Manipulating Semantic Values in Kernel Data Structures: Attack Assessments and Implications

Aravind Prakash*, Eknath Venkataramani*, Heng Yin*, Zhiqiang Lin†
{arprakas, evenkata, heyin}@syr.edu, zhiqiang.lin@utdallas.edu
*Department of EECS, Syracuse University
†Department of Computer Science, University of Texas at Dallas

***Abstract*—Semantic values in kernel data structures are critical to many security applications, such as virtual machine introspection, malware analysis, and memory forensics. However, malware, or more specifically a kernel rootkit, can often directly tamper with the raw kernel data structures, known as DKOM (Direct Kernel Object Manipulation) attacks, thereby significantly thwarting security analysis. In addition to manipulating pointer fields to hide certain kernel objects, DKOM attacks may also mutate semantic values, which are data values with important semantic meanings. Prior research efforts have been made to defeat pointer manipulation attacks and thus identify hidden kernel objects. However, the space and severity of Semantic Value Manipulation (SVM) attacks have not received sufficient understanding. In this paper, we take a first step to systematically assess this attack space. To this end, we devise a new fuzz testing technique, namely - *duplicate-value directed semantic field fuzzing*, and implement a prototype called *MOSS*. Using MOSS, we evaluate two widely used operating systems: Windows XP and Ubuntu 10.04. Our experimental results show that the space of SVM attacks is vast for both OSes. Our proof-of-concept kernel rootkit further demonstrates that it can successfully evade all the security tools tested in our experiments, including recently proposed robust signature schemes. Moreover, our duplicate value analysis implies the challenges in defeating SVM attacks, such as an intuitive cross checking approach on duplicate values can only provide marginal detection improvement. Our study motivates revisiting of existing security solutions and calls for more effective defense against kernel threats.**

I. Introduction

Operating system (OS) manages the hardware resources and provides a higher-level abstraction to the user-level applications. This higher-level abstraction can be described using the OS-level semantic knowledge, such as what processes are active in the system, which process is currently running, what modules are loaded into a specific process, which files are opened by a process, which network connections have been opened, and so on. This knowledge is crucial for many computer security applications, including virtual machine introspection (VMI), malware detection and analysis, and memory forensics. The functionality and trustworthiness of these security applications critically depend on the correctness of the obtained OS-level semantic knowledge.

However, OS kernel can be compromised. Particularly, a family of attacks, called Direct Kernel Object Manipulation (DKOM) can directly tamper with values (including both pointers and data values) in important kernel data structures, in order to hide malicious activities and confuse security tools. For instance, the FU rootkit [1] has capabilities of hiding a process, escalating the privilege of a process, hiding a network connection, etc. Consequently, a great deal of work has been designed to detect the hidden objects [2]–[5], based on the notion that DKOM rootkits often manipulate kernel pointers to hide their presence. For instance, KOP [3] and MAS [5] can generate nearly complete traversal template to discover nearly all kernel objects. Two robust signature schemes (value-based [3] and pointer-based [4]) are used to scan the memory dump and can identify hidden objects more reliably.

Unfortunately, in addition to manipulating pointers to hide specific kernel objects, attackers may also manipulate data values in kernel data structures to mislead security tools. To distinguish from pointer manipulation based DKOM attacks, we call such attacks as *Semantic Value Manipulation* (SVM) attacks. It is still unclear how large the attack space of SVM is and how severe SVM attacks can be on OS kernels, specially on closed-source operating systems (like Windows). On one hand, with the highest privilege, an attacker can modify arbitrary memory locations; on the other hand, she does not want these modifications to introduce noticeable differences in system behavior (e.g., crashes, instability, and malfunction).

Therefore in this paper, we conduct the first systematic study to assess the attack space of SVM attacks on both Windows and Linux, the two most widely used operating systems. In order to conduct this study, we propose a new fuzz testing technique to automatically mutate data structure fields of interest. There are two unique features in our system: (1) It is *semantic-field oriented*, namely it can cooperate with the test program and automatically locate the data structure fields that hold specified OS semantic information and mutate their values; and (2) it is *duplicate-value directed*, because semantic values are often duplicated in various data structures. Test coverage is increased by fuzzing these duplicates both individually and simultaneously.

Value duplicates might lead to a more robust defense against such attacks. For instance, as a hypothetical defense, a security tool may conduct consistency checking across these duplicate values to detect any mutation attempts. Therefore, during the monitoring of the binary execution of the OS kernel, we would like to automatically locate the data structure fields of interest exercised by our test program as well as their duplicates in other data structures. By fuzzing these values individually, we can identify which copy is subject to mutation. By fuzzing values simultaneously, we can determine whether this entire value duplicate set is subject to mutation. This helps us determine if the security tools can indeed perform consistency checking on the set.

978-1-4799-0181-4/13/$31.00 ©2013 IEEE

To automatically identify duplicate values from the binary execution of the OS kernel, we devise *dynamic duplicate value analysis algorithm*, a new dynamic dataflow analysis algorithm. This algorithm monitors the execution of each instruction and maintains a duplicate value set for each variable (i.e., memory location and register). Since our analysis directly works on the binary execution of an operating system, it is general enough to evaluate any operating system (including the closed-source OSes, such as Windows).

We have implemented this new fuzz testing technique into a prototype system, named *MOSS* (short for "Mutating OS Semantics"). Using MOSS, we conduct experimental analysis on Windows XP and Ubuntu 10.04.

To further demonstrate the attack impact, we implemented a proof-of-concept kernel rootkit, based on FUTo [1] (a well-known DKOM rootkit for Windows). Specifically, we installed a real-world bot, TDSS [6] in a controlled Windows XP guest OS and using the rootkit, we performed simultaneous mutations to all vulnerable semantic fields identified by MOSS. The mutations were targeted at hiding and/or misleading the state-of-the-art security tools without leading to system crash.

Paper Contribution. In summary, this paper makes the following contributions:

- We conduct the first systematic study to assess the attack space and severity of SVM attacks. Specifically, we propose *duplicate-value directed semantic field fuzzing* technique, and devise *dynamic duplicate value analysis* technique to automatically identify duplicate fields. We have implemented these techniques in our prototype MOSS.
- We perform an empirical evaluation on both Windows and Linux OSes using fuzzing based tests, and show that many semantic values can be manipulated without any adverse effects on system stability and program functionality, implying that the space of SVM attacks is vast for both Windows and Linux.
- We implement a proof-of-concept SVM rootkit that confirms the findings from our fuzz testing. Protected by our rootkit, a realworld bot program can successfully mislead or worse, hide from all the security tools we tested, including recently proposed robust signature schemes.
- Our study also assesses the difficulty of defeating SVM attacks. We show that consistency checking on duplicate values is effective on some semantic fields, but not all.

II. Background & Problem Statement

A. Semantic Value Manipulation Attacks

The OS manages hardware resources and provides services such as system calls to user level programs. The semantic abstraction of OS, the focus of this paper, consists of a variety of entities, including processes, threads, files, directories, network connections, kernel modules, etc. Each entity is associated with a set of attributes, such as ID, name, status, etc.

