FAROS: Illuminating In-Memory Injection Attacks via Provenance-based Whole-System Dynamic Information Flow Tracking

Meisam Navaki Arefi†, Geoffrey Alexander†, Hooman Rokham†, Aokun Chen°,
Michalis Faloutsos⊕, Xuetao Wei*, Daniela Seabra Oliveira° and Jedidiah R. Crandall†
University of New Mexico† University of Cincinnati* University of California at Riverside⊕ University of Florida°
mnavaki@unm.edu, alexandg,hrokham,crandall@cs.unm.edu, chenaokun1990@ufl.edu,
michalis@cs.ucr.edu, weix2@ucmail.uc.edu, daniela@ece.ufl.edu

***Abstract*—In-memory injection attacks are extremely challenging to reverse engineer because they operate stealthily without leaving artifacts in the system or in any easily observable events from outside of a virtual machine. Because these attacks perform their actions in memory only, current malware analysis solutions cannot expose their behavior. This paper introduces FAROS[1], a reverse engineering tool for Windows malware analysis based on dynamic information flow tracking (DIFT), which can flag stealthy in-memory-only malware injection attacks by leveraging the synergy of: (i) whole-system taint analysis; (ii) per security policy-based handling of the challenge of indirect flows via the application of tags of different types, and (iii) the use of tags with fine-grained provenance information. We evaluated FAROS with six advanced in-memory-injecting malware and it flagged the attacks for all samples. We also analyzed FAROS' false positive rate with 90 non-injecting malware samples and 14 benign software from various categories. FAROS presented a very low false positive rate of 2%, which shows its potential towards practical solutions against advanced in-memory-only anti-reverse-engineering attacks.**

***Index Terms*—Dynamic Information Flow Tracking, In-memory Injection, Malware Analysis.**

I. Introduction

In-memory injection attacks are extremely challenging because they perform their malicious actions *in memory only* without leaving artifacts in the file system or in any easily observable event in the system or in a virtual machine (VM). The hallmark of this stealthy type of attack is the hiding (or injection) of malicious payloads inside a legitimate process address space.

Current malware analysis solutions (e.g., reverse engineering and memory snapshot forensic tools) are no match for such attacks because of the inability of these solutions to expose the attack in-memory-only behavior. Memory injection is a common benign behavior in Windows (e.g., used for debugging), and such techniques are easily evaded via the Windows OS API [1]. More specifically, event-based monitoring techniques do not look into system memory, where the obfuscation attack occurs.

Reverse engineering tools, such as the sandbox-based Cuckoobox [2], cannot expose in-memory-only attacks because these tools rely on easily observable VM events, such as system or library calls [3], file system activity, or specific library function calls. Memory snapshot forensics tools, such as Volatility [4] with the malfind plugin [5], assume that the Portable Executable format of a binary file will be intact and that important memory artifacts will not be destroyed. These solutions look at a snapshot of memory at one single point in time. In-memory injection attacks are typically transient, i.e., once the malicious payload is injected and executed, there is nothing stopping the attacker from cleaning up memory before the VM is stopped for a memory snapshot. Thus, while Volatility can give visibility into memory, it does so up to a certain point in time. An analyst needs visibility into memory *throughout* the execution of the sandboxed VM environment to flag transient in-memory attacks in a way that can be adapted to emerging and future threats.

In this paper, we propose FAROS, a reverse engineering tool for Windows malware analysis based on dynamic information flow tracking (DIFT), which can flag stealthy in-memory-only malware injection attacks. The key novelty of FAROS is the synergy of: (i) whole-system DIFT; (ii) a per security policy-based strategy to overcome the challenge of handling indirect flows via the application of tags of different types and using their unique confluence on a memory location as attack invariant, and (iii) the use of tags with fine-grained provenance information. In this context, the term fine-grained provenance means aggregate information about how netflow, process, and/or file activities are associated with a particular byte in memory, for example, whether and when a process accessed a particular byte or a byte was read/written into a file.

The first key feature of FAROS is the application of whole system DIFT [6] (also known as dynamic taint analysis), a promising technology for making systems transparent. DIFT tags certain inputs or data with meta information and then propagates tags as the program or system runs, usually at the instruction level, to achieve information flow transparency. FAROS taints objects at the byte level and performs DIFT at the instruction level.

The second key feature of FAROS is overcoming the challenge of handling indirect flows, inspired by the concept

[1]FAROS means lighthouse in Greek. In this paper, FAROS helps illuminate in-memory-only attacks.

of data-flow tomography [7]. Capturing indirect flows is one of the major challenges and impedances for the widespread usage of DIFT in security applications. To have a principled way for making decisions about whether or not to propagate tags for address or control dependencies, one would need to do a full analysis of a target program/system to consider, for example, code paths that do not happen, which moves away from the main attraction of DIFT: operation without access to source code. To address this challenge, FAROS addresses indirect flows in a per security policy fashion, via the application of tags of different types (e.g., *netflow*, *file*, *process/memory*) and using their unique confluence on a memory location as attack invariants. In particular, the hallmark of in-memory-only injection attacks is when a byte is associated with tags of type *netflow* (indicating that the byte originated in a particular network connection) and *export-table* (indicating that the byte is part of the OS kernel region where linking and loading operations occur).

The third key feature of FAROS is using tags containing rich provenance information. In contrast with current DIFT systems, FAROS' tags convey detailed information about the byte's lifetime in the system (e.g., chronology of all memory, file system, and network activities associated with the byte).

The synergy of these three features represents the novelty of our work and allows FAROS to flag stealthy in-memory-only injection attacks in a general way, that makes no assumptions about the format of data structures or the injected code's persistence in memory.

We implemented FAROS as a PANDA/QEMU plugin [8], with introspection for the Windows OS, and evaluated its effectiveness in flagging malicious behavior with six sophisticated malware, implementing the following types of stealthy in-memory-only attacks: (i) reflective DLL injection; (ii) process hollowing/replacement, and (iii) code/process injection. FAROS flagged in-memory-only malicious behavior for all samples. We also analyzed FAROS' false positive rate with 90 non-injecting malware samples and 14 benign software for different behaviors (e.g. download, upload, and remote desktop). FAROS produced a very low false positive rate of 2% for our dataset. FAROS' false positives can be dismissed/whitelisted by an analyst in a straighforward fashion, because they always involve well-known Just-In-Time compilers (e.g., Java).

