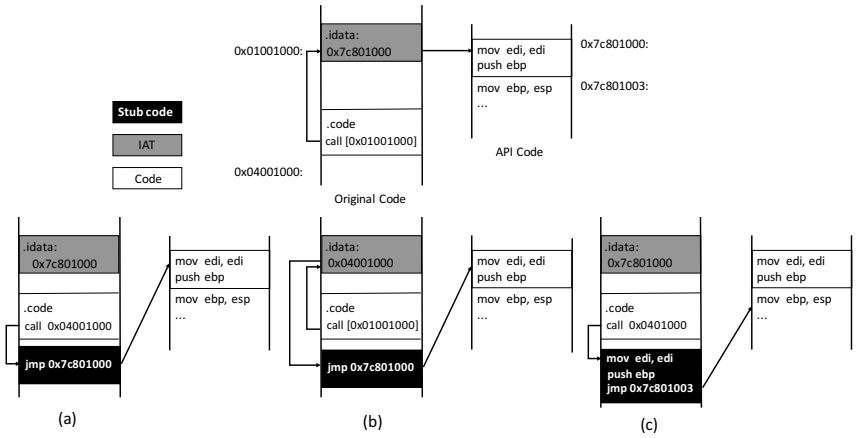

Fig. 2.3 Three patterns of API redirection. Top is a normal Windows executable before applying API redirection. (a) Pattern in which the reference of the call instruction is modified, (b) that in which the entry of the IAT is modified, and (c) that in which API redirection is conducted with stolen code.

comparison to identify the existence of a specific API, the address of the local buffer does not match that where the API is expected to reside. As a result, API name resolution fails.

Semantic evasion is a technique that hides the location information of specific data that store important system objects. In this thesis, semantics means the same thing as the positions, i.e., the virtual memory address, of specific targets, such as APIs exported from each loaded system DLL. Analysis tools construct the semantics of an analysis environment by collecting information from an OS by querying it via APIs or system calls, parsing specific data structures, or finding variables that are managed by the OS. However, when malware successfully interferes with collection by hiding the data structures or modifying returned values from the OS with semantic evasion, malware can control the view of semantics to be read by analysis tools. Thus, this makes analysis tools incorrectly understand the current OS situation. Semantic evasion affects dynamic and static analyses. There are kernel-land and user-land approaches to evade semantics. Here, we focus on only user-land approaches because kernel-land approaches are difficult to apply to recent

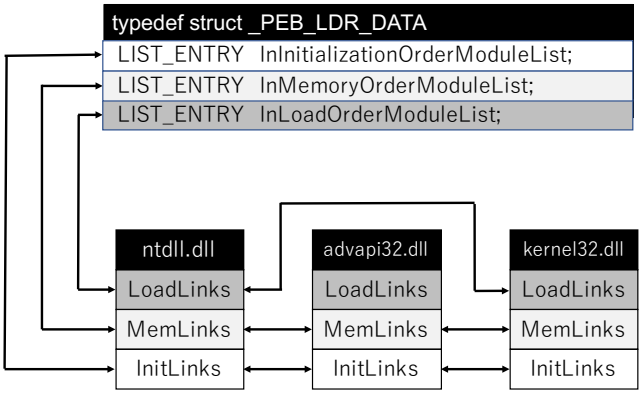

Fig. 2.4 DLL Unlinking

Windows OSs since recent Windows OSs have advanced kernel protection such as patchguard [33]. Therefore, we explain DLL unlinking, name confusion, and static linking as user-land techniques belonging to semantic evasion.

DLL unlinking [16] is a technique that interferes with static analysis by hiding the data structure that is used for managing loaded DLLs in a PEB, as shown in Fig. 2.4. DLL unlinking hides the metadata of loaded DLLs that could become the destination of the flows so that control flows from call instructions cannot reach any API. Since a control flow of an external function call cannot 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 Windows data structure 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.

Name confusion involves copying a system DLL to another file path while changing its file name. The copied DLL exports the same functions as the original DLL, so the malware loading the copied DLL can still call the same functions as those in the original DLL. If the name has been changed, some analysis systems [14][23][22][24] that depend on the names of the module to identify their target can be evaded. Also, name confusion is often used in target evasion, e.g., malware changes its name to that

of a system executable file installed as default such as svchost.exe or winlogon.exe.

Static linking is a technique that links Windows-system DLLs statically with an executable and then removes imported API information from the PE header of the linked DLL to prevent analysis tools from identifying the calls of the functions exported from the linked DLL as API calls. Since imported API information has already been removed from the linked DLLs, we cannot obtain information related to the APIs exported from the statically-linked DLL such as the name of the API. Therefore, even if we acquire an executed address in the hook phase and identify the virtual memory address in the memory area where the executable file linked with the DLL is mapped, we cannot resolve the virtual address with the corresponding API name. As a result, we fail to recognize the execution as an API call. However, statically-linked DLLs may have some troubles. They probably lose the portability as a PE executable since system DLLs tend to depend heavily on specific Windows versions. For example, it may be difficult to generate an executable with static links with kernel32.dll in one environment and run it in a different environment. Also, the size of malware linked with system DLLs is likely to become large since the size of a system DLL is not so small. Therefore, we do not consider this technique as being a real-world threat in this thesis.

2.4 Literature

In this section, we present literature related to dynamic analysis and static analysis techniques by focusing on their API-oriented analysis components such as how to identify its targets, how to hook API calls, and how to resolve the API names.

2.4.1 Dynamic Analysis: API Monitoring

API monitoring is a fundamental technique for malware analysis. Several systems have been proposed that precisely monitor malware activities based on API monitoring. We divide these systems into five categories from the viewpoint of API hooking techniques: binary rewriting, address matching, control transfer interception, simulation, and DLL hijacking. Additionally, we mention other related studies

that do not API calls, but monitor system calls or instruction executions to compare their architectures in the next subsection.

Binary Rewriting

CWSandbox [27] and Cuckoo Sandbox [24] employ the inline-hooking technique that replaces instructions at the entry of an API with a `jmp` instruction pointed to a prepared API handler, which is a function for collecting information for further investigations and then transferring the control back to the original API code. CWSandbox also set hooks on LoadLibrary and LoadLibraryEx to handle dynamic API resolution, while Cuckoo Sandbox derives the base address of a loaded DLL with a callback mechanism provided by a Windows OS. JoeBox [34] hooks APIs using a data rewriting technique, i.e., EAT hooking, that replaces a function pointer in the EAT of the PE header with the address to a prepared API handler. API Monitor [35] modifies the entries in the IAT in the PE header for hooking, i.e., IAT hooking, with the pointer to a prepared API handler. It also hooks the GetProcAddress API calls so that it sets hooks on its newly resolved API code to handle dynamic API resolution.

DRAKVUF [11] is an analysis environment built on Xen hypervisor [36]. It hooks executions of a process running in a guest OS with the breakpoint injection technique proposed in [37]. It starts analysis of a malware sample by injecting a code snippet into a process running in a guest OS from the virtual machine monitor (VMM) layer and identifies its target instance based on CR3. It acquires the semantics of a guest OS using memory forensics techniques. More specifically, it uses Rekall [38] to do that.

Address Matching

VAMPiRE [39] actualizes stealth breakpoints by making use of the virtual memory trap mechanism, i.e., disabling the present bit in a page table entry, and an original page fault handler in a kernel driver. SPiKE [40] is a framework built on VAMPiRE for performing code instrumentation. SPiKE rewrites the EAT of a DLL to redirect the execution to a prepared API handler when an API exported by the DLL is invoked. It loads analysis modules into a target process by monitoring a process or

thread creation APIs such as CreateProcess, OpenProcess, Suspend/ResumeThread. egg [41] is also a framework for binary instrumentation, and it is implemented as a kernel module. It hooks API calls with the same mechanism as VAMPiRE. A feature of egg is a taint analysis capability. egg actualizes coarse-grained taint analysis, which can track the data flow of files, virtual memory, and threats with its kernel-level instrumentation.

TTAnalyze [14] (ancestor of Anubis [12]) monitors APIs and system calls invoked from malware in the VMM layer using address matching. TTAnalyze is built on QEMU, which is a whole system emulator [42]. TTAnalyze determines target processes using CR3 that are passed from a probe module running on the guest OS and then it retrieves semantics information of a guest OS by relying on a callback mechanism that the guest OS provides. Panorama [23] is a malware analysis environment established on a whole-system emulator, TEMU [22], which is an extended version of QEMU with a taint analysis capability. Panorama is designed to analyze and detect malware based on taint tracking. Panorama, actually TEMU, hooks APIs based on address matching while capturing DLL loading events based on the callback mechanism and parsing the EAT of the load DLL for the RVA of each API exported from the DLL. It identifies its target processes based on CR3.

Control Transfer Interception

IntroLib [43] and CXPInspector [44] define API calls as control transfers from memory regions for malicious code and for API code. To intercept a control transfer, they use a similar technique to VAMPiRE. The difference between IntroLib and CXPInspector is that IntroLib relies on shadow page tables for interception, while CXPInspector relies on a hardware-assisted virtualization support feature such as Extended Page Tables or Nested Page Tables. IntroLib parses the headers of a DLL file to acquire the API names and RVAs of each API exported from the DLL and then calculates the virtual addresses of the APIs by adding the base address acquired from the memory layout to the RVA. CXPInspector uses VADs to obtain the base address of a loaded DLL and it either uses symbol information (if available) or parses the DLL PE header to acquire the RVA of each DLL API.

Simulation

Norman Sandbox [45] and Zero Wine [46] simulate Windows APIs. For that purpose, they must prepare almost the same number of API call handlers as that for Windows APIs. The handlers simulate the behaviors of a corresponding API and are used for logging. That is, when a handler is invoked, it simulates the behaviors of the API and collects the arguments passed to an API call, writes them into a log file, and returns an appropriate value to the API caller.

DLL Hijacking

BinUnpack [47] is a tool for unpacking based on the IAT rebuilt-then-called technique. It reconstructs the IAT of a packed malware based on stack tracing with API monitoring, which is the same technique as QuietRIATT [48]. API monitoring is designed to hook API calls invoked from the packed malware with kernel-level DLL hijacking. Kernel-level DLL hijacking hooks system calls that are used to load or map a DLL file into memory such as NtMapViewOfSection and replaces the mapping executable file with a fake one, while traditional DLL hijacking manipulates the order in which DLLs are searched [15].

Others

Ether [30] is a malware analysis system built on Xen hypervisor [36]. Ether monitors only system calls, and not user-land API calls with the same mechanism applied in VAMPiRE. However, it actualizes this monitoring with Intel virtualization technology. That is, it turns off the present bit of a target virtual memory page and captures page fault exceptions at the VMM layer to actualize instrumentation. Ether identifies its target process using the process name and process page directory entry. Also, it acquires the semantics of a guest OS through a series of memory reads.

Alkanet [49] is a system call monitor built on BitVisor, which is a thin hypervisor [50]. Alkanet hooks system calls by setting hardware breakpoints at the entrances and exits of system calls. It performs system call monitoring with thread-level granularity, i.e., identifies its targets based on a TID, and acquires OS semantics such as the system call number using techniques such as memory forensics.

Tartarus [51] is an analysis environment using taint analysis to track its target codes, and it is implemented on DECAF [52]. Tartarus does not monitor API calls, but it mainly monitors instruction traces. Its taint tracking capability allows Tartarus to be independent of the ways of code injections so that they can handle various types of code injections. In [51], a method was also proposed for tracking the executions of a program constructed with a return-oriented-programming (ROP) chain, which is partially used in new code injection techniques such as PowerLoaderEx [7] and AtomBombing [5].

BareBox [53] is a malware analysis system built on bare metal, i.e., it does not use any virtualization technology. BareBox runs two OSs on the same hardware. One analyzes malware within the OS and the other, Meta-OS, manages the analysis environment by taking a snapshot and restoring the state of the analysis environment after the analysis is completed. BareBox hooks system calls with system service descriptor table (SSDT) hooking in the analysis environment.

2.4.2 Static Analysis: IAT Reconstruction

There are also several tools for IAT reconstruction in static analysis. We classified them into two categories: forensics and hybrid. Forensics are tools used in a situation where there is only a memory dump of an infected computer. That is, there is no information collected using dynamic analysis techniques. The hybrid category represents tools that are used in a situation where there is an executable file. So, we can run the file to collect information using dynamic analysis techniques and then perform IAT reconstruction using the information collected during the dynamic analysis.

Forensics

Volatility [54] is a memory forensic framework that includes a set of plugins, each of which is designed to do a specific analytical job. Volatility reads a memory dump file and analyzes it while invoking some of the plugins. `impscan` is a plugin that is used for IAT reconstruction. It identifies IATs by collecting the reference address of indirect call instructions in code regions of its target process. Then, it resolves the

API name of each entry in the identified IAT by acquiring the base address of each loaded DLL from a PEB and the offset of each API exported from the DLL. Rekall [38] is a memory forensics framework cloned from Volatility. Rekall incorporates extended features such as a disk analysis capability and the ability to identify loaded DLLs based on a global unique identifier (GUID) in the PE header of an executable found in a memory dump.

Hybrid

Eureka [55] is a framework for malware unpacking and reverse engineering. It takes several techniques to identify API references from target codes using both dynamic and static analyses. When a DLL is loaded at its standard virtual address, it calculates the positions of each API exported from the DLL whose position is calculated from the `image base` address in the DLL PE header and the RVA stored in the EAT in the header. When a DLL is loaded at a non-standard virtual address, it identifies the base addresses of each loaded DLL by monitoring the NtOpenSection and NtMapViewOfSection system service calls during dynamic analysis. Additionally, it performs in-depth static analysis such as control flow graph recovery and a partial data-flow analysis to identify the targets of call sites found in code regions in static analysis.

RePEconstruct [56] is a tool built on DynamoRIO [57]. It collects instruction traces, especially focusing on the branch and indirect reference instructions. With the trace information collected during dynamic analysis and exported function information, which may be obtained from some OS-managed data structures, RePEconstruct performs IAT reconstruction. QuietRIATT [48] focuses on API monitoring for IAT reconstruction. It monitors API calls and collects the return address of each API call. After API monitoring is completed, it identifies the instruction immediately before the collected return addresses. The instructions should be API call sites and are expected to be an indirect call instruction. By collecting the reference addresses of these indirect call instructions, QuietRIATT identifies the IAT. Quist et al. [58] proposed an unpacking system that is built on Ether [30]. This system mainly focuses on the phase for finding the original entry point and repairing the PE header

of malware under analysis after extracting the original code of a packed executable. Their technique for finding the IAT is the same as that of other IAT reconstruction tools such as impscan, while that for API resolution relies on VADs to extract the file name of mapped DLLs in the target memory area. BinUnpack, which was described in the previous subsection, performs IAT reconstruction with API monitoring. The employed technique for reconstructing IAT is the same as that for QuietRIATT. It also uses rebuilt IATs to identify the original entry point of an obfuscated executable.

2.5 Problem Analysis

We introduce the target-gap problem in this section. We first summarize the architecture of existing malware analysis systems and tools that were presented in the previous section. Then we qualitatively evaluate the capability of the systems and tools to evade detection (resistance capabilities). Next, we describe the target-gap problem which represents the root cause of why malware can evade most current analysis systems and tools.

Table 2.2 Dynamic Analysis Literature Comparison

System Name	Layer	Monitor	Unit	Hook	Resolution	Target Evasion		Hook Evasion			Resolution Evasion	
						T1	T2	H1	H2	H3	R1	R2
CWSandbox[27]	User-land	API	Process	Inline-hook	EAT	✓	✓	✓			✓	N/A
Cuckoo Sandbox[24]	User-land	API	Process	Inline-hook	Callback, EAT	✓	✓	✓			✓	N/A
JoeBox[34]	User-land	API	Process	EAT hook	EAT	✓	✓	1,3	1		✓	N/A
API Monitor[35]	User-land	API	Process	IAT hook	IAT	✓	✓	1,3	1	✓	✓	N/A
DRAKVUF [11]	VMM	System Call, API	Process	BP injection	Memory Forensics	✓	✓	✓			✓	N/A
SPiKE[40]/VAMPiRE[39]	Kernel-land	API	Process	Address-matching	EAT	✓	✓	✓	✓		✓	N/A
egg[41]	Kernel-land	API	Thread	Address-matching	EAT	✓	✓	✓	✓		✓	N/A
TTAnalyze[14]	VMM	System Call, API	Process	Address-matching	Callback, EAT	✓	✓	✓	✓		✓	N/A
Panorama[23](TEMU[22])	VMM	API	Process	Address-matching	Callback, EAT	✓	✓	✓	✓		✓	N/A
IntroLib[43]	VMM	API	Process	Control Transfer Interception	Memory Forensics, EAT	✓	✓		✓		✓	N/A
CXPInspector[44]	VMM	API	Process	Control Transfer Interception	VAD, EAT	✓	✓		✓		✓	N/A
Norman Sandbox[45]	User-land	API	Process	Simulation	-	✓	✓	✓	✓		✓	N/A
Zero Wine[46]	User-land	API	Pprocess	Simulation	-	✓	✓	✓	✓		✓	N/A
BinUnpack[47]	User-,Kernel-land	API	Process	DLL hijacking	API Monitoring	✓	✓	3			✓[2]	N/A
Ether[30]	VMM	System Call	Process	Address-matching	-	✓	✓	N/A	N/A	N/A	N/A	N/A
Alkanet[49](BitVisor[50])	VMM	System Call	Thread	Hardware BP	-	✓	✓	N/A	N/A	N/A	N/A	N/A
Tartarus[51](DECAF[52])	VMM	Instruction	Code	N/A	N/A			N/A	N/A	N/A	N/A	N/A
BareBox[53]	Kernel-level	System Call	Process	SSDT hooking	N/A	✓	✓	N/A	N/A	N/A	N/A	N/A

✓indicates that the system does not successfully satisfy the requirement. – indicates that that is unknown.

T1: capability for handling a new code injection technique. T2: capability to distinguish API calls invoked from benign part of codes and malicious ones in the same process. H1: capability for detecting API calls even when the first few instructions of the API are skipped. H2: whether or not the hook mechanism is bond to virtual addresses. H3: capability for detecting API calls even when no IAT is referenced. R1: resistance against semantic evasion. R2: resistance against control-flow obfuscation.

[1] If malware identifies the address of a target API from the IAT or EAT in which the hooks for monitoring are installed, the hooks still work for hooking.

[2] The authors argue that BinUnpack is strong against state-of-the-art resolution evasion techniques. However, if the OS is compromised, BinUnpack can be evaded.

[3] If the first few instructions of the hook handler for an API are different from the ones of the API, the execution consistency is not kept when malware applies the sliding call technique for the API. Then, it may lead a program crash.

Table 2.3 Static Analysis Literature Comparison

Tool Name	Input	Target	Find-table	Resolution	Target Evasion		Code Obfuscation	Resolution Evasion	
					T1	T2		R1	R2
impscan(Volatility)[54]	Memory Dump	Process Name	Disasm.	PEB	✓	✓	N/A	✓	✓
impscan(Rekall)[38]	Memory Dump	Process Name	Disasm.	PEB, GUID	✓	✓	N/A	✓	✓
Eureka[55]	Executable	N/A	Disasm.	System Call Monitoring	N/A	N/A	N/A	✓	
RePEconstruct[56]	Executable	N/A	Insn. Trace	-	N/A	N/A	N/A	✓	✓
QuietRIATT[48]	Executable	N/A	API Monitoring	API Monitoring	N/A	N/A	N/A	✓	[1]
Quist at el.[58]	Executable	N/A	Disasm	VAD	N/A	N/A	N/A	✓	✓
BinUnpack[47]	Executable	N/A	API Monitoring	API Monitoring	N/A	N/A	N/A	✓[2]	[1]

✓indicates that the system does not successfully satisfy the requirement. - indicates that that is unknown.

R1: resistance against semantic evasion. R2: resistance against control-flow obfuscation.

[1] API monitoring itself is a challenging problem, as previously mentioned in this subsection.

[2] The authors argue that BinUnpack is strong against state-of-the-art resolution evasion techniques. However, if the OS is compromised, BinUnpack can be evaded.

Table 2.4 Evaluation Viewpoint Summary

Symbol	Description
T1	Track target codes injected with an unknown technique.
T2	Distinguish API calls invoked from inserted codes, not benign ones.
H1	Capture the calls of an API even when the first few instructions of the API are skipped.
H2	Identify the API code without depending on their located addresses.
H3	Capture the calls of an API even when malware does not reference any IAT.
R1	Understand the current semantics of the analysis environment without depending on the OS.
R2	Analyze control-flow even when junk codes exist between an API call site and the code of the API.

2.5.1 Architecture Comparison

We summarize the architecture of each dynamic analysis system in Table 2.2 and that of each static analysis tool or system in Table 2.3. We qualitatively evaluate their resistance capabilities as explained in Section 2.3. First, we present the evaluation viewpoints, as shown in Table 2.4, and then explain the evaluation results in dynamic and static analyses separately.

Dynamic Analysis

The resistance capabilities are qualitatively evaluated by surveying relevant papers or source codes if they are available on the Internet. We present the viewpoints of resistance capabilities against target, hook, and resolution evasions hereafter.