These attributes are stored in the data fields of various kernel data structures. We refer to such data fields that hold OS semantic information as *semantic fields* (for example, in flavors of Windows OS, "UniqueProcessId" and "ImageFileName" in `EPROCESS` hold the pid and the process name, respectively). Sometimes, one semantic value may be replicated in multiple kernel data structures. For instance, in Windows, the program name is stored in `EPROCESS` as the name of the process and the main module name as one of the loaded modules.

It is common for security analysis tools to refer to such attributes to retrieve sensitive information from the kernel. While there are numerous techniques to ensure kernel code integrity [7]–[9], as well as control flow integrity (e.g., [2], [10]–[12]), there is no reliable data integrity protection techniques yet. As such, to evade all the existing defense mechanisms, adversaries are motivated to launch data-only attacks, particularly DKOM attacks wherein, adversaries directly modify the pointer fields and data fields of certain kernel objects to hide and manipulate certain OS semantic fields [1].

While unlinking a kernel object by manipulating pointers is an effective hiding technique, defeating such a technique using data structure traversal [2], [5] and scanning [3], [4] based approaches is relatively easy. What is more interesting is the direct modification on semantic values. For example, can an attacker directly modify the process name in the process object and the name of an opened file to deceive security analysis tools? What other semantic values can be freely mutated by attackers? Although we are aware of specific DKOM or SVM attacks, these questions in general have not been well understood.

B. Problem Statement

In this paper we aim to conduct a systematic study to assess the space and severity of SVM attacks. In particular, we aim to answer the following two questions:

(1) Which semantic fields are subject to direct mutation attacks? Attackers have incentives to manipulate values in the semantic fields, but cannot make arbitrary changes. Some of these changes will lead to system crash or malfunction, which attackers will try to avoid because their goal is to maintain stealth. A semantic field is not sensitive to mutation if, after a change to it, the OS continues to function normally. However, security tools can depend on a semantic value if it is sensitive to mutation, i.e., changing it will impact system or program stability.

The answer to this question may also heavily depend on each individual OS version, due to the different data structure models and different ways to operate on these values. From a security standpoint, a semantic value is untrustworthy if several common mutations cause no adverse effects on the system or the program. However, it is difficult to conclude that a semantic value is completely trustworthy. A failed mutation attempt on a semantic field under certain system states does not mean that this semantic value is not mutable at all. Under other circumstances and with multiple mutations, it might be possible to safely change the semantic value. We do not intend to completely explore the attack space, because it is impossible to iterate through all circumstances and combinations. To

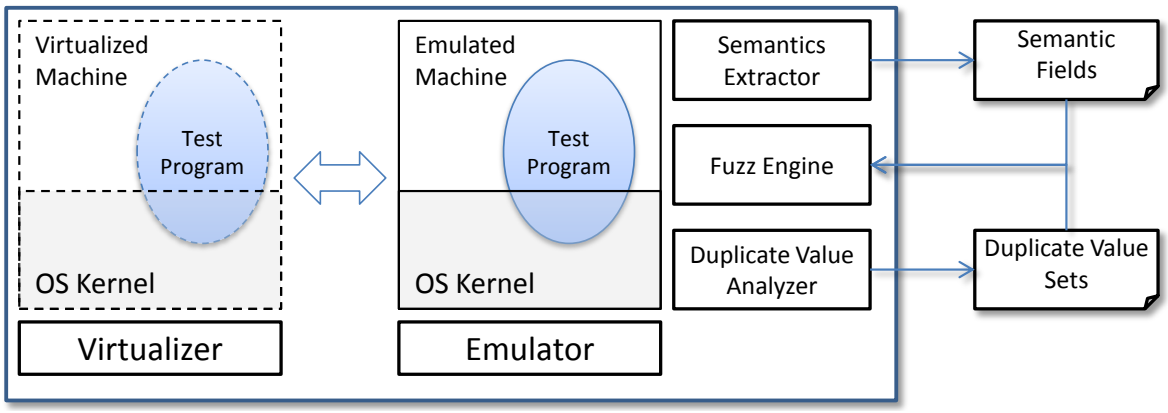

Fig. 1. Architecture of Dupcliate-Value directed Fuzzing

be more realistic, we aim to evaluate single value mutation attacks, in which each semantic field is mutated individually. This evaluation at least serves as a lower bound of the actual attack space.

(2) Can consistency checking help detect SVM attacks? An OS often manages its semantic information in a redundant fashion. We know that this is true at least for some semantic information such as PID and process name. From the perspective of defenders, we may be able to leverage the information redundancies in the semantic values to detect SVM attacks. If we know several semantic values are always the same and one semantic value tends to be less mutable than the others, we should check this field instead of the other fields. Alternatively, we may perform a consistency checking on the set of semantic values. To evade such consistency checking, attackers would have to change these semantic values in the entire set, increasing the chances of system instability.

III. Our Techniques

In order to answer the above two questions, we propose a new fuzz testing technique called *duplicate-value directed semantic value fuzzing*. The target of this fuzz testing is an OS kernel (such as Windows or Linux), and the data to be mutated are the important semantic values along with their duplicates.

A. System Overview

Figure 1 illustrates an overview of our fuzz testing system. We run the OS of interest within TEMU [13], [14], a whole-system binary analysis platform. Such a virtualized testing environment facilitates fuzz testing for several reasons. First, it is simple to modify arbitrary memory values. Second, it can easily revert the virtual machine to the previously saved state to conduct fuzz testing in the next round. Last and most importantly, it can dynamically switch between emulation and virtualization mode for during testing. In the emulation mode, we can perform fine-grained binary analysis to locate duplicate semantic values, and then we can switch to the virtualization mode to fuzz these duplicate values for better testing efficiency.

More specifically, inside the virtual machine, we run a test program to activate the kernel side execution. Note that we are mutating the semantic values that are related to malicious activities. That is, the attacker attempts to manipulate semantic values about her own behavior, such as the name of the malicious process, the file that has been accessed, and so on. These malicious activities are often stealthy and have infrequent interactions with the victim system. To mimic these malicious activities, our test program does not need to achieve the high test coverage of the OS kernel code. Instead, our test program just need conduct some common tests to exercise different OS subsystems, such as task management, file system, network stack, etc. Therefore, if all the mutation attempts on a semantic value do not cause adverse effect in these test cases, we can conclude that this semantic value is mutable. Otherwise, if a semantic value is sensitive enough to all the mutation attempts on it, we have confidence that this semantic value is immutable and thus tend to be trustworthy. The situation for some semantic values is in between: some mutations cause system instability while some others do not. These semantic values are *partially* mutable.

On top of TEMU, we develop three components: semantics extractor, fuzz engine, and duplicate value analyzer. The semantics extractor, which will be discussed in Section III-B, locates the semantic values from the memory snapshot of the guest system. The duplicate value analyzer monitors the kernel execution and perform dynamic duplicate value analysis, which will be detailed in Section III-C. At a high level, it clusters the memory locations into sets, each of which holds the same semantic value. The fuzz engine coordinates with the other two components to conduct automated fuzz testing, which will be discussed in Section III-D.

