Recommended Paper

Taint-assisted IAT Reconstruction against Position Obfuscation

Yuhei Kawakoya[1,a)] Makoto Iwamura[1,b)] Jun Miyoshi[1,c)]

Received: April 18, 2018, Accepted: September 7, 2018

Abstract: Windows Application Programming Interface (API) is an important data source for analysts to effectively understand the functions of malware. Due to this, malware authors are likely to hide the imported APIs in their malware by taking advantage of various obfuscation techniques. In this paper, we first build a formal model of the Import Address Table (IAT) reconstruction procedure to keep our description independent of specific implementations and then formally point out that the current IAT reconstruction is vulnerable to position obfuscation techniques, which are anti-analysis techniques obfuscating the positions of loaded APIs or Dynamic Link Libraries (DLLs). Next, we introduce an approach for API name resolution, which is an essential step in IAT reconstruction, on the basis of taint analysis to defeat position obfuscation techniques. The key idea of our approach is that we first define taint tags, each of which has a unique value for each API, apply the taint of the API to each of its instructions, 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 demonstrate that a system in which our proposed API name resolution has been implemented enables us to correctly identify imported APIs even when malware authors apply various position obfuscation techniques to their malware.

Keywords: malware, IAT-reconstruction, Taint Analysis, Anti-analysis, Windows API

1. Introduction

Manual static analysis by an analyst, i.e., reading machine instructions one by one, is our last resort to fight against malware because anti-analysis techniques implemented by recent malware have been drastically sophisticated and can hinder dynamic analysis. However, a manual static analysis is time-consuming since it requires analysts to read a vast number of machine instructions, which are more difficult to read than high-level programming languages, such as C or Python. In this situation, Windows Application Programming Interfaces (APIs), which are called from some of the instructions, are a key data source to efficiently proceed static analysis since they are well-documented and human-friendly.

Malware authors understand this situation well, so they apply various types of obfuscation to their malware to hide APIs used in them [28], [29]. One example of obfuscation is that they remove the metadata of imported APIs or make it unreachable from the header of their malware. More concretely, they remove Import Name Tables (INTs), dereference both INTs and Import Address Tables (IATs) from the Portable Executable (PE) header and then replace them with their own loader that resolves API dependencies at runtime. This removal or dereferences cause disassemblers, e.g., IDA [9], to fail to recognize imported APIs in disassembled code. We call this process *IAT obfuscation*.

Background. When we (malware analysts) statically analyze malware whose imported APIs are obfuscated, we try to restore the removed and dereferenced metadata of imported APIs, i.e., INTs and IATs, before starting to read machine instructions. We call this process *IAT reconstruction*, and its steps are as follows: (1) Acquire a memory dump, (2) Identify IATs, (3) Resolve API names, and (4) Restore the PE header.

In step (1), we execute malware in an environment under control and stop its execution when all APIs imported by the malware have been dynamically resolved by their own loader and then generate a memory dump. In step (2), we first search the executable memory regions of the malware's process in the memory dump for the indirect call or jump instructions, such as `call [0x1001000]` or `jmp [0x1001000]`. The memory area referenced by an indirect instruction becomes a candidate for an IAT entry. Then, we regard as an IAT the memory area where many IAT entry candidates are gathered in a cluster like a table. In step (3), we relate all addresses in the IATs with the API name by matching them with those calculated from the base address of a loaded Dynamic Link Library (DLL) and the offset of each export API listed in the Export Address Table (EAT) in its PE header and then rebuild the INT corresponding to an IAT. Many analysis tools acquire the base address of a loaded DLL from the Process Environment Block (PEB) or a Virtual Address Descriptor (VAD), which are Windows-managed data structures. In step

1 NTT Secure Platform Laboratories, Musashino, Tokyo 180–8585, Japan
a) kawakoya.yuhei@lab.ntt.co.jp
b) iwamura.makoto@lab.ntt.co.jp
c) miyoshi.jun@lab.ntt.co.jp

The preliminary version of this paper was presented at Computer Security Symposium 2017 (CSS2017) in October 2017, and recommended to be submitted to Journal of Information Processing (JIP) by the program chair of CSS2017.

(4), we optionally restore the removed parts of the PE header and fix the pointers to the IATs and INTs. As a result of IAT reconstruction, disassemblers become able to correctly recognize the imported APIs.

Problem. However, when malware authors apply position obfuscation techniques to their malware, they could make IAT reconstruction, especially the API name resolution step, infeasible. Position obfuscation is an anti-analysis technique to place API code or a DLL on a memory area, which is different from the memory area where the program loader originally places. They are classified into two types on the basis of the target: API or DLL. Stolen Code [29] and Copied API Obfuscation [28] are examples of position obfuscation techniques. These techniques are used to obfuscate the positions of placed APIs by making copies of them. DLL Unlinking [16] and Stealth Loader [12] are examples of DLL position obfuscation techniques. These techniques obfuscate the locations of loaded DLLs by hiding data structures containing their metadata.