FAROS is designed to operate as an off-line reverse engineering tool for malware analysis. Our performance evaluation showed that FAROS introduced a 56x performance slowdown compared to standard QEMU. FAROS is designed to plug into a reverse engineer's toolbox in the same way as CuckooBox [2]: a VM that an analyst can run in parallel with other tasks and that will report its findings back to him automatically. Thus, while short response time is desirable, it is not a priority. FAROS turns CPU cycles into actionable information about in-memory injection attacks that may be hiding important information from the reports generated by other automated tools. Thus, FAROS' goal is detecting a wide variety of attacks with a low false positive rate, and providing contextual information about processes, files, and network activities, rather than performance. FAROS also provides the reverse engineer with the full provenance of the injected code, saving him valuable analysis time. Without this information, finding where the injected code came from would be an arduous, time-consuming, and error-prone manual effort. In summary, this paper presents the following contributions:

1) FAROS, a DIFT-based reverse engineering tool for malware analysis, which can flag malicious activities that are happening in-memory-only, such as reflective DLL injection, process hollowing/replacement, and code/process injection.
2) An exploration of the limits of DIFT systems by empowering them with the synergy of whole-system taint analysis, per-security policy strategy for addressing the challenge of indirect flows, and the use of tags with rich provenance information.
3) FAROS' source-code as a PANDA plugin for the Windows 7 OS.[2]
4) FAROS evaluation, showing that it can effectively flag in-memory-only malicious behaviors for all test samples, while achieving a very low false positive rate, 2%.

The paper is organized as follows. Section II presents FAROS' threat model. Section III provides background information on DIFT. Section IV explains how FAROS overcome the challenges of handling indirect flows. Section V discusses FAROS architectural design and implementation details. Section VI presents FAROS' experimental evaluation regarding effectiveness in flagging in-memory injection attacks, analysis of false positives, and performance overhead. Section VII discusses related work and section VIII concludes this paper.

II. Threat Model

FAROS is a whole-system DIFT-based reverse engineering tool for Windows malware analysis capable of *flagging* stealthy in-memory injection attacks. In this paper, we focus on three types of such attacks: (i) reflective DLL injection [9]; (ii) process hollowing/replacement [10], and (iii) code/process injection [11]. The purpose of these types of in-memory injection attacks is to write the malware directly into the memory of the victim's machine, instead of writing the malware into the hard drive, where it can be detected by anti-viruses or file-system monitoring tools.

Today's malware analysis solutions, such as CuckooBox [2] leverage system call information, file system activities, specific library functions, and functionalities that are externally visible in a VM to gather information about malware. However, advanced in-memory injection attacks and APT (Advanced Persistent Threats) techniques, such as reflective DLL injection, process hollowing/replacement, and code/process injection, operate in a stealthy mode and without leaving their footprints, by performing their malicious actions in-memory only, without invoking system calls, library calls or performing file I/O.

[2]*https://github.com/mnavaki/FAROS*

In-memory injection attacks are usually implemented by using a small loader that unpacks/downloads the in-memory payload and injects it in the process memory. During the attack, the loader usually has the appropriate level of privilege on the system to be able to perform injection into other programs' memory. After the injection, the loader is commonly deleted from the system to prevent its detection. In contrast to traditional malware, which tends to leave artifacts in the file system, these type of attacks have only an in-memory fingerprint hiding themselves inside a benign process. This stealthiness makes detection by traditional anti-malware approaches (signature and machine learning-based) extremely hard. FAROS can shed light on activities happening in-memory only, thus allowing an analyst to better understand and automatically *flag* such attacks. The advanced in-memory injection attacks considered in this paper are detailed below.

Reflective DLL injection. In these attacks, the reflective programming technique (which allows a program to examine and modify its own structure and behavior at runtime) is employed to load a malicious DLL from memory into the target process. In this scenario, the DLL should be loaded from memory rather than from disk. Since Windows does not provide such loading function, a separate loader is required to load the DLL into the target process. This leads to a bypass in the procedure of registering the DLL with a process and on the operation of DLL loading monitoring tools. The DLL parses the host processes kernels export table to calculate the addresses of three functions required by the loader, namely LoadLibraryA, GetProcAddress and VirtualAlloc, to load itself.

Process hollowing/replacement. In this technique, the malware starts a benign process in a suspended state and replaces the process address space contents with a malicious payload. The process is then resumed and the entry point of the new image is executed. More specifically, the malware forks a benign child process in a suspended state, unmaps and overwrites the memory of the child process, and resolves imports and exports for the new code. The attack is stealthy because the exploit code hides behind a legitimate process, bypassing process monitoring tools.

Code/Process injection. Malware leverages code injection to force another process to perform actions on its behalf, such as downloading another malware or stealing information from the system. Code injection is accomplished by writing directly into a remote target process's memory or making the new process load a malicious DLL [12]. After the code injection, the malicious process may obtain access to the target process's memory, elevated privileges, and system resources, while remaining undetected.

III. Background - DIFT

Dynamic information flow tracking (DIFT [6], also called dynamic taint analysis) is a promising technology for making systems transparent. DIFT works by tagging certain inputs or data and then propagating tags as the program or system runs, so that something can be learned about the flow of information.

```
const char str1 = "Tainted string";
char str2[80];
char lookuptable[256];
for (i = 0; i < 256; i++)
    lookuptable[i] = i;
for (j = 0; j < strlen(str1); j++)
    str2[j] = lookuptable[str1[j]];
```

Fig. 1: An example of address dependencies in C code.

There are two main types of flows that DIFT systems can track [6]: direct and indirect. Direct flows are data-flow based, while indirect flows are control-flow based. For example, in $x = y + 1$, there is a direct flow from y to x. However, in code $x = 0;\ if\ (y == 1)\ x = 1;$, the value of x is dependent of y, meaning that there is an indirect flow from y to x.

There are two types of direct flows: copy and computation dependencies. In a copy dependency, a value is copied from one location to another, where a location can be a byte, a word of memory or a CPU register. The straightforward way to track this information flow is to propagate the tag so that the destination location is tainted with the same tag as the source location. In computation dependencies, tags must be combined. For example, after the computation for $x = y + z$, the tag for x should contain the union of the tags for y and z; or, in single-bit tag DIFT systems, x should be tainted if either y or z is tainted.

An indirect flow occurs when information dependent on program input determines from where and to where information flows. There are two types of indirect flows: *address* and *control dependencies*. Figure 1 provides an example of address dependencies. In this figure, if the characters in `str1` are tainted, then the characters in `str2` at the end should also be tainted, because they carry the exact same information. This example may seem contrived at first, but versions of it occur in many situations from special handling of ASCII control characters to ASN.1 encodings. The only way to ensure that `str2` is properly tainted is to propagate tags through the address dependency where `str1[j]` is used as an offset to index into `lookuptable`.

Control dependencies, illustrated in Figure 2, present a dilemma for all existing DIFT systems. In Figure 2 `taintedinput` is copied to `untaintedoutput` bit by bit. Information flows one bit at a time through the control dependency in the `if` statement.

IV. Overcoming the Challenge of Indirect Flows

One challenge with existing DIFT systems is that they cannot track indirect flows effectively. Should all address and control dependencies lead to tag propagation? Not propagating tags in these cases leads to undertainting, where important information flows are missed. Propagating tags for all address and control dependencies leads to overtainting, where quickly

```
char taintedinput;
char untaintedoutput = 0;
for (bit = 1; bit < 256; bit <<= 1){
    if (bit & taintedinput)
        untainedoutput |= bit;
}
```

Fig. 2: An example of control dependencies in C code.

every piece of data in the system is tagged with every tag because the tagging system is too conservative. In either case, very little is learned about the actual information flow of the system. Thus, the handling of indirect flows forces DIFT systems to choose between undertainting or overtainting.