Viewpoint. The resistance capability against target evasion is considered from the two aspects. The first aspect is if the system can handle a new code injection technique **(T1)**. This aspect comes from the following considerations. Most of the existing analysis systems have some capability for handling code injections. However, they mainly assume heuristic approaches toward handling them such as hooking specific API calls, e.g., CreateRemoteThread or WriteProcessMemory. With heuristic approaches, we could adequately handle known code injections, but we could not deal with unknown ones. Because new code injection techniques emerge every year, it is essential to have the capability to handle code injections without depending on heuristics. The second aspect is if the system has the capability

to distinguish correctly API calls invoked from the same process space in which both benign and malicious codes reside **(T2)**. This situation occurs when code is injected into a benign process or a file is infected in an executable. If the system does not have this capability, it aggressively collects API calls invoked from even a code region belonging to a benign program. Then, these collected APIs result in false positives and confuse system users.

The hook resistance capability is measured based on three aspects. The first is if the system has a capability for capturing the calls of an API even when the first few instructions of the API are not executed by malware **(H1)**. This aspect comes from the property of stolen code and sliding call. Most existing analysis systems set hooks for monitoring on the first instruction of each API. Even when we employ the address-matching technique for hooking, we also monitor the executions of the address of the first instruction of each API. Malware authors, of course already know this fact, so they avoid the hooks by not executing the first instruction of each API when their malware calls an API. The second aspect is if the system has a capability of identifying the code of APIs without depending on their located addresses **(H2)**. This aspect comes from the property of stolen code and copied API obfuscation. If a hook mechanism strongly depends on the virtual memory address to identify the API codes, malware can easily avoid this hook by moving their target code to another location. The last is if the system has a capability for capturing the calls of an API even when malware does not call API via any IAT **(H3)**, i.e., malware directly jumps to the API code to call the API. This aspect comes from the property of dynamic API resolution. When malware dynamically resolve APIs by itself without depending on a program loader, malware directly jumps to the API code from its call site. If the system sets hooks on IATs for hooking, it could be evaded.

The resolution resistance capability is considered from the two aspects: control-flow and semantic validity. These aspects come from the fact that API name resolution is performed based on the comparison between the destination address of a call instruction in malware code and the addresses where target APIs are expected to be loaded. That is, to make the resolution failure, an attacker should retarget the call instruction to the different address from the API one, i.e., control-flow

obfuscation, or disturb the calculation of the address where an API is loaded for comparison, i.e., semantic evasion. We explain only semantic evasion resistance here because control-flow obfuscation does not affect the dynamic analysis.

Semantic evasion resistance, which is a part of resolution evasion, is dependent on OS semantics. That is, semantic evasion resistance depends on how much a system correctly understands the current situation and if it can keep track of changes in the situation even when the underlying OS is compromised and unreliable **(R1)**. This represents the degree of independence from the OS. Analysis instances are likely to trust the OS, which is installed in an analysis environment or running under the analysis instances. Thus, the instances construct their analysis logics based on the information provided by the OS. For example, when we want to collect the base address of a loaded DLL, we acquire that information from a PEB or VAD, which is a data structure that the OS manages. However, this hypothesis, i.e., that the OS is reliable, is not always true. When malware intrudes into the kernel layer, it can easily modify any data in the OS. Even when malware does not go into the kernel layer, it could modify some system data stored in the user-land, such as PEB. Therefore, whether or not analysis instances can understand the current semantics without any help from the OS is a metric when we consider semantic evasion resistance.

Result. Table 2.2 shows the results of qualitative evaluations considering the resistance capability viewpoint. Regarding target evasion, all systems except for Tartarus are vulnerable to new code injection techniques. In addition, they do not distinguish API calls invoked from a process memory space where benign and malicious codes reside together. This is because these systems identify their targets based on a PID, while Tartarus identifies targets based on target codes marked with taint analysis. When we try to track the movement of an injected code with PID-based targeting, we need to know the PID of the process to which the code is injected. To accomplish this, we have to know the injection methodology and prepare hook points in advance to monitor the behavior. However, it is difficult to predict a new emerging code-injection technique before the malware is used. Therefore, PID-based targeting is vulnerable to new code injections. Furthermore, the granularity of PID-based and TID-based targeting approaches is too rough to distinguish API

calls from benign parts or malicious parts of code in a running process. If these codes are mixed and placed in the same memory region, a single thread may execute both codes. Therefore, even TID-based targeting is insufficient to distinguish them. Thus, PID- and TID-based targeting are neither robust against code injections nor file infections.

Regarding hook evasion resistance, IAT hooking is not robust against dynamic API resolution because malware directly dumps to the API code without referencing any IAT. Whether IAT hooking is effective against the other hook evasion techniques depends on how malware identifies the positions of target API codes. If malware identifies it from the corresponding entry of an IAT, the hook set on the entry is still active even after stolen code or copied API obfuscation are applied. However, if malware identifies it from other sources, e.g., the EAT of the DLL, and directly jump to the API code without referencing any IAT, malware can bypass IAT hooking. EAT hooking is also effective if malware identifies the addresses of APIs from the EAT. Since EAT hooking replaces the entries of the EAT of a DLL with the address of a hook handler, when malware acquires the address of a target API for applying evasion techniques from the EAT of the DLL, malware wrongly recognizes the address of the hook handler as the one of the API. Even if some instructions of the hook handler are stolen or copied, the functionality of the hook handler is not lost because the code of the hook handler is executed. However, we need to care about the sliding call cases. If the first few instructions of the handler for an API are different from the ones of the API, the execution context could be destroyed when malware skips the few instructions for sliding call. This may lead a program crash.

Code rewriting techniques are robust against stolen code and copied API obfuscation, but they are not robust against sliding call. This is because when the first instruction of an API is copied with stolen code or copied API obfuscation, the inserted instruction for hooking such as `int3` or `jmp` is also copied to a newly-allocated buffer. If the copied instruction is executed in the buffer, this execution generates an exception in the case of `int3` or transfers the execution to a hook handler in the case of `jmp`. However, if the first instruction is skipped with sliding call, the inserted instruction is never executed. Then, this situation fails to hook the API calls.

On the other hand, control transfer interception is resistant to stolen code and sliding call, but not to copied API obfuscation. This is because there are control transfers from malicious code to any API code in the cases of stolen code and sliding call, but there are no control transfers from malicious code to any API code in the case of copied API obfuscation. In copied API obfuscation, since all API instructions are copied to a local buffer in a malware process, no control transfer to API codes occurs. Thus, there are only transfers between malicious codes.

Address matching is vulnerable against all hook evasion techniques since it focuses on only the single virtual memory address to hook a specific API call. It is sufficient for malware to evade this technique by simply moving the code of an API to a different location or skipping the address. So, all hook evasion techniques can evade address matching. In short, there is no hooking technique that satisfies both properties of binary rewriting and transfer interception, i.e., a technique that can track the movement of a target code and capture the control transfers between different purposed memory regions, respectively.

DLL hijacking, which is used in BinUnpack, is resistant against stolen code and copied API obfuscation because actual API codes reside behind the fake DLL injected into malware under analysis. When the malware tries to steal some of the code of an API, what it steals is the code of a function exported from the fake injected DLL. So, the hook set on the first instruction of the API in the real DLL is still active and works toward capturing the executions of the API. However, when malware uses the sliding call technique, BinUnpack may crash in the following case. The first instruction of a function exported from the fake DLL is not the same as that of the corresponding API exported from the real DLL. This difference may cause the context of the execution to break because the malware is designed to perform sliding call with the expectation that the skipped instruction exists at the entrance of the API.

It is worth mentioning that hook techniques involving any modification, especially code modifications, such as inline hooking or software breakpoints, are not suitable for malware analyses because they may expose the existence of analysis modules to malware. That is, malware could easily detect evidences for hooking in the analysis

environment and may stops its execution or changes its behaviors. For the same reason, DLL hijacking, IAT hooking, and EAT hooking also expose the existence of analysis modules to malware. So, they possibly affect the execution of malware.

Regarding resolution evasion resistance, most of systems and tools are vulnerable to incorrect OS semantics. If an underlying OS is compromised and returns an incorrect view of its semantics, malware can evade the analysis tools built on the semantics. The authors of the BinUnpack paper argue that BinUnpack is sufficiently strong against state-of-the-art resolution evasion techniques. However, theoretically speaking, if an underlying OS is compromised, BinUnpack also does not collect accurate OS semantics. Then, it may fail to resolve the APIs of the process running on the OS.

Static Analysis

We surveyed papers or source codes of static analysis tools to evaluate their resistance capabilities against each evasion technique. We present viewpoints of the evaluation and results.

Viewpoint. For static analysis, we qualitatively evaluate the resistance capability of static analysis tools from two viewpoints: semantic evasion and control flow obfuscation in resolution. We do not explain the resistance capabilities against target evasion and code obfuscation techniques. This is because the capabilities against target evasion techniques, i.e., **T1** and **T2**, are already explained in the dynamic analysis part in this subsection and code obfuscation techniques are outside the scope of this thesis. So, we first explain the two viewpoints of resistance capabilities against resolution evasion and then describe the evaluation results.

Resolution evasion resistance against semantic evasion is the same as that mentioned in the dynamic analysis part, i.e., **(R1)**. That is, it represents the degree of dependency on OS semantics. If IAT reconstruction tools strongly depend on OS semantics and the OS is already compromised, the semantics collected from the OS may be incorrect. More specifically, if IAT reconstruction tools are designed to collect the positions of each API based on the OS-provided information, they are likely to fail to reconstruct the correct IAT.

Resolution evasion resistance against control-flow obfuscation handled cases where junk code exists between API call instructions and their API code **(R2)**. IAT reconstruction tools are designed to resolve the API of each entry in an IAT from the code directly pointed to by the entry. If the entry points to a junk code that does not contain any API code, analysis tools are likely to fail to resolve the API from the entry.

Result. Table 2.3 shows the results of this evaluation. Regarding semantic evasion, all tools can be evaded using this type of evasion. The tools reconstruct a view of OS semantics, i.e., the API locations, based on some OS-managed data structures such as a PEB or VAD. Even the tools with API monitoring rely on the underlying OS mechanism, e.g., callbacks or API monitoring, to identify the API names they capture. So, these tools are vulnerable to semantic evasion if the OS is unreliable.

Regarding control-flow obfuscation, IAT reconstruction tools without API monitoring can be evaded, while those with API monitoring cannot. This is because dynamic analysis techniques are not affected by control-flow obfuscation. Therefore, these tools can overcome this type of evasion technique by combining with dynamic analysis techniques. However, as we mentioned in the previous subsection, it is not easy to perform precisely API monitoring. Precise API monitoring itself is a challenging task. Additionally, Eureka has a capability of performing control-flow analysis to relate a `call` instruction to any API. It probably analyzes the control-flow containing junk codes if junk code does not have complicated conditionally-protected execution parts.

Eventually, as discussed above, there are no existing analysis tools that are sufficiently designed against the various evasion techniques. We considered the reason why these many analysis tools can be evaded and have reached a conclusion. We explain the reason in the next subsection.

2.5.2 The *Target-Gap* Problem

We consider the reason why malware can evade existing analysis techniques, and then present our one conclusion, i.e., the *target-gap* problem. The target-gap problem

is a design problem that commonly exists in current dynamic and static analysis techniques. In short, we consider that there is a gap between what we really want to analyze and what we actually analyze.

We illustrate three example situations in which we can address the target-gap problem. The first example is when we analyze malware with a dynamic analysis technique, we identify the target instances in an analysis environment based on PIDs. Then, we handle the executions of the instances with the target PID as to-be-analyzed executions, even though what we really want to analyze is the executions of the code of the malware under analysis. The PID is an identifier of a process that is an instance of a malware executable, which possibly includes our target codes. However, the process of malware is not always equal to the code of the malware. Given a case that malware injects a part of its code into another process, the injected process becomes an instance executing a part of the malicious code, but the PID of the code-injected process is not the original malware instance. Existing dynamic analysis environments set hooks on specific APIs, as previously explained, to acquire the PID of the process to which the code is injected. However, this approach does not work for unknown code injection techniques.

We consider that the main reason for this inability is the gap problem. That is, although what we really want to analyze is the malware code, what we actually analyze is a process that may contain the malware code. Since there is a gap, we have to fill the gap with heuristics when the malware code is moved to another process. To accomplish this, we must know the path for the injection in advance; however, as previously explained, it is impossible to know code injection techniques before they appear. If we focus on the code, we do not need to consider what process is executing the code. Thus, it is not necessary to know the injection methodology beforehand.

The second example is when we try to capture the execution of a specific API, we monitor the executions of the address where the API is expected to be loaded, even though what we really want to capture is the execution of the API code. The existence of the API code and its address calculated from other data sources are usually equivalent. However, most codes can be executed anywhere in the memory

since they are not bound to their addresses or at least with a small re-calculation of its address. So, malware can easily move the API code to another buffer and execute them there. When malware moves the code of an API to another location, some of the existing API monitors fail to identify the API execution. We consider that the main reason why malware can do this is the gap problem. That is, we do not directly capture the execution of the API code. Instead, we assume the execution of the API code based on the execution of the memory address where the API code is expected to be located. There is a gap between the two. Malware takes advantage of this gap to evade existing analysis techniques.

The third example is when we try to identify the memory layout at a particular execution point after starting an analysis, what we actually identify is the memory layout that the OS recognizes at that time even though what we really want to identify is the real memory layout of the current execution environment. More specifically, we identify the positions of the APIs exported from loaded DLLs, and this is accomplished by communicating with the underlying OS using API calls or directly parsing the data structures of the OS to emulate the communication. We trust the semantics that the OS provides as an oracle for analysis. However, OSs can also be evaded. One example is a case where a rootkit has been installed. When malware succeeds in gaining administrator privileges and intrudes into the kernel layer, malware commonly modifies the value of specific variables in certain data structures to hide malicious activities and artifacts from analysis instances such as anti-virus software. Despite this, we trust the OS and build analysis logic on the data that the OS provides. We consider that this failure also comes from the gap problem. Analysis tools implicitly have a strong dependency on the OS and trust the veracity of the provided OS semantics. This is a cause that malware can evade analysis tools and systems.

In summary, the reason why existing analysis systems and tools are insufficient against evasion techniques is the target-gap problem. We presented three situations that may have occurred in the past. In the rest of this thesis, we address the target-gap problem. To accomplish this, we first demonstrate the target-gap problem with a new semantic evasion technique and then propose taint-based dynamic and static

analysis techniques to address the problem.

Chapter 3

A New Threat: Trace-Free Program Loader

3.1 Introduction

In this chapter, we demonstrate a threat of the target-gap problem by introducing a new semantic evasion technique. The reason why we focus on semantic evasion is that we can attack both dynamic and static analysis techniques at the same time with one technique and semantic evasion techniques have not been well-studied compared to other evasion techniques. Specifically, we present *Stealth Loader*, a program loader to evade all existing API-oriented analysis 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 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

[*1] a DLL loaded by Stealth Loader

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 chapter 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-oriented analysis 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 [24]. 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.

3.2 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.

3.2.1 Overview

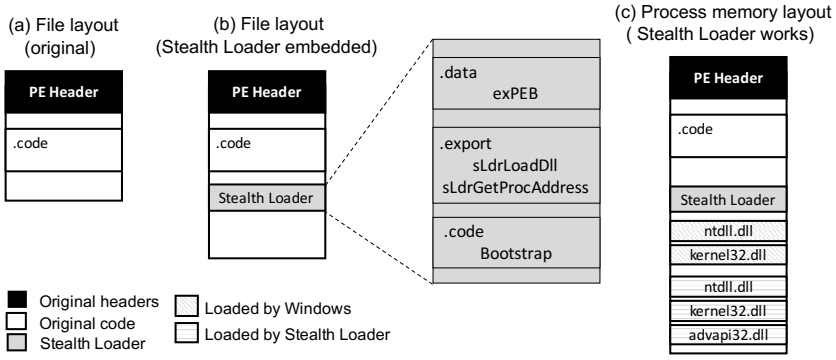

Fig. 3.1 How Stealth Loader works and its components. (a) The file layout of an executable before Stealth Loader is embedded, (b) that after Stealth Loader is embedded and the components of Stealth Loader are also described, and (c) the process memory layout after Bootstrap resolves the dependencies of an executable and stealth-loaded DLLs.

Figure 5.1 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 evasion 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 oriented analyses. 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.

3.2.2 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.

Code Mapping

■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

*2 DLLs loaded by Windows

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 [16] acquires the list of loaded DLLs on memory by parsing VADs and extracting the file path of each mapped file.

■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.

■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.

Dependency Resolution

■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 Figure 3.2-(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.

■Design To avoid this, Stealth Loader loads dependent DLLs to resolve the dependency of a loading DLL. In the case in Figure 3.2, 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 Figure 3.2-(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.

■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.

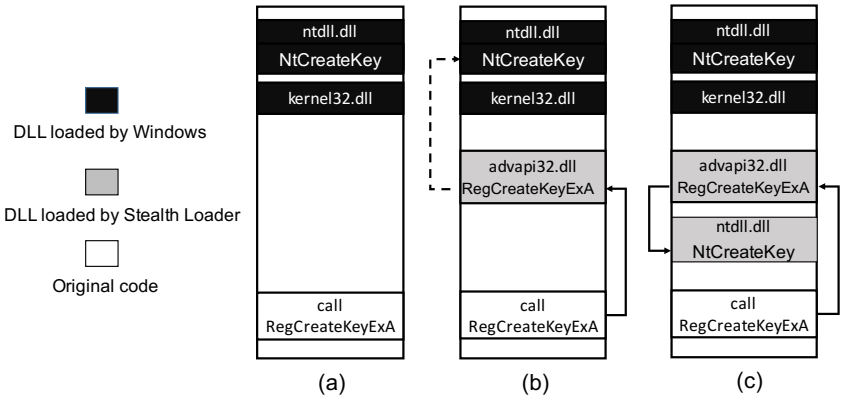

Fig. 3.2 Example of resolving dependency with Stealth Loader. (a) The layout before Stealth Loader starts, (b) the stealth-loaded advapi32.dll does not create a dependency on the Windows-loaded ntdll.dll, and (c) the stealth-loaded advapi32.dll creates a dependency on the stealth-loaded ntdll.dll.

Initialization & Registration

■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.

■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.

■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 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.

3.2.3 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.

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 [59] 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.

Reflective Loading

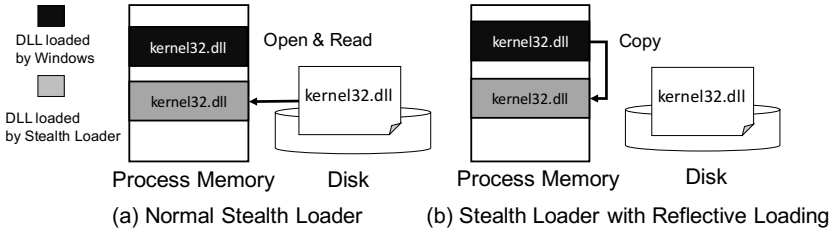

Fig. 3.3 Behaviors of normal Stealth Loader and Reflective Loading. (a) Stealth Loader loads kernel32.dll from a disk, and (b) Stealth Loader with Reflective Loading loads kernel32.dll from the memory, i.e., the Windows-loaded one.

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 (Figure 3.3-(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 (Figure 3.3-(b)). This idea is inspired by Reflective DLL injection, introduced by Fewer [60], 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.

3.3 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.

3.3.1 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.

3.3.2 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 3.3.2. 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 [61], 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 [24], sets hooks for dynamic analysis. Regarding the remaining 25% of APIs, they separately reside in several DLLs in a small group.

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.

■ **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.

■ **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.

■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.

3.3.3 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.

3.4 Experiments

To show the feasibility of Stealth Loader, we conducted three experiments: one for comparing its resistance capability against current analysis tools to other evasion 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.

3.4.1 Analysis Resistance

To show the resistance capability of Stealth Loader against current API-oriented analysis tools, such as API monitoring and IAT reconstruction, 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 Chapter 2. Regarding the other techniques which are not covered by these tools, we qualitatively discuss the resistance capability of Stealth Loader against them in Subsection 3.5.3 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 evasion techniques, i.e., IAT obfuscation, API redirection, which is the pattern explained in Figure 2.3-(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.

Static Analysis

Table 3.1 Results of Static and Dynamic Analysis Resistance Experiment

Evasion Techniques	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 evasion technique successfully evaded the tool. Stealth Loader evaded all the tools.

[1] IAT Obfuscation and API Redirection are techniques for API evasion 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.

In this experiment, we analyzed each protected executable with four major static analysis tools, IDA [62], Scylla [17], impscan (The Volatility Framework [16]), and ldrmodules (The Volatility Framework [16]). IDA is a de-facto standard disassembler 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 addresses from the export address table (EAT) of each loaded DLL. ldrmodules acquires the base addresses from VADs.

■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.

■Results The left part of Table 3.1 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 Chapter 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.

Dynamic Analysis

In this experiment, we analyzed each protected executable with three dynamic analysis tools, Cuckoo Sandbox [24], traceapi [63], and mini_apitracer [64]. 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 3.1, is a plugin of DECAF [52], which is a binary analysis framework built on QEMU [42].

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.

■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.

■Results The right part of Table 3.1 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 Chapter 2, 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.

3.4.2 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.

Procedure

First, we collected 117 pieces of malware from VirusTotal [65] 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

Table 3.2 Results of Real-world Malware Experiment

	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.