B. Locating Semantic Values

At certain execution point, we need to locate the semantic values to be mutated. Semantic values for mutation are selected in cooperation with the test program inside the virtual machine. A test point has been defined within the test program, dictating which semantic value or which set of values need to be mutated. More details will be discussed in Section III-D. Then the semantic extractor needs to locate the selected semantic value in the guest kernel memory space.

We leveraged Volatility memory forensics framework [15] and we implemented a plug-in to locate the semantic values

of interests. More specifically, at the test point, the virtual machine is paused, and a memory snapshot is taken. Then our Volatility plug-in will parse the kernel data structures in the memory snapshot and identify both virtual and physical address for the selected value. The virtual address will then be used as input to find duplicate value sets, which will then be mutated individually and simultaneously in the subsequent fuzz testing.

C. Dynamic Duplicate Value Analysis

Many memory locations share the same value at a given moment, either coincidentally, or because of program logic. Our interest is the latter case since such duplicates hold values which have the same semantic meaning. We call these variables to be *truly* duplicate. To identify true duplicate values, we devise a dynamic binary analysis algorithm that classifies variables (memory locations or registers) into clusters. Variables belonging to the same cluster hold the same semantic value because of the program logic in this particular program execution.

To better explain the idea of dynamic duplicate value analysis, consider the example code in Table I. After executing the 6 statements under "Statement" column of Table I, variables a, c, e, and f should have the same value, so these variables should belong to the same cluster. b belongs to this cluster till line 5, where b is assigned to a different value. Suppose that e is identified to have a semantic meaning such as `pid` of a process, we can conclude that the other variables (a, c, and f) in the same cluster should also hold the `pid` of that process. Therefore, we need to perform dataflow analysis to compute these clusters.

Yet, the existing forward dataflow analysis (i.e., taint analysis [16]) and backward dataflow analysis (i.e., backward slicing [17]) cannot solve this problem. For taint analysis, the taint source needs to be known in advance. However, in our case, semantic values can only be identified at a later stage. Backward slicing is not a solution either. Starting from line 4 and walking backward the code snippet, backward slicing can identify e is directly copied from a and b, but c and f are missing. Moreover, b should not be a redundant value, because b is later assigned to a different value at line 5. To solve this problem, we devise a new dynamic dataflow analysis algorithm called *dynamic duplicate value analysis* to compute the clusters at runtime. The basic algorithm is shown in Algorithm 1.

The basic idea of this algorithm is as follows. At memory byte granularity, we treat each memory byte as a variable r and a redundancy cluster S_r is associated with each variable r. Based on each instruction's semantics from the execution traces, we perform data flow analysis. More specifically,

- **Direct Assignment** For each instruction i in the execution trace, we check if i is an assignment operation. In x86, assignment operations include `mov`, `push`, `pop`, `movs`, `movzx`, `movsx`, etc. As a variable represents a memory byte, we break an assignment into one or more per-byte assignments, and for each source and destination byte pair (u, v), we update the duplicate sets accordingly (as shown in `DoAssign`). First of all, the destination v is no longer equivalent to the other variables r in its old duplicate set S_v, and thus v needs to be removed from S_r. Then, as now v is equivalent to u, v also needs to be added into the duplicate set S_r, where $r \in S_u$. Lastly, the duplicate set of v will be updated to that of u. In general, a membership change of a variable in its duplicate set needs to spread around to maintain consistent membership information. SSE and MMX instructions may also serve as data transfer operations. We do not consider these instructions because we found in our experiments that these instructions rarely appear in the kernel execution.
- **Other Operations** For the rest of the instructions, while the duplicate sets for the source operands remain the same, the duplicate set for the destination operand needs to be reset. Therefore, for each byte v of the destination operand, `DoRemove` notifies all variables in v's duplicate set that v is no longer a duplicate value to them.

Algorithm 1 Basic Algorithm for Dynamic Duplicate Value Analysis

```
procedure DynValueAnalysis(Trace t)
    for all instruction i ∈ t do
        if i.type is assignment operation then
            for each src & dst byte pair(u, v) do
                DoAssign(u, v)
            end for
        else
            for each byte v in the dst operand do
                DoRemove(v)
            end for
        end if
    end for
end procedure

procedure DoAssign(u, v)
    for all variable r ∈ S_v do
        S_r ← S_r − {v}
    end for
    for all variable r ∈ S_u do
        S_r ← S_r + {v}
    end for
    S_v ← S_u
end procedure

procedure DoRemove(v)
    for all variable r ∈ S_v do
        S_r ← S_r − {v}
    end for
end procedure
```

Table I gives a step-by-step demonstration of how the algorithm executes on the above code snippet.

Extension for String Conversions. However, the basic algorithm only handles literal value equivalence. For strings, the operating system kernel often makes conversions, such as from ANSI to UNICODE or vice versa, or from upper

TABLE I. ALGORITHM EXECUTION ON THE SAMPLE CODE

Statement	S_a	S_b	S_c	S_d	S_e	S_f
`1: a = b`	{a, b}	{a, b}				
`2: c = a`	{a, b, c}	{a, b, c}	{a, b, c}			
`3: d = b + c`	{a, b, c}	{a, b, c}	{a, b, c}			
`4: e = a`	{a, b, c, e}	{a, b, c, e}	{a, b, c, e}		{a, b, c, e}	
`5: b = 2`	{a, c, e}		{a, c, e}		{b, d, e}	
`6: f = c`	{a, c, e, f}		{a, c, e, f}		{a, c, e, f}	{a, c, e, f}

case to lower case or vice versa. Semantically, a converted string is equivalent to the original string. Therefore, we have to extend the basic algorithm to maintain the equivalence relation between the converted and original strings. We hook the string handling functions in Windows and directly call `DoAssign` to make the duplicate value association between the input and the output.

Discussion. This algorithm captures how normal program execution operates on duplicate values, through direct assignments and restricted string conversions. Thus, it is able to correctly identify duplicate values in regular programs. However, a program may be obfuscated to evade our analysis. As an example, a direct assignment can be replaced by a sequence of arithmetic or logic operations. As we apply this algorithm to benign kernel code analysis, this limitation does not apply.

Moreover, as a dynamic analysis technique, the identified duplicates depend on the program execution. In our setting, we trace the kernel execution from the start of the test program to a designated test point, so the creation and propagation of the semantic values associated with the test program should be completely captured and analyzed.

D. Testing Procedure

Testing Cycle. As depicted in Figure 2, a testing cycle proceeds as follows:

1) In the virtualization mode, start the virtual machine and boot up the guest system.
2) Switch to emulation mode, run the test program and start to trace kernel execution for duplicate value analysis.
3) At a predetermined test point, pause the virtual machine and save the current VM state; in the meantime consult the semantics extractor to locate important semantic values and query the duplicate value analyzer to compute duplicate value sets; and switch to the virtualization mode.
4) Choose to mutate a single value or a set of duplicate values, and resume the virtual machine;
5) The test program finishes normally or prematurely or system crashes; revert to the saved VM state and go the step 4 to fuzz another semantic value or another duplicate value set.