The example of indirect flow illustrated in Figure 1 represents types of information flows that occur in real systems all the time. In modern systems, operations such as indexed data structures, compression and decompression, encryption and decryption, hashing, switch statements, encodings, and string manipulations are the rule, not the exception, and all of the aforementioned operations involve address and control dependencies.

Capturing indirect flows is one of the major impedances for the widespread usage of DIFT in security applications, including reverse engineering applications. On the surface, DIFT is a perfect technology for reverse engineering: it gives visibility into how information flows in a live system. However, because of the challenge of addressing indirect flows, DIFT systems are not accurate enough for reverse engineers to get usable results.

To have a principled way for making decisions about whether or not to propagate tags for address or control dependencies, one would need to do a full analysis of a target program/system to discover, for example, the range an offset used to index into a lookup table could take or how commonly a branch is taken vs. not taken. In other words, one would have to consider code paths that do not happen, which moves away from the main attraction of DIFT: operation without access to source code.

In this paper, we overcome the challenge of capturing indirect flows in a per security policy fashion and inspired by the concept of dataflow tomography introduced by Mazloom et al. [7]. The idea is as follows. The DIFT system support tags of different types, for example, *netflow* (the byte came from a network source), *string* (the byte is known to be part of an ASCII or UTF8 string based on context), *pointer* (the byte represents a pointer), and *export-table* (the byte came from the kernel area where linking and loading operations occur). Two or more tags of different types can "come together" (like a river confluence) in a certain memory address as DIFT instructions are executed. In other words, when a byte tag contains at least two tags of different types, we say that these tags came together (tag confluence). For example, a byte can enter the system and be associated with a tag of type *netflow*. Then, if this byte is written into a memory area corresponding to kernel areas where linking/loading occur, then the byte

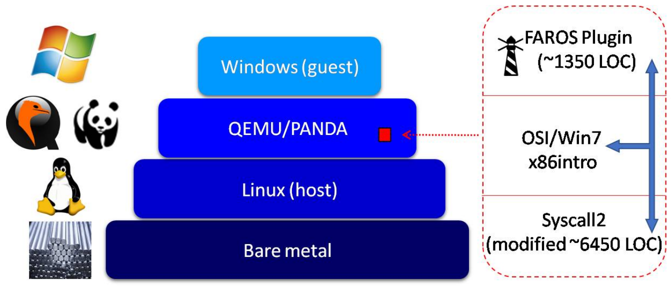

Fig. 3: FAROS architecture.

becomes associated with two different types of tags: *netflow* and *export-table*.

FAROS leverages the unique properties of tag confluence in a given memory location as attack invariant. In particular, the hallmark of in-memory-only attacks from a DIFT perspective is the association of tags of type *netflow* and *export-table* to the same byte in memory. We say these tags "came together" – indicating that data from the network is being executed as part of a linking/loading process.

V. FAROS Design and Implementation

FAROS was implemented as a plugin to PANDA [8], which is an open-source platform for dynamic analysis. PANDA is built upon the QEMU whole-system emulator [13] and adds to it three main features: (i) the ability to record and replay executions to enable whole-system analyses; (ii) an architecture to streamline the addition of other plugins to QEMU, and (iii) the ability to use LLVM [14] as the QEMU's intermediate language instead of TCG [15] for analysis.

Figure 3 shows an overview of FAROS architecture. The FAROS/PANDA plugin runs on top of Linux OS Ubuntu 14.04 (host OS) and supports Windows 7 as the guest OS.

FAROS interacts with two other plugins, namely *syscalls2* and *OSI/Win7x86intro*. The syscalls2 plugin provides callbacks that allow notifications whenever system calls are invoked in the guest. We have modified the syscalls2 plugin to get the system calls arguments and follow their pointer arguments. The OSI plugin provides callbacks whenever a process-related event happens in the system, such as start/end of a new process. It also provides APIs to get the current process information.

A. FAROS Provenance and Taint Tracking System

Compared to basic DIFT (1-bit tags), provenance tracking can provide much richer information about what is happening in the system. Provenance can capture various types of information about a byte, such as (i) what the origin of a byte is; (ii) which bytes are affected by another byte, and (iii) what the life cycle of a byte looks like (e.g. came from a network source, then was copied to the address space of a process, then was written to a file, whose data was consumed by another process, etc.). In this context, a byte could come from different sources such as network, file system, and memory.

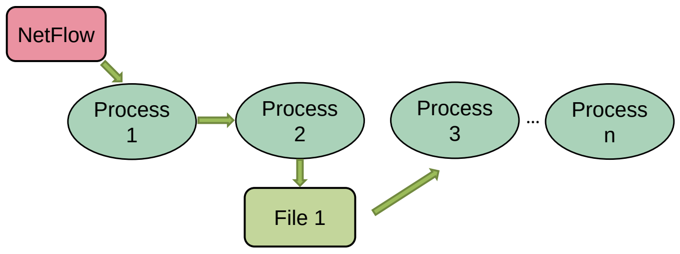

Fig. 4: An example of a provenance list for a particular byte.

FAROS uses provenance information to flag an in-memory injection attack and to give an analyst additional information about how the attack was conducted or where it was originated.

FAROS combines DIFT with provenance in a DIFT-provenance system, whose tags track the source and the history of a given byte. To implement provenance, FAROS introduces different tags for three basic types of system activities: network (*netflow* tag), process/memory (*process* tag), and file-system (*file* tag). FAROS also introduces an (*export-table*) tag to represent bytes coming from the memory region where the OS keeps its export table.

FAROS maintains a provenance list for each byte in the system. The provenance list for a byte is a combination of *process*, *netflow*, *file*, and/or *export-table* tags. A *process* tag is a CR3 value, which uniquely identifies a process at the architecture level. A *netflow* tag is a data structure containing source and destination IP addresses and port numbers. A *file* tag has a file name and a version that indicates how many times a file has been accessed.

Figure 4 shows the high-level view of a provenance list for a particular memory address. In this list, data comes in from network and goes to Process 1. Next, it goes to Process 2, and then it is written into File 1, which is read by Process 3.

Tag Insertion. For network activity, once a network packet comes in, FAROS constructs a *netflow* tag and taints every byte of that packet with the created *netflow* tag. The complete tracking of this tag type requires whole-system taint analysis (tainting with tags and propagating them as the CPU operates on the tainted data) including inside the kernel. For process/memory activity, if a process accesses a byte in memory, FAROS adds a *process* tag into the head of that byte's provenance list. For file system activity, when a file gets loaded into memory, FAROS taints the file contents with a *file* tag. Further, when a buffer is written into a file, FAROS taints the buffer with a *file* tag. FAROS' tag insertion mechanism is implemented at the OS level via a driver hooking into corresponding Windows system calls. For example, for *file* tag insertion, FAROS leverage 26 filesystem-related system calls (e.g. `NtFileRead()` and `NtFileWrite()`) to get the file names and buffers. Finally, for export table read activity, FAROS scans all loaded modules and taints the function pointers in the export tables with the *export-table* tag.