We build a formal model of IAT reconstruction, which is independent of specific implementations and suitable for analyzing problems in its design, which most existing IAT reconstruction tools follow. Then, we analyze the formal model to find design flaws. So far in our analysis, we have found that the current IAT reconstruction is vulnerable to position obfuscation techniques, and we believe this is because it is implicitly dependent on two assumptions.

- The addresses in an IAT, i.e., the pointers to the codes of each API, are directly computed from the offsets in the EAT and the base address of the DLL.
- The loaded addresses of each DLL are correctly managed by an Operating System (OS) and can be accessed through specific data structures of the OS.

Position obfuscation techniques attack these two assumptions by intentionally creating a situation in which either or both are not satisfied. Since this causes each address in an IAT to not match any calculated addresses, we fail to resolve the API names from the addresses. As such IAT reconstruction fails. Since position obfuscation techniques can directly affect API identification, which is a fundamental component in most API-based security products and research, malware may hide specific behaviors, especially the most harmful ones, from those security mechanisms by using position obfuscation techniques. This would be an advantage for attackers to potentially evade security in many places.

Our Approach. To overcome position obfuscation techniques, we introduce a new API name resolution approach based on taint analysis. The key idea is that 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 approach has two advantages for countering position obfuscation 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 two assumptions and not be affected by position obfuscation techniques.

We have implemented this approach 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) [3], and then set taint tags on them. For dynamic analysis, we use API Chaser [11], 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.

Experiments. To assess the effectiveness of our system, we have conducted three types of experiments. The first one is for evaluating whether our approach is more resistant to position obfuscation techniques than existing IAT reconstruction tools. For that purpose, we prepared several Windows executables and obfuscated their APIs by using the 4 position obfuscation techniques mentioned above. Then, we analyzed them using our system and 3 other IAT reconstruction tools: impscan, impscan++ [*1], and Scylla [18]. The results show that only our system could correctly identify all imported APIs, whereas the others could not because of position obfuscation 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.

Contribution. We make the contributions of this paper to be

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

as follows.

- We formally model the IAT reconstruction process and clearly illustrate that the process is vulnerable to position obfuscation techniques because it is implicitly dependent on the two above assumptions. We also experimentally illustrate that existing major IAT reconstruction tools in which the process is implemented are ineffective against position obfuscation techniques.
- We introduce an API name resolution approach based on taint analysis, which is generic and effective against position obfuscation techniques. Then, we also describe an implementation of a system that uses this approach for IAT reconstruction. This implementation contains *TaintVolatility*, which is the first integration of Volatility with TSK, which can be used for offline memory dump analysis without being pre-configured.
- We demonstrate the effectiveness of our API name resolution through the experiments and show it is independent of the two assumptions and robust against position obfuscation techniques.

2. Position Obfuscation

In this section, we first define a formal model for IAT reconstruction and we use the abstract formal model to keep our description independent of implementation. Next, we explain position obfuscation techniques, which are anti-analysis techniques for hindering IAT reconstruction. Finally, we analyze a problem of the existing IAT reconstruction procedure using the defined model.

2.1 Abstract Model of IAT Reconstruction

First, we model the memory dump of a program and then define both procedures of IAT obfuscation and IAT reconstruction with the model. For preparation, we define several notations and functions. We denote by A the set of addresses of the data storage for a program such as a memory and disk. In this model, we do not distinguish a memory from disk as a data storage. We can access the data stored at the specific position of the storage with the address $a \in A$. We also denote by P the set of programs in a process memory dump. In addition, we denote by $P_{exe} \subseteq P$ the set of executable programs and $P_{dll} \subseteq P$ the set of library programs in a memory dump. For notation, let F_p denote the set of functions in a program $p \in P$, $expF_p \subseteq F_p$ the set of functions exported from a program p, and $impF_{p,q} \subseteq F_q$ the set of functions imported by $p \in P$ from $q \in P_{dll}, q \neq p$. We let S denote the set of all possible symbols of the functions of all programs in a memory dump. We denote by $\delta : F_p \rightarrow A$ the function to return the address of a function. We denote by $\psi : P \rightarrow A$ the function to return the base address of a program. We denote by $\varphi : A \rightarrow S$ the function to resolve the symbol of an address if the address has the symbol. Otherwise, the function returns $\emptyset$.