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 [24].

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 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.

Results

Table 3.2 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.

3.4.3 Memory Consumption

Table 3.3 Results of Memory Consumption Comparison

	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 [66]. 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.

As we mentioned in Section 3.2, 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.

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[66] 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.

Results

Table 3.3 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.

3.5 Discussion

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

3.5.1 Platform Dependency

As we mentioned in Section 3.3, the current Stealth Loader is implemented to run on the Windows 7 environment. However, we believe that the design explained in Section 3.2 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 Subsection 3.3.2. In other words, the other part of this paper, i.e., all sections except for Subsection 3.3.2 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.

3.5.2 Other API-oriented Analysis Techniques

Eureka [55] 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.

3.5.3 Countermeasures

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.

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 [67], 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.

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 [68], 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. [69], we need different counter-approaches to solve this type of problem.

Inferring DLLs from Visible Traces

Since the current Stealth Loader avoids supporting some APIs, as we explain in Subsection 3.3.2, 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 Chapter 2, 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 Figure 2.3-(c) in Chapter 2. 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.

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 Subsection 3.2.3, 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. [69], have been proposed to avoid byte-pattern-based detection. 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 3.3 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.

Restricting Untrusted Code

One more direction is to prevent Stealth Loader from working at each phase. Policy enforcement, which is mentioned in safe loading [70], 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) [71]. In case of Control Flow Guard [72], 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.

3.6 Conclusion

We presented Stealth Loader as a proof-of-concept implementation to demonstrate the target-gap problem. That is, existing analysis techniques are vulnerable to new evasion techniques. We showed that Stealth Loader actually evaded all major analysis tools. We also qualitatively showed that Stealth Loader can evade previously proposed API-oriented analysis techniques in academic studies. The reason why these evasions happen is that all existing tools depend on OS and establish their analysis logics on OS recognitions. However, OS is not always reliable. As we showed, even from user-land, attackers can evade OS, and then this evasion leads to

evade analysis tools.

To make analysis tools more robust against evasive malware, we need to recognize that OS is evadable and not always reliable, and then be independent from OS. From the next Chapter, we will introduce our taint-based approaches as an evasion-resistance-inclusive design for analysis tools.

Chapter 4

Taint-Assisted Dynamic Malware Analysis

4.1 Introduction

In this chapter, we present a practical API monitor called *API Chaser*, which is resistant to various evasion techniques. API Chaser is built on a whole system emulator, QEMU[42] (actually Argos[73]), and executes monitored malware in a guest operating system (OS) running on it. In API Chaser, we use a *code tainting* technique to identify precisely the execution of monitored instructions. The core idea of code tainting is that we identify our target code based on taint tags. To this end, we first prepare a taint tag targeted for a specific analysis purpose and add the tag to the target instructions before executing them. Then, we begin to run the executable file containing the monitored instructions. At the virtual CPU of an emulator, we confirm whether or not a fetched instruction contains the taint tag targeted for analysis. If it does, it is executed under analysis. If not, it is executed normally, i.e., it is outside the scope of the monitoring.

We apply the code tainting technique to API monitoring. We call this technique *taint-based control transfer interception*. This technique works as follows. We use three types of taint tags for three different types of instructions: the instructions for APIs, those for malware, and those for benign programs. First, we add the three

types of taint tags to the respective target instructions. Then, when the CPU fetches an instruction and it has a taint tag for the API, it confirms which type of taint tag the caller instruction has. There are three cases: a taint tag for malware, one for benign, and one for API. Each case respectively corresponds to the following situations: an API call from malware, that for a benign process, and that for another API (nested call). Our monitoring target is the call only from malware and we exclude the others.

Taint-based control transfer interception is resistant to evasion techniques because it is able to distinguish between the target instructions and others at byte granularity even when they exist in the same process memory space. In addition, this technique is able to track the movement of monitored instructions by propagating taint tags attached to them when malware injects a malicious code into another process. This technique is independent of OS semantic information such as virtual addresses, Process ID (PID) or Thread ID (TID), and file names. Therefore, it is no longer influenced by the changes in these types of information by malware for evading analysis systems.

In API Chaser, there are also several unique implementations for enhancing the resistance against evasion techniques, i.e., *pre-boot disk tainting* and *code taint propagation*, and for improving the practical capability for large-scale analysis, i.e., *hot-boot* and *one-time disk image*. These techniques contribute to achieving precise API monitoring and a practical malware analysis sandbox, respectively. In the proposed API Chaser implementations, we use 32-bit Windows XP and 7 as the guest OS. However, we do not limit API Chaser to only these two platforms. We believe that the API Chaser design is neutral and we can apply it to other platforms such as a 64-bit Windows 8 or 10 as the guest OS while following the same design.

To show the effectiveness of API Chaser, we conducted several experiments using real-world malware with a wide range of evasion techniques. We evaluated the API monitoring accuracy by analyzing some malware on API Chaser and in comparative environments in which APIs are monitored using existing techniques. Then we compared the logs output by each environment. These experimental results indicate that API Chaser is able to capture precisely the API calls from all sample malware without being evaded. We also evaluated API Chaser with several synthetic malware,

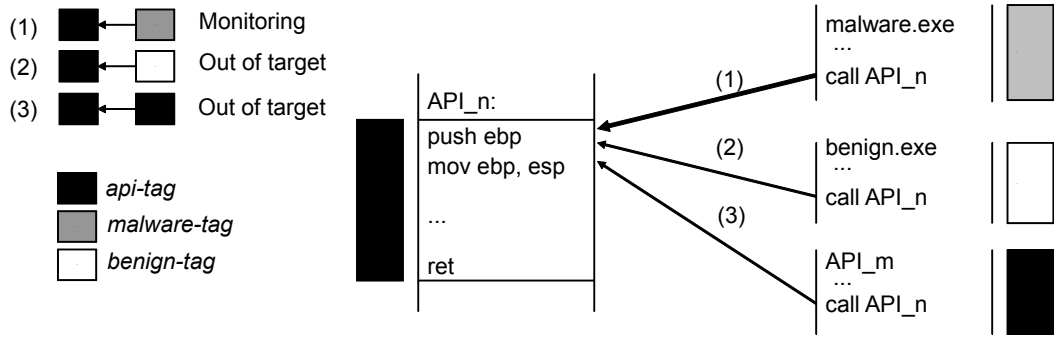

Fig. 4.1 Taint-based Control Transfer Interception

in which state-of-the-art evasive techniques were implemented. The experimental results show that API Chaser is sufficiently robust against new emerging techniques such as Process Hollowing [3], AtomBombing [5], PowerLoaderEx [7], Shim-based DLL Injection [8], and Stealth Loader [9].

Moreover, we analyzed 8,879 malware samples collected from the Internet for a large-scale experiment. The samples were classified into 421 malware families with AVClass [74]. Through the analyses, we found 701 hook-evasive ones$*^1$, which belonged to 36 families, while we found 344 target-evasive malware samples$*^2$, which belonged to 84 families. We consider that these numbers allow us to argue that hook evasion and target evasion techniques are actually major techniques and widely used among real-world malware.

4.2 Our Approach

To address the target-gap problem that existing API monitors have, we propose the *taint-based control transfer interception* API monitoring technique that uses *code tainting* to identify precisely the execution of APIs. First, we define some terms and the scope of this paper. Second, we present code tainting. Third, we describe the types of monitored instructions. Fourth, we present taint-based control transfer

$*^1$ malware uses hook evasion techniques
$*^2$ malware uses target evasion techniuques

interception, i.e., how to capture API calls invoked from malware and exclude the ones invoked from benign processes and nested API calls.

4.2.1 Definitions and Scope

We define three important terms used in this chapter: API, API call, and API monitoring.

- *API* is a function comprising more than one instruction to conduct a specific purpose and we use it interchangeably with a user-land Windows API, which is a function provided from the Windows operating system and libraries.
- *API call* is a control transfer with valid arguments from an instruction outside of an API to an instruction within the API.
- *API monitoring* is a technique to detect the first execution of an instruction of monitored APIs immediately after control has been passed from an instruction outside of the API.

We explain the scope of this chapter. The evasion techniques in the scope are those that were mentioned in Section 2.3, i.e., those used for hiding API calls that malware has actually invoked. We exclude conditionally-protected executions, e.g., trigger-based ones [20] and stalling code [75], from the scope of this chapter. We also exclude the case that malware invokes functions of modules, e.g., DLLs, statically linked to the malware, which do not execute any instructions of system modules that we prepared in our analysis environment.

4.2.2 Code Tainting

Code tainting is a taint analysis application and a technique used to identify the execution of monitored instructions based on taint tags attached to them. It adds taint tags to the target instructions before executing them. Then, when the CPU fetches an instruction, it confirms if the instruction (actually the opcode of the instruction) has a taint tag. If the instruction has a taint tag targeted for analysis, it will be executed under analysis. If not, it will be executed normally. When

monitored instructions are operated as data, taint tags added to the instructions are propagated in the same way as data tainting. That is, we can track the movement of monitored instructions based on the taint tags.

There are three advantages of code tainting for monitoring malware activities. First, it becomes possible to conduct fine-grained monitoring. This property is effective against malware using target evasion techniques. Code tainting can distinguish the target instructions and others at byte granularity based on taint tags, even though there are both injected malicious instructions and benign ones mixed together in the same process space or the same executable file. Second, it allows us to track the movement of the target instruction by propagating taint tags attached to them. This property is effective against both target evasion and hook evasion techniques. For example, when malware injects its malicious code into other processes or other executable files, code tainting can track the injection by propagating taint tags added to the malicious code. Third, it is no longer influenced by changing the semantic information of an OS, e.g., virtual addresses, PID or TID, and file names. This property is also effective against both target evasion and hook evasion techniques such as name confusion or Stealth Loader because it does not depend on these types of semantic information at all for monitoring API calls, and depends solely on taint tags.

A similar technique to code tainting has been used in previous research [73][76] to detect attacks by tainting received data from the Internet and then monitoring a control transfer to the tainted data. We leverage the technique for malware analysis on API monitoring. The difference is that code tainting adds taint tags to the code with the obvious intention of monitoring its execution, whereas the previous research taints all received data to detect a control transfer to it.

4.2.3 Tag Types and Monitored Instructions

We use the following three types of taint tags to identify the execution of three types of instructions for API monitoring.

- *Api-tags* target instructions in each API

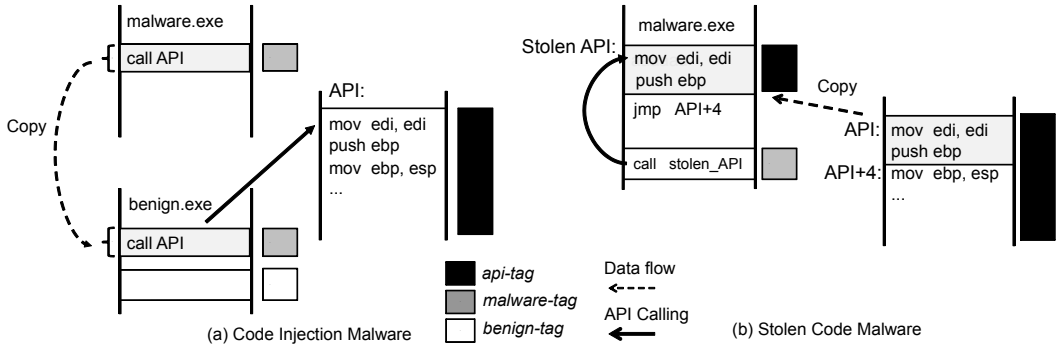

Fig. 4.2 Taint-based Control Transfer Interception Against Evasion Techniques

- *Malware-tag* targets instructions in malware
- *Benign-tag* targets instructions in benign programs

We taint all instructions in each API with api-tags. We use this type of tag to detect the execution of APIs at the CPU. Moreover, we embed API-identifier information in each api-tag which we can use to distinguish the execution of each type of API. Regarding malware-tag, we taint all bytes in a malware executable file and dynamically generated code with malware-tags. We use malware-tags to identify the caller instruction of APIs and detect the execution of malware instructions. On the other hand, we taint all bytes in benign programs with benign-tags. By benign programs, we mean all files that have been installed on Windows by default, or in other words, all instructions except for those in malware and APIs. We mainly use this type of taint tag to identify the caller instruction of APIs and then exclude the API calls from the monitoring target.

4.2.4 Taint-Based Control Transfer Interception

We use code tainting with the three types of taint tags to monitor APIs invoked from malware. We call this API-call-hooking technique *taint-based control transfer interception*. When a CPU fetches an instruction and the instruction has an api-tag, it confirms the taint tag attached to the caller instruction. There are three cases as shown in Fig.4.1: the API is called from malware, a benign process, and the internal

of other APIs (nested call). As for the first case, shown in Fig.4.1 (1), if the caller instruction has a malware-tag, it determines that the API call is from malware. Thus, it captures the API call and collects the information related to the API call such as its arguments. With regard to the second, shown in Fig.4.1 (2), if caller one has a benign-tag, it determines that the API call is from a benign process. Thus, it is outside the target monitoring and does not need to capture this API call. As for the third, shown in Fig.4.1 (3), if the caller has an api-tag, it is a nested API call. Nested API calls are also excluded from the monitoring target, so that we can focus only on API calls directly invoked from malware. This makes the behaviors of malware clearer and easier to understand.

In Fig.4.2, as a running example, we explain the behaviors of taint-based control transfer interception against the two evasion techniques: code injection and stolen code. Fig.4.2(a) shows the behavior against code injection. When malware injects code from malware.exe to benign.exe, the taint tags of the code are propagated. The API call from the injected code is a control transfer from an instruction with a malware-tag to an instruction with an api-tag. Then, we can identify it as our target API call. On the other hand, Fig.4.2(b) shows the behavior of calling a stolen API. When few instructions at the entry of the API are copied to the allocated memory area in malware.exe, the taint tags added to the instructions are also propagated. The call instruction, `call stolen_API`, has a malware-tag and the copied instruction, `mov edi, edi`, has an api-tag, so we detect the API call and include it in the monitoring target.

4.3 System Description

In this section, we present an overview of *API Chaser*, which uses taint-based control transfer interception for monitoring API calls. First, we briefly explain the main components of API Chaser. Second, we illustrate its malware analysis process. Third, we present the enabling techniques used in API Chaser: *pre-boot disk tainting* and *code taint propagation*.

4.3.1 Components

API Chaser is built on a whole system emulator, QEMU (actually on Argos). API Chaser has the following components: virtual CPU for API monitoring and taint propagation, shadow memory to store taint tags for virtual physical memory (hereafter “physical memory"), and shadow disk to store taint tags for a virtual disk (hereafter “disk").

A virtual CPU is the core component of API Chaser. It is a dynamic binary translator that translates a guest instruction to host native instructions. With dynamic binary translation, it conducts API monitoring as mentioned in the previous section and taint propagation based on our propagation policy, which is explained in a later subsection. The shadow memory is a data structure for storing taint tags added to data on physical memory. When the virtual CPU fetches an instruction, it retrieves the taint tag added to the instruction from the shadow memory. The shadow disk is also a data structure for storing taint tags added to data on a disk. When data with taint tags are written to a disk, the taint tags are transferred from the shadow memory to the shadow disk and stored in the corresponding entries of the shadow disk. When transferring data with taint tags from a disk to physical memory, the taint tags are also transferred from the shadow disk to the shadow memory.

4.3.2 Analysis Process

Fig.4.3 shows the analysis process for API Chaser. There are two steps for API Chaser to analyze malware: taint setting and analysis.

Taint Setting Step

In the taint setting step, API Chaser conducts pre-boot disk tainting, which adds taint tags to all the target instructions in a disk image file before booting a guest OS. The details of pre-boot disk tainting are given in the following subsection.

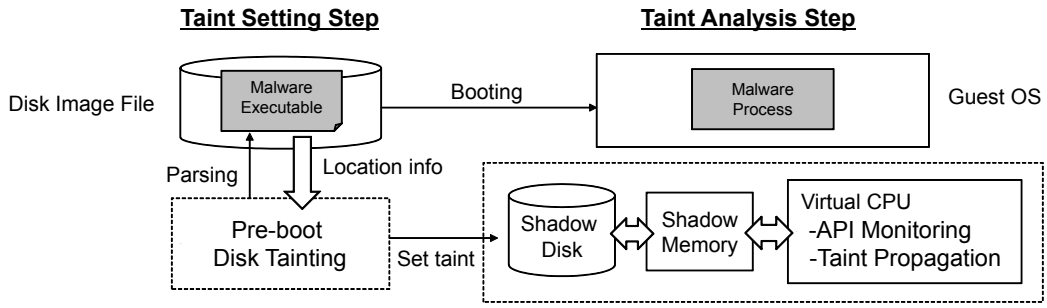

Fig. 4.3 Analysis Process for API Chaser

Analysis Step

In the analysis step, API Chaser first boots the guest OS installed on the disk image file. During the boot, target files containing target instructions are loaded onto physical memory. At the same time, the taint tags added to the target instructions are also transferred from the shadow disk to the shadow memory. After completing the boot, API Chaser executes malware and initiates analysis. During the analysis, API Chaser conducts API monitoring and taint propagation based on our policy.

4.3.3 Enabling Techniques

We explain the enabling techniques used in API Chaser to support the API monitoring: pre-boot disk tainting and code taint propagation.

Pre-boot Disk Tainting

Pre-boot disk tainting is a technique that adds taint tags to target instructions on a disk image file before booting a guest OS. Properly adding taint tags to all target instructions is not an easy task because they may be copied and widespread over the system after a guest OS has booted. For example, after booting a guest OS, an API instruction may be on a disk, loaded onto memory, swapped out to disk, or swapped into memory. When we add taint tags to a target instruction, we must identify all the locations of widespread instructions and add tags to all of them. If we miss adding a tag to any one of them, it may allow malware to evade the API monitoring.

	Rule1 and Rule2	Rule3
Target Instruction	mov [edi], eax	call CryptEncrypt(,,, pbData, pdwDataLen,,,);

Code Taint Propagation Handling Code	if eax is tainted:
	  set the tag of eax on [edi];
	else:
	  if 'mov' has a malware-tag
	  set a malware-tag on [edi];
		if 'call' has a malware-tag:
		  for(i = 0; i < *pdwDataLen; i++) {
		  set a malware-tag on pbData[i];
		  }

Fig. 4.4 Code Taint Propagation Example

To avoid this troublesome task, we use pre-boot disk tainting. The procedure is given hereafter. First, it parses a disk image file containing target instructions and identifies the location where the target instructions are stored. We use disk forensic tools [77] to identify files containing target instructions, and then, if necessary, we acquire the offsets of the target instructions from the PE header of the files and identify the locations of each API using disassemble tools [78][62]. Second, it adds taint tags to the corresponding entries of a shadow disk based on the calculated location. Before launching a guest OS, all instructions surely reside on a disk and they are not widespread yet. Pre-boot disk tainting simplifies the tainting task because only target instructions on a disk require attention. We no longer need to care whether or not target instructions have been loaded.

Our Taint Propagation Policy

API Chaser conducts taint propagation based on pre-defined rules. The pre-defined rules are mainly composed of two types: basic rules and rules for code taint propagation.

Basic rules are defined based on each instruction type, such as data-move, unary arithmetic, or binary arithmetic operations. We basically use the rules of Argos [73] for API Chaser as they are. More concretely, when API Chaser handles a data-move operation, such as `mov`, it propagates its taint tag to the destination if the source operand is tainted. When it handles an unary arithmetic operation, such as `inc`, it preserves the same tag of the operand if the operand has a taint tag. When it handles a binary arithmetic operation, such as `add`, it propagates the tag of the source operand

to the destination if any one of the source operands is tainted. If both operands are tainted, it propagates the tag of the first operand. However, this propagation rule may cause it to disconnect a taint propagation with the tag of the second operand because we have to discard the tag of the second operand even when the tag plays an important role for tracking a specific data-flow. Regarding this problem, we discuss it in Subsection 4.6.2.

In addition to the above rules, we use our original taint propagation rules for memory-write operations called *code taint propagation* to prevent malware from avoiding code tainting by generating a code using implicit-flow-like code extraction. Implicit flow is a process where a value with a taint tag affects the decision making of the following code flow. However, there is no direct dependency between the value and other values operated in the following code. Thus, a taint tag is not propagated over the implicit flow, even though they are semantically dependent on each other. It is reported that taint tags are not properly propagated in some Windows APIs that use implicit-flow-like processing [23]. Actually, we observed some cases in which malware-tags added to the code of malware were not propagated to its dynamically generated code. This is because most obfuscated malware has encrypted or compressed original code in its data section and it uses implicit-flow-like data-processing to unfold compressed or encrypted code and extract its original code. If we fail to propagate malware-tags properly, we miss identifying the execution of malware instructions.

To address this, we use code taint propagation for code dynamically generated by malware. Code taint propagation has the following rules.

- **Rule1**: If an executed instruction is tainted with a malware-tag and its source operand is not tainted, the taint tag of the instruction, i.e., malware-tag, is added to the destination operand.
- **Rule2**: If an executed instruction is not tainted or tainted with the other tags, it does not propagate the taint tag of the instruction to its destination.
- **Rule3**: If an instruction calling an API is tainted with a malware-tag, the taint tag of the instruction, i.e., malware-tag, is added to the written data by the API.