Tag propagation. After a basic block (i.e. a list of instructions terminated by a branch instruction) is executed in the guest OS, FAROS gets a list of CPU instructions for that basic block. It then processes these instructions and propagates the taint information into a shadow memory and a shadow register according to DIFT rules for propagation of tags (explained below). Table I shows our propagation policy. Of note, an address can be a byte in memory or a register. The details of each operation are as follows:

Taint operation	Taint propagation policy
$copy(a,b)$	$prov(a) \longleftarrow prov(b)$
$union(a,b)$	$prov(c) \longleftarrow prov(a) \cup prov(b)$
$delete(a)$	$prov(a) \longleftarrow \emptyset$

TABLE I: FAROS propagation rules. $prov$ indicates the provenance list associated with an address.

- **copy(a,b):** Copy the provenance list associated with address *b* to address *a*. Copy operations happen in instructions such as MOV, STR, and LD.
- **union(a, b, c):** Assign the union of provenance lists associated with address *b* and address *c* to address *a*. Union operations happen in instructions such as AND, OR and MUL.
- **delete(a):** Delete the provenance list associated with address *a*. Delete operations happen in instructions such as MOVI, or XOR when the operands are the same.

Data Structures. FAROS maintains three hash maps, one for each tag types *netflow*, *process*, and *file* (Figure 5). All entries in these hash maps have been involved in activities related to tainted bytes. The *Process* hash map contains the processes currently captured by taint analysis. The value field represents the value of the CR3 register (which uniquely identifies a process at the architecture level) and the key represents an index. The *Netflow* hash map contains the *netflow* tags currently captured by taint analysis. The value field is a netflow tag and the key is an index. The *File* hash map contains the files currently captured by taint analysis. The value is a *file* tag and the key is an index. FAROS current implementation does not incorporate a hash map for export table activity because its corresponding tag does not contain additional information (besides the tag itself) as the other tag types do. For example, a *netflow* tag contains additional information about IP addresses and port numbers. The current implementation of FAROS just required information about the existence of *export-table* tags to flag in-memory-only attacks. As future work and to further aid an analyst, we plan to augment this tag with information about function name, which will require the addition of a corresponding hash map.

FAROS uses three bytes to represent a tag data structure, which we call *prov_tag*. The first byte indicates the tag type (i.e. *netflow*, *process*, *file*, or *export-table*), and the next two bytes indicate the tag index in the corresponding hash map as described above (see Figure 6). FAROS maintains all provenance tags in main memory for fast access.

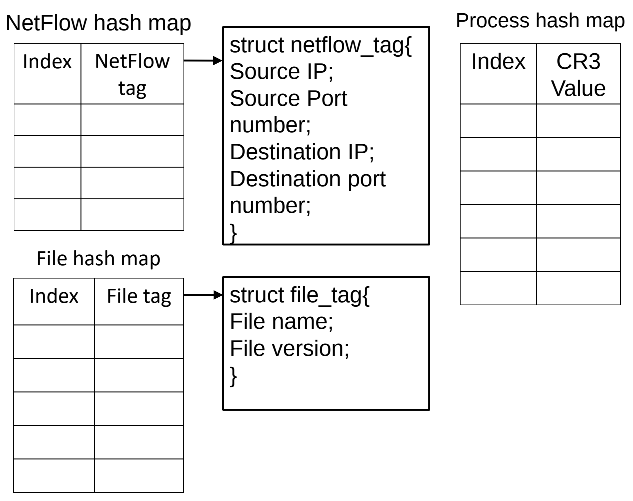

Fig. 5: Structure of *Process*, *Netflow*, and *File* hash maps.

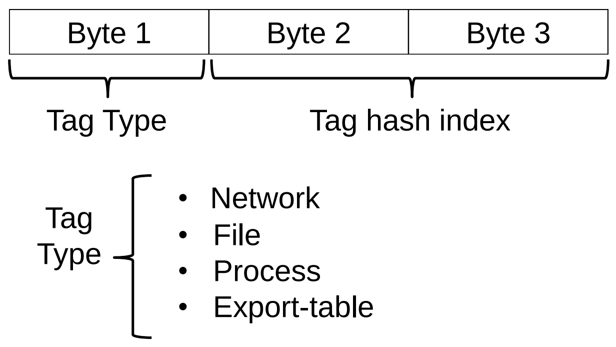

Fig. 6: Structure of prov_tag.

FAROS also adds two hash maps to keep a shadow memory and a shadow register bank. The shadow memory stores provenance information associated with the corresponding memory address in the actual memory. Similarly, the shadow register bank stores provenance information associated with actual CPU registers. In both shadow memory and shadow register bank, a hash map entry contains a provenance list.

B. *Flagging in-memory injection attacks*

As explained in Section IV the hallmark of in-memory-only attacks is the colliding of *netflow* and *export-table* tags. Of note, as FAROS is a reverse engineering tool, the analyst already knows which process is malicious and which process is the victim, and his goal is to detect and analyze the attack. The flagging criteria is as follows. All in-memory injection attacks have a common information flow: a flow coming from the network goes into a client process and that process injects the malicious payload into a target process memory. Then, the injected code tries to resolve imports/exports and gets executed. For example, consider a victim machine running malware X, which entered the system via a phishing attack. Then malware X downloads a malicious payload and injects it into a benign process Y, to make the attack harder to detect. Then, the injected malicious payload resolves its imports/exports and then gets executed within process Y. Note that to resolve the imports and exports, this malicious payload needs to read the export table.

As previously mentioned, FAROS tags the bytes corresponding to the export table with an *export-table* tag, so that any readings of the export table can be monitored. Moreover, FAROS monitors all load/mov instructions to check whether they read the export table. Thus, whenever FAROS processes a load/mov instruction that has a provenance list containing a *netflow* tag, a *process* tag corresponding to the malicious process (CR3), and a *process* tag corresponding to the target process (CR3), and this instruction attempts to read the export table (i.e. reads a memory location associated with an *export-table* tag), FAROS flags this activity as an in-memory injection attack. In other words, FAROS considers this instruction as part of the DLL/process/code that has been copied into the target process. In addition, FAROS outputs the address of this instruction along with its provenance info, as shown in Figure II.

One might argue that FAROS does not need to keep the whole provenance information for the flagging criteria. However, FAROS' design goal is to give reverse engineers enough information so that they can start their analysis, in case an in-memory injection attack has been detected, and save the analyst hours of reverse engineering effort. FAROS is not a malware detector, but rather a reverse engineering tool, whose main goal is to help an analyst to better understand in-memory injection attacks. By providing provenance information, an analyst can answer the following questions: (i) what the life cycle of the malicious process looks like?; and (ii) where does it come from? Such information can only be obtained via the provenance lists.

Further, one may wonder whether there are other ways to get a `libc` function pointer that does not rely on the export table. In practice, any pointers in a process's memory that would lead to a desired system service will likely have been derived in some way from the kernel's export tables that are mapped into the process's address space.