The memory dump of a process comes from the execution of multiple program components including the main program and the libraries it depends on. To model this, we define a program $p \in P$ in the memory dump as a tuple (I_p, ET_p, ID_p, L_p) where I_p is the ordered set of instructions in the code regions of a program $p \in P$, ET_p is the ordered set of the metadata of functions exported from $p \in P$, ID_p is the set of $IT_{p,q}$ where $IT_{p,q}$ is the ordered set of the metadata of functions imported by $p \in P$ from $q \in P_{dll}$, and L_p is the set of library programs loaded by $p \in P$: $L_p = \{l : l \in P_{dll}\}$. For ET_p, the metadata of an exported function is defined as (rva, sym) where rva is the relative virtual address of a function, i.e., the offset from the base address of a loaded library program and calculated from $\delta(f) - \psi(p), f \in expF_p; sym$ is the symbol of the function and is acquired from $\varphi(\delta(f))$. ID_p is the set of $IT_{p,q \in L_p}$ and each $IT_{p,q}$ is the ordered set of a tuple (va, sym) where va is the virtual address of a function and is calculated from $\delta(f), f \in impF_{p,q \in L_p}; sym$ is the symbol of the function and is acquired from $\varphi(va)$.

```
input : $I_p, L_p$
output: $\Pi$
1 $\{iat_q : q \in L_p\} \leftarrow$ IdentifyIAT($I_p$);
2 for $iat \in iat_q, q \in L_p$ do
      // $iat$ is the ordered set of virtual addresses.
3     $imp\_table \leftarrow \emptyset$;
4     for $va \in iat$ do
5         $sym \leftarrow$ ResolveName($va, L_P$);
          // We define '$\Leftarrow$' as 'adjoin'
6         $imp\_table \Leftarrow (va, sym)$;
7     $\Pi \leftarrow \Pi \cup \{imp\_table\}$;
8 return $\Pi$
```
Algorithm 1: IAT Reconstruction

IAT Obfuscation. When malware authors apply IAT obfuscation for their malware, the IATs and INTs of the malware become unreachable from its PE header and unavailable for analysis. To model this action, we define IAT obfuscation as a function to drop ID_p from the memory dump of p. That is, when we define the memory dump of a program whose API is obfuscated as p', p' is modeled as a tuple $(I_p, ET_{p'}, L_{p'})$, $L_p \subset L_{p'}$. To keep the explanation simple, we do not consider any obfuscation for the code regions. In addition, $ET_{p'}$ is $\emptyset$ in the case of $p' \in P_{exe}$. That is, only I_p and $L_{p'}$ remain and are available for later analysis.

IAT Reconstruction. Algorithm 1 gives an overview for IAT reconstruction. IAT reconstruction is a process of identifying IATs, rebuilding the removed INTs by resolving API names, and optionally restoring the PE header for INTs and IATs. As a result of this, it allows disassemblers to recognize an imported API properly. Algorithm 1 receives I_p and L_p and then outputs Π, which is the set of *imp_table*. Since each *imp_table* is the same as $IT_{p,q}$, the set of *imp_table*, i.e., Π, is the same as ID_p. The algorithm first identifies IATs with `IdentifyIAT` (line 1), which we will explain in detail in Algorithm 2. Then, the algorithm enumerates all entries of each identified IAT and then resolves the API name of each entry with `ResolveName` (line 5). The resolved API name *sym* is appended to *imp_table* as a tuple with *va*. At line 7, a *imp_table* is appended to Π.

Algorithm 2 gives an algorithm for IAT identification. For notation, let i[TA] [*2] denote the address referenced by the instruction i, i.e., $i[\text{TA}] \in A$, when i is an indirect call or jump and let $*i$[TA] denote the value stored in the memory at the address i[TA].

[*2] TA means **T**arget **A**ddress of indirect call or jump instructions.

```
1  Function IdentifyIAT($I_p$)
       Data: $I_p$
       Result: $\Pi$
2      for $i \in I_p$ do
           // $ISA_{indirect}$ is a set of indirect call
              instructions.
3          if $i \in ISA_{indirect}$ then
4              candidate $\Leftarrow (i[TA], *i[TA])$;
5      sort(candidate);
6      $\Pi \leftarrow \emptyset$;
7      $iat \leftarrow \emptyset$;
8      for $j \leftarrow 0$ do
9          $iat \Leftarrow$ candidate [j][1] ;
10         if candidate[$j$][0] != candidate[$j+1$][0] $-$ 4 then
11             $\Pi \leftarrow \Pi \cup \{iat\}$;
12             $iat \leftarrow \emptyset$;
13         $j \leftarrow j + 1$;
14     return $\Pi$;
```

Algorithm 2: IAT Identification

```
1  Function ResolveName ($va$, $L_p$)
       Data: $va$, $L_p$
       Result: $symbol$
2      for $l \in L_p$ do
3          for $(rva, sym) \in ET_l$ do
               // Here is an attack vector
4              if $va == rva + \psi(l)$ then
5                  return $sym$ ;
6      return $\emptyset$;
```

Algorithm 3: Name Resolution

`IdentifyIAT` receives a set of instructions I_p, and then for each instruction, it checks if the instruction is an indirect call or jump (line 4). If it is, i[TA] and $*i$[TA] are included in `candidate` [*3] as a candidate for an IAT entry. Then, `candidate` is sorted on the basis of the address of each entry (line 5). After that, for each entry in `candidate`, the algorithm checks if the addresses of the two sequential entries in `candidate` are 4-byte aligned (line 11). If so, the two entries belong to the same IAT; otherwise, they belong to different IATs.

Algorithm 3 shows the overview for API name resolution. For each library l, it first enumerates the elements of ET_l, which is a set of (rva, sym) (line 3). The algorithm also calculates the base address of l with $\psi(l)$, adds rva to the base address to calculate the virtual address of a function exported by l, and then compares the virtual address with va, which is an entry in an identified IAT (line 4). If they match, there is the function at va. Thus, the algorithm returns the sym as the name of the function placed at va.