The bottom-left pseudocode in Fig.4.4 is an example of **Rule1** and **Rule2**, illustrating the case of `mov [edi], eax`. If the source operand of the target instruction, eax, does not have any taint tags and the opcode, `mov`, has a malware-tag, we add malware-tags to the destination operand, `[edi]`. Consequently, it appears as if it propagates taint tags of opcode to the destination operand of the opcode. The bottom-right pseudocode in Fig.4.4 is an example of **Rule3**, illustrating the case of `call CryptEncrypt` whose prototype is shown below. The `call` instruction has a malware-tag and it calls CryptEncrypt API, which is a function that encrypts the passed data and writes its output to the memory area pointed to by the argument, `pbData`. The argument, `pdwDataLen`, indicates the size of the output data.

```
BOOL WINAPI
        CryptEncrypt(
                _In_ HCRYPTKEY hKey,
                _In_ HCRYPTHASH hHash,
                _In_ BOOL Final,
                _In_ DWORD dwFlags,
                _Inout_ BYTE *pbData,
                _Inout_ DWORD *pdwDataLen,
                _In_ DWORD dwBufLen);
```

We detect the moment when execution is returned from the API by monitoring a control transfer from an instruction with the api-tag to one with the malware-tag, and then add malware-tags to the written bytes by acquiring the location of the written bytes from `pbData`. It seems as if the taint tag of the `call` instruction is propagated to the written bytes of the API called from the instruction. Owing to code taint propagation, we can taint all generated codes with malware-tags and identify the execution of the code based on its taint tags. We will discuss the side-effects of code taint propagation in Subsection 4.6.3.

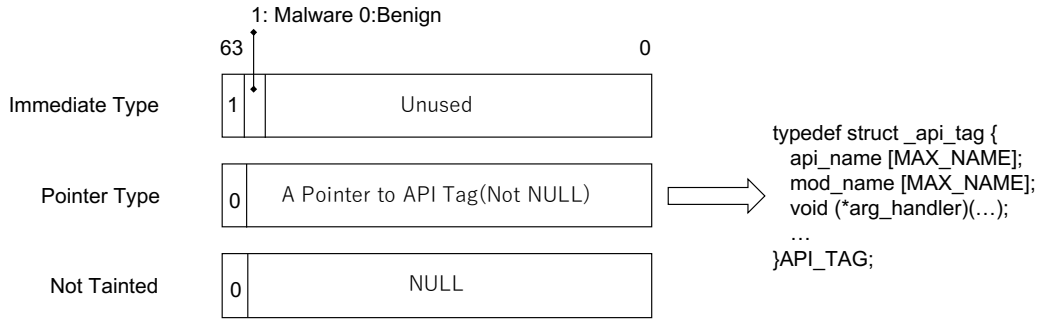

Fig. 4.5 Taint Tag Format

4.4 Implementation

In this section, we explain the details of the API Chaser implementation focusing on extensions from Argos [73] and techniques for making API Chaser practical for industrial use-cases. We present the taint tag format, virtual CPU, shadow memory, shadow disk, virtual direct memory access (DMA) controller, API argument handlers, hot-boot, and one-time disk image.

4.4.1 Taint Tag Format

We introduce the format of a taint tag stored in the shadow memory and shadow disk. The size of a taint tag is eight bytes. There are three format types, as shown in Fig.4.5: immediate format type for malware-tags and benign-tags, pointer format type for api-tags, and not-tainted type. The format is chosen depending on the type of taint tag. We distinguish the format type based on the highest bit of a tag. In the case of the immediate type, we distinguish malware tags from benign tags based on the second highest bit. API Chaser uses only the highest two bits, and the other bits are unused. On the other hand, in the case of the pointer type, a taint tag is a pointer to an `API tag` data structure. An API tag structure is a data structure that stores information related to an API such as the API name, DLL name, and API argument handling functions. We create an API tag data structure for each API, and

all instructions in each API have a taint tag with a pointer to the same API tag data structure.

4.4.2 Virtual CPU

The virtual CPU of QEMU (Argos) achieves virtualization with dynamic binary translation. It translates instructions from a guest OS to instructions for a host OS to emulate consistently the guest OS on the host OS. Argos adds a taint tracking mechanism to the dynamic binary translation. That is, it propagates taint tags from source operands to the destination based on its taint propagation policy after executing each instruction. In API Chaser, we added two new functions to the virtual CPU: an API monitoring mechanism and code taint propagation.

Fig.4.6(a) shows the API monitoring mechanism of API Chaser in the virtual CPU. When an API call is invoked from malware, i.e., the execution transferring the instruction with a malware-tag to that with an api-tag, the virtual CPU retrieves the information related to the API through its API tag data structure pointed to by the taint tag, and generates host native instructions for handling the API, i.e., invoking the API handler function. An API handler outputs the API name and DLL name, and internally invokes argument handling functions.

As for code taint propagation, Figs.4.6(b) and 4.6(c) show the difference in the behaviors between Argos and API Chaser. In the case of Argos, when it reads a guest OS instruction for writing memory, it generates a taint handling function as host native code. The function propagates taint tags from source operands to the destination if the source has any taint tags. In the case of API Chaser, it generates its original taint handling function for code taint propagation. The function adds malware-tags to the writing destination if the source operand does not have any taint tags and the opcode has a malware-tag.

4.4.3 Shadow Memory, Disk, and Virtual DMA Controller

Shadow memory is an array of eight-byte entries where each entry corresponds to a byte on physical memory. Argos originally has shadow memory, but it has only a

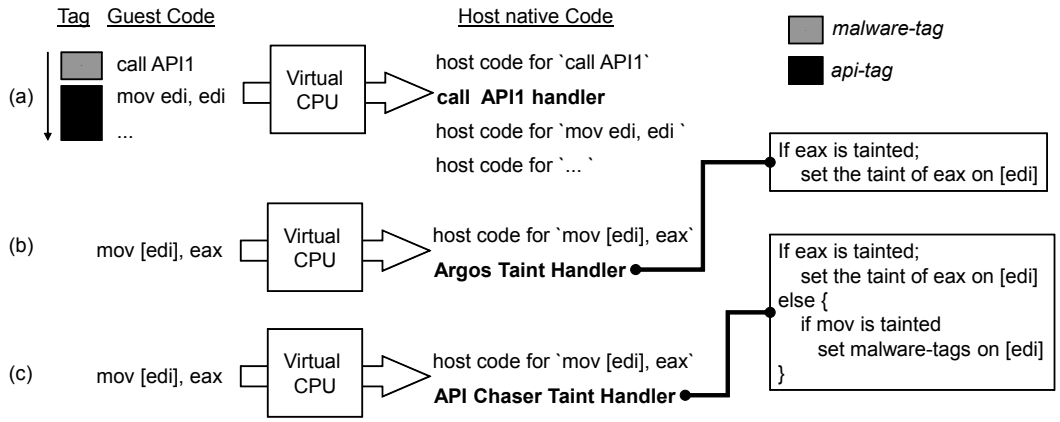

Fig. 4.6 Examples of Dynamic Binary Translation

one-byte taint tag space for one byte on the physical memory, which is only used for determining the taint state. We extend it to an eight-byte taint tag space for one byte to store a pointer to the API tag data structure. Therefore, we need memory space eight times as large as the physical memory for the shadow memory. For example, if the size of the physical memory is 256 Mbytes, the size of the shadow memory is 2 Gbytes. Considering large-scale analysis, we design the shadow memory to be dynamically allocatable at the time when it becomes necessary in order to suppress the increase in simultaneous memory consumption when running multiple API Chaser instances at the same time.

The shadow disk is a binary-tree data structure for storing taint tags added to data on a disk. The entries for the structure contain information related to tainted data on a disk such as the sector number, offset, size, taint tag buffer, and pointers represented by the nodes of the binary tree. A taint tag entry for one-byte data on a disk takes eight bytes of space, so we need eight times as large a memory space as a disk for the shadow disk. However, the size of the disk is much larger than that of the physical memory, so it is difficult to allocate sufficient memory space to store the taint tags of all data on a disk beforehand. Thus, we design the memory space for the shadow disk to be dynamically allocated as needed. Argos does not have a shadow disk, so we newly implemented it for API Chaser.

In API Chaser, the virtual DMA controller transfers taint tags between the shadow memory and a shadow disk. API Chaser monitors DMA commands at the virtual DMA controller, and when it finds a request for transferring data, it acquires the data location from the request and confirms whether or not the transferred data has taint tags. If it does, the virtual DMA controller transfers the taint tags between the shadow memory and shadow disk. Argos does not have this mechanism either, so we newly implemented it for API Chaser as well.

4.4.4 API Argument Handler

To obtain more detailed information of API calls, we extract argument information passed to them when they are called and when the execution is returned from them. To do this, we prepare an API argument handler for each API. We extract the argument information such as the number of arguments, variable types, size, and whether it is an input or output argument from the Windows header files provided by the Windows software development kit (SDK). For undocumented APIs, we extract their information from the web site [79] and source code of React OS [80]. We register an API argument handler to an API tag data structure when we create the data structure for adding api-tags to the instructions of each API. The handler is invoked from the virtual CPU when it detects an API call invoked from malware and outputs detailed argument information related to the API into a log file.

4.4.5 Hot-boot

We use the snapshot capability of QEMU for hot-boot, which skips the boot process of a guest OS and enables quick initialization of analysis. Taint analysis provides deep insight into malware behavior by adding data-flow analysis. However, one drawback to taint analysis is performance degradation as measured in Subsection 4.5.5. In our brief experiment, it took more than 10 mins to boot a guest OS on API Chaser from the cold disk image, i.e., cold-boot, and be ready for analysis. This performance penalty may become a bottleneck for API Chaser for use in industrial use-cases considering that an anti-virus company reportedly collects a plethora of

large-scale malware samples per day from the Internet or their customers.

To compensate for such a performance penalty, we implemented a hot-boot capability using `savevm` and `loadvm` QEMU Monitor Protocol (QMP) commands, which save the state of a guest OS running on QEMU and restore that state, respectively. We extended these two commands with a capability for handling shadow memory. That is, the hot-boot capability stores the state of the shadow memory when `savevm` is executed by taking a snapshot and restores the state when `loadvm` is issued for a guest OS to restart running from the taken snapshot. With the extended `savevm` and `loadvm` commands, we achieve hot-boot in API Chaser. This capability allows us to initiate an analysis immediately after launching API Chaser and makes it practical when analyzing large-scale malware.

We can use hot-boot and pre-boot tainting at the same time without compromising any advantages from pre-boot disk tainting. When the `savevm` command is issued after a guest OS has been booted and ready for analysis, the API code, which is tainted with api-tags, may exist in both the memory and disk. To handle this situation, when we take a snapshot, we can save the status of the shadow memory using the extended `savevm`. When we load the taken snapshot, we can restore the status of the shadow memory with the extended `loadvm` command for the API code on memory and perform pre-boot disk tainting to taint the API code on the disk.

4.4.6 Parallel Analyses

To analyze large-scale malware samples, it is natural to run multiple instances in parallel. For that purpose, we must reduce the simultaneous consumption of memory and disks because the physical resources such as memory or disks are limited on a machine. To reduce the memory consumption, we developed dynamic shadow memory and disk allocations as explained in Subsection 4.4.3. Additionally, to reduce disk space consumption, we developed a one-time disk image. We explain this in this subsection.

One-time Disk Image

Disk space is a physically-limited resource. So, one requirement for API Chaser as a practical analysis environment is to reduce the consumption of disk space to run multiple API Chaser instances concurrently. Another requirement for API Chaser is that it must have a capability for returning to a clean state after one analysis has completed because the environment may be destroyed or compromised by the malware under analysis. One option for this requirement with QEMU is to use a `-snapshot` option, which keeps the original disk read-only and redirects all disk-writes to a temporal disk. However, unfortunately, QEMU version 1.1.50 on which Argos was built does not support this `-snapshot` option for a guest OS booted with `loadvm` [81].

To satisfy these two requirements at the same time, we developed a one-time disk image for API Chaser. The one-time disk image is an extended qcow2 image file format[82]. We leverage the backing_file capability of the qcow2 image format to retain the clean state. The one-time disk image works as follows. First, we cold-boot a guest OS with the `-snapshot` option and take a snapshot with `savevm` after the booting has completed and is ready for analysis. Due to the `snapshot` option, QEMU creates a temporal disk image in the `tmp` directory. Second, we configure the backing_file option of the temporal disk image with the original disk image. This configuration allows read-access to the data that do not exist in the temporal disk image to be redirected to the original disk image. In addition, this configuration allows write-access to be routed to the temporal disk image.

Using this capability, we retain the read-only nature of the original disk image and redirect all write access to the temporal disk image even after a guest OS is restarted with the `loadvm` command. When we begin analysis, we simply copy the configured temporal disk image, rename it, and boot a guest OS from the copied temporal disk image as an analysis environment of API Chaser. After analysis is complete, we simply discard the copied temporal disk image and restart a new analysis by copying a new temporal disk image from its original. This approach allows us to prepare only one original disk image and multiple copies of the temporal disk image for multiple

analysis environments. Since the size of a temporal disk image is much less than that of the original disk image, we can reduce the consumption of disk space for preparing multiple disks for multiple instances. Additionally, we can retain the clean environment state for each analysis.

4.5 Experiments

To show the effectiveness of API Chaser, we conducted four experiments to evaluate the accuracy, the analysis capability for new emerging evasion techniques, the capability for large-scale analysis, and the performance of API Chaser.

4.5.1 Experimental Environment

All experiments were conducted on a computer with Intel Xeon CPU E5-1650 v4 3.6 GHz, 64 G memory and 360 G SSD. API Chaser runs on Ubuntu Linux 14.14, and the guest OS was Windows XP Service Pack 3 (WinXPsp3) or Windows 7 Service Pack 1 (Win7sp1). The guest OS was allocated 256 Mbytes for its physical memory in the case of WinXPsp3 and 1 Gbyte in that of Win7sp1. We targeted 6,862 APIs in major Windows system DLLs for monitoring.

4.5.2 Accuracy Experiments

We evaluated API Chaser from the viewpoint of its resistance against hook evasion and target evasion. We prepared several malware executable files that have various evasive functions and we used them to evaluate the resistance of API Chaser against hook evasion and target evasion. As a comparative environment, we prepared two different implementations of API Chaser that respectively use existing techniques to detect API calls (Type I) or identify target code (Type II). In each experiment, we executed some malware on API Chaser and one of these comparative environments for five minutes, acquired API logs that were respectively output by the two environments, and then compared them. When there were some differences between the logs, we revealed the causes of the differences by manually analyzing malware and

Table 4.1 Results of Hook Evasion Resistance Experiment

Virus Name	API Chaser	Type I	Unmatched	Reason	Evasion Technique
Win32.Virut.B	6,361	4,852	1,509	F.N. of Type I	API Hook
Themida	43,994	41,028	2,966	F.N. of Type I	Stolen Code
Infostealer.Gampass	38,382	1,397	37,485	F.N. of Type I	Sliding Call
Packed.Mystic!gen2	97,364	97,363	1	F.N. of Type I	Sliding Call

investigating the infected environment using IDA [62] and The Volatility Framework [54] to determine whether the fault was in API Chaser or in the comparative environments. We used WinXPsp3 as a guest OS of API Chaser for the experiments.

Hook Evasion Resistance Experiment

We used four real-world malware samples which have hook evasion functions with stolen code, sliding call, or API hooking. We include malware using API hooking into the samples for this experiment because the behavior of API hooking is similar with the one of stolen code and it is also able to evade API monitoring as stolen code does. Regarding the other two hook evasion techniques introduced in Chapter 2, i.e., name confusion and copied API obfuscation, we could not find any malware sample using them in the wild. So, we qualitatively discuss the resistance capability of API Chaser against these two techniques in Chapter 2.

With the four malware samples, we executed them on both API Chaser and a comparative environment (Type I). Type I is another implementation of API Chaser with a different technique to detect API calls. It detects API calls by comparing an address pointed to by an instruction pointer to addresses where APIs should reside, which is a common existing technique. The other components of Type I are the same as API Chaser.

■Results Table 4.1 lists the results of this experiment. We manually investigated the causes of the differences in captured API calls and revealed that all of them were caused by false negatives of Type I. We explain the details of the two cases, Themida and Mystic!gen2, although the others also had the same reason for their differences. In the case of Themida, API Chaser captured 2,966 more API calls than Type I. All

the unmatched API calls were detected in the dynamically allocated and writable memory area. On the other hand, all the matched API calls were detected in the memory area where system DLLs were mapped. We manually confirmed that all API calls, except for API calls with no arguments that API Chaser detected, had valid argument information. Thus, these were not false positives of API Chaser, but false negatives of Type I. As we mentioned, API Chaser can detect the stolen API call by propagating taint tags added to an API to the stolen instructions, while Type I cannot because it does not track the movement of the stolen instructions. This capability contributes to the resistance of API Chaser against hook evasion techniques. In the case of Packed.Mystic!gen2, we confirmed that it used the sliding call technique. The following code snippet is from a sliding call in this malware.

```
0x00408175 push ebp
0x00408176 mov ebp, esp
0x00408178 sub esp, 20h
0x0040817B cmp dword ptr [eax], 8B55FF8Bh
0x00408181 jnz loc_40818C
0x00408187 add eax, 2
0x0040818C add eax, 6
0x00408191 jmp eax ;to API+2 or API+6
```

The `cmp` instruction at 0x0040817B confirms the existence of the following four bytes, 0x8B, 0xFF, 0x55, and 0x8B at the address stored in `eax`, which points to the head of an API. These four bytes may indicate the assembler instructions, `mov edi, edi; push ebp; mov ebp, esp;`, which is a prologue for a hotpatch-enabled API, which has sufficient space for hooking before the first instruction of the API. In fact, the total size of the three assembler instructions is a total of six bytes. If the malware finds these four bytes at the entry of the API, it jumps to a location at six bytes after the entry of the API to avoid monitoring. API Chaser adds taint tags to all instructions in each API, so it was able to detect the execution of the instruction at API entry + 0x6 and identified it as an API call from malware.

Table 4.2 Results of Target Evasion Resistance Experiment (Tracking)

Virus Name	Description of Evasion Technique	Result
Win32.Virut.B	Infecting files with CreateFileMapping	✓
	Injecting code with WriteProcessMemory	✓
Trojan.FakeAV	Injecting code with WriteProcessMemory	✓
	Changing the name of rundll32.exe to jahjah06.exe	✓
Infostealer.Gampass	Injecting code with WriteProcessMemory and the injected code loads a dropped DLL	✓
	Changing its name to svchost.exe	✓
Spyware.perfect	Injecting a dropped DLL with SetWindowsHookEx	✓
Trojan.Gen	Injecting a dropped DLL via AppInit_DLLs registry key	✓
Backdoor.Sdbot	Executing a dropped EXE as a service	✓

Target Evasion Resistance Experiment

We prepared six real-world malware with target evasion functions. Using these malware, we evaluated the following two capabilities of API Chaser: tracking the movement of target code and identifying the target code in a code-injected process or executable file. As for the tracking capability, we confirmed that API Chaser can capture API calls from a process or executable code-injected file by the six malware. In regard to the identifying capability, we prepared another comparative environment (Type II). The Type II environment is different from API Chaser in identifying target code and tracking code injection. It identifies its target depending on the PID and tracks code-injection based on invocation of specific API calls and DLL loading events. For example, Type II hooks the invocations of WriteProcessMemory API calls and extracts the PID of the destination process of the writing from its arguments. Then, it includes the PID into its monitoring targets. The Type II components except for those for identifying and tracking target code are the same as API Chaser.

■Results Table 4.2 lists the results of the tracking experiment. API Chaser successfully tracked all the behaviors of the injected code without being evaded. We consider that Type II can also track them if it knows how target malware evades monitoring and it prepares mechanisms for tracking the behaviors beforehand. However,

it is practically difficult to know all code injection methods and prepare for them before executing target malware because there are many unpublished functions in Windows and third party software. On the other hand, API Chaser can track code injection by propagating taint tags added to target malware. Since API Chaser does not depend on individual code injection mechanisms, we can say that it is more generic than the existing approach depending on each injection method for tracking them.

Table 4.3 lists the results of the identifying experiment. We manually investigated the causes of the unmatched API calls and revealed that all the unmatched API calls were caused from false positives of Type II. That is, API Chaser successfully identified all API calls invoked from an injected code in a benign process and eliminated API calls invoked from the benign part of code in the process. We explain the details of the two specific cases, Trojan.FakeAV and Infostealer.Gampass, although the others also yield the same results. In the case of Trojan.FakeAV, all the matched API calls were invoked from the dynamically allocated memory area which was allocated and written by Trojan.FakeAV, while unmatched API calls were invoked from the memory area where explorer.exe was mapped. This indicates that API Chaser captured the API calls invoked from the code injected by Trojan.FakeAV and Type II additionally captured API calls invoked from original code in the code-injected benign process. In the case of Trojan.Gen, all the matched API calls were invoked from tzdfjhm.dll, while all the unmatched calls were from the memory area where notepad.exe was mapped. Library tzdfjhm.dll was registered to the registry key, AppInit_DLLs, which is used by malware for injecting a registered DLL into a process. The DLL was dropped and registered to the key by Trojan.Gen.