These insights show that defining attacks in terms of information flows allows analysis tools to better adapt their operations against future and more sophisticated malware and exploits.

C. *Usage scenario*

To use FAROS, an analyst needs to set up a Windows 7 VM, start PANDA in recording mode (to enable instruction emulation), and then run the malware he wants to analyze along with any other applications or activities that he is interested in observing inside the VM. Once the interesting activities are completed, the analyst stops the PANDA recording mode and initiates the PANDA replay of the recorded capture with the FAROS plugin loaded and performing taint analysis. FAROS will generate an output file indicating whether there are any in-memory injection attacks. If such an attack has been captured, FAROS provides the memory addresses of the instructions that were captured as part of the malicious injected payload, along

Memory Address	Provenance List
`0x83B07019`	NetFlow: {src ip,port: 169.254.26.161:4444, dest ip.port: 169.254.57.168:49162} ->Process: inject_client.exe ->Process: notepad.exe;
`0x83B07018`	NetFlow: {src ip,port: 169.254.26.161:4444, dest ip.port: 169.254.57.168:49162} ->Process: inject_client.exe ->Process: notepad.exe;
`0x83B07017`	NetFlow: {src ip,port: 169.254.26.161:4444, dest ip.port: 169.254.57.168:49162} ->Process: inject_client.exe ->Process: notepad.exe;
`0x83B07016`	NetFlow: {src ip,port: 169.254.26.161:4444, dest ip.port: 169.254.57.168:49162} ->Process: inject_client.exe ->Process: notepad.exe;
`0x83B07006`	NetFlow: {src ip,port: 169.254.26.161:4444, dest ip.port: 169.254.57.168:49162} ->Process: inject_client.exe ->Process: notepad.exe;
`0x83B07005`	NetFlow: {src ip,port: 169.254.26.161:4444, dest ip.port: 169.254.57.168:49162} ->Process: inject_client.exe ->Process: notepad.exe;

TABLE II: An example of FAROS output for a particular in-memory injection attack.

with the provenance list associated with each one of these memory addresses (see Table II where the provided memory addresses indicate the address of *mov* instructions that have read the export table).

VI. Experimental Evaluation

FAROS' experimental evaluation considered three types of advanced in-memory attacks, as described below.

Reflective DLL injection. We performed three reflective DLL injection experiments using three different modules in Metasploit [16]: *reflective_dll_inject*, *reverse_tcp_dns*, and *bypassuac_injection*. FAROS flagged these attacks successfully. The experiment set-up and results for the attacks are as follows.

1) *reflective_dll_inject:* We used the Meterpreter module in Metasploit. We set up two VMs (victim and attacker), and put them in the same network. Then, we set up Metasploit on the attacker machine, and generated a Meterpreter shell code using Kali Linux [17] to run on the victim machine. We ran the shell code (*inject_client.exe*), in the victim machine to open a session for the attacker, and then performed a remote reflective DLL injection targeting *notepad.exe* from the attacker machine using Metasploit. The injected DLL only showed a pop-up message from the target process, representing a successful injection.
Figure 7 shows the output of our system for this experiment. A *mov* instruction has a provenance list that shows this instruction is coming from the network, then going to *inject_client.exe* and next to the target process *notepad.exe*. Also, this instruction is reading the memory address of 7fff0960 which is already tainted as export-table. These two types of provenance information coming together (netflow and export-table) represented a true positive in-memory injection attack.
2) *reverse_tcp_dns:* The environment setup was the same as the one used in the previous experiment. However, in this experiment the shell code and the target process were the same. Figure 8 depicts the result of this experiment, where we can observe the same flow of information as shown in the previous example – *netflow* and *export-table* tags coming together at the same memory location, thus enabling FAROS to flag the tag confluence as an in-memory injection attack.

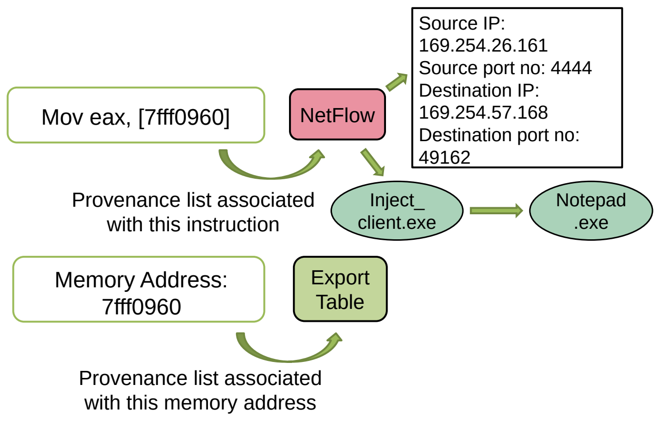
Mov eax, [7fff0960]
NetFlow
Source IP: 169.254.26.161
Source port no: 4444
Destination IP: 169.254.57.168
Destination port no: 49162
Provenance list associated with this instruction
Inject_ client.exe
Notepad .exe
Memory Address: 7fff0960
Export Table
Provenance list associated with this memory address

Fig. 7: Provenance tracking for reflective DLL injection using the Meterpreter module.

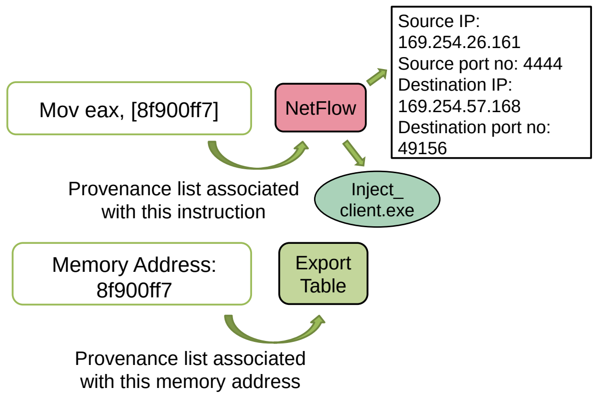
Mov eax, [8f900ff7]
NetFlow
Source IP: 169.254.26.161
Source port no: 4444
Destination IP: 169.254.57.168
Destination port no: 49156
Provenance list associated with this instruction
Inject_ client.exe
Memory Address: 8f900ff7
Export Table
Provenance list associated with this memory address

Fig. 8: Provenance tracking for reflective DLL injection using the *reverse_tcp_dns* module.

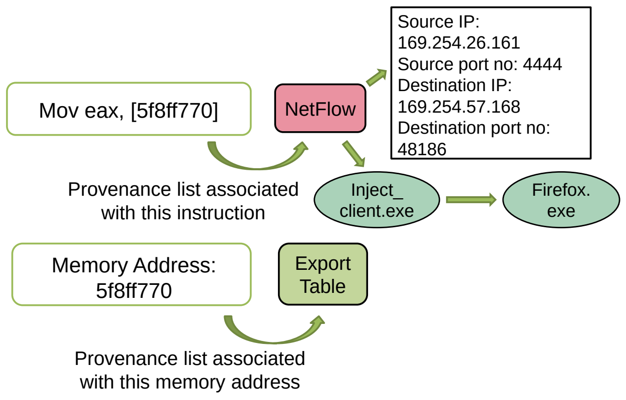

Fig. 9: Provenance tracking for reflective DLL injection using the *bypassuac_injection* module.