The testing cycle shown above is done for one test point. In reality, we define multiple test points to mutate different sets of semantic values. Therefore, this testing cycle will be conducted multiple times, one for each test point.

TABLE II. TEST CASES AND TEST POINTS

No	**Test Case**
1	Start test program Test Point 1: mutate process&thread related values Run other test cases
2	Load a user DLL Test Point 2: mutate DLL related values Call a function in the DLL repeatedly Unload the DLL
3	Load a kernel module Test Point 3: mutate kernel module values Send IO requests to the kernel module Unload the kernel module
4	Open two files for read and write Test Point 4: mutate file values Read and write these two files repeatedly close these files
5	Open a TCP connection Test Point 5: mutate values related to this connection Send and receive data through this connection Close this connection
6	Open a registry key (Windows only) Test Point 6: Mutate registry key related values Read and write this registry key repeatedly Close this key

Test Program. We design our test program to exercise basic and common operations that are commonly performed by programs and that are typically exhibited by malware. More specifically, our test program includes the following test cases and identify appropriate test points, as shown in Table II. We can see that this test program exercises process and thread management, DLL load and unload, kernel module management, file operations, network operations, and registry key accesses (for Windows only). Totally six test points are defined at precise moments, when the virtual machine will be paused and selected semantic values will be mutated. These test points capture the moment when certain kind of values have been created and will be used for later operations. For example, for file related semantic values, the test point is defined after the files are open and before read and write operations are performed on these files.

As a preliminary step to conducting the network related tests, we launch a light weight HTTP server on the guest OS. This is important because our fuzz testing repeatedly reverts back to a previous VM state. If the server program is on a different host, the connection states for the client and the server will become out-of-sync once the VM is reverted back to an earlier state.

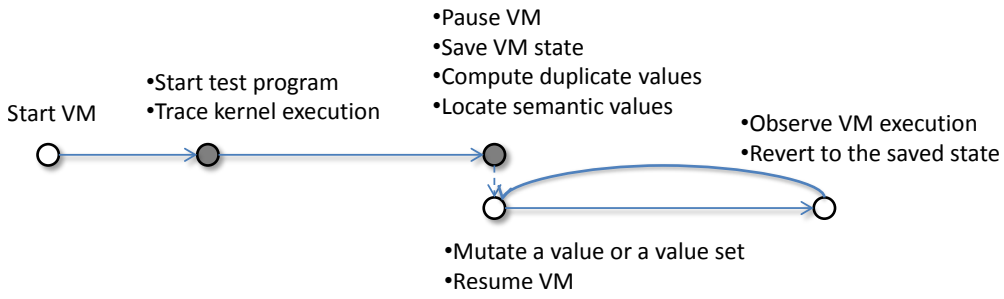

Fig. 2. **Fuzz Testing Cycle.** A gray node indicates the virtual machine at that moment is running in the emulation mode, whereas a white node stands for the virtualization mode.

TABLE III. VALUE MUTATION RULES

Type	Mutation Rules
ID	0, copy from another ID, increment or decrement by a small constant
Size/Offset	0, increment or decrement by a small constant
String	"", copy from another string, mutate one character

The identified test points are tested individually. For instance, when we conduct fuzzing on the first test point, the other test points are simply skipped. Here, though the test points are skipped the test program continues to perform all the operations listed in Table II during all the tests. This is important since a change in a test point could have an implication in multiple functionalities. For instance, a change to a thread related semantic values might result in dropping of the connection that thread has made. Also, the order of the test cases listed in Table II does not reflect the actual order of our fuzz testing. Suppose that we are conducting test case 5 for the network connection. We actually move this test case earlier, immediately after the test program starts, such that we can observe if the mutation of network-related semantic values will affect the execution of the other test cases.

Mutation Rules. To avoid system instability due to mutation, the changes have to satisfy the type constraint of the original value. In other words, the mutation rules depend on the type of the semantic value to be mutated. In contrast, other fuzz testing projects (such as in [3], [18]) aim to randomly fuzz certain data values to identify their value constraints or to explore the program space.

We list the mutation rules in Table III. For example, for an ID (e.g., pid, tid), we consider 0 as an input, because 0 is often reserved for system process and thread. Similarly, for a string, we use an empty string as an input since the OS may have special handling for empty strings, such as ignoring and skipping an object if its name is empty. Attacker may exploit this feature to hide certain objects.

IV. EMPIRICAL STUDY

We perform our empirical study on two popular operating systems, which are Windows XP with service pack 3 (XPSP3) and Ubuntu 10.04 with Linux kernel version 2.6.32-25 (Linux). We conducted our experiments on a Pentium Core i7 with 3GHz and 4GB RAM. The host operating system is 32-bit Ubuntu 10.04 with kernel version 2.6.32-38. We analyzed both operating systems individually as a virtual machine running inside QEMU. 512MB RAM was allocated for the virtual machine.

We compiled two lists of semantic fields, one for Windows XP (Table IV) and the other for Linux (Table V). Forensic tools (such as Volatility [15], a comprehensive memory forensic framework) query these semantic fields to extract semantic information from a memory dump. Although these lists are not nearly complete, we believe that they provide a fairly good coverage on important semantic fields.

Using the value mutation rules listed in Table III, we designed 3 mutation tests (including 1 whole-set mutation) for each field in Table IV and Table V resulting in a total of 258 test cases. The test cases were distributed across 12 test points (6 test points in each of the 2 OSs), with average trace gathering time of approximately 15 minutes per test point. Depending on the test point in question and the size of trace, redundancy identification and semantic value location took between 7 min (best case) to 32 min (worst case) with 92 percent of the time consumed during redundancy identification. Each test case execution involving VM restoration and fuzzing 25 to 60 seconds. After fuzzing, the execution continued for 3 minutes as a part of behavior assessment. Additionally, we wrote a rootkit to examine the effects of semantic mutations on the OS information retrieval tools. In one shot, we mutated the primitives listed in Table VIII and observed the impact on the system.

Furthermore, within the guest OS, we run administration tools (such as `netstat` for both Windows and Linux, Task Manager and process explorer [19] for Windows, and `ps`, `pmap`, `top`, etc. for Linux, and so on), to observe the effects of these mutation tests within the guest OS.