2.2 API Position Obfuscation

API position obfuscation is a technique for calling an API that is copied to an allocated buffer from its original one. This involves making a copy of all or a part of the code of an API and then transferring the execution to the API via the copied instructions. This technique allows malware to avoid the hooks for monitoring, which are often set on the first instruction of an API. There are two types of API position obfuscation techniques: Stolen Code [29] and Copied API Obfuscation [28].

[*3] `candidate` is the ordered set.

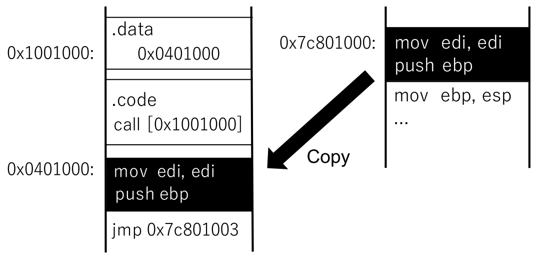


Fig. 1 Stolen Code.

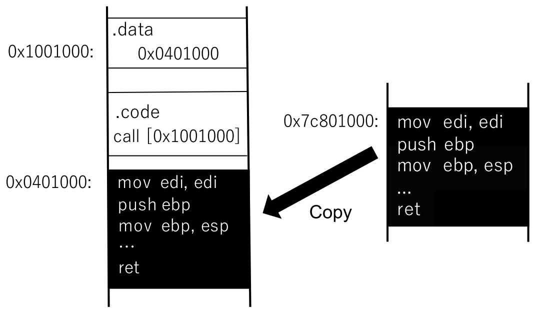


Fig. 2 Copied API obfuscation.

Stolen Code invokes an API via a few instructions copied from the head of an API. In the example in **Fig. 1**, `mov edi, edi` and `push ebp` are copied from `0x7c801000` to `0x0401000`, and `jmp` is put after them. This `jmp` transfers the execution to the instruction right after the copied instructions, i.e., `mov ebp, esp` at `0x7c801003`. Since many analysis tools set hooks on the head of an API for monitoring, malware can avoid these hooks by using Stolen Code.

Copied API Obfuscation is an evolved version of Stolen Code. Whereas Stolen Code makes a copy of a few instructions from the API head, Copied API Obfuscation makes a copy of all instructions of an API. It invokes an API by executing the copied instructions without jumping to the instructions placed at the original position. In the example in **Fig. 2**, all instructions from `mov edi, edi` at `0x7c801000` to `ret` are copied to the buffer at `0x0401000` and then executed.

2.3 DLL Position Obfuscation

DLL position obfuscation is a technique to hide the existence of loaded DLLs by hiding their metadata. Since the base address of a loaded DLL is often used for calculating the position of an API, if it is hidden, the calculation results in a wrong address. There are two types of DLL position obfuscation techniques: DLL Unlinking [16] and Stealth Loader [12].

DLL Unlinking unlinks the entry of a specific DLL from the linked lists in PEB, which are used for managing loaded DLLs. **Figure 3** shows a case when the entry of advapi32.dll is unlinked from InLoadOrderModuleList. Due to this, even if an analysis tool searches the list for the entry of advapi32.dll, it fails to find it since the entry is unreachable from the list. As we already mentioned, if the loaded address of a DLL is not found, API name resolution fails.

Stealth Loader [12] is a program loader totally independent of the Windows standard program loader and is embedded into an executable. When the executable starts to run, Stealth Loader

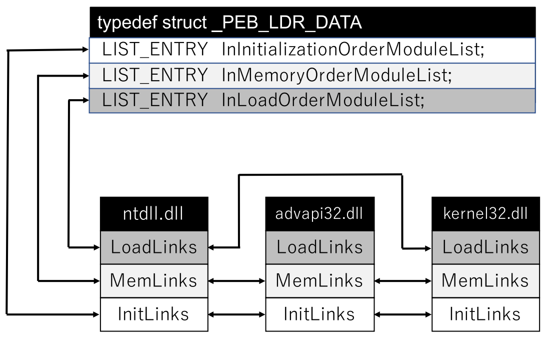

typedef struct _PEB_LDR_DATA
LIST_ENTRY InInitializationOrderModuleList;
LIST_ENTRY InMemoryOrderModuleList;
LIST_ENTRY InLoadOrderModuleList;
ntdll.dll
advapi32.dll
kernel32.dll
LoadLinks
MemLinks
InitLinks

Fig. 3 DLL Unlinking.