4.5.3 Synthetic Malware Experiment

The purpose of this experiment is to show the feasibility of API Chaser against state-of-the-art evasion techniques including those introduced in academic studies. For that purpose, we collected proof-of-concept (PoC) codes of the following techniques, Process Hollowing[3][4], AtomBombing[5][6], PowerLoaderEx[7], Shim-based DLL Injection[8], and Stealth Loader[9]. Then, we generated synthetic

Table 4.3 Results of Target Evasion Resistance Experiment (Code identification)

Virus Name	Injected Process	API Chaser	Type II	Unmatched	Reason
Win32.Virut.B	notepad.exe	315	3,020	2,705	F.P. of Type II
Win32.Virut.B	winlogon.exe	184	783	599	F.P. of Type II
Trojan.FakeAV	explorer.exe	20	1,782	1,762	F.P. of Type II
Infostealer.Gampass	explorer.exe	147,646	149,408	1,762	F.P. of Type II
Spyware.perfect	notepad.exe	4,792	7,511	2,719	F.P. of Type II
Trojan.Gen	notepad.exe	230	3,222	2,992	F.P. of Type II

malware samples based on the PoC codes and analyzed them with API Chaser. We used Win7sp1 as a guest OS of API Chaser for this experiment. The reason why we focus on these five techniques is that they appeared or became major after our paper was first published in 2013 [83]. So, these techniques represent new techniques for API Chaser and if we can precisely analyze the malware with these techniques with API Chaser, we can demonstrate that the design of API Chaser is possibly strong enough for analyzing future-emerging techniques.

Process Hollowing

Process Hollowing is a technique that hides the presence of a malware process. First, it creates a benign executable file process that is suspended. Next, it overwrites the contents of the suspended process with those of a malicious executable file, and then resumes execution of the process after overwriting is completed. The PoC code for this technique, [4] creates a process of svchost.exe that is suspended and overwrites its contents with that of the specified (malicious) executable file.

■Result API Chaser successfully captured the APIs invoked from the specified executable file overwritten in the process of svchost.exe. API Chaser kept tracking the movement of the overwritten code of the specified executable code by propagating tags added to the code so that APIs invoked from the code were the control transfers from tainted code to API code.

AtomBombing

AtomBombing is a technique that injects a (malicious) code snippet into explorer.exe and executes it without using the APIs commonly used for code injection: WriteProcessMemory, VirtualAllocEx, and CreateRemoteThread. AtomBombing takes advantage of the atom table which is a shared data structure already mapped into explorer.exe. AtomBombing maps a code snippet into the virtual memory space of explorer.exe by writing it in the atom table. After the code mapping is completed, AtomBombing invokes the NtQueueApcThread API to hijack a thread with sleeping status in explorer.exe. Then, when the sleeping thread wakes up, the thread starts executing the mapped code. AtomBombing uses the return-oriented-programing (ROP) technique to avoid the issue in which the memory areas of the atom table do not have the executable permission.

■Result API Chaser successfully captured the API calls invoked from the code injected into explorer.exe since the caller instructions of these APIs were written by this synthetic malware, i.e., they were tainted. However, API Chaser missed capturing the API calls invoked from the ROP gadgets. The reason for this was that the caller instructions of these API calls in the ROP gadgets belonged to a benign code. So, these caller instructions have the benign-tag and control transfer of this API call is from benign-tag to api-tag. We discuss this issue in more depth in Subsection 4.6.3.

PowerLoaderEx

PowerLoaderEx is an evolved version of PowerLoader[7]. This is also a technique that injects a code snippet into a benign process and executes it without using the three APIs. PowerLoaderEx first writes code in a desktop heap memory, which is a shared area among GUI applications. Second, it overwrites the function pointer that handles a specific type of window message using the SetWindowLongPtr API. Then, it intentionally generates the window message to kick the handler with the SendNotifyMessage API. Since the desktop heap is basically not executable, PowerLoaderEx employs the ROP technique to add executable permission to the injected

code in a manner similar to AtomBombing.

■Result Similar to the AtomBombing case, API Chaser could capture the API calls invoked from the injected code, while it failed to capture the ones from ROP gadgets. This is also discussed more in Subsection 4.6.3.

Shim-Based DLL Injection

Shim-based DLL Injection is a technique that injects a DLL into a benign process by taking advantage of the application compatibility mechanism of Windows; Microsoft officially prepares a mechanism that ensures backward compatibility in most of their products. This is currently implemented by the Application Compatibility Framework (ACF). The ACF is capable of intercepting API calls, controlling the loading process of DLLs, and patching memory. The PoC code for this technique [84] generates an sdb file while configuring its inject-target program, an injecting DLL installs the sdb file with the `sdbinst` command, and the PoC code kicks the target program. When the target program is executed, the ACF injects the configured DLL into the process of the target program.

■Result API Chaser can track the injection of the DLL if the DLL is tainted. Taint analysis allows API Chaser to track the movement of its target code with taint tags without being affected by the injection manner. In real-world malware, since an injecting DLL is downloaded or dropped by the malware, the DLL becomes tainted on API Chaser. So, we do not miss capturing API calls invoked from the DLL injected into a process.

Stealth Loader

Stealth Loader is a program loader that loads Windows system DLLs such as kernel32.dll and ntdll.dll without leaving any trace to be detected. By loading a system DLL with Stealth Loader, the loaded DLL is not recognized as being 'loaded' by the Windows OS or even analysis tools. Since analysis tools fail to recognize the existence of the loaded DLL, they also fail to capture API calls of the functions exported from the unrecognized system DLL.

■Result API Chaser recognizes the calls of the function exported from stealth-loaded system DLLs as 'API calls'. This is because API Chaser identifies an API call based on taint tags added to the API before initiating an analysis (pre-boot disk tainting), and it does not rely on any metadata managed by the Windows OS, which is the portion Stealth Loader attacks. So, if Stealth Loader deceives the Windows OS by not leaving any trace identifying the existence of loaded DLLs, API Chaser is not affected by that.

4.5.4 Large-Scale Malware Analysis Experiment

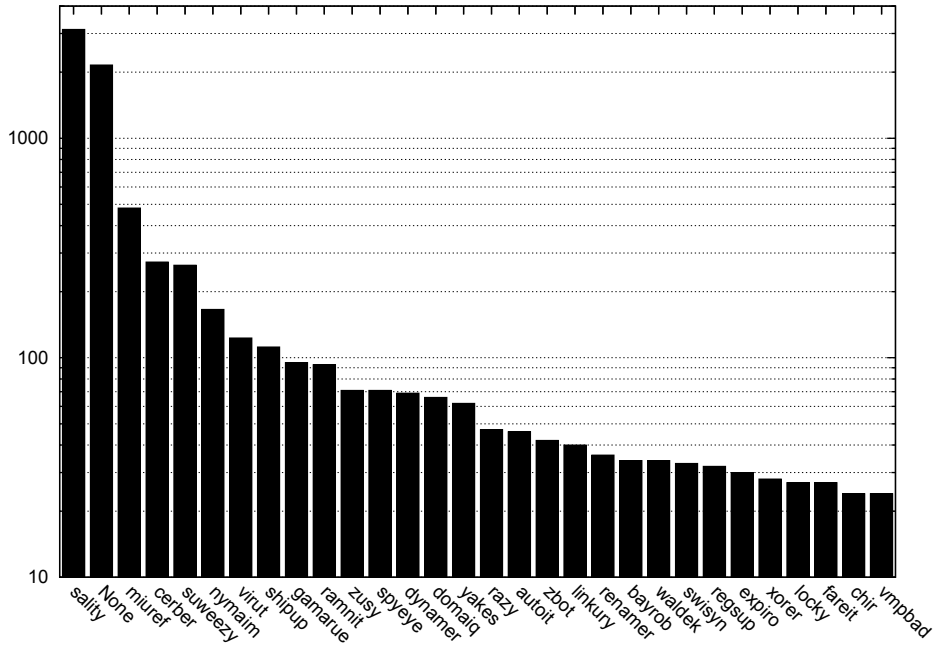

Fig. 4.7 Top 30 malware families in 6,722 samples. X-axis is malware family, while y-axis is the number of samples in each family with logarithmic scale.

The goal of this experiment is to show how much major hook and target evasion techniques are among real-world malware samples. To achieve this goal, we col-

lected a certain number of malware from various sources and analyzed them with API Chaser to find malware using these techniques. We call malware using hook evasion techniques *hook-evasive* malware, whereas we call malware using target evasion techniques *target-evasive* one in this experiment.

Dataset

As the dataset for this experiment, we totally collected 8,979 malware samples from various data sources including malware exchange with an industrial vendor and our own honeypots. Then, we filtered out 100 samples whose hash value was duplicated. We used the 8,879 samples for this experiment. Next, we downloaded the anti-virus scan reports of 6,722 samples from VirusTotal [65] because we needed them for classification with AVClass [74]. The other 2,157 samples did not have any report in VirusTotal, which means they had never been uploaded for scanning onto VirusTotal, so we made None family for them. As a result, we classified the 6,722 samples into 420 families with AVClass and thus we classified 8,879 samples into 421 families including None.

Figure 4.7 shows the major top 30 malware families in the samples. Sality is the most major family in them and it has 3,131 samples, which occupies about 28% of the dataset. None is the second major family and it has 2,157 samples, which occupies about 24%. Since with only the two families, we can occupy about 60% of the total dataset. So, this dataset has a certain amount of bias. Thus, when we show the results of this experiment, we show not only the number of samples, but also the number of the families to mitigate this bias.

Procedure

As an analysis environment, we prepared three API Chaser instances to analyze the samples in parallel and we configured the analysis time to be 5 mins per analysis. After 5 mins elapsed, we forcibly terminated the API Chaser process even though the malware under analysis was still running. We used Win7sp1 as the guest OS of API Chaser for this experiment.

After all analyses had been done, we found malware samples using hook evasion or target evasion techniques by parsing each analysis log. We found hook-evasive

malware samples from API call logs as follows. We first calculated the virtual addresses of each API from the base address of a loaded DLL and its export function table. Next, we compared the address logged in a log file as an API call destination to the calculated address, and then if the addresses were not matched, we identified the malware as hook-evasive.

We also found target-evasive malware samples as follows. We first created a list of processes which were in parent-child relationship with monitored malware based on specific API calls, such as CreateProcessA/W. When we found an API call invoked from a process which was not on the list, we identified the malware sample as target-evasive because the API call possibly came from a injected code with target evasion.

Result

We found 701 hook-evasive malware samples in the dataset. 476 samples out of 701 belonged to miuref malware family and this was the most popular one in the dataset. 104 samples belonged to None family and this is the second popular one. The others belonged to any one of 34 families. The top 5 families were miuref(476), None(104), ramnit(66), sality(3), and dynamer(3) whose numbers in the parentheses express the number of samples belonging to these families. Regarding target-evasive malware, we found 344 target-evasive malware samples. 56 samples out of 344 belonged to None family and this was the most popular one in the dataset, while the others belonged to any one of 83 families. The top 5 families were None(56), bayrob(32), sality(31), gamarue(21), and parite(19). Since hook evasion techniques were used in 8.5% families of all dataset families and target evasion techniques are used in 19.9%, we argue that these evasion techniques are major and often used among malware to intensionally hide API calls.

Through these analyses, we collected a total of 5,133,292,748 API call logs. 4,771,863 out of 5,133,292,748 API calls, which were about 0.09% of the total API calls, were intentionally hidden with hook evasion techniques, while 172,859,468 API calls, which were about 30% of the total API calls, came from a injected code with target evasion techniques. As we showed in the accuracy experiments,

API Chaser was able to capture these API calls without being evaded and then we could collect the arguments passed to these API calls correctly, which possibly contain useful information as an indicator for detecting malware, i.e., indicator of compromise (IOC). However, if you analyze these evasive malware with an analysis environment whose architecture for API monitoring is based on the ones of Type I or Type II, you may miss capturing some of these API calls or excessively capture them, respectively, if the analysis environment does not care about evasion techniques at all. These inaccuracies of API call monitoring possibly lead to both false negatives and positives.

Lastly, regarding analysis times, we started the analysis at 2017/2/7 19:50:59 and finished at 2017/2/23 1:35:33. We spent approximately 382 hours to analyze the 8,879 samples. On average, we spent 7.75 minutes (= (382*60)/(8879/3)) to analyze one sample with one API Chaser instance, even though we configured the analysis time to be set to 5 mins for the analysis. This time difference comes from the time for preparing the analysis environment before initializing the analysis and that for compressing logs after analysis.

4.5.5 Performance Experiment

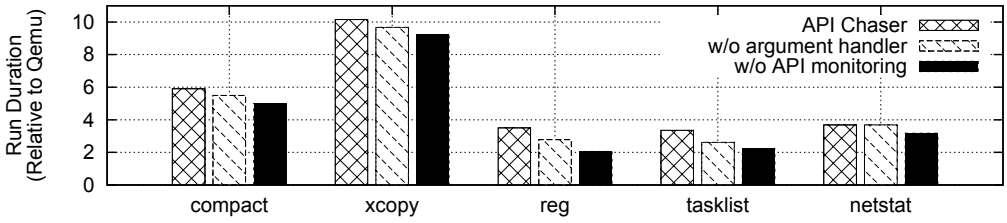

Fig. 4.8 Results of performance experiment: Number of captured API calls during the execution of each command is as follows: compact is 28,464, xcopy is 1,222, reg is 44,059, tasklist is 8,271, and netstat is 103.

The goals of this experiment are to show how much of performance degradation API Chaser has, compared to a vanilla QEMU, and where the degradation mainly comes from. This is because we put a higher priority on precision for API monitoring

rather than performance as the fundamental design of API Chaser. This design choice may impose on performance penalties. To know the impact of the penalties, we conduct this experiment and clarify the most influential part in API Chaser.

We used five Windows standard commands for this experiment. With these commands, we could cover APIs of major behaviors which we should focus on, such as file, registry, network, process, or memory-related behaviors.

As comparative environments, we prepared three environments: vanilla QEMU (`QEMU`), API Chaser without API monitoring (`w/o API monitoring`), and API Chaser without argument handlers (`w/o argument hander`). The reason why we prepared these comparative environments was to clarify which functionality of API Chaser mainly causes performance degradation. As we explained, we added several functionalities to QEMU to implement API Chaser; we could roughly classify the added functionalities into three groups: functionality related to taint analysis, API hooking (API monitoring), or argument handling. API Chaser, of course, has the three functionalities, i.e., taint analysis, API monitoring, and argument handling. The `w/o argument handler` environment has taint analysis and API monitoring but it does not have argument handing. The `w/o API monitoring` environment has only taint analysis and it does not have both API monitoring and argument handling. `QEMU` does not have any of them. By comparing the performances of them, we could identify the most influential functionality in the three ones to the performance of API Chaser. In addition, we used WinXPsp3 as the guest OS of API Chaser for this experiment.

Result

Figure 4.8 shows the relative run duration of these five commands on each environment compared to relative QEMU, which is set to 1. The results show that the degradation in performance of API Chaser was approximately 3 to 10 fold compared to that for `QEMU`. As you can see, the performance degradation mainly came from the taint analysis functionality because the difference between `QEMU` and `w/o API monitoring` was larger than the others, i.e., difference between `w/o argument handler` and `w/o API monitoring` or one between API Chaser and

`w/o argument hander`.

We consider that the degradation is not a severe limitation to API Chaser because the current version of API Chaser has not been optimized to reduce its overhead. We consider that there is much room for improvement, for example, applying the work done in [85] to API Chaser. In addition, we discuss in Subsection 4.6.3 an issue that arises from the performance degradation when we analyze malware in terms of checking the delay of execution.

4.6 Discussion

In this section, we discuss the resistance capability of API Chaser against hook evasion techniques which we did not have experiments in Subsection 4.5.2, multiple operands handing for taint propagation, and the limitations of API Chaser.

4.6.1 Other Hook Evasions

In the accuracy experiment in Subsection 4.5.2, API Chaser was not affected by hook evasion techniques used in real-world malware. However, we could not find any malware in the wild using name confusion or copied API obfuscation. So, we qualitatively discuss the resistance capability of API Chaser against these two techniques.

We consider that API Chaser is not affected by name confusion because of the following considerations. When a DLL is copied, the taint tags set on the DLL are propagated to the copied DLL. Even if the name of the DLL is changed, it does not affect taint propagation at all because taint propagation is conducted at (virtual) hardware layer without depending on the semantics of an OS or a file system and the change of file names is a matter of OS or file system layers. Thus, when API code in the copied DLL is executed, we can capture the execution of the API correctly because the propagated taint tag has existed on the code of the API.

We also consider that API Chaser is not affected by copied API obfuscation. As we explained in Chapter 2, copied API obfuscation is similar with stolen code. Copied API obfuscation copies all instructions of an API, while stolen code does

the first few instructions of the API. Considering a case that a copied or stolen API is invoked from a malicious code, there is no difference in control transfers between them. That is, an execution control is transferred from the malicious code to the first instruction of the copied or stolen API. Since we have already demonstrated that API Chaser handles stolen code properly, we believe that it could do copied API obfuscation as well.

4.6.2 Multiple Operands for Taint Propagation

When API Chaser handles an instruction which has more than two operands and both of them have different taint tags, a taint propagation depending on one of the tags is disconnected. This is because, in our implementation of API Chaser, we simply propagate the taint tag of the first operand and discard the one of the second operand when we encounter this situation.

To mitigate this issue, an approach is to make a priority based on the types of taint tags for propagation. When we encounter this situation, we decide which tag should be propagated to the destination operand based on the priority. Another one is to generate a new tag and make a relationship between the new tag and the tags of each operand to construct the data-flow with them in an offline analysis. In either case, we need to add a code to the handlers for these types of instructions, i.e., instructions with multiple operands. Since these instructions often appear during an analysis, the impact of the additional code is not so small. We need to achieve a balance between performance and precision when we adopt one of these approaches to API Chaser.

4.6.3 Limitation

We discuss limitations of API Chaser from viewpoints of detection-type anti-analyses, scripts, return-oriented-programming, and implicit flow.

QEMU Detection and Timing Attack

With the exception of evasion techniques, malware often uses QEMU detection and timing attacks [2] to detect analysis environments. API Chaser is also possibly

affected by these two techniques. So, we discuss the two techniques below.

Several methods for detecting QEMU have been studied and proposed [18][86][87][88]. To avoid these detections, we individually managed to let QEMU-specific artifacts become invisible to malware. For example, we changed the product names of virtual hardware in QEMU for detection techniques that depends on these names. We also changed the behaviors of specific instructions by finding the execution of these instructions and dynamically patching them at runtime.

Timing attacks are a technique that checks the delay in executing a specific code block. We designed API Chaser to focus on accuracy rather than performance; therefore, it takes several more seconds to execute part of a code block than in real hardware environments. As for this technique, we can overcome this with the same approach as that used in our previous study [29], which controls the clock in a guest OS on API Chaser by adjusting the tick counts in the emulator to remove the delay.

Scripts

API Chaser has a limitation for analyzing script-type malware, e.g., a visual basic script or a command script. These scripts are executed on some platforms such as an interpreter or a virtual machine. Although these scripts have the taint tags of malware, API Chaser cannot detect their execution because the instructions executed on the virtual CPU are those of their platform and not those of the tainted script. To address this problem, we are currently considering a way to identify the target code with both taint tags and semantic information such as PID and TID.

Return Oriented Programming

API Chaser faces a limitation when an API is called in a manner similar to ROP [89]. For example, when an attacker constructs a ROP chain by pushing the address of a specific API and executing the `ret` instruction in a benign code region to jump to the API, the control transfer in this case is from a benign-tag to an api-tag. So, API Chaser fails to identify this control transfer as a monitoring target.

We may be able to handle this case by extending taint-based control transfer interception using the approach that Korczynski *et al.* [51] proposed. That is,

when a control transfer instruction such as `ret`, `call`, or `jmp` is executed and the destination address of the instruction is tainted, we identify such a control transfer and the first basic block at the destination address as a part of malicious code.

Another option is to detect the ROP code. If we can detect the ROP code, we may be able to identify the execution of APIs called from malware via the detected ROP code. Detection of ROP is outside the scope of this paper and we leave it for other studies. Many studies leverage the unique behavioral characteristics of the ROP code such as its use of many ret instructions, jumps to the middle of an API, or jumps to an instruction of non-exported functions.

Implicit Flow

Another limitation of API Chaser is due to feasibility issues of taint propagation, e.g., implicit flow. If malware authors know the internal architecture of API Chaser, especially code taint propagation, it may be possible to cause intentionally API Chaser to yield false positives or false negatives using implicit flow. For example, malware reads a piece of code in a benign program and processes the code through implicit flow which does not change its value. Then it writes the code back to the same position. As a result, the taint tags on the code are changed from benign to malware. Due to this, if malware executes the written code, API Chaser identifies the execution as the one of malware, even though the code is truly equivalent to benign code. On the other hand, if malware reads a piece of code in an API and conducts the same process, it overwrites the taint tags for API with those for malware. Thus, API Chaser deals with the execution of the code as one of malware. To address this problem, we must improve the strength of the taint propagation, for example, as done in [90][91]. We consider this as our future work.