The key component of MOSS is *Duplicate Semantic Value Analysis*, which in theory is independent of the OS. Therefore, with the kernel data structure information for the key kernel data structures, careful identification of test points and a corresponding test program one can perform single-field and duplicate-field mutations on any guest OS to identify the semantic fields susceptible to mutation. In this paper, as a proof-of-concept, we consider Windows XP SP3 and Linux 2.6.32-25 to perform the empirical study. However, it is often

TABLE IV. SEMANTIC FIELDS SELECTED FOR WINDOWS XP SP3 AND THEIR MUTABILITY

Category	Semantic Field	Mutability
Process	EPROCESS.UniqueProcessId	✓
	EPROCESS.InheritedFromUniqueProcessId	✓
	EPROCESS.ImageFileName	✓
	EPROCESS.CreateTime	✓
	EPROCESS.ExitStatus	✓
	EPROCESS.ActiveThreads	p
	EPROCESS.GrantedAccess	✓
	EPROCESS.Token	✗
	EPROCESS.ObjectTable.HandleCount	✓
	EPROCESS.Flags	p
	EPROCESS.ObjectHeader.ObjectType	✓
	EPROCESS.PoolHeader.PoolTag	✓
	EPROCESS.PoolHeader.BlockSize	✓
Thread	ETHREAD.PoolHeader.PoolTag	✓
	ETHREAD.PoolHeader.BlockSize	✓
	ETHREAD.ObjectHeader.ObjectType	✓
	ETHREAD.Cid.UniqueProcess	✗
	ETHREAD.Cid.UniqueThread	✓
	ETHREAD.StartAddress	✓
DLL & Kernel Module	_LDR_DATA_TABLE_ENTRY.DllBase	✓
	_LDR_DATA_TABLE_ENTRY.EntryPoint	✓
	_LDR_DATA_TABLE_ENTRY.FullDllName	✓
	_LDR_DATA_TABLE_ENTRY.BaseDllName	✓
	_LDR_DATA_TABLE_ENTRY.Flags	✓
	_LDR_DATA_TABLE_ENTRY.LoadCount	✓
	_LDR_DATA_TABLE_ENTRY.PatchInfo	✓
Registry Key	CM_KEY_NODE.Name	✓
	CM_KEY_NODE.NameLength	✓
	CM_KEY_NODE.LastWriteTime	✓
	CM_KEY_NODE.SubkeyCounts	✓
	CM_KEY_NODE.Flags	✓
	CM_KEY_NODE.Signature	✓
	CM_KEY_NODE.Parent	✓
	CM_KEY_NODE.Security	✗
Network	TCPT_OBJECT.RemoteIpAddress	✗
	TCPT_OBJECT.RemotePort	✗
	TCPT_OBJECT.LocalIpAddress	✗
	TCPT_OBJECT.LocalPort	✗
	TCPT_OBJECT.Pid	✗
	TCP_LISTENER.AddressFamily	✓
	TCP_LISTENER.Owner	✓
	TCP_LISTENER.CreateTime	✓
	TCP_ENDPOINT.State	✓
Memory Pool	POOL_HEADER.PoolTag	✓
	POOL_HEADER.BlockSize	✓

TABLE V. SEMANTIC FIELDS SELECTED FOR LINUX AND THEIR MUTABILITY

Category	Semantic Field	Mutability
Task	task_struct.state	✓
	task_struct.flags	✓
	task_struct.pid	p
	task_struct.fds	✗
	task_struct.comm	✓
	task_struct.start_time	✓
	task_struct.stime	✓
	task_struct.exit_code	✓
File	task_struct.files.fd[i].f_owner	✓
	task_struct.files.fd[i].f_mode	✓
	task_struct.files.fd[i].f_pos	✓
	dentry.d_name	✓
	dentry.d_iname	✓
	dentry.d_flags	✓
	dentry.d_time	✓
	inode.i_uid	✓
	inode.i_gid	✓
	inode.i_size	✓
	inode.i_atime	✓
	inode.i_ctime	✓
	inode.i_mtime	✓
Module	module.name	✓
	module.num_syms	✓
	module.state	✓
	module.core_size	✓
	module.core_text_size	✓
	module.num_kp	✓
	vm_area_start.vm_start	✗
	vm_area_start.vm_end	✗
	vm_area_start.vm_flags	✓
Network	inet_sock.saddr	✗
	inet_sock.daddr	✗
	inet_sock.sport	✗
	inet_sock.dport	✗
	sock_common.skc_family	✓
	sock_common.skc_refcount	✓
	sock_common.skc_state	✓
	sock.sk_protocol	✓
	sock.sk_flags	✓
	sock.sk_type	✓
	sock.sk_err	✓

the case that a new version of an OS retains a significant part of the previous version. Therefore, it is possible that the mutability results tabulated in Table IV and Table V are applicable to other versions of Windows and Linux OSes, respectively.

A. Single Field Mutation

We consider a semantic field to be immutable only if all of the mutation attempts on it cause system or program instability. If some of the mutations do not cause critical failures, then attackers may potentially make similar modifications and thus mislead the security tools. Based on this standard, we have listed the results in the last column of Table IV and Table V. A `'p'` in the mutability column indicates that the semantic field showed no system or program instabilities for certain mutations, while it did for some others.

From the mutability column in Table IV and Table V, we can see that most of the semantic fields, including process name, file name, module name and many others can be changed by an attacker without adverse effects on the system or a program. This observation immediately raises a question about the trust issue for all the security applications (such as memory forensics and virtual machine introspection) that critically rely on the correctness of these semantic fields.

For both operating systems, network related semantic fields tend to be reliable. Mutations to source and destination IP addresses and port numbers immediately cause failures to subsequent operations on the network connection. This is good news, which means network security tools that make security decisions based on the network connections can be trusted, as long as these connection objects can be reliably located.

For Windows XP, the `UniqueProcessId` in `ETHREAD` tends to be reliable. A mutation will either crash the entire system or the test program. The `Pid` in the TCP connection object (`TCPT_OBJECT`) can also be relied upon. A mutation on it will immediately drop this connection. It

is worth noting that security tools usually read Pid from EPROCESS.UniqueProcessId, which turns out to be not reliable at all, because none of the mutations on it causes severe failures. This finding suggests to retrieve the UniqueProcessId in the ETHREAD objects or Pid in the TCPT_OBJECT objects (if available) instead.

Interestingly, strings are completely mutable (that is, all occurrences of the string can be mutated without adverse effect on the system) for both the operating systems we tested. OS kernels usually rely on pointers and integers (such as handles and IDs) for operations as opposed to strings. String mappings for resources (e.g., file handle to file name) are often maintained in instances that involve interpretation by a human. This observation is particularly worrisome since strings like process name, file name, registry key name, etc., have severe security relevance and are fully mutable.

Similarly, it turns out that all the time related information (such as, process creation time, exit time, etc.) are also fully mutable and therefore not reliable. This observation has far reaching impacts. For instance, time information is crucial in a memory forensic context. One may need to use the time stamps of certain malicious activites as crime evidence. With DKOM as a possibility, such time stamps cannot be assumed correct.

B. Duplicate Field Mutation

In addition to mutating the selected semantic fields individually, we also identified their duplicate fields and mutated these duplicates both separately and simultaneously. We present these results in Table VI and Table VII for Windows XP and Linux, respectively. For each primary semantic field that has at least one duplicate, we list the number of duplicates (including the primary) identified through MOSS, the types of these duplicates, the immutable duplicates if any, and whether the entire duplicate set is mutable. Due to the dynamic nature of our analysis, the number of duplicates depends on the start execution point, the end execution point, and the particular execution path. In our experiment, duplicate values were identified by dynamic duplicate value analysis from the start of the test program to a predetermined test point. Therefore, these duplicates may not always hold true for different test cases. For each duplicate value, we further identify in which data structure and which field the value is located whenever possible. Again, we use Volatility for locating kernel data structures. Due to the limited coverage of Volatility, we may not always be able to recognize the corresponding data structures. In such cases, we list only the virtual addresses in the third column.