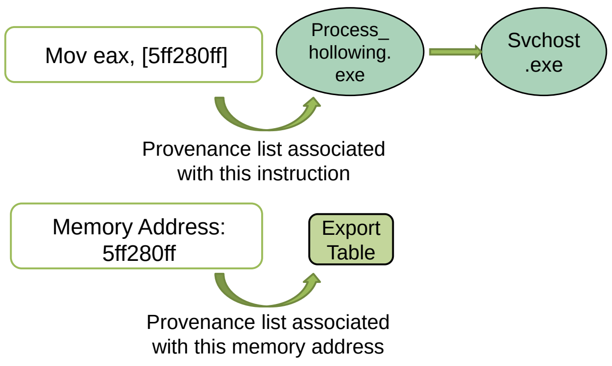
Mov eax, [5ff280ff]
Process_ hollowing. exe
Svchost .exe
Provenance list associated with this instruction
Memory Address: 5ff280ff
Export Table
Provenance list associated with this memory address

Fig. 10: Provenance tracking for process hollowing/replacement.

3) *bypassuac_injection:* Similarly, we performed another reflective DLL injection using the *bypassuac_injection* module in Metasploit. The environment setup was the same as in the previous experiments, however, the target process was *firefox.exe*. Figure 9 represents the result of this experiment. The results are similar to the previous experiments.

Process hollowing/replacement. Our simulated attack used a malware sample from Lab 3-3 [11], which performs process replacement on *svchost.exe* to launch a keylogger. FAROS successfully flagged the malware as memory injecting. Figure 10 shows the provenance list captured for this experiment. This list shows the flow of data from *process_hollowing.exe* to *svchost.exe*, and to the export table. In this list *svchost.exe* is a child of *process_hollowing.exe*.

Java Applets	AJAX websites
acceleration	gmail.com
equilibrium	maps.google.com
pulleysystem	kayak.com
projectile	netflix.com/top100
ncradle	kiko.com
keplerlaw1	backpackit.com
inclplane	sudokucarving.com
lever	pressdisplay.com
keplerlaw2	rpad.com
collision	brainking.com

TABLE III: Java applets selected from http://www.walter-fendt.de/ph14e/ and AJAX websites used in FAROS false positive evaluation.

Code/Process injection. We used two real-world code-injecting malware in this evaluation: DarkComet and Njrat (for remote shell behavior). DarkComet is one of the most popular Remote Administration Tools (RAT) in use today. This RAT allows a user to control the system with a Graphical User Interface (GUI). Njrat allows attackers to hack and steal information from the victim machine. FAROS successfully flagged the malware as in-memory injecting. Results for the provenance list for these malware samples were similar to those for the reflective DLL injection experiment.

A. False positive analysis

Languages that use JIT compilation (e.g., Java and .NET) can be a source of false positive for FAROS. To analyze FAROS' false positive rate we tested it with 20 different Java applets and AJAX websites. Table III shows the list of the Java applets and websites we evaluated. FAROS flagged only two of the Java applets (10%) as memory injecting. Java applets operate similarly to memory injection attacks: the system receives data over the network, which is linked and loaded with export tables. However, JITs software (e.g., JAVA and AJAX) is relatively uncommon and can be white-listed by an analyst.

In addition, we evaluated FAROS' false positive rate with 102 non-in-memory injecting malware samples and benign software from different categories. FAROS produced a 0% false positive rate. Table IV shows the list of malware samples and benign software used in this evaluation along with their associated behavior.

B. Comparison with CuckooBox

As FAROS focuses on flagging in-memory injection attacks, we compared it with the most popular open source analysis tool in this category. *Cuckoo sandbox* is an open-source malware analysis system, which retrieves: (i) traces of system and function calls made by the malware; (ii) files created and modified by the malware; (iii) a memory dump of the malware; (iv) network traffic traces; (v) screen shots, and (vi) a full machine memory dump. It is also an expandable system compatible with plugins such as Volatility and malfind.

Behavior / Program	Idle Run	Audio Record	File Transfer	Key logger	Remote Desktop	Upload	Download	Remote Shell
Real-world malware								
Pandora v2.2	✓	✓		✓	✓	✓	✓	✓
Darkcomet v5.3	✓	✓		✓	✓	✓	✓	
Njrat v0.7	✓	✓		✓	✓	✓	✓	
Spygate v3.2	✓	✓		✓	✓	✓	✓	✓
Blue Banana	✓				✓	✓		✓
Blue Banana v2.0	✓				✓	✓		✓
Blue Banana v3.0	✓				✓	✓		✓
Bozok	✓			✓	✓	✓	✓	✓
Bozok v2.0	✓			✓	✓	✓	✓	✓
Bozok v3.0	✓			✓	✓	✓	✓	✓
DarkComet v5.1.2	✓	✓		✓	✓	✓	✓	
DarkComet legacy	✓	✓		✓	✓	✓	✓	
Extremerat v2.7.1	✓	✓		✓	✓	✓	✓	✓
Jspy	✓			✓	✓			✓
Jspy v2.0	✓			✓	✓			✓
Jspy v3.0	✓			✓	✓			✓
Quasar v1.0				✓	✓			✓
Benign software								
Remote Utility	✓	✓	✓		✓			✓
TeamViewer	✓		✓		✓			
Win7-snipping tool	✓		✓		✓			
Skype	✓	✓	✓					

TABLE IV: FAROS' false positive analysis dataset: Malware samples (non-in-memory injecting) and benign software along with their behaviors. Each checkmark identifies a malware/benign sample behavior.

To analyze a Cuckoo sandbox memory dump, we used the Volatility plugin. Similarly to the tests we performed with FAROS, we tested Cuckoo sandbox with reflective DLL injection, process hollowing/replacement, and process/code injection. For these experiments, we ran Cuckoo sandbox 2.0.0 and Volatility 2.6 on Ubuntu 14.04. The guest VM for Cuckoo sandbox was a 32-bit Windows 7 with 2 GB of RAM. We analyzed Cuckoo sandbox with and without *malfind* plugin.

Reflective DLL injection. We analyzed reflective DLL injection attack via Cuckoo sandbox by observing a reflective injector injecting a benign DLL into *Explorer.exe*. Without the malfind plugin, we failed to identify a trace of our DLL under the DLL list either under the injector or the victim process which in this case was *Explorer.exe*. Although we could find that both *Explorer.exe* and the DLL were referred to separately by the injector in the injector trace, CuckooBox could not flag the attack. Using the malfind plugin to analyze the victim process, multiple reflective DLL injection memory segments were found, which allowed the attack to be detected. However, CuckooBox could not link the attack directly to the DLL or the process injector. Furthermore, in this experiment, CuckooBox could not trace the provenance of the injected DLL back to a *netflow* record.