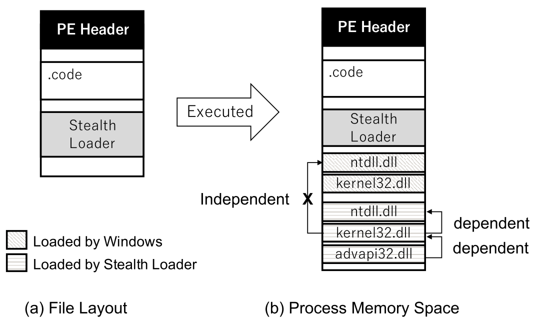


Fig. 4 Stealth Loader.

loads dependent Windows system DLLs, such as kernel32.dll or ntdll.dll, by itself without leaving any footprints of loaded DLLs. To do that, it avoids using map-related functions for allocating code on memory or makes dependencies on only stealth-loaded DLLs, as shown in **Fig. 4**. Since Stealth Loader does not leave any footprints of loaded DLLs, a system DLL loaded with Stealth Loader is not recognized as *'loaded'* by analysis tools or even Windows OS. In addition, the APIs exported from the DLL loaded with Stealth Loader are callable with the same interface as the original ones. Nevertheless, these API calls are not recognized as *'API call'* and look like unknown external function calls since there is no evidence to identify the function as an API.

2.4 Problem Analysis

We analyze design problems in the current IAT reconstruction procedure, which is modeled in Section 2.1 and on which many IAT reconstruction tools depend. So far in our analysis of the algorithms, we have found that Algorithm 3 contains an attack vector at line 4, which malware can exploit to evade IAT reconstruction. Position obfuscation techniques are designed to attack this part. More concretely, they attack API name resolution, which is an essential part of IAT reconstruction. In other words, malware authors intentionally create a situation where $va == rva + \psi(l)$ (line 4 in Algorithm 3) is unsatisfiable to make API name resolution fail.

When a malware uses API position obfuscation techniques, it makes a copy of the instructions of a certain API. To define this action in our model, we denote I_f as the ordered set of instructions i of a function $f \in F_p$: $I_f = (i_0, i_1, ..., i_n)$. When a malware, for example, performs Stolen Code, it copies the first-$m, (0 < m < n)$ instructions to a buffer. We denote the copied instructions as $I'_f = (i'_0, i'_1, ..., i'_m)$, and we also denote by $\mu : I_f \to A$ the function to return the address of an instruction. Then the malware updates the entry in an IAT for f with $\mu(i'_0)$. Due to $\mu(i_0)$ $! = \mu(i'_0)$, va, which is the updated entry in the IAT and passed to `ResolveName` as an argument, is not matched with any address calculated from $\psi(l)$ and rva. As a result, $va == rva + \psi(l)$ is not satisfied through any ET_l entries, and then `ResolveName` returns $\emptyset$. DLL position obfuscation techniques also create a situation where $va == rva + \psi(l)$ is not satisfied by making $\psi(l)$ infeasible, which means the base address of a loaded DLL l is unknown. As a result, we fail to calculate the correct addresses where each API of hidden DLLs is placed and then `ResolveName` returns $\emptyset$.

In our analysis so far, we have found that existing API name resolutions implicitly depend on two assumptions.

- The addresses in an IAT, i.e., the pointers to the code of each API, are directly computed from the entries in the EAT of a DLL and the base address of the DLL.
- The loaded addresses of each DLL are correctly managed by an OS and can be accessed through specific data structures of the OS.

API position obfuscation techniques attack the first assumption, whereas DLL ones attack the second. Both intend to make $va == rva+\psi(l)$ unsatisfied so that IAT reconstruction fails. Since API identification is a fundamental component of API-based security mechanisms, such as API monitoring, API behavior-based malware detections or IAT reconstructions, all of these security mechanisms could be affected by position obfuscation techniques. That means, if a malware can intentionally create situations in which either of these two assumptions does not hold, it can evade security mechanisms. This design flaw possibly becomes an attack vector for malware to evade security products and technologies with the simple technique and thus conduct malicious activities without being detected.

3. Our Approach

In this section, we introduce a taint-based API name resolution approach, which is independent of the two assumptions mentioned in Section 2. First, we define the goal and scope of this paper. Second, we introduce a taint-based API name resolution. Finally, we explain a system for IAT reconstruction using our taint-based API name resolution.

3.1 Goal and Scope

Our goal in this paper is to present the first generic approach against position obfuscation techniques and to ensure a system in which the approach implemented works well in practice. *'Generic'* in this context means an approach commonly applicable for both API and DLL position obfuscation techniques without heuristics or adjustments depending on targets.

The scope of this paper is to solve the problem of API name resolution, i.e., the existing API name resolution is vulnerable to position obfuscation techniques. In other words, we focus on an approach for realizing an API name resolution that is independent of the two assumptions and robust enough against all position obfuscation techniques mentioned in Section 2. Other anti-analysis techniques targeting the other steps in the IAT reconstruction, such as memory dump acquisition or IAT identification, are beyond the scope of this paper.

3.2 Taint-based API Name Resolution

We introduce a new API name resolution approach for IAT reconstruction, taint-based API name resolution, which is generic and resistant to position obfuscation techniques. Our approach 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.

We define our approach with the abstract model. To extend the model defined in Section 2.1 with taint analysis capability, we introduce two more notations and three more functions. T denotes the set of defined tag_f where tag_f is the unique identifier for a function f: $T = \{tag_f : f \in \bigcup expF_l, l \in L_p\}$. SS denotes the set of tags used for tainting: $SS = \{s_a : s_a \in T, a \in A\}$ where s_a is the tag which is set on the address $a \in A$. We denote by $\sigma : T \rightarrow S$ the function to map $tag \in T$ to the symbol of the corresponding function f of tag_f, $\xi : F_p \rightarrow T$ the function to map f to tag, and $\gamma : \mathbb{P}(SS) \times A \rightarrow T$ the function to return the tag related to the address $a \in A$ where $\mathbb{P}(SS)$ is the power set of SS, which is all possible subsets of SS.