The immutable duplicates, if any, indicate which duplicate fields (other than the primary) *may* be reliable. The knowledge about immutable duplicates is valuable, because it means that security tools could examine these alternative fields instead of the primary ones to obtain more reliable OS semantics.

The last column indicates if the entire duplicate set is simultaneously mutable. If not, security tools may be able to perform a consistency check on the entire set to obtain more reliable outputs. Of course, the underlying assumption is that the security tool is smart enough to locate all the duplicate fields, which in practice may be difficult, especially for closed-source operating systems like Windows.

From the results in Table VI and Table VII, we can see that information redundancy does exist for some important OS semantics. This is the case for both operating systems. For example, in Windows, EPROCESS.UniqueProcessId appears as the UniqueProcess in all the ETHREAD objects belonging to that process, and also appears in the HANDLE_TABLE. For a process which has established at least one TCP connection, the pid should also appear in the TCPT_OBJECT.pid [19], which MOSS could not identify at test point 1. This is because the network operations happened after test point 1 in our experiment and the corresponding TCPT_OBJECT was not created at that point. In fact, at test point 5, we confirmed that TCPT_OBJECT.pid indeed is one of the duplicates. For the process name EPROCESS.ImageFileName, we also found duplicates in OBJECT_NAME_INFORMATION.Name and RTL_USER_PROCESS_PARAMETERS.ImagePathName. As the main module, the process name also appears in the base module name BaseDllName and full module name FullDllName in LDR_DATA_TABLE_ENTRY. These results are also consistent with publicly available Windows documentation [19].

For Linux, we found that the pid of the test program replicates in the group id task_struct.t_gid, and also the light-weight process (lwp)'s group id, which specifies the pid of the hosting process of a thread in Linux. Similarly, the process name in task_struct.comm also share the same value with its light-wight processes. vma.vm_start has a duplicate in vma.vm_end of the preceding vma structure, and vma.vm_end has a duplicate in vma.vm_start of the subsequent vma structure. We also found that the source IP address and the destination IP address are duplicate to each other. This is because in our test, both the server and the client programs are running in the localhost, so both source and destination IP addresses are 127.0.0.1. These findings are in agreement with the source code of the OS kernel.

Unfortunately, our results show that most of these duplicate are mutable both individually and simultaneously. In very limited cases, the information redundancy can help improve the integrity of semantic information. As discussed earlier, though UniqueProcessId in EPROCESS is mutable, its duplicate, UniqueProcess in ETHREAD is immutable. ETHREAD.StartAddress in Windows is another such case. The primary ETHREAD.StartAddress can be manipulated, but its duplicate StartingVa in _SECTION_OBJECT is more sensitive to mutations.

Table VI and Table VII also show that the result of mutating the entire duplicate set is the same as mutating the individual duplicate fields. This indicates that the operating systems process these semantic fields separately, and perform no cross checking on these duplicates. From the defender's perspective, if one can reliably locate one immutable field (either the primary or a duplicate), checking the entire duplicate set is not necessary.

TABLE VI. DUPLICATE FIELDS FOR WINDOWS XP AND THEIR MUTABILITY

Primary Field	# of Dups	Type of Duplicates	Immutable Duplicates	Set Mutability
_EPROCESS.UniqueProcessId	36	_ETHREAD.Cid.UniqueProcess, _HANDLE_TABLE.UniqueProcessId, _CM_KEY_BODY.ProcessId, _EPROCESS.InheritedFromUniqueProcessId, _ETIMER.Lock, _TEB.ClientId, _TEB.RealClientId, 0x9b57b6d0, 0x9ccdaef0, 0x9cce697c...	_ETHREAD.Cid.UniqueProcess	✗
_EPROCESS.ImageFileName	4	_OBJECT_NAME_INFORMATION.Name, _RTL_USER_PROC_PARAMS.ImagePathName, _SE_AUDIT_PROCESS_INFO.ImageFileName	None	✓
_EPROCESS.CreateTime	2	_ETHREAD.CreateTime	None	✓
_EPROCESS.ActiveThreads	2	_EPROCESS.ActiveThreadsHighWatermark	None	✓
_HANDLE_TABLE.HandleCount	2	_HANDLE_TABLE.HandleCountHighWatermark	None	✓
_FILE_OBJECT.FileName (Data file)	7	0x003a948e, 0x822df33a, 0x822df35c, ...	None	✓
_LDR_DATA_TABLE_ENTRY.FullDllName	3	_LDR_DATA_TABLE_ENTRY.BaseDllName, _FILE_OBJECT.FileName	None	✓
_LDR_DATA_TABLE_ENTRY.BaseDllName	3	_LDR_DATA_TABLE_ENTRY.FullDllName, _FILE_OBJECT.FileName	None	✓
_CM_KEY_NODE.LastWriteTime	2	0x9b43ea60	None	✓
_CM_KEY_NODE.Parent	4	0x94d20a20, 0x9adc7940, 0x9adc7948	None	✓
_CM_KEY_NODE.Security	2	0x822c7880	_CM_KEY_NODE.Security	✗
_ETHREAD.StartAddress	2	_SECTION_OBJECT.StartingVa	_SECTION_OBJECT.StartingVa	✗

TABLE VII. DUPLICATE FIELDS FOR LINUX AND THEIR MUTABILITY

Primary Field	# of Dups	Type of Duplicates	Immutable Duplicates	Set Mutability
task_struct.pid	4	task_struct.t_gid, task_struct.t_gid(lwp), 0xf63916dc	None	✓
task_struct.comm	2	task_struct.comm(lwp)	None	✓
task_struct.static_prio	3	task.parent.static_prio, task.static_prio (lwp)	None	✓
task_struct.exit_code	3	task.parent.exit_code, task.exit_code (lwp)	None	✓
task_struct.fds	3	0xf7179080, 0xf61bae84	0xf7179080, task.fds	✗
module.name	2	0xd93c524c	None	✓
module.num_syms	12	module.num_kp, 0xe086c15c, 0xe086c170...	None	✓
vma.vm_start	2	vma.vm_end	vma.vm_start	✗
vma.vm_end	2	vma.vm_start	vma.vm_end	✗
dentry.d_name	2	0xf583f0d8	None	✓
inet_sock.saddr	24	inet_sock.rcv_saddr inet_sock.daddr 0xde49147c 0xde49148c ...	inet_sock.rcv_saddr inet_sock.daddr 0xde49147c 0xde49148c ...	✗
inet_sock.daddr	24	inet_sock.rcv_saddr inet_sock.saddr 0xde49147c 0xde49148c ...	inet_sock.rcv_saddr inet_sock.daddr 0xde49147c 0xde49148c ...	✗

C. Interesting Observations

In these mutation tests, we also observed some interesting behaviors, of which attackers may take advantage. In Windows XP, changing the test program's `pid` to 0 will effectively hide the test program from the process list in the Task Manager. Attackers can use this technique to effectively hide a process.