Process hollowing/replacement. Similarly to the reflective DLL injection experiment, we setup Cuckoo sandbox to analyze process hollowing as it injected a Hello World program into *svchost*. We tried to analyze the attack using the Volatility *pslist* command to list all the processes, but we could not find the Hello World process in the process list. However, having used *vadinfo* to examine each *svchost*, we discovered that one of them was different from the rest, which allowed attack detection. However, an analyst would still not know anything about how the attack was conducted and he would not have any provenance information related to the attack.

Code/Process Injection. For code/process injection, we used Cuckoo sandbox to analyze a bot program from a RAT, while running the RAT server. We successfully found the process in the memory dump list, and also with the trace of the privilege of the bot program, we discovered that it had unusual privileges. Using these insights along with the package capture, we could identify it as a RAT. But again, an analyst would be blind about how the attack was conducted.

In sum, even though Cuckoo sandbox can flag all attacks with the malfind plugin, compared to FAROS, malfind does not provide the following forensics information about the attack: netflow, memory addresses, and full provenance history. As malfind does not necessarily target in-memory-only attacks, its underlying assumptions can be easily violated by such attacks. More specifically, malfind assumes that the injected memory has a certain structure and that this structure persists for a long time to be captured in a memory dump. Lastly, the malfind plugin provides the data that an analyst needs to find an in-memory injection attack, but does not detect all types of in-memory attacks automatically. FAROS gives visibility in a way that no other tool does: into memory throughout the execution

Application	Replay time w/o FAROS (second)	Replay time w/ FAROS (second)	X overhead
Skype	69	1260	18.2
Team Viewer	25	322	12.8
Bozok	7	50	7.1
Spygate	30	420	14
Pandora	4	28	7
Remote Utility	67	1320	19.7

TABLE V: Performance evaluation: PANDA vs PANDA + FAROS

of a sandboxed VM environment. Thus, while it may be possible to evade FAROS' specific policy for detecting in-memory injection attacks, it will in turn be possible to update the policy and even to do so proactively because of FAROS' flexibility. ROP (Return Oriented Programming) chains, techniques that search for functions in memory to avoid tainted library linking pointers, or methods to launder taint marks are examples of evasion techniques that can be addressed by the policy given to FAROS, because fundamentally they all happen in memory and are based on information flow, which is exactly what FAROS is designed to give visibility into. Because FAROS is based on whole system taint analysis and generates rich provenance tags, it has large flexibility in terms of what to taint and how to propagate it, proving itself a general tool for reverse engineering that can be adapted to emerging and future attack techniques.

C. *Performance Evaluation*

Whole-system DIFT is intrinsically heavy-weight, and thus, performance is not a priority for FAROS. Instead, FAROS' design and implementation focused on providing a low false positive rate. In spite of that, we have analyzed the performance overhead of FAROS to PANDA.

All experiments were done on a system with an Intel Core i7-6700K 4.00GHz processor, and 32G RAM running on Ubuntu 14.04. The guest was Windows 7 Ultimate 32-bit with 4GB memory. Table V shows the slowdown of our system for seven random samples of software. FAROS exhibited a 14x slowdown on average compared to PANDA replay. In addition, PANDA replay is almost 4x slower than standard QEMU [8], which makes FAROS' slowdown a 14*4 = 56x compared to QEMU. Moreover, table V illustrates that FAROS' performance overhead depends on the workload. PANDA recordings with more complex behavior have more performance overhead.

D. *Discussion and Limitations*

Any automated malware analysis tool is vulnerable to evasion by an attacker who has knowledge of the tool's technical details and has the time, resources, and motivation to evade it. For example, FAROS maintains a large amount of provenance information in the form of provenance tags. An evasion technique could leverage this design to exhaust FAROS' memory.

Another example is the execution of code specifically designed to evade FAROS' information flow tracking logic (e.g., by using control flow dependencies), by generating a great amount of tagged data, and by increasing FAROS' performance/memory overhead to a level higher than the guest system can handle. For example, because we do not directly track control dependencies, a dedicated attack could copy data bit-by-bit using an `if` statement in a `for` loop that simply sets zero-initialized bits to 1 only if they were 1 in the input. The output produced by such a loop would be identical to the input but would be untainted.

Further, an important caveat about FAROS' current implementation, is that FAROS currently requires the analyst to identify the process to be analyzed. Reverse engineers typically have a small set of processes of interest, but we still plan to address this issue via performance improvements in future work so that FAROS can analyze the whole system at once in only one deterministic replay.

VII. Related Work

Our work intersects the areas of reverse engineering, malware analysis, and DIFT. This section summarizes the state-of-the-art in these areas, and highlights under-studied areas.

A. *Reverse Engineering and Malware Analysis*

Dynamic analysis tools, such as CWSandbox [18], Norman Sandbox [19], Cuckoo Sandbox (namely, CuckooBox) [20], and Anubis [21], were proposed for reverse engineers to analyze malware. These tools typically execute malware in a controlled environment via sandbox techniques [22] and monitor malware's behavior via extracting system calls or API function calls. However, these tools make assumptions about the memory layout used by the attack (e.g., the relevant memory will be persistent) and do not provide full provenance information about memory. Although CWSandbox hooks the APIs related to the DLL library [23], it does not provide the meaningful and comprehensive provenance information provided by FAROS. The Volatility plugin for Cuckoo Sandbox is also specific to only reflective DLL loading, and cannot flag other types of in-memory injection attacks. To streamline dynamic malware analysis, some tools adopt emulation and hardware-aided visualization techniques. For example, Panorama [24] and VMscope [25] were proposed as whole system QEMU-based malware analysis systems [13]. Ether performs transparent malware analysis leveraging hardware virtualization [26]. B. Dolan-Gavitt et. al. developed a QEMU plugin (PANDA) to automate vulnerability detection [27]. Bacs et al. [28] proposed a semi-automated approach to recover infected systems via DIFT. None of these aforementioned DIFT-based tools were designed to detect advanced in-memory injection attacks, and are, therefore complementary to FAROS.

B. *DIFT*

Most past works attempt to resolve the undertainting *vs.* overtainting dilemma, rather than re-gaining accuracy lost by the absence of handling indirect flows, such as FAROS does. For example, in Suh et al. [6] address dependencies are not propagated if the address is calculated using a scaled index base (an x86 construct for calculating addresses). It is debatable whether this is effective for even the simple application considered in that paper of tainting network inputs and checking control data transfers, and this heuristic is very unlikely to be effective for other uses of DIFT. A scaled index base is necessary for looking up 32-bit pointers (by shifting the offset left by 2), but if the data of interest to DIFT is strings or pixels, for examples, the heuristic does not apply.

In the Minos system [29], [30], address dependencies are only propagated for 8- and 16-bit loads and stores, but not for 32-bit loads and stores. Additionally, as an attempt to mitigate control dependencies, 8- and 16-bit immediate values (i.e., constants that are compiled into the program's machine code) were tainted automatically even if the code did not come from a tainted source. Without these heuristics, there were several attacks the authors tested that Minos would not have been able to catch. Even with these heuristics, an existing attack at the time (CVE-2003-0818, which was not designed to evade Minos specifically) was not detected by Minos at the control flow transfer because the exploit code caused a heap link operation, involving 32-bit address dependencies, to link a heap object into the control flow of the program without using any base pointers that came from the tainted input [30].