Algorithm 4 gives an overview of the initial taint setting. It sets the corresponding taint tag to all the instructions of each API. The corresponding tag is acquired through the ξ function. After the tags are set, they are stored in SS.

```
Data: $\bigcup expF_{l \in L_p}$
Result: $SS$
1 $SS \leftarrow \emptyset$;
2 for $f \in \bigcup expF_{l \in L_p}$ do
3     for $i \in I_f$ do
4         $s_{\mu(i)} \leftarrow \xi(f)$;
5         $SS \leftarrow SS \cup \{s_{\mu(i)}\}$;
6 return $SS$;
```

Algorithm 4: Preprocessing: Initial Tainting

Algorithm 5 gives an overview of taint-based API name resolution. SS' is the shadow storage after dynamic analysis has been done. That is, some taint tags were propagated and stored in the different addresses from SS. If the passed address va, which is an entry in an IAT, has the taint tag, it acquires the symbol of the API related to the tag using σ and then returns it. Otherwise, since the address does not have any tag, it simply returns $\emptyset$. The difference from Algorithm 3 is that Algorithm 5 resolves the API name of va on the basis of tag, whereas Algorithm 3 relies on the calculation with $va == rva + \psi$, which is the attack vector targeted by position obfuscation techniques. By avoiding the attack vector in our algorithm, we can resolve the API name without depending on the calculation.

```
1 Function TaintResolveName (va, SS')
      Data: $va$, $SS'$
      Result: $symbol$
2     $tag \leftarrow \gamma(va, SS')$;
3     if $tag \neq \emptyset$ then
4         return $\sigma(tag)$;
5     else
6         return $\emptyset$;
```

Algorithm 5: Taint-based Name Resolution

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 [20] or identifying sensitive information leaking [7]. A new aspect in this paper 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, at instruction-level granularity. 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 are managed in the VMM layer and the VMM is isolated from the malware running environment.

3.3 Taint-assisted IAT Reconstruction System

We have developed a system for IAT reconstruction whose API name resolution is realized with our taint-based approach. **Figure 5** 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.

3.3.1 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

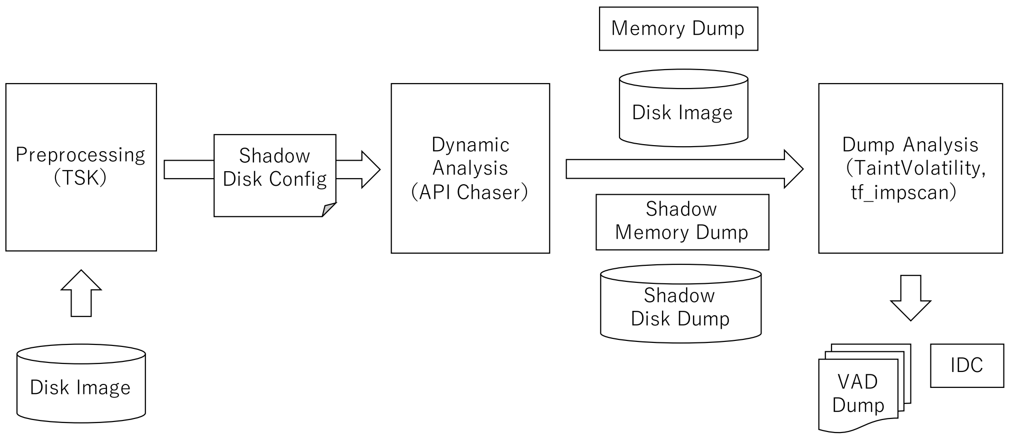


Fig. 5 Workflow of Our System.

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 approach 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 original 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 by an OS.

3.3.1.1 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 [19], 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.

3.3.2 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.

3.3.2.1 Implementation

For a dynamic analysis engine, we use API Chaser [11], which is an API monitoring system with taint analysis capability. API Chaser is built on QEMU [2] (Argos [20]) 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.

3.3.3 Dump Analysis

In the dump analysis phase, we conduct IAT reconstruction as a preparation for static analysis, following Algorithm 6. 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 our approach explained in Algorithm 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 Section 3.2. This function allows us to resolve API names without being evaded by position obfuscation techniques, as we have already explained.

```
input : $I_p, SS'$
output: $\Pi$
1 $\{iat_q : q \in L_p\} \leftarrow$ IdentifyIAT($I_p$);
2 for $iat \in iat_q, q \in L_p$ do
3     $imp\_table \leftarrow \emptyset$;
4     for $va \in iat$ do
5         $sym \leftarrow$ TaintResolveName($va, SS'$);
6         $imp\_table \Leftarrow (va, sym)$;
7     $\Pi \leftarrow \Pi \cup \{imp\_table\}$;
8 return $\Pi$
```

Algorithm 6: Taint-assisted IAT Reconstruction

3.3.3.1 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 [21], 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 changes.