Another interesting observation was when the file name in the file object was changed to an empty string. In both operating systems we experimented on, the file name shows up as empty. If all the duplicate file names are also changed to empty, then this file becomes completely inaccessible. The file cannot be opened using either the original file name or the empty string. This is another technique an attacker might use to hide certain files. Also, once the file name and its duplicates are set to empty string, an administrator (or an anti-virus software) would not be able to delete the file, which is very concerning.

Furthermore, in Windows when the EPROCESS.Flags value is changed to $0xFFFFFFFF$, the system perceives the process as being a system process. Upon attempting to terminate the process, a dialog pops up and says that the process is a system process and terminating it would result in a restart. Force quitting it abruptly restarts the OS. We feel that this behavior is not only abnormal, but it has dangerous security implications. For example, an attacker may use this trick to prevent her malicious process from being killed.

D. Impact on Security Tools

To further confirm the results from fuzz testing based findings in Table VII and Table VI, we implemented a proof-of-concept SVM rootkit for Windows. Given a specific malware process, this SVM rootkit manipulates all the mutable semantic fields associated with this process, and their duplicates that can be identified. The SVM rootkit changes the integer values to 0 and string values to empty string. The rootkit changes the pool tags to "None" to indicate that the object is associated with the default pool. To demonstrate the power of this rootkit, we ran a bot named TDSS [6] in Windows XP SP3 in a controlled environment. We evaluated a variety of security tools, including Process Explorer [19], Task Manager,

TABLE VIII. IMPACT OF SVM ROOTKIT ON SECURITY TOOLS

Category	Primary Fields Mutated	Task Mgr	Proc Exp	Volatility (scan)	Volatility (traversal)	Sig Graph	Robust Sign	VMI
Process	EPROCESS.UniqueProcessId	H	H	H	N	N	H	N
	EPROCESS.InheritedFromUniqueProcessId	-	H	H	N	N	H	N
	EPROCESS.POOL_HEADER.PoolTag	-	-	H	N	N	H	N
	EPROCESS.POOL_HEADER.BlockSize	-	-	H	N	N	H	N
	EPROCESS.CreateTime	-	H	H	N	N	H	N
	EPROCESS.ExitTime	-	H	H	N	N	H	N
	EPROCESS.ImageFileName	H	H	H	N	N	H	N
	EPROCESS.ExitStatus	-	-	H	N	N	H	N
Thread	ETHREAD.CreateTime	-	H	H	N	-	-	N
	ETHREAD.ExitTime	-	H	H	N	-	-	N
	ETHREAD.Cid.UniqueThread	-	H	H	N	-	-	N
	ETHREAD.StartAddress	-	H	H	N	-	-	N
	ETHREAD.POOL_HEADER.PoolTag	-	H	H	N	-	-	N
	ETHREAD.POOL_HEADER.ObjectSize	-	H	H	N	-	-	N
Kernel Module & User DLL	LDR_DATA_TABLE_ENTRY.DllBase	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.EntryPoint	-	-	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.SizeOfImage	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.FullDllName	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.BaseDllName	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.Flags	-	-	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.PatchInformation	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.LoadCount	-	H	N	N	-	-	N

Volatility memory forensic tools (including both traversal and scan versions), Sig-Graph [4][1] and Robust-Signature [3]. To further complete the list of security tools, we also included an in-house VMI tool that we implemented for inspecting guest execution from QEMU. This VMI tool retrieves semantic information from a guest operating system by first locating the global data structures in the guest memory and traversing them. Specifically, it can retrieve process, thread and module information from the guest operating system. We kept TDSS running for over 3 hours before recording the outputs of various security tools, to ensure that neither the system nor the program crashed because of the SVM rootkit.

Table VIII lists the impact of the SVM rootkit on the selected security tools. It presents what primary semantic fields are manipulated and the mutation impacts on the security tools. We can see that there are mainly two kinds of symptoms: either the OS entities become hidden (H), or the misleading new values are fetched and displayed (N). Volatility traversal tools (i.e., `pslist`, `threads`, `modules`) are misled to show meaningless values. For scan tools, whereas the process and thread information is hidden from `psscan` and `thrdscan`, `modscan` can still identify the module information, which unfortunately has been manipulated and thus has become meaningless. The reason why process and thread objects are hidden is because their pool tags have been manipulated and `psscan` and `thrdscan` rely on these pool tags to identify these objects. The two robust signature schemes are also misled or evaded. The graph signature [4] for EPROCESS is reliable enough to find malware's process, but the obtained process information is all invalid. The value-invariant signature [3] is even worse. It failed to identify the malware process because the `ExitTime` of the malware process has been manipulated and the signature checks this value to remove noisy and dead process objects. The result of our VMI tool is similar to that of the Volatility traversal tools. Although the information about the malware execution can be extracted, it is incorrect. Consequently, we cannot leverage the knowledge obtained from this VMI tool to perform analysis on the malware execution.

[1]We implemented SigGraph as a plugin to Volatility and created a signature for EPROCESS

V. DISCUSSION

SVM attack space is vast. Our experiments show that several OS semantic fields can be mutated without hurting the system stability. Once the kernel has been infiltrated, an attacker can arbitrarily manipulate any semantic value in general to accomplish her malicious goals. In our experiments, we have limited our changes to single value mutations and duplicate set mutations, but an attacker is not restricted to making these changes.

In general, we did not attempt to test multiple mutations, since it leads to a very large number of combinations, which are infeasible to test. However, our current testing infrastructure does support multi-mutation based tests and can be extended in future. The key focus of our tests are to highlight the seriousness of single value and duplicate set mutations, which we believe is a large attack space in itself.

Memory forensics may fail. Memory forensic involves obtaining digital evidence from the live system. Our study shows that the digital evidence (particularly the OS semantics) obtained from a memory snapshot cannot be assumed correct, given the possibility of SVM attacks. Recent effort in robust signature schemes [3], [4] can help detect hidden kernel objects, but the extracted semantic information can still be completely misleading.

We need more trustworthy VMI techniques. The current VMI techniques [20]–[23] more or less rely on memory analysis, and can therefore be incorrect. Triggered by certain events (e.g., system calls) or demanded by the administrator, the current VMI techniques traverse important kernel data structures of the guest system, and then extract the operating system semantics. Virtuoso [22], VMST [23], and Exterior [24]

have greatly narrowed the semantic gap and improved the usability of VMI, but these new approaches do not change the fact that they directly read from the virtual machine memory, disregard of other runtime events. Once the guest kernel is compromised by DKOM attacks, the current VMI techniques will fail, just like the memory forensics.