4.7 Conclusion

Analyzing evasive malware in a dynamic manner is a challenging problem for anti-malware research especially in developing practical malware analysis environments. In this chapter, we provided a solution by using API Chaser, which is a prototype system of our API monitoring technique. API Chaser was designed and implemented

to prevent malware from evading API monitoring. We conducted experiments using actual malicious code with various types of evasion techniques to show that API Chaser correctly works according to its design of being difficult to evade. We believe that API Chaser will be able to assist malware analysts in understanding malware activities more correctly without spending a large amount of effort in reverse engineering and contribute to improving the effectiveness of anti-malware research based on API monitoring.

Chapter 5

Taint-Assisted Static Malware Analysis

5.1 Introduction

In this chapter, we introduce a new API name resolution technique based on taint analysis to fight against semantic evasion techniques in static analysis. The key idea is that we assign a unique taint tag to each API and resolve the API name from the assigned taint tag. To this end, we first define taint tags, each of which has a unique value for each API. We then apply the taint of the API to each of its instructions. Next, we run a malware under analysis in an isolated environment while performing taint analysis using the tags. In particular, we propagate them by following pre-defined rules when the malware under analysis moves or makes a copy of API code. Then, we acquire the dump files of taint tags as well as that of (virtual) physical memory when dynamic analysis has completed. Finally, we perform IAT reconstruction on the basis of the dump files. In the IAT reconstruction, we resolve the API names of each address in an IAT from the taint tags of the code pointed to from each address in the IAT.

This technique has two advantages for countering semantic evasion techniques. First, we can conduct fine-grained tracking for API code and identify it correctly even when it is wholly or partly placed out of the memory range where a DLL has

been mapped. This is because we can track the movement of the instructions of each API at instruction-level granularity with taint analysis. Second, we can identify the positions of each DLL or API without depending on specific data structures that the OS manages. This is because we manage them with the tags used for tainting API code and propagated independently from OS's behaviors. These advantages allow our API name resolution to be independent of the OS semantics and realize reliable API name resolution on the unreliable OS without being evaded by semantic evasions.

We have implemented this technique in a system, which is composed of preprocessing, dynamic analysis, and dump analysis phases. In the preprocessing phase, we correctly identify the position of each target DLL in a disk image by using a disk forensics tool, The Sleuth Kit (TSK) [77], and then set taint tags on them. For dynamic analysis, we use API Chaser [83], which is a sandbox with taint analysis capability. We have extended API Chaser to generate dump files at any arbitrary time during the execution of the malware. For dump analysis, we have extended The Volatility Framework (Volatility) [16] with capabilities of reading taint tags and disk forensics. We call this extended Volatility *TaintVolatility*. We have developed a plugin running on TaintVolatility on the basis of `impscan`[16]. We call this plugin *tf_impscan*. Using tf_impscan with TaintVolatility, we identify the IATs and resolve the API names from taint tags for IAT reconstruction. Finally, for an output of this system, we generate an IDC (IDA script) for adding resolved API names to the disassembled code of IDA.

To assess the effectiveness of our system, we have conducted three types of experiments. The first one is for evaluating whether our technique is more resistant to semantic evasion techniques than existing IAT reconstruction tools. For that purpose, we prepared several Windows executables and obfuscated their APIs by using the five semantic evasion techniques mentioned in Chapter 2 with Stealth Loader. Then, we analyzed them using our system and three other IAT reconstruction tools: impscan, impscan++[*1], and Scylla[17]. The results show that only our system could correctly identify all imported APIs, whereas the others could not

[*1] A tool we developed for this experiment.

because of semantic evasion techniques. The second one is for showing the effects of disk forensics integration. For that purpose, we analyzed several Windows executables with differently configured TaintVolatility, i.e., with or without disk forensics integration. The results of this experiment show that the disk forensics capability of TaintVolatility works correctly and contributes to generating better results for IAT reconstruction. The third one is for measuring the performance degradation of TaintVolatility, compared to impscan. The results of this experiment show the degradations are within practical range and do not impose serious impacts on the effectiveness of TaintVolatility.

5.2 Our Approach

In this section, we introduce a taint-based API name resolution technique, which is resistance against semantic evasions. First, we define the goal and scope of this chapter. Second, we introduce a taint-based API name resolution. Finally, we explain a system for IAT reconstruction using our taint-based API name resolution.

5.2.1 Goal and Scope

Our goal in this chapter is to present the first generic technique against semantic evasion techniques and to ensure a system in which the approach implemented works well in practice. *'Generic'* in this context means a technique commonly applicable for all types of semantic evasion technique, which is explained in Chapter 2, without heuristics or adjustments depending on targets.

The scope of this chapter is to solve the problem of API name resolution, i.e., the existing API name resolution is vulnerable to semantic evasion techniques. In other words, we focus on a technique for realizing an API name resolution that is robust enough against all types of semantic evasion technique mentioned in Chapter 2. We exclude two types of resolution evasion techniques from our scope in this chapter: static lining and control-flow obfuscation. We will qualitatively discuss the feasibility of our system against the two types of evasion techniques in Section 5.4. Other attack techniques targeting the other steps in the IAT reconstruction,

such as memory dump acquisition or IAT identification, are beyond the scope of this chapter.

5.2.2 Taint-based API Name Resolution

We introduce a new API name resolution technique for IAT reconstruction, *taint-based API name resolution*, which is generic and resistant to semantic evasion techniques. Our technique resolves the API name of a pointer stored in an IAT by taking advantage of taint tags that are used to taint the API code before starting the analysis and propagated during the analysis. In particular, we first identify the positions of each API code in a disk and then taint their codes with the unique taint tags. Next, we begin to run the malware. When the malware under analysis operates the codes of the APIs on which taint tags have been set, we track the movement of the code by propagating the taint tags. After running the malware for a certain amount of time, we generate a memory dump as well as dumps of taint tags and then analyze them for IAT reconstruction. In the API name resolution phase in IAT reconstruction, we resolve the API names from taint tags that have a unique value to distinguish one API from the others.

A taint tag is a piece of data structure related to target data and is used in taint analysis for tracking the flow of the data in the host. Taint analysis itself is not new and has been used in much security research, such as for detecting zero-day attacks [73] or identifying sensitive information leaking [92]. A new aspect in this chapter is that we use taint analysis for assisting static analysis. In other words, we use taint analysis for bridging the semantic gap between dynamic and static analysis. In particular, we relate a taint-tag with the symbol of code, manage the taint-tag in the virtual machine monitor (VMM) layer independently from an OS during dynamic analysis, and access the symbol of the code from the tag for static analysis. IAT reconstruction is an application of this approach. We believe that this approach is reasonable for malware analysis because the symbols that a tag possesses are not affected or modified by malware, unlike those of OS-managed data structures. That means that even when a malware intrudes the OS layer and successfully gains the root privilege, the malware cannot access and modify the taint tags because they

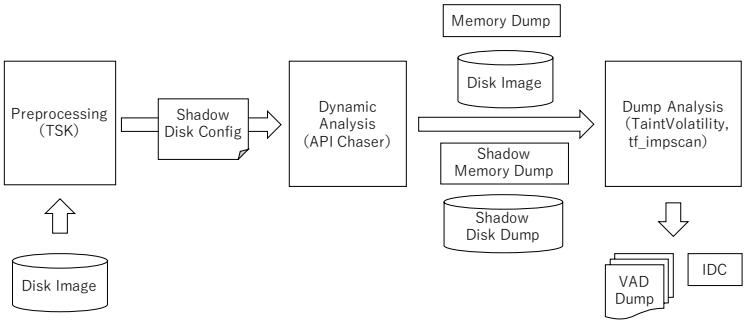

Fig. 5.1 Workflow of Our System

are managed in the VMM layer and the VMM is isolated from the malware running environment.

5.2.3 Taint-assisted IAT Reconstruction System

We have developed a system for IAT reconstruction whose API name resolution is realized with our taint-based technique. Figure 5.1 shows the overview of the system and its workflow. The workflow is mainly composed of three phases: preprocessing, dynamic analysis, and dump analysis. We explain the details of each phase and their implementations below.

Preprocessing

In the preprocessing phase, we first define taint tags, each of which has a unique value for each API and taint the instructions of each target API in a disk with the tag. For an input for this phase, we receive a disk image file on which a guest OS has been installed. For an output, we generate a configuration file for setting up a shadow disk, which is a data structure for storing taint tags related to data on a disk. To do that, we first parse the received disk image by using a forensics tool, analyze the file system installed on it, and then identify the positions of each target API in a disk. After that, we set a unique taint tag on each API. Specifically, we store the taint tag to the corresponding entry in a shadow disk.

This design (i.e., tainting data on a disk, not memory, before starting an analysis)

has two advantages. First, we do not need to care about when the target code in a DLL is loaded onto a physical memory. It is not easy to correctly capture this timing because of on-demand loading. The timing depends on the behaviors of a running process on the OS and is difficult to predict correctly. In contrast, our technique allows us to focus on only taint propagation after setting taint tags. This design could make the implementation simpler. Second, we can comprehensively set taint tags on targets. It is also not easy to correctly identify all locations of specific data originally contained in a file after an OS has been booted because an OS may copy or cache the data in a temporal buffer or its own data structures on the memory. However, before booting an OS, we can easily identify the location of a target file on a disk, and we can say that the data has not been copied to other locations yet by an OS.

■Implementation We use The Sleuth Kit (TSK) to identify the positions of target DLLs in a disk image in which a guest OS has been installed. TSK is a disk forensics tool for parsing a disk image and analyzing various file systems. Using TSK, we acquire the sector number, offset, and size of a target DLL and then extract the DLL file. After that, we acquire the Relative Virtual Address (RVA) of each target API from the PE header of the extracted DLL and then calculate the position of each target API in a disk by adding the RVA to the base address of DLL. Finally, we set the taint tags on each API code. This process has to be completed before starting the dynamic analysis.

We define 22 Windows system DLLs and 7,222 APIs exported from the 22 DLLs as our target for our system. These 22 DLLs include kernel32.dll, ntdll.dll, advapi32.dll and the ones that export APIs often used in a malware. The target DLLs are selected by referencing IDA scope [61], which is an open-sourced IDA script for accelerating static analysis. Using these 22 DLLs, we have covered most APIs defined in IDA scope as remarkable ones, which means we should pay attention to them in static analysis. Thus, we consider that the current numbers of target DLLs and APIs are enough for our purpose. If we need to add support DLLs in the future, we can easily increase the number for a small cost, i.e., simply add one line to a source code.

Dynamic Analysis

In the dynamic analysis phase, we boot an analysis environment as a guest OS from a disk image, run the malware on the booted environment, and perform taint analysis. For an input, we receive the configuration file for setting up a shadow disk. For an output, we generate a set of dumps after executing the malware for a certain amount of time. These dumps contain the (virtual) physical memory, the shadow memory, the shadow disk, and the disk image.

■Implementation For a dynamic analysis engine, we use API Chaser [83], which is an API monitoring system with taint analysis capability. API Chaser is built on QEMU [42] (Argos[73]) and performs API monitoring and taint analysis in the VMM layer. API Chaser leverages taint analysis for API monitoring. In particular, there are two applications of taint analysis for precise API monitoring. The first is code tainting, which is a technique to identify the executions of target code on the basis of taint tags set on the code. The second is to capture the events of API call invocations on the basis of taint tags. That is, we set taint tags on each API code before starting analyses and then recognize as an API call the execution transfer from an instruction that has the taint tags expressing a target code to one that has the taint tags indicating API code.

We have extended API Chaser to enable it to generate dump files on various types of events that have happened in a guest OS. These events include API calls, process creation or termination, module load or unload, or system shutdown. API Chaser provides call-back mechanisms that invoke registered handlers whenever specific hardware or software events happen. Therefore, to generate dump files at specific timings, we have simply developed a small piece of code for calling the function for generating dump files and register the code to appropriate call-backs as a handler.

Dump Analysis

In the dump analysis phase, we conduct IAT reconstruction as a preparation for static analysis. For IAT identification, we first manually select a target process and then identify the positions of the IATs in the virtual memory space of the process using

the `IdentifyIAT` function, in which the find-pointer phase explained in Chapter 2 is implemented. For API name resolution, we resolve the API names of each entry in the identified IATs using `TaintResolveName`, which we explained in Subsection 5.2.2. This function allows us to resolve API names without being evaded by position obfuscation techniques, as we have already explained.

■Implementation We have developed *TaintVolatility* and *tf_impscan* for dump analysis. *TaintVolatility* is an extended version of The Volatility Framework (Volatility) with two new features: a capability for reading dumps of taint tags and disk forensics integration. For the capability of taint tag analysis, we have added options to parse a shadow memory dump and then extract necessary information for API name resolution from taint tags. We have implemented this capability as an extension of the `Address Space` module of Volatility. This design comes from the consideration that when we provide this capability as a function of a framework, we can allow any plugin to use this capability. Specifically, we provide the `taint_resolver(vaddr)` interface for plugins, and this function returns the information related to the taint tags set on the virtual address specified with `vaddr`.

In addition, we have extended Volatility to integrate with TSK, i.e., disk forensics capability. This capability is essential for our system because we have often faced cases in which necessary data has not been loaded to physical memory due to on-demand loading when we generate a memory dump. To solve this, when we parse a memory dump with TaintVolatility and find a memory page that is not mapped on memory, we first find the VAD(s) related to the page and then extract the path of the file containing the memory page from the found VAD(s). After that, we use TSK to parse the disk image and then obtain the location of the page in the disk image. Then, we acquire the taint tags related to the page in a disk from a shadow disk dump. We use pytsk[93], which is a python wrapper for TSK to invoke TSK functions from TaintVolatility.

tf_impscan is a plugin designed to run on TaintVolatility and perform IAT reconstruction by using the `taint_resolver` interface. tf_impscan is built on `impscan`, which is a plugin for IAT reconstruction. An extension of tf_impscan over impscan is API name resolution. tf_impscan resolves the API name using the `taint_resolver`

of TaintVolatility, whereas impscan does this by reading the metadata of loaded DLLs from the PEB. For the other steps of the IAT reconstruction procedure, i.e., IAT identification and PE header restoration, we reuse the code of impscan with small adjustments.

5.3 Experiments

In this section, we describe the experiments we conducted to evaluate the effectiveness of our system. Specifically, we conducted three types of experiments. The first is for showing the resistant capability against semantic evasion techniques. The second is for showing the effectiveness of the integration of TaintVolatility with a disk forensics tool. The third is for measuring performance degradation of TaintVolatility, compared to vanilla impscan, and then showing the degradation is within acceptable range.

5.3.1 Semantic Evasion Resistance

The goal of this experiment is to determine whether our system is effective enough to resolve APIs obfuscated with the known semantic evasion techniques better than current common IAT reconstruction tools.

Procedure

We prepared several Windows executables whose imported APIs were already known. Then, we obfuscated the APIs imported by these executables using semantic evasion techniques which we explained in Chapter 2 with Stealth Loader. We used these obfuscated executables as a dataset for evaluating our system. For simplicity, we did not apply any obfuscation to the code of these executables. In this experiment, we focus on only imported APIs exported from our target DLLs. Moreover, we set out of scope the cases in which an ordinal number, instead of an address, is stored in an entry in IATs.

For comparison, we prepared 3 other IAT reconstruction tools, such as impscan, impscan++, and Scylla. impscan and impscan++ are plugins for Volatility. imp-

Table 5.1 Results of Resistant Capability Experiment

-	Stolen Code	Sliding Call	Copied API	DLL Unlinking	Name Confusion	Stealth Loader
tf_impscan	✓	✓	✓	✓	✓	✓
impscan	-	-	-	-	✓	-
impscan++	-	-	-	✓	✓	-
Scylla	✓*	-	-	-	✓	-

✓means that the tool successfully identified all APIs without being affected by position obfuscation techniques. On the other hand, - means that it failed because it was affected by them. ✓* means when we gave the address of the original entry point, it could successfully identify imported APIs.

scan++ is our developed plugin, which resolves API names using VADs, whereas the original impscan does this using PEB. Scylla is an open-sourced IAT reconstruction tool popular among malware analysts.

Results

Table 5.1 shows the results of this experiment. Our system successfully defeated all semantic evasion techniques, whereas the others were evaded with some of them. This is because these tools are designed to resolve API names by comparing the addresses in an IAT with ones calculated from the base address of a loaded DLL and RVA acquired from the EAT of the loaded DLL. To resolve the API name, these addresses need to exactly match. However, due to Stolen Code, Sliding Call, and Copied API Obfuscation, the addresses filled in the IATs point to buffers prepared for the stolen, sliding or copied code, not the positions where APIs were originally placed by a program loader. Thus, they did not match the calculated ones. As a result of this, the comparison tools failed to resolve API names. The reason impscan++ can defeat DLL Unlinking is that it acquires the base addresses of loaded DLLs from VADs, whereas DLL Unlinking hides them from the PEB. All tools are not affected by Name Confusion because they do not filter any DLLs by their name. When name confusion changes the name of a DLL, e.g., lernel32.dll from kernel32.dll, all tools simply identify the IAT of the DLL, e.g., the IAT of lernel32.dll.

5.3.2 Disk Forensics Integration

The goal of this experiment is to measure how much the integration of disk forensics capability with TaintVolatility contributes to a better result of IAT reconstruction.

Procedure

We first prepared several Windows executables and executed them on our system. When we analyzed their memory dumps, we resolved API names using TaintVolatility without disk forensics capability. Then, we enabled the capability and resolved API names again. Lastly, we compared the results of each resolution, i.e., resolutions with or without disk forensics capability.

Results

Table 5.2 shows the results of this experiment. These results shows that disk forensics capability is essential for our system. Without it, our system failed to resolve some APIs. Whether an API stays on a memory or not is totally dependent on the behaviors of each running process. If an API has been already called by a process during dynamic analysis, the API is loaded on a memory and is likely to be still on the memory when we generate a memory dump. However, if it is not, it may not be loaded on a memory when we make a memory dump.

As we have explained, our technique can handle both cases properly. When the code of API stays on memory, we simply extract the information of the API from shadow memory. Additionally, when the code is on a disk, we do this from a shadow disk using disk forensics capability.

5.3.3 Performance Measurement

The goal of this experiment is to measure performance degradation of TaintVolatility, compared to vanilla impscan and show that the degradation does not impose significant impact on the effectiveness of tf_impscan.

Table 5.2 Results of Disk Forensics Integration Experiment

-	calc	notepad	taskmgr	services	iexplore	lsass	cmd
# of Imported APIs	380	242	363	300	143	91	233
# of Target APIs	215	173	275	168	95	52	161
# of Resolved APIs w/o DF	193	144	249	164	91	50	107
# of Resolved APIs w/ DF	215(22)	173(29)	275(26)	168(4)	95(4)	52(2)	161(54)

# of Imported APIs is the number of APIs imported by each executable. # of Target APIs is the number of APIs exported from our target DLLs, which are defined in Subsection 5.2.3. # of Resolved APIs w/o DF and # of Resolved APIs w/ DF are APIs that our system could resolve without and with disk forensics capability. The numbers in parentheses are the APIs resolved from the shadow disk. DF means **D**isk **F**orensics capability.

Procedure

We used the same memory dumps as the second experiment (Subsection 5.3.2) to do this experiment. When we analyzed the memory dumps with the three tools, impscan, *tf_impscan* without disk forensics capability, and *tf_impscan* with disk forensics capability, we measured the elapsed seconds to complete their task, i.e., IAT reconstruction, with the `time` command. We used the sum of `user` and `system` times as an indicator for comparison.

We conducted this experiment on a virtual machine on which Ubuntu Linux 14.04 was installed. We assigned 2 CPU cores and 2 GB memory for the virtual machine. This virtual machine ran on MacBook Pro, which had 3.1 GHz Intel Core i7, 16 GB memory, and 1TB flush storage.

Results

Table 5.3 shows the results of this experiment. The degradation rates of *tf_impscan* w/o DF were from x1.2 to x1.4, while the ones of *tf_impscan* w/ DF were from x1.4 to x2.0. These overheads mainly come from taint_resolver. Especially, when it needs to access a shadow disk, i.e., when a memory page containing the target virtual address is not loaded onto a physical memory yet, it takes more overhead because it has to take more steps for the resolution, such as identifying the mapped

Table 5.3 Results of Performance Experiment

-	calc	notepad	taskmgr	services	iexplore	lsass	cmd
impscan	11.6	8.5	10.4	8.5	13.8	8.3	7.5
tf_impscan w/o DF	14.4	11.8	13.3	11.2	16.3	11.0	10.1
tf_impscan w/ DF	19.9	15.9	20.0	15.4	20.3	14.3	13.7

The unit of the numbers is seconds. We measured the performance with `time` command. These numbers are the sum of `user` and `system` times of `time` command.

file and the position where the file is stored on a disk, and calculating the offset of the corresponding address in the disk. Nevertheless, since even the maximum degradation in this experiment was less than x2.0, we consider that these performance degradations are not a serious limitation of TaintVolatility and may be accepted in practical fields.

5.4 Discussion

In this section, we discuss the limitations of our technique, the validity of our experiments and platform dependency.

5.4.1 Limitation

We explain some limitations of our technique and then show our considerations for each of them.