4. 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 position obfuscation 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.

4.1 Position Obfuscation Resistance

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

4.1.1 Procedure

We prepared several Windows executables whose imported APIs were already known. Then, we obfuscated the APIs imported by these executables using position obfuscation techniques which we explained in Section 2 and Code Insertion, which is a technique for obfuscating control flow between API call sites to corresponding API codes. 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. impscan++ 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.

4.1.2 Results

Table 1 shows the results of this experiment. Our system successfully defeated all obfuscation techniques except for Code Insertion, 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 and Copied API Obfuscation, the addresses filled in the IATs point to buffers prepared for the stolen 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.

Regarding Code Insertion, all tools including tf_impscan failed to identify all imported APIs. The reason of this failure in tf_impscan will be discussed in Section 6.

4.2 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.

Table 1 Results of resistant capability experiment.

-	Stolen Code	Copied API	DLL Unlinking	Stealth Loader	Code Insertion
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.

Table 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 Section 3.3. # 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.

Table 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.

4.2.1 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.

4.2.2 Results

Table 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 approach 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.

4.3 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 TaintVolatility.

4.3.1 Procedure

We used the same memory dumps as the second experiment (Section 4.2). 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 1 TB flush storage.

4.3.2 Results

Table 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 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. Related Work

In this section, we mention related work. We mainly focus on differences and similarities between this work and our approach.

API Chaser [11] is a dynamic analysis environment using taint analysis specialized for analyzing advanced malware armored with anti-analyses. The basic idea of this paper comes from API Chaser's API monitoring mechanism, i.e., tainting each API code with a unique taint tag. However, since API Chaser is designed for dynamic analysis, it can resolve the API names of only the executed part of code when we apply API Chaser for API name resolution for static analysis. On the other hand, our approach is applicable for resolving the API names referenced from the parts of code that are not executed during dynamic analysis.

Eureka [26] identifies API call references by analyzing the control flow graph (CFG) of the code of malware. To identify the loaded addresses of each DLL for API name resolution, it relies on monitoring a call of the NtMapViewOfSection API and extracting its input and output arguments. Eureka relates the mapped address, i.e., the loaded address of a DLL, with the DLL name given to the API call. However, this approach is also vulnerable to Stealth Loader, because Stealth Loader does not use map functions at all for loading a DLL.

Choi [5] proposed an approach for handling Stolen Code by monitoring all memory accesses and identifying the destination addresses of a copy of the code. Even though the implementations are different, his idea is similar to ours in that both ideas focus on tracking the movement of API code. We use taint analysis for tracking, whereas Choi uses memory trace.

Rekall [25], which is a memory forensics tool cloned from Volatility, can analyze both memory and disks but can only be used in two situations. The first is when Rekall is running in live forensics mode. That is, Rekall can analyze disks in the environment where it is running. The second is when analysts explicitly specify files when they acquire a memory dump in AFF4 [17] file format. Using this format, they can add specified file contents to a memory dump, and Rekall can analyze the added file contents when analyzing the memory dump. On the other hand, TaintVolatility is mainly designed for analyzing both memory and disks in offline mode, so we can use it without any pre-configuration before starting an analysis.

Formal representations for malware analysis have been studied in Refs. [6], [14], and [15]. The main difference between their works and our paper is that their models are built on execution traces, whereas our model is on a memory dump. To deal with a memory dump, we introduce several new notations and functions.

Quist et al. [22] use VADs for identifying the mapped DLLs for IAT reconstruction, whereas Raber et al. [23] use API hooking. As with the approaches explained in Section 2, these approaches are also vulnerable against position obfuscation techniques. For example, Stealth Loader does not leave any traces of loaded DLLs in VADs. Moreover, Stealth Loader does not execute the API codes on which hooks are installed because it loads dependent DLLs by itself.

6. Discussion

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

6.1 Limitation

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

Code Insertion Our approach fails when code snippets are inserted in the control flow between each entry in an IAT and the corresponding API code. This technique is used in API redirection [29]. 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 approach by applying CFG analysis, as used in Eureka [26]. 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.

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 [8], [24]. 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 [6], [13].

DLL Static Linking When a system DLL is statically linked to a malware executable, we cannot identify the APIs exported from the DLL [1]. 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.

Implicit Information Flow The implicit flow problem is a general limitation of taint analysis. Taint propagation fails when tainted data is processed with implicit flow [4]. 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 [10], [27] to our system.

6.2 Validity of Experiments

We consider that the experiments we had are reasonable for achieving our goal in this paper, even though we did not have any experiment using real-world malware. Our goal is to propose an approach and develop a system resistant to position obfuscation. To measure its resistance capability against position obfuscation, 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.

6.3 Platform Dependency

We have developed our system with targeting for 32 bit-Windows 7 platform as an analysis environment (guest OS). However, our approach 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 approach 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 approach to resolve the names of the addresses in GOT.