VI. Related Work

Fuzz Testing. Plenty of research (such as SAGE [18], KLEE [25], and S2E [26]) has gone into performing fuzz testing to explore program execution space and discover bugs and security vulnerabilities. The purpose of our fuzz testing is different. Our fuzz testing targets the OS semantic fields. By mutating the values in these semantic fields as wells their duplicates, we aim to evaluate the mutability of OS semantic fields. Dolan-Gavitt et al. proposed a fuzz testing technique to detect value invariants in the kernel data structures, and use these invariants to construct more robust signatures for kernel objects [3]. Although our fuzz testing is also targeting at the kernel data structures, it is different in several ways: 1) our focus is OS semantic fields, so our system can automatically identify not only the semantic fields but also their duplicates and then perform fuzzing on these fields; and 2) our test cases and mutation rules are also designed differently as our goal is not to crash the system but to explore the potential attacks.

Dynamic Dataflow Analysis. In this paper, we devise a new dynamic dataflow analysis algorithm to track duplicate values. At some level, this algorithm is related to the abstract variable binding technique for automatically reverse engineering malware emulators [27], in which two kinds of dataflow algorithms (i.e., forward binding and backward binding) are proposed.

This algorithm also shares some similarity with dynamic type inference [28] and data structure reverse engineering [29], [30]. In these systems, the identified type for one variable is propagated to the other variables, whereas in MOSS we need to update the membership information to all the variables in the duplicate value set.

Virtual Machine Introspection. Introspecting a virtual machine often requires interpreting the low level bits and bytes of guest OS kernel to high level semantic state. This is a non-trivial task, because of the semantic-gap [31]. Early approaches (e.g., [8], [21], [32], [33]) have been using manual efforts to locate the kernel objects by traversing from the exported kernel symbols or searching for invariant numbers. Recent advances show that we can largely automate this process [22]–[24]. Our work sheds some light on the VMI techniques. We show that most of the semantic knowledge extracted by VMI cannot be trusted, and we call for more trustworthy VMI techniques.

VII. Limitations and Future Work

In this paper, we attempt to identify the semantic fields susceptible to mutation by an attacker. Though we identify several fields that are mutable both in Windows and Linux OSes, the list of such mutable semantics is not close to being complete. A thorough and complete analysis of all the semantic fields and their mutability is needed. Moreover, while most semantic fields have a direct correlation with the kernel data structures, it is not always the case. It is possible that a semantic field is derived as a result of one or more operations on multiple data structures. We intend to address such cases in future work. Furthermore, we have considered freely and easily available security tools in our experiments to detect the impacts of DKOM. However, a more complete result will include the impact on anti-virus software. We intend to include some AV software in our future experiments.

Trustworthy VMI needs to be dynamic in nature and be more involved with the guest kernel execution. Instead of querying semantic values that are statically available in memory, a more trustworthy VMI should capture the moment when semantic values are created, modified, and deleted, and make sure these operations on these semantic values are not from attackers. We aim to explore this direction as our future work.

Certain attacks on kernel code modify the interpretation of different kernel data structures. Such modifications will alter the mapping between the semantic meaning and the kernel data structures. MOSS does not address such attacks.

VIII. Conclusion

In this paper, we conducted a systematic assessment on Semantic Value Manipulation attacks in two widely-used operating systems, Windows XP and Ubuntu Linux. In a prototype system MOSS, we implemented a new fuzz testing technique - *duplicate-value directed semantic field fuzzing* to explore space of SVM attacks. We evaluated 45 semantic fields for Windows and 41 fields for Ubuntu Linux and conducted a total of 258 tests. Our results demonstrate that most of the security sensitive semantic fields can be freely mutated for both operating systems. Furthermore we found that consistency checking for duplicate values only help in some cases but not all. We also implemented a proof-of-concept SVM rootkit, which manipulated all mutable semantic values regarding a realworld bot sample TDSS. The selected security tools have been misled or worse - completely bypassed.

Our study implies that memory forensics and the current VMI techniques will completely fail if attackers fully exploit the power of SVM attacks. We call for revisiting of the existing security solutions and motivate serious research study for effective SVM attack mitigation.

Acknowledgement

We would like to thank anonymous reviewers for their comments. This research was supported in part by NSF grant #1018217, NSF grant #1054605, McAfee Inc, and VMware Inc. Any opinion, findings, conclusions, or recommendations are those of the authors and not necessarily of the funding agencies.

References

[1] "FU Rootkit," http://www.rootkit.com/project.php?id=12, 2005.

[2] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of ACM CCS*, 2009.

[3] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of ACM CCS conference*, 2009.

[4] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Proceedings of NDSS*, 2011.

[5] W. Cui, M. Peinado, Z. Xu, and E. Chan, "Tracking rootkit footprints with a practical memory analysis system," in *Proceedings of USENIX Security Symposium*, Aug. 2012.

[6] S. Golovanov, "Analysis of tdss rootkit technologies," Securelist, Tech. Rep., Aug 2010. [Online]. Available: http://www.securelist.com/en/analysis/204792131

[7] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of SOSP*, 2007.

[8] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, 2008, pp. 1–20.

[9] M. C. Grace, Z. Wang, D. Srinivasan, J. Li, X. Jiang, Z. Liang, and S. Liakh, "Transparent protection of commodity os kernels using hardware virtualization," in *Conference on Security and Privacy in Communication Networks (SecureComm)*, 2010.

[10] J. Nick L. Petroni and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of ACM CCS*, 2007.

[11] Z. Wang and X. Jiang, "Countering persistent kernel rootkits through systematic hook discovery," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*, September 2008.

[12] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook prevention," in *Proceedings of ACM CCS*, 2009.

[13] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, Hyderabad, India, Dec. 2008.

[14] "TEMU: The BitBlaze dynamic analysis component," http://bitblaze.cs.berkeley.edu/temu.html.

[15] "Volatility: Memory Forencis System," https://www.volatilesystems.com/default/volatility/.

[16] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of 12th Annual NDSS conference*, 2005.

[17] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990.

[18] P. Godfroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[19] M. Russinovich, "Windows sysinternals utilities," http://technet.microsoft.com/en-us/sysinternals.

[20] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.

[21] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)*, October 2007.

[22] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2011.

[23] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2012.

[24] ——, "Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery," in *Proceedings of the 9th Annual International Conference on Virtual Execution Environments*, Houston, TX, March 2013.

[25] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of OSDI 2008*, 2008.

[26] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '11. New York, NY, USA: ACM, 2011, pp. 265–278.

[27] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009, pp. 94–109.

[28] M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin, "Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis," in *In Proceedings of 19th Annual Network & Distributed System Security Symposium*, 2012.

[29] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, February 2010.

[30] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a dynamic excavator for reverse engineering data structures," in *Proceedings of NDSS 2011*, San Diego, CA, 2011.

[31] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, 2001.

[32] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[33] F. Baiardi and D. Sgandurra, "Building trustworthy intrusion detection through vm introspection," in *Proceedings of the Third International Symposium on Information Assurance and Security*. IEEE Computer Society, 2007, pp. 209–214.