Control-flow Obfuscation

Our technique fails when code snippets are inserted in the control flow between each entry in an IAT and the corresponding API code. This technique is called control-flow obfuscation and used in API redirection [2]. In this situation, an entry in an IAT points to the inserted code, not any API code, and the code does not have any taint tags. Thus, we cannot resolve the API name simply by looking at the taint tag of the instruction directly referenced from the IAT entry. To overcome this, we are considering extending our technique by applying CFG analysis, as used in

Eureka [55]. When an entry in an IAT points to the code that has no taint tags, we begin analyzing the control flow starting at the entry and proceeding until it reaches an instruction that has taint tags related to any API.

DLL Static Linking

When a system DLL is statically linked to a malware executable, we cannot identify the APIs exported from the DLL [95]. This is because the codes of the APIs exported from the DLL do not have any taint tags, even though we need taint tags to resolve the API names. However, we consider that it is not easy to link a system DLL to a malware executable in practice because there are several technical challenges. One example is that doing so may cause a dependency problem between incompatible DLL versions. Another is that the statically linked system DLL loses its portability because the file is enlarged. We consider that these difficulties probably reduce the attractiveness of static linking for malware in practical fields.

Incomplete Dynamic Analysis

If the execution of a malware does not reach the code for API name resolution during the dynamic analysis, our system cannot resolve the API names of IAT entries. This is because the addresses in IAT are not filled in when we generate a memory dump. As you know, there are several anti-analysis techniques to detect the existence of VMM or analysis environments [18][88], i.e., conditionally-protected executions. Thus, if our analysis environment is detected with some of these techniques, the execution is possibly stopped in the middle. We consider that this is a different problem from the target of our paper, so we set it as beyond the scope because there have been several studies for tackling this problem [30][94].

Implicit Flow

The implicit flow problem is a general limitation of taint analysis. Taint propagation fails when tainted data is processed with implicit flow [96]. If malware processes API code with an implicit flow without changing its value before performing position obfuscation techniques, the taint tags set on the API code are cleared, i.e., malware can wash the taint tags set on the data. As a result, we lose the relationship between

API code and the API name that we make in the preprocessing phase. To overcome this, we will adapt the results of existing research [90][91] to our system.

5.4.2 Validity of Experiments

We consider that the experiments we had are reasonable for achieving our goal in this chapter, even though we did not have any experiment using real-world malware. Our goal is to propose a technique and develop a system resistant to semantic evasion. To measure its resistance capability against semantic evasion, a malware in the wild is not appropriate because we do not know its correct answers and a malware is basically a complex of many anti-analyses and functions, i.e., it is difficult to identify the causes when we fail to get expected results from real-world malware. On the other hand, we used Windows executables in our experiments because we can download the symbols for these executables and get the correct answers, i.e., which APIs are imported by an executable. Considering this fact, answer-known Windows executables are more appropriate and reasonable than real-world malware in terms of evaluating our system.

5.4.3 Platform Dependency

We have developed our system with targeting for 32bit-Windows 7 platform as an analysis environment (guest OS). However, our technique is not limited to a specific environment. That is, taint-based name resolution is independent of platforms and architectures. We believe that we could also apply our techiniqeu to an Executable and Linkable Format (ELF)-format executable. In ELF-format executable, it has the Global Offset Table (GOT) to store the addresses of each external function. Theoretically speaking, we can use the taint-based name resolution technique to resolve the names of the addresses in GOT.

5.5 Conclusion

In this chapter, we proposed a new Application Programming Interface (API) name resolution based on taint analysis to solve the target-gap problem in static analysis. We also described system components for IAT reconstruction whose API name resolution is realized with our technique. Additionally, we demonstrated that this system is generically effective for various types of semantic evasion techniques through experiments.

Chapter 6

Conclusion

Malware analysis is time-consuming, error-prone, and requires in-depth domain knowledge. Considering the amount of work required to analyze malware and the number of newly emerging malware, it is natural to want to automate the malware analysis process as much as possible. Security industry responses to this demand have produced many automated malware analysis tools such as sandboxes or disassemblers. However, most of those automated tools are not capable of keeping pace with malware evolution. Notably, the speed of emerging new evasion techniques is faster than the security industry expectations. As a result, existing automated analysis tools lack resistance capabilities against evasion techniques. That is, malware can evade these tools.

A reason why malware can evade these analysis tools is that there is a gap between what we really want to analyze and what we actually analyze. We call this gap the *target-gap* problem. The target-gap problem is a design problem that commonly exists in many analysis tools. One example of this problem is that we identify our targets for analyses based on a PID of a malware process in cases of dynamic analysis, even though what we really want to analyze is the malware code. Another example is when we capture the executions of a specific API call, what we really capture is the executions of the virtual memory address where the API should be located. Similar to these examples, most existing analysis tools are designed to focus on objects slightly different from what we really want to analyze. We presume the

executions of our targets based on their indicators. This gap gives malware a chance to evade those analysis techniques.

To make matters worse, users of these tools do not recognize the fact that they fail to capture the essential behavior of malware under analysis. This is because the users tend not to pay sufficient attention to false negatives. For example, when an anti-virus product detects a legitimate program as malware by mistake, this false positive gathers public attention [97][98]. On the other hand, when an anti-virus fails to identify a single malware by mistake, this false negative is not given attention because people rarely notice that it generates a false negative. Evasive malware causes analysis tools to generate false negatives and, as mentioned earlier, users of the tools are not likely to find these false negatives. Thus, evasion problems have been left unsolved for a long time.

This thesis sheds light on the problem and proposes two new analysis techniques based on taint tracking to solve the problem. We re-designed malware analysis systems such as a sandbox and IAT reconstruction tool by considering the resistance capability against evasion techniques as an inclusive-by-design property. Then, we developed our tools with the new design. These tools are not only patching a problem that exists in current systems but is sufficiently strong against evasion techniques because they do not have the root cause of the problem. Due to this design, our techniques are effective for existing evasion techniques and will be for future-emerging ones as we proved in Subsections 4.5.2 and 4.5.3.

We summarize what we have achieved through this thesis as follows. In Chapter 2, we discussed the target-gap problem, which is a design problem that commonly exists in API-oriented analysis techniques. We first explained both dynamic and static techniques, i.e., API monitoring and IAT reconstruction, respectively. Next, we mentioned evasion techniques for bypassing API-oriented analysis techniques. Then, we described literature pertaining to both dynamic and static malware analysis systems. Last, we described the target-gap problem and showed several situations that malware analysts may face with existing analysis tools. In Chapter 3, we presented Stealth Loader as a proof-of-concept implementation that takes advantage of the target-gap problem. We then experimentally showed that Stealth Loader evaded

all primary analysis tools. We also qualitatively showed that Stealth Loader could evade previously proposed API-oriented analysis techniques in academic studies. In Chapter 4, we introduced an API monitoring capability based on taint analysis to solve the target-gap problem in dynamic analysis using API Chaser. API Chaser was designed and implemented to prevent malware from evading API monitoring. We conducted experiments using actual malicious codes with various evasion techniques to show that API Chaser correctly works according to its intended design of being difficult to evade. In Chapter 5, we proposed an API name resolution capability based on taint analysis to solve the target-gap problem in static analysis. We also described system components for IAT reconstruction whose API name resolution is achieved using our approach. Additionally, we showed based on experimental results that this system is generically effective for various resolution evasion techniques.

Through the research for this thesis, we have solved many technical problems. However, there are still two more unsolved subjects to establish a comprehensive countermeasure against evasion techniques: implicit flow and performance. Implicit flow is a challenging problem commonly implemented in all systems with a taint analysis capability. Unfortunately, this problem has not been solved yet even though there are several studies that investigated this problem [90][91][99]. API Chaser is also affected by implicit flow similar to the way other taint-analysis-based systems are affected. This means that malware authors are intentionally able to disturb taint propagation if they embed a taint cleaning function that contains an implicit flow logic in their malware, and the malware calls the function between every two data-dependent functions of the malware. To make our systems more effective and robust against real-world malware, we plan to investigate this problem and hope to develop a solution in the near future.

The other remaining subject is performance degradation. Taint analysis increases the performance penalty as a side-effect of executing a single machine instruction, i.e., when an instruction is being executed, a taint-analysis-based system must check the taint tags of the instruction and its operands, and then propagate one of them to the destination operand by following pre-defined propagation rules. The impact of these side-effects accumulates to a tremendous level considering the number of

executed instructions while running malware on a VM for analysis. To mitigate this performance degradation, several approaches [100][101] have been proposed so far, including separation of the taint propagation module from the original executions or pushing a part of the taint analysis to a hardware unit. We have already achieved reasonable performance with API Chaser, as we evaluated in Subsection 4.5.5. Here, reasonable means that API Chaser could execute malware without being detected using a timing attack, as we discussed in Subsection 4.6.3. However, to prepare for cases in which malware narrows the time slot for detection, we must improve the performance of API Chaser, for example, by applying existing research results to our systems.

In conclusion, this thesis brings evasion techniques involving target code executions to an end. In other words, as far as malware executes its codes, we can capture the executions and monitor its behaviors without being evaded in dynamic analysis. Most existing evasion techniques involve code executions. For example, target-evasive malware changes the instance, i.e., process, to execute its malicious codes, but the code itself is executed. Another example is that hook-evasive malware moves a part of the code of a specific API to a different address from its original position, but the code of the API is executed. Similar to these examples, as far as malware performs malicious actions involving its code executions, we can capture the malware behaviors even if malware uses evasion techniques.

As future directions for this research, we describe two possible future topics: the generalization of taint-based approaches and evasion techniques without code executions. The generalization of taint-based approaches means that we would explore new applications for taint analysis in security and non-security fields. Taint analysis was previously used only for data-flow analysis, but we showed that it could be used to capture the execution of target codes with code tainting. If we take a different viewpoint to this new application of taint analysis, this approach can add meaning to inorganic bytes in data storage such as memory or disks as we regularly do with symbols or debug headers for analyses or forensics. We call the symbols or debug headers *static symbols* for convenience. A static symbol comprises position information such as a virtual address or offset and semantic information such as

the names of functions or variables. We can add semantic information to a byte in memory or a disk if the address of the byte matches the position of a static symbol. However, since malware intentionally removes symbols and debug headers from its executable to disrupt analyses, we cannot make use of static symbols for malware analysis in many cases. Furthermore, malware can evade the position of specific data or code by copying them from its original location to dynamically allocated memory as we showed in Chapter 2. As a result, static symbols are not sufficiently practical for malware analysis in many cases.

We showed that taint analysis is useful in adding the meaning to inorganic bytes in an analysis environment, even if malware drops static symbols. We relate an API to a taint tag to which a unique value is assigned and then set it to the code of the API. This results in complementing the lack of static symbols for precise analyses since we acquire the semantics of code or data from the taint tag set to it. We call these taint tags *taint symbols*. Taint symbols have two advantages over static symbols for malware analysis. The first advantage is that we can safely manage taint symbols without being compromised or disturbed by malware running in a guest OS since taint analysis is performed in the VM layer, which is not directly affected by malware running in a guest OS. The second is that we can track the movement of the target code and data by propagating the respective taint tags set to the code and data. So, if malware dynamically defines a data structure by collecting variables defined in some known data structure, we can identify each variable in the malware data structure with the taint tags propagated from their source data structures. This property allows us to accelerate the analysis of unknown data structures.

We believe that taint symbols can be applied to other analysis tools, not only for malware analysis purposes but also to cases that require precise analyses such as program testing or debugging in software engineering fields. So, we have two future directions for this research. The first is that we will extend other malware analysis tools except for sandboxes or forensics with taint analysis as we showed in this thesis. Taint symbols are generic and can be applied in many security situations such as vulnerability research or disassemblers. The second is that we will apply taint symbols to debugging or software engineering. Taint symbols may be useful

in analyzing dynamically generated code or data with a just-in-time compiler, even though the code for analysis may not have malicious intent unlike malware.

Another avenue of research for our work that we wish to pursue is to address a new type of evasion technique that does not involve code executions. An example of this type of evasion technique is full ROP malware. When a malware author writes his/her entire malware using the ROP technique, the malware possibly does not have any code inside of it to perform a malicious action. Specifically, this malware has only a set of return addresses for jumping to each gadget and arguments for them. So, it has no code to execute inside. Since our taint-based approaches require executions of target codes for monitoring, they are not effective with respect to ROP malware.

To cover ROP malware, we must extend our target identification logic to be finer grained. ROP malware does not involve malicious code executions, but its behaviors have characteristics. That is, there are many features when using the ROP technique. These features allow us to identify the behaviors triggered by ROP. For example, we first taint the entire ROP malware with a taint tag, and then we monitor a control transfer whose destination address is tainted, but the instruction itself is not tainted. This situation indicates that the control transfer destination is determined by tainted data, which are controlled by malware. So, we should monitor the behaviors that occur after this control transfer is observed.

Another new type of evasion technique that does not involve code executions is side effect malware. We consider that it may be possible to take advantage of the side effects of some behaviors to achieve a different purpose. An example is Meltdown and Spectre [102]. A characteristic feature of these attacks is that they take advantage of side effects of modern processor out-of-order executions. Recent CPUs have many functionalities and have become so complicated that there are few people who understand the entire CPU architecture in detail. Moreover, when we consider software running on hardware such as an OS, drivers, middleware, and applications, the complexity of the entire architecture to perform one action on a computer exceeds the limit that one person can recognize. So, when we execute one action on a computer, a hundred or thousand side effects occur along with performing that action. When malware authors conduct in-depth investigations on

side effects and find a useful combination of them to achieve their goal, they can accomplish the goal with only side effects without executing malicious code. Again, since our taint-based approaches require executions of target codes for monitoring, it is no longer effective against this type of evasion technique.

To cover side effect malware, we must extend the definition of taint propagation. We now propagate taint tags from one data set to another only when there is a direct relationship such as data movement or arithmetic computations between the two data sets. However, to create a relationship between more sensitive data, we propagate taint tags when there is an indirect relationship as well such as in cases of code dependency. This approach may currently lead to over tainting, but it also has the possibility of finding more sensitive relationships between data, which may contain semantic relationships or side effects. We will investigate an approach to achieve them without losing practicality as a malware analysis system for solving real-world problems.

Acknowledgement

First of all, I would like to express my sincere gratitude to my supervisor, Professor Tatsuya Mori, for providing me with the opportunity to challenge a doctoral degree. He has been very supportive and encouraging. He also provided me with his insightful knowledge and guided me to complete this thesis. In addition, I would like to thank my sub-adviser, Professor Tachio Terauchi. His advice made me think about the true contributions of this thesis through fundamental questions, such as what cybersecurity research should be and what is the ultimate goal of security research. I would also like to thank another sub-adviser, Professor Hironori Washizaki. His advice helped to have me develop a better understanding of this thesis and clear its motivations and limitations. His valuable feedback led to improving the quality of this thesis. I would also like to thank the other sub-adviser, Associate Professor Yoshihiro Oyama, for his meaningful comments based on his expertise from the perspective of malware analysis. His practical advice also helped me improve the quality of this thesis, especially the technical aspects of this thesis.

I would like to extend my gratitude to Professor Yoichi Muraoka. MURAOKA laboratory at Waseda University was the place where I found my interest in research during my undergraduate and master's course. I also extend my gratitude to Professor Shigeki Goto. My challenge to a doctoral degree was started when I visited his laboratory in December 2011. He gave me a lot of valuable and practical advises and then encouraged me to this challenge.

I would like to thank Dr. Makoto Iwamura. He motivated me to engage the field of cybersecurity. He is one of the researchers whom I greatly respect in computer security fields. I am honored to work with him for fourteen years since I joined NTT. I would like to thank the current and former members of NTT Secure Platform

Laboratories, in particular, Takeo Hariu, Takeshi Yada, and Jun Miyoshi, for their continuous support. Without their support, I could never complete this thesis. Many thanks to my colleague of Cyber Security Project of NTT Secure Platform Laboratories, in particular, Dr. Yuto Otsuki, Yuma Kurogome, and Toshinori Usui. All research seeds in this thesis were cultivated through discussions with them. The time spent with them has empowered me to complete this thesis. I also would like to thank Dr. Eiji Kuwana and Dr. Kenji Takahashi for supports when I was at NTT Innovation Institute, Inc. They gave me an opportunity to work in Silicon Valley, where the most exciting place for people who engaged in computer science. That experience opens my eyes to the world and pushes me around to take a new challenge.

Most of all, I would like to thank my friends and family for their warm support. They have been the cornerstone of all my accomplishments. I thank my parents for teaching me how to read, write and calculate when I was a child. It was all started from there. Most notably, my greatest gratitude goes to my wife Yoshie, my son Minato, and my daughter Hazuki, who have supported me through all of life with their smiles.

Bibliography

[1] M. Suenaga, "A Museum of API Obfuscation on Win32." https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/security-response-museum-API-win32-09-en.pdf, 2009.

[2] M.V. Yason, "The Art of Unpacking," Black Hat USA Briefings, August 2007.

[3] J. Leitch, "Process Hollowing." https://www.autosectools.com/Process-Hollowing.html.

[4] m0n0ph1, "Process-Hollowing." https://github.com/m0n0ph1/Process-Hollowing.

[5] T. Liberman, "ATOMBOMBING: BRAND NEW CODE INJECTION FOR WINDOWS." https://blog.ensilo.com/atombombing-brand-new-code-injection-for-windows.

[6] BreakingMalwareResearch, "atom-bombing." https://github.com/BreakingMalwareResearch/atom-bombing.

[7] BreakingMalware.com, "PowerLoaderEx." https://github.com/BreakingMalware/PowerLoaderEx.

[8] S. Pierce, "Defending Against Malicious Application Compatibility Shims," Black Hat Europe Briefings, November 2015.

[9] Y. Kawakoya, E. Shioji, Y. Otsuki, M. Iwamura, and T. Yada, "Stealth Loader: Trace-free Program Loading for API Obfuscation," Proceedings of 20th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID '17, pp.217–237, September 2017.

[10] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," Proceedings of the 10th Annual Net-

work and Distributed System Security Symposium, NDSS '03, pp.191–206, February 2003.

[11] T.K. Lengyel, S. Maresca, B.D. Payne, G.D. Webster, S. Vogl, and A. Kiayias, "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System," Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC '14, New York, NY, USA, pp.386–395, ACM, December 2014.

[12] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A View on Current Malware Behaviors," Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More, LEET '09, pp.8–8, April 2009.

[13] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction," ACM Trans. Inf. Syst. Secur., vol.13, no.2, pp.12:1–12:28, March 2010.

[14] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A Tool for Analyzing Malware," Proceedings of the European Institute for Computer Antivirus Research Annual Conference, EICAR '06, April 2006.

[15] M. "RIVAL", "Dynamic-Link Library Hijacking." https://www.exploit-db.com/docs/english/31687-dynamic-link-library-hijacking.pdf.

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

[17] NtQuery, "Scylla." https://github.com/NtQuery/Scylla.

[18] P. Ferrie, "Attacks on Virtual Machine Emulators." https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/attacks-on-virtual-machine-emulators-07-en.pdf, 2006.

[19] J. Rutkowska, "Subverting Vista Kernel for Fun and Profit," Black Hat USA Briefings, August 2006.

[20] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D.X. Song, and H. Yin,

"Automatically Identifying Trigger-based Behavior in Malware," (eds) Botnet Detection, vol.36, pp.65–88, 2008.

[21] A. Vasudevan and R. Yerraballi, "Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions," Proceedings of 2006 IEEE Symposium on Security and Privacy, Oakland '06, pp.264–279, May 2006.

[22] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M.G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," Proceedings of the 4th International Conference on Information Systems Security, ICISS '08, pp.1–25, December 2008.

[23] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis," Proceedings of the 14th ACM conference on Computer and communications security, CCS '07, pp.116–127, October 2007.

[24] D. Oktavianto and I. Muhardianto, Cuckoo Malware Analysis, Packt Publishing, 2013.

[25] M. Polychronakis, K.G. Anagnostakis, and E.P. Markatos, "Network–Level Polymorphic Shellcode Detection Using Emulation," Proceedings of the 3rd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA '06, Berlin, Heidelberg, July 2006.

[26] M. Polychronakis, K.G. Anagnostakis, and E.P. Markatos, "Comprehensive shellcode detection using runtime heuristics," Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10, New York, NY, USA, pp.287–296, ACM, December 2010.

[27] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," IEEE Security and Privacy, vol.5, no.2, pp.32–39, March 2007.

[28] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis?," ACM Comput. Surv., vol.49, no.1, pp.4:1–4:37, April 2016.

[29] Y. Kawakoya, M. Iwamura, and M. Itoh, "Memory Behavior-Based Automatic Malware Unpacking in Stealth Debugging Environment," Proceedings of 5th IEEE International Conference on Malicious and Unwanted Software, MALWARE '10, October 2010.

[30] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," Proceedings of the 15th ACM conference on Computer and communications security, CCS '08, pp.51–62, Oct. 2008.

[31] M.G. Kang, P. Poosankam, and H. Yin, "Renovo: A Hidden Code Extractor for Packed Executables," Proceedings of the 2007 ACM Workshop on Recurring Malcode, WORM '07, pp.46–53, November 2007.

[32] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," Proceedings of 22st Annual Computer Security Applications Conference, ACSAC '06, pp.289–300, December 2006.

[33] windowsvistasecurity, "An Introduction to Kernel Patch Protection." https://blogs.msdn.microsoft.com/windowsvistasecurity/2006/08/12/an-introduction-to-kernel-patch-protection/.

[34] Stefan Buhlmann, "Joebox Sandbox." http://www.joesecurity.org/.