More recent DIFT systems that are designed for flexibility [31]–[33] enable address and/or control dependencies to be tracked, but provide no satisfactory method for doing so in practice. Panorama [24], for instance, relies on a human to manually label which address and control dependencies tags should be propagated. Other systems such as DTA++ [34] or DYTAN [35] rely on off-line analysis, which does not scale to full systems.

Systems designed with correctness as the primary goal, such as RIFLE [36] or GLIFT [37], propagate all tags all the time unless a compiler statically analyzes the information flow and deems a particular operation safe. Fenton's data mark machine [38] suffers from the same problem. Specifically, Fenton assumes that tainted information is only ever put in tainted registers and untainted information is only ever put in untainted registers, which implies that the compiler already knows the taint information of every piece of data for every program point because, the compiler needs to decide what type of register to store any given piece of data in. TaintDroid [39] detects data leakage of Android applications using variable-level tracking within the VM interpreter. It does not track taints for native code, and only applies a heuristic that propagates taints from input arguments to that of the return value of functions.

Other works have discussed the applicability of DIFT for malware analysis. Cavallaro et al. [40] focused on the possibility of an attacker specifically evading DIFT. Because FAROS focuses on posthoc attack analysis it does not address this concern, since it assumes that the malicious code is not specifically attempting to evade DIFT. More research is needed to address these type of DIFT evading threats. Slowinska and Bos [41] quantified the DIFT overtainting problem. FAROS does not provide general-purpose tainting, which was the focus of Slowinska and Bos's analysis. Rather, FAROS shows that, for certain applications, focusing on tag confluence can provide good precision, overcoming the challenge of overtainting in a per security policy fashion.

VIII. Conclusion

In this paper, we presented FAROS, a DIFT-based reverse engineering tool, which can illuminate in-memory injection attacks by leveraging the synergy of (i) whole-system taint analysis; (ii) the mitigation of the inacuracy of traditional DIFT in a per security policy fashion via the application of tags of different types and observing their confluence as attack invariants, and (iii) the use of tags with fine-grained provenance information. FAROS accurately flagged all six in-memory injection attacks evaluated in this paper, while achieving a very low false positive (2%). FAROS not only saves reverse engineers substantial time and effort in practice, but also provides them with valuable information about any in-memory injection attacks, including process information for injecting and injected processes, relevant memory addresses, and netflows.

Acknowledgments

We would like to thank our shepherd Etienne Riviere for guidance in writing the final version of the paper and to the DSN 2018 anonymous reviewers for valuable feedback. This research has been supported by DARPA Trusted Computing Project (Grant No. FA8650-15-C-7565) and the U.S. National Science Foundation (Grant Nos. #1518523, #1518878).

References

[1] J. Lundgren, "Detecting stealthier cross-process injection techniques with windows defender atp: Process hollowing and atom bombing," https://blogs.technet.microsoft.com/mmpc/2017/07/12/detecting-stealthier-cross-process-injection-techniques-with-windows-defender-atp-process-hollowing-and-atom-bombing/, Accessed August 2, 2017.

[2] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, "Malware? tear it apart, discover its ins and outs and collect actionable threat data." https://cuckoosandbox.org/, Accessed May 13 2017.

[3] A. King, "Reflective injection detection," https://www.defcon.org/images/defcon-20/dc-20-presentations/King/DEFCON-20-King-Reflective-Injection-Detection.pdf, DEFCON 2012.

[4] "The volatility foundation," http://www.volatilityfoundation.org/, Accessed May 13 2017.

[5] "Malfind," https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal#malfind, Accessed May 13 2017.

[6] G. E. Suh, J. Lee, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," in *Proceedings of ASPLOS-XI*, Oct. 2004.

[7] B. Mazloom, S. Mysore, M. Tiwari, B. Agrawal, and T. Sherwood, "Dataflow tomography: Information flow tracking for understanding and visualizing full systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 1, p. 3, 2012.

[8] B. F. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering for the greater good with panda," Columbia University Computer Science Technical Reports, New York, NY, Tech. Rep. CUCS-022-14, 2014. [Online]. Available: http://dx.doi.org/10.7916/D8WM1C1P

[9] S. Fewer, "Reflective dll injection," https://pdfs.semanticscholar.org/d30c/d54b10948fef0046f61fd05cd18eed4cc561.pdf, 2008.

[10] J. Leitch, "Process hollowing," http://www.autosectools.com/process-hollowing.pdf, Accessed May 13, 2017.

[11] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. William Pollock, 2012.

[12] M. Hale Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics*. Wiley, 2014.

[13] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of Annual Technical Conference*. USENIX, 2005.

[14] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[15] "Tiny code generator (tcg)," http://wiki.qemu.org/Documentation/TCG, Accessed May 13, 2017.

[16] "Penetration testing software," https://www.metasploit.com/, Accessed May 13, 2017.

[17] "Kali linux," https://www.kali.org/, Accessed May 13, 2017.

[18] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, no. 2, 2007.

[19] "Norman sandbox," http://download.norman.no/product_sheets/eng/SandBox_analyzer.pdf, Accessed May 13, 2017.

[20] "Cuckoo sandbox," https://cuckoosandbox.org/, Accessed May 13, 2017.

[21] "Anubis," http://anubis.iseclab.org/, Accessed May 13, 2017.

[22] M. Graziano, D. Canali, L. Bilge, A. Lanzi, and D. Balzarotti, "Needles in a haystack: mining information from public dynamic analysis sandboxes for malware intelligence," in *USENIX Security Symposium*. USENIX Association, 2015, pp. 1057–1072.

[23] "Dynamic link library (dll)," https://support.microsoft.com/en-us/help/815065/what-is-a-dll, Accessed May 13, 2017.

[24] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 116–127.

[25] X. Jiang and X. Wang, "out-of-the-box monitoring of vm-based high-interaction honeypots," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 198–218.

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

[27] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 110–121.

[28] A. Bacs, R. Vermeulen, A. Slowinska, and H. Bos, "System-level support for intrusion recovery," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 144–163.

[29] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, 2004, pp. 221–232.

[30] J. R. Crandall, S. F. Wu, and F. T. Chong, "Minos: Architectural support for protecting control data," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 3, no. 4, pp. 359–389, 2006.

[31] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 482–493.

[32] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," *MICRO-39*, pp. 135–148, December 2006.

[33] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation." in *HPCA*, 2008, pp. 173–184.

[34] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2011.

[35] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 196–206.

[36] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[37] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ASPLOS*, 2009, pp. 109–120.

[38] J. S. Fenton, "Information protection systems," in *Ph.D. Thesis, University of Cambridge*, 1973.

[39] W. Enck, P. Gilbert, 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," in *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[40] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *International conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 143–163.

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