7. Conclusion

In this paper, we first mentioned the problem from which existing approaches for Import Address Table (IAT) reconstruction currently, or will, suffer, i.e., position obfuscation techniques, and then proposed a new Application Programming Interface (API) name resolution based on taint analysis to solve the problem. We also described system components for IAT reconstruction whose API name resolution is realized with our approach. Additionally, we demonstrated that this system is generically effective for various types of position obfuscation techniques through experiments.

As far as we know, our paper is the first to describe position obfuscation techniques, clarify the impact they can have on existing security mechanisms, and introduce a generic and effective approach against them. We consider that position obfuscation techniques are a significant problem for us to work on because they enable malware authors to effectively evade current security mechanisms.

References

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

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

[3] Carrier, B.: The Sleuth Kit (TSK) (online), available from ⟨http://www.sleuthkit.org/⟩ (accessed 2017-08-17).

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

[5] Choi, S.: API Deobfuscator: Identifying Runtime-obfuscated API calls via Memory Access Analysis, *Black Hat Asia* (2015).

[6] Dinaburg, A., Royal, P., Sharif, M. and Lee, W.: Ether: Malware analysis via hardware virtualization extensions, *Proc. 15th ACM Conference on Computer and Communications Security, CCS '08*, pp.51–62, ACM (2008).

[7] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P. and Sheth, A.N.: 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 (online), DOI: 10.1145/2619091 (2014).

[8] Ferrie, P.: Attacks on Virtual Machine Emulators, *Symantec Security Response* (2006).

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

[10] Kang, M.G., McCamant, S., Poosankam, P. and Song, D.: DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation, *NDSS* (2011).

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

[12] Kawakoya, Y., Shioji, E., Otsuki, Y., Iwamura, M. and Yada, T.: Stealth Loader: Trace-free Program Loading for API Obfuscation, *Research in Attacks, Intrusions, and Defenses: Proc. 20th International Symposium, RAID 2017* (2017).

[13] Kirat, D., Vigna, G. and Kruegel, C.: Barecloud: Bare-metal Analysis-based Evasive Malware Detection, *Proc. 23rd USENIX Conference on Security Symposium, SEC'14*, Berkeley, CA, USA, USENIX Association, pp.287–301 (2014) (online), available from ⟨http://dl.acm.org/citation.cfm?id=2671225.2671244⟩.

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

[15] Korczynski, D. and Yin, H.: Capturing Malware Propagations with Code Injections and Code-Reuse Attacks, *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*, pp.1691–1708 (online), DOI: 10.1145/3133956.3134099 (2017).

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

[17] Michael Cohen, S.G. and Schatz, B.: Extending the Advanced Forensic Format to Accommodate Multiple Data Sources, Logical Evidence, Arbitrary Information and Forensic Workflow, *The Digital Forensic Research Conference DFRWS 2009 USA* (2009).

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

[19] Plohmann, D. and Hanel, A.: SimpliFiRE.IDAScope, *Hacklu* (2012).

[20] Portokalidis, G., Slowinska, A. and Bos, H.: Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation, *Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, EuroSys '06*, pp.15–27, ACM (2006).

[21] py4n6 (online), available from ⟨https://github.com/py4n6/pytsk⟩ (accessed 2017-08-17).

[22] Quist, D., Liebrock, L. and Neil, J.: Improving antivirus accuracy with hypervisor assisted analysis, *Journal in Computer Virology*, Vol.7, No.2, pp.121–131 (online), DOI: 10.1007/s11416-010-0142-4 (2011).

[23] Raber, J. and Krumheuer, B.: QuietRIATT: Rebuilding the Import Address Table Using Hooked DLL Calls, *Black Hat DC Briefings* (2009).

[24] Raffetseder, T., Krügel, C. and Kirda, E.: Detecting System Emulators, *ISC*, pp.1–18 (2007).

[25] Rekall, available from ⟨http://www.rekall-forensic.com/⟩ (accessed 2017-12-19).

[26] Sharif, M.I., Yegneswaran, V., Saidi, H., Porras, P.A. and Lee, W.: Eureka: A Framework for Enabling Static Malware Analysis., *ESORICS*, Jajodia, S. and Lopez, J. (Eds.), *Lecture Notes in Computer Science*, Vol.5283, pp.481–500, Springer (2008).

[27] Slowinska, A. and Bos, H.: Pointless tainting?: Evaluating the practicality of pointer tainting, *Proc. 4th ACM European Conference on Computer Systems, EuroSys '09*, pp.61–74, ACM (2009).

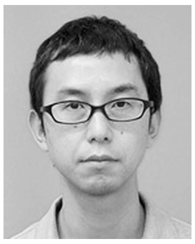
[28] Suenaga, M.: A Museum of API Obfuscation on Win32, *Symantec Security Response* (2009).

[29] Yason, M.V.: The Art of Unpacking, *Black Hat USA Briefings* (2007).

Editor's Recommendation

This paper shows malware obfuscation methods, the limits of existing methods for that, so has high material value. The proposed method considers practical use including cooperation with IDA Pro, thus this paper is selected as a recommended paper.

(Program Chair of Computer Security Symposium 2017 (CSS2017), Yuji Suga)



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

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

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