[35] Rohitab Batra, "API Monitor." http://www.rohitab.com/apimonitor.

[36] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and The Art of Virtualization," Proceedings of the nineteenth ACM symposium on Operating systems principles, SOSP '03, New York, NY, USA, pp.164–177, ACM, October 2003.

[37] Z. Deng, X. Zhang, and D. Xu, "SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization," Proceedings of the 29th Annual Computer Security Applications Conference, ACSAC '13, New York, NY, USA, pp.289–298, ACM, December 2013.

[38] M. Cohen, "Rekall." http://www.rekall-forensic.com/.

[39] A. Vasudevan and R. Yerraballi, "Stealth Breakpoints," Proceedings of 21st Annual Computer Security Applications Conference, ACSAC '05, pp.381–392, December 2005.

[40] A. Vasudevan and R. Yerraballi, “SPiKE: Engineering Malware Analysis Tools Using Unobtrusive Binary-instrumentation,” Proceedings of the 29th Australasian Computer Science Conference - Volume 48, ACSC ’06, Darlinghurst, Australia, Australia, pp.311–320, Australian Computer Society, Inc., January 2006.

[41] S. Tanda, “"egg" - A Stealth fine grained code analyzer,” RECON Briefings, June 2011.

[42] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” Proceedings of the annual conference on USENIX Annual Technical Conference, FREENIX Track, ATEC ’05, pp.41–46, April 2005.

[43] Z. Deng, D. Xu, X. Zhang, and X. Jiang, “IntroLib: Efficient and transparent library call introspection for malware forensics,” Digital Investigation, vol.9, pp.S13 – S23, 2012.

[44] T.H. Carsten Willems, Ralf Hund, “CXPInspector: Hypervisor-Based, Hardware-Assisted System Monitoring,” Tech. Rep. Technical Report TR-HGI-2012-002, Ruhr University Bochum, November 2012.

[45] Norman, “Norman Sandbox Analyzer.” http://download01.norman.no/product_sheets/eng/SandBox_analyzer.pdf.

[46] Joxean Koret, “Zero Wine: Malware Behavior Analysis.” http://zerowine.sourceforge.net/.

[47] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J.Y. Marion, “Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost,” Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS ’18, New York, NY, USA, pp.395–411, ACM, October 2018.

[48] J. Raber and B. Krumheuer, “QuietRIATT: Rebuilding the Import Address Table Using Hooked DLL Calls,” Black Hat DC Briefings, 2009.

[49] Y. Otsuki, E. Takimoto, T. Kashiyama, S. Saito, E. Cooper, and K. Mouri, “Tracing malicious injected threads using alkanet Malware analyzer,” IAENG Transactions on Engineering Technologies - Special Issue of the World

Congress on Engineering and Computer Science 2012, Lecture Notes in Electrical Engineering, Germany, pp.283–299, Springer Verlag, January 2014.

[50] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "BitVisor: A Thin Hypervisor for Enforcing I/O Device Security," Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '09, New York, NY, USA, pp.121–130, ACM, March 2009.

[51] D. Korczynski and H. Yin, "Capturing Malware Propagations with Code Injections and Code-Reuse Attacks," Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS, CCS '17, pp.1691–1708, October 2017.

[52] A. Henderson, A. Prakash, L.K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform," Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA '14, pp.248–258, July 2014.

[53] D. Kirat, G. Vigna, and C. Kruegel, "BareBox: Efficient Malware Analysis on Bare-metal," Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11, New York, NY, USA, pp.403–412, ACM, December 2011.

[54] The Volatility Framework, "The Volatility Foundation." `https://www.volatilityfoundation.org/`.

[55] M.I. Sharif, V. Yegneswaran, H. Saidi, P.A. Porras, and W. Lee, "Eureka: A Framework for Enabling Static Malware Analysis," Proceedings of European Symposium on Research in Computer Security, ESORICS '08, pp.481–500, October 2008.

[56] D. Korczynski, "RePEconstruct: reconstructing binaries with self-modifying code and import address table destruction," Proceedings of 11th International Conference on Malicious and Unwanted Software, MALWARE' 16, pp.31–38, October 2016.

[57] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent Dynamic Instrumentation," Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, VEE '12, New York, NY, USA, pp.133–144, ACM, March 2012.

[58] D. Quist, L. Liebrock, and J. Neil, "Improving antivirus accuracy with hypervisor assisted analysis," Journal in Computer Virology, vol.7, no.2, pp.121–131, May 2011.

[59] Heaven Tools. http://www.heaventools.com/remove-debug-information.htm.

[60] S. Fewer, "Reflective DLL Injection." http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf.

[61] D. Plohmann and A. Hanel, "simpliFiRE.IDAScope," Hacklu Briefings, 2012.

[62] Hex-Rays, "Hex-Rays Home." https://www.hex-rays.com/.

[63] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," Proceedings of the 3rd conference on USENIX Windows NT Symposium, WINSYM '99, pp.14–14, USENIX, July 1999.

[64] Sycurelab, "DECAF." https://github.com/sycurelab/DECAF.

[65] VirusTotal, "VirusTotal." https://www.virustotal.com/.

[66] M. Russinovich, "VMMap." https://docs.microsoft.com/en-us/sysinternals/downloads/vmmap.

[67] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pp.190–200, June 2005.

[68] Zynamics, "BinDiff." https://www.zynamics.com/bindiff.html.

[69] C. Kruegel, E. Kirda, and A. Moser, "Limits of static analysis for malware detection," Proceedings of the 23rd Annual Computer Security Applications Conference, ACSAC '07, pp.421–430, December 2007.

[70] M. Payer, T. Hartmann, and T.R. Gross, "Safe Loading - A Foundation for Secure Execution of Untrusted Programs," Proceedings of the 2012 IEEE

Symposium on Security and Privacy, Oakland '12, pp.18–32, May 2012.

[71] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS '05, pp.340–353, November 2005.

[72] Microsoft, "Control Flow Guard." https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.

[73] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation," Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, EuroSys '06, pp.15–27, April 2006.

[74] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: A Tool for Massive Malware Labeling," Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID '16, pp.230–253, September 2016.

[75] C. Kolbitsch, E. Kirda, and C. Kruegel, "The Power of Procrastination: Detection and Mitigation of Execution-stalling Malicious Code," Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pp.285–296, October 2011.

[76] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," Proceedings of the 12th Annual Network and Distributed System Security Symposium, NDSS '05, February 2005.

[77] B. Carrier, "The Sleuth Kit(TSK)." http://www.sleuthkit.org/.

[78] M. Iwamura, M. Itoh, and Y. Muraoka, "Towards Efficient Analysis for Malware in the Wild," Proceedings of IEEE International Conference on Communications, ICC '11, pp.1–6, June 2011.

[79] T. Nowak and A. Sawicki, "The Undocumented Functions." http://undocumented.ntinternals.net/.

[80] React OS Project, "ReactOS." http://www.reactos.org/.

[81] M. Coppola, "loadvm snapshot as read-only." https://bugs.launchpad.net/qemu/+bug/1184089.

[82] M. McLoughlin, "The QCOW2 Image Format." https://people.gnome.org/~markmc/qcow-image-format.html.

[83] Y. Kawakoya, M. Iwamura, E. Shioji, and T. Hariu, "API Chaser: Anti-analysis Resistant Malware Analyzer," Proceedings of 16th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID '13, pp.123–143, October 2013.

[84] S. Chevet, "dllinjshim.cpp." https://gist.github.com/w4kfu/95a87764db7029e03f09d78f7273c4f4.

[85] A. Ermolinskiy, S. Katti, S. Shenker, L.L. Fowler, and M. McCauley, "Towards Practical Taint Tracking," Tech. Rep. UCB/EECS-2010-92, EECS Department, University of California, Berkeley, June 2010.

[86] L. Martignoni, R. Paleari, G.F. Roglia, and D. Bruschi, "Testing CPU Emulators," Proceedings of the 18th International Symposium on Software Testing and Analysis, ISSTA '09, pp.261–272, July 2009.

[87] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, "Testing System Virtual Machines," Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pp.171–182, July 2010.

[88] T. Raffetseder, C. Krügel, and E. Kirda, "Detecting System Emulators," Proceedings of the 10th International Conference on Information Security, ICS '07, pp.1–18, October 2007.

[89] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting Return-Oriented Programming Malicious Code," Proceedings of the 5th International Conference on Information Systems Security, ICISS '09, pp.163–177, December 2009.

[90] M.G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation," Proceedings of the 18th Annual Network and Distributed System Security Symposium, NDSS '11, February 2011.

[91] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," Proceedings of the 4th ACM European conference on Computer systems, EuroSys '09, pp.61–74, April 2009.

[92] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," ACM Trans. Comput. Syst., vol.32, no.2, pp.5:1–5:29, June 2014.

[93] py4n6, "pytsk." "https://github.com/py4n6/pytsk.

[94] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal Analysis-based Evasive Malware Detection," Proceedings of the 23rd USENIX Conference on Security Symposium, SEC '14, pp.287–301, August 2014.

[95] B. Abrath, B. Coppens, S. Volckaert, and B. De Sutter, "Obfuscating Windows DLLs," Proceedings of 2015 IEEE/ACM 1st International Workshop on Software Protection, SPRO '15, pp.24–30, May 2015.

[96] L. Cavallaro, P. Saxena, and R. Sekar, "On the Limits of Information Flow Techniques for Malware Analysis and Containment," Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA '08, pp.143–163, July 2008.

[97] nsoftware, "Symantec/Norton Products detecting WS.Reputation.1." https://www.nsoftware.com/kb/xml/11011701.rst.

[98] NirBlog, "Antivirus companies cause a big headache to small developers.." http://blog.nirsoft.net/2009/05/17/antivirus-companies-cause-a-big-headache-to-small-developers/.

[99] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic spyware analysis," 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference, ATC'07, pp.18:1–18:14, 2007.

[100] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: Pipelined Symbolic Taint Analysis," Proceedings of the 24th USENIX Conference on Security Symposium, SEC'15, pp.65–80, August 2015.

[101] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The World's Fastest Taint Tracker," Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection, RAID'11, September 2011.

[102] Graz University of Technology, "Meltdown and Spectre." https://meltdownattack.com/.

List of Research Achievements

Journal Papers

1. Yuhei Kawakoya, Eitaro Shioji, Makoto Iwamura, and Jun Miyoshi, “API Chaser: Taint-Assisted Sandbox for Evasive Malware Analysis”, Journal of Information Processing (in printing).
2. Yuhei Kawakoya, Makoto Iwamura, and Jun Miyoshi, “Taint-assisted IAT Reconstruction against Position Obfuscation", Journal of Information Processing Vol.26, pp.813-824, December 2018 (Recommended Paper, Specially Selected Paper).
3. Yuhei Kawakoya, Eitaro Shioji, Yuto Otsuki, Makoto Iwamura, and Jun Miyoshi, “Stealth Loader: Trace-free Program Loading for Analysis Evasion", Journal of Information Processing Vol.26, pp. 673-686, September 2018.
4. Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu, “Tracing Malicious Code with Taint Propagation”, Journal of Information Processing Society of Japan (IPSJ) Vol.54 No.8, pp.2079-2089, August 2013 (in Japanese)(Recommended Paper, Specially Selected Paper, IPSJ Yamashita SIG Research Award).
5. Mitsuaki Akiyama, Kazufumi Aoki, Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh, “Design and Implementation of High Interaction Client Honeypot for Drive-by-download Attacks”,IEICE Transactions on Communication, Vol.E93-B No.5 pp.1131—1139, May 2010.
6. Kazufumi Aoki, Yuhei Kawakoya, Mitsuaki Akiyama, Makoto Iwamura, and Mitsutaka Itoh, “Investigation and Understanding Active/Passive Attacks”,

Journal of Information Processing Society of Japan (IPSJ) Vol.50, No.9, pp. 2147—2162, September 2009 (in Japanese).

Conference Papers

1. Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi and Kazuhiko Ohkubo, "Building stack traces from memory dump of Windows x64", Proceedings of the Digital Forensic Research Workshop EU 2018 (DFRWS EU), March 2017.
2. Yuhei Kawakoya, Eitaro Shioji, Yuto Otsuki, Makoto Iwamura, and Takeshi Yada, "Stealth Loader: Trace-free Program Loading for API Obfuscation" Proceedings of 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), September 2017.
3. Yuhei Kawakoya, Makoto Iwamura, Eitaro Shioji, and Takeo Hariu, "API Chaser: Anti-analysis Resistant Malware Analyzer", Proceedings of 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), October 2013.
4. Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu, "Code Shredding: Byte-Granular Randomization of Program Layout for Detecting Code-Reuse Attacks", Proceedings of 28th Annual Computer Security Applications Conference (ACSAC), December 2012.
5. Mitsuaki Akiyama, Yuhei Kawakoya, and Takeo Hariu , "Scalable and Performance-Efficient Client Honeypot on High Interaction System", Proceedings of 2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet (SAINT), July 2012.
6. Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh , "Memory behavior-based automatic malware unpacking in stealth debugging environment", Proceedings of 5th IEEE International Conference on Malicious and Unwanted Software (MALWARE), October 2010.
7. Mitsuaki Akiyama, Yuhei Kawakoya, Makoto Iwamura, Kazufumi Aoki, Mitsutaka Itoh, "MARIONETTE: Client Honeypot for Investigating and Un-

derstanding Web-based Malware Infection on Implicated Websites”, Joint Workshop on Information Security 2009 (JWIS), August 2009.

8. Yuhei Kawakoya, “VM-Based Malware Detection System”, Proceedings of 16th USENIX Security Symposium, WORK-IN-PROGRESS, August 2007.
9. Yuhei Kawakoya and Yoichi Muraoka, “Proposal and Implementation of Router-Based Traceback Technique”, Proceedings of International Multi-conference in Computer Science and Computer Engineering (SAM), June 2004.

Invited Talks

1. Yuhei Kawakoya, “Analysis Evasion. Introduction of Self-Made Packer”, SecurityCamp 2018, August 2018.
2. Yuhei Kawakoya, “Unpacking Introduction”, anti Malware engineering WorkShop 2011 (MWS2011), October 2011.
3. Yuhei Kawakoya, “The Secrets of the Development of Stealth Debugger”, RSA Conference Japan 2010, September 2010.

Books

1. 川古谷他 5 名（監訳）, “サイバーセキュリティプログラミング —Python で学ぶハッカーの思考”, オライリー・ジャパン, 2015 年 10 月.
2. 川古谷他 7 名（翻訳）, “実践 Metasploit——ペネトレーションテストによる脆弱性評価”, オライリー・ジャパン, 2012 年 5 月.
3. 川古谷他 4 名, “アナライジング・マルウェア——フリーツールを使った感染事案対処", オライリー・ジャパン, 2010 年 12 月.

Others

1. Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Tatsuya Mori, “IOC Conversion with Symbolic Execution”, Proceedings of the Computer Security Symposium 2018 (CSS2018), October 2018 (in Japanese).
2. Toshionori Usui, Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, and Jun Miyoshi, “Automatic Enhancement of Script Engines by Appending Behavior Analysis Capabilities”, Proceedings of the Computer Security Symposium 2018 (CSS2018), October 2018 (in Japanese)(MWS Best Paper Award).
3. Takeo Hariu, Daiki Chiba, Mitsuaki Akiyama, Takeshi Yagi, Yuhei Kawakoya, Yukio Nagafuchi, and Takaaki Koyama, “Cyberattack Countermeasure Technology to Support NTT’s Security Business”, NTT Technical Review, vol.16, no.5, pp.1-9, May 2018.
4. Yuhei Kawakoya, Makoto Iwamura, and Jun Miyoshi, “Taint-assisted Forensics for IAT Reconstruction”, Proceedings of the Computer Security Symposium 2017 (CSS2017), October 2017 (in Japanese)(CSS Excellent Paper Award).
5. Asuka Nakajima, Ren Kimura, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu, “An Investigation of Method to Assist Identification of Patched Part of the Vulnerable Software Based n Patch Diffing", Proceedings of the Multimedia, Distributed, Cooperative, and Mobile Symposium 2017 (DICOMO2017), June 2017 (in Japanese).
6. Hiroaki Anada, Yuhei Kawakoya, and Kuniyasu Suzaki, “A Report on International Conference NDSS2015", IEICE Technical Report, vol.115, no.81, pp.63-68, June 2015 (in Japanese).
7. Yuhei Kawakoya, Eitaro Shioji, Makoto Iwamura, and Takeo Hariu, “Identifying the Contents of Malware Communication Using Data Dependency Between API Calls”, Proceedings of the Computer Security Symposium 2013 (CSS2013), pp. 745 - 752, October 2013 (in Japanese).
8. Yuhei Kawakoya, Eitaro Shioji, Makoto Iwamura, and Takeo Hariu, “Tracing malicious code with Taint Propagation”, Proceedings of the Computer Security Symposium 2012 (CSS2012), pp. 1-8, October 2012 (in Japanese)(MWS Best Paper Award).

9. Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu, “Identifying the Code to be Analyzed with Taint Tags”, IEICE Technical Report, vol.112, no.128, pp.77-82, July 2012 (in Japanese).
10. Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu, “Detecting Invalid Control Flow with Pseudo-Dispersion of Program Code”, IEICE Technical Report, vol.112, no.128, pp. 103-108, July 2012 (in Japanese)(ICSS Research Award).
11. Makoto Iwamura, Yuhei Kawakoya, and Takeo Hariu, “Specifying the Addresses of IAT Entries”, Proceedings of the Computer Security Symposium 2011 (CSS2011), pp.12-17, October 2011 (in Japanese).
12. Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu, “Dynamic Packer Identification Based on Instruction Trace”, Proceedings of the Computer Security Symposium 2011 (CSS2011), pp.18-23, October 2011 (in Japanese).
13. Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh, “Dense Ship : Virtual Machine Monitor Specialized for Server-Type Honeypot”, IEICE Technical Report, vol.111, no.81, pp. 63-68, June 2011 (in Japanese).
14. Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu, “Automatic Unpacking Based on Entropy of Memory-Access Values", IEICE Technical Report, vol.110, no.475, pp. 41-46, March 2011 (in Japanese).
15. Shinta Nakayama, Kazufumi Aoki, Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh, “Address Independent Breakpoints Using Extended Memory Function", Proceedings of the Computer Security Symposium 2011 (CSS2010), pp.213-218, October 2010 (in Japanese).
16. Kazufumi Aoki, Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh, “Investigation about Malware Execution Time in Dynamic Analysis", Proceedings of the Computer Security Symposium 2011 (CSS2010), pp.543-548, October 2010 (in Japanese).
17. Mitsutaka Itoh, Takeo Hariu, Naoto Tanimoto, Makoto Iwamura, Takeshi Yagi, Yuhei Kawakoya, Kazufumi Aoki, Mitsuaki Akiyama, and Shinta Nakayama, “Anti-Malware Technologies”, NTT Technical Review, vol.8, no.7, pp.1-7, July 2010.

18. Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh, “Automatic OEP Finding Method for Malware Unpacking”, IEICE Technical Report, vol.110, no.79, pp. 13-18, June 2010 (in Japanese).
19. Kazufumi Aoki, Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh, “Malware Behavior Analysis using Semipermeable Network", Proceedings of the Computer Security Symposium 2009 (CSS2009), pp.1-6, October 2009 (in Japanese).
20. Mitsuaki Akiyama, Makoto Iwamura, Yuhei Kawakoya, Kazufumi Aoki, and Mitsutaka Itoh, “Implementation and Evaluation of Detection Methods on Client Honeypot", Proceedings of the Computer Security Symposium 2009 (CSS2009), pp.1-6, October 2010 (in Japanese).
21. Yuhei Kawakoya, Mitsuaki Akiyama, Kazufumi Aoki, Mitsutaka Itoh, and Hiroki Takakura, “Investigation of Spam Mail Driven Web-Based Passive Attack”, IEICE Technical Report, vol.109, no.33, pp. 21-26, May 2009 (in Japanese).
22. Mitsuaki Akiyama, Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh, “Investigation and Understanding about Web-based Malware using Client Honeypot", Proceedings of the Computer Security Symposium 2008 (CSS2008), pp.319-324, October 2008 (in Japanese).
23. Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh, “Analyzing Malware with Stealth Debugger”, Proceedings of the Computer Security Symposium 2008 (CSS2008), pp.115-120, October 2008 (in Japanese).
24. Yuhei Kawakoya, Tadaaki Yanagiwara, Makoto Iwamura, and Mitsutaka Itoh, “Honey Patch : A Method of Intrusion Detection on Honeypot”, Proceedings of the 2006 IEICE General Conference, No.211, May 2006 (in Japanese)
25. Yuhei Kawakoya and Yoichi Muraoka, “Proposal and Implementation of Rental-Retrieve Lottery Resource Management Algorithm against Denial Resource Consumption Attack", Proceedings of the 10th Workshop for Programming and Application System of Japan Society for Software Science and Technology (SPA2005), March 2005 (in Japanese).
26. Koichi Kato, Yuhei Kawakoya, Yusuke Shibata, Katsunobu Ishida, Tomonari

Sonoda, and Yoichi Muraoka, “Internet Telephony Protocol using no Dedicated Location Server IPSP:InetPhone Signaling Protocol", IPSJ SIG Technical Report, No.2, pp.49-55, January 2004 (in Japanese).

Copyrights