In Proceedings of the 2012 International Conference on Network and System Security (NSS 2012), Fujian, China, Nov 21-23 2012, LNCS, Springer.

Identifying OS Kernel Objects for Run-time Security Analysis

Amani S. Ibrahim, James Hamlyn-Harris, John Grundy and Mohamed Almorsy

Centre for Computing and Engineering Software Systems
Faculty of ICT, Swinburne University of Technology
Melbourne, Australia
[aibrahim, jhamlynharris, jgrundy, malmorsy]@swin.edu.au

Abstract. As dynamic kernel runtime objects are a significant source of security and reliability problems in Operating Systems (OSes), having a complete and accurate understanding of kernel dynamic data layout in memory becomes crucial. In this paper, we address the problem of systemically uncovering all OS dynamic kernel runtime objects, without any prior knowledge of the OS kernel data layout in memory. We present a new hybrid approach to uncover kernel runtime objects with nearly complete coverage, high accuracy and robust results against generic pointer exploits. We have implemented a prototype of our approach and conducted an evaluation of its efficiency and effectiveness. To demonstrate our approach's potential, we have also developed three different proof-of-concept OS security tools using it.

Keywords: Operating Systems, Kernel Data Structures, Runtime Objects

1 Introduction

An OS kernel has thousands of heterogeneous data structures that have direct and indirect relations between each other with no explicit integrity constraints, providing a large attack surface to hackers. In Windows and Linux Operating Systems (OSes), from our analysis nearly 40% of the inter-data structure relations are pointer-based relations (indirect relations), and 35% of these pointer-based relations are generic pointers (*e.g.* null pointers that do not have values, and void pointers that do not have associated type declarations in the source code). Such generic pointers get their values or type definitions only at runtime according to the different calling contexts in which they are used [1]. In such a complex data layout, the runtime memory layout of the data structures cannot be predicted during compilation time. This makes the kernel data a rich target for rootkits that exploit the points-to relations between data structure instances in order to hide or modify system runtime objects. Hence, accurately identifying the running instances of the OS kernel data structures and objects is an important task in many OS security solutions such as kernel data integrity checking [2], memory forensics [3], brute-force scanning [4], virtualization-aware security solutions [5], and anti-malware tools [6]. Although discovering runtime objects has been

an aim of many OS security research efforts, these still have many limitations. Most fall into two main categories: *Memory Mapping Techniques* and *Value Invariants Approaches*.

Memory mapping techniques identify kernel runtime objects by recursively traversing the kernel address space starting from the global variables and then follow pointer dereferencing until reaching running object instances, according to a predefined kernel data definition – for each kernel version – that reflects the kernel data layout in the memory [5,7,8]. However, such techniques are limited and not very accurate. They are vulnerable to a wide range of kernel rootkits that exploit the points-to relations between data structures instances to hide the runtime objects or point to somewhere else in the kernel address space. They require a predefined definition of the kernel data layout that accurately disambiguates indirect points-to relations between data structures, in order to enable accurate mapping of memory. However – to the best of our knowledge – all of the current efforts (with the exception of KOP [7] and SigGraph [4]) depend on security expert knowledge of the kernel data layout to manually resolve ambiguous points-to relations. Thus, these approaches only cover 28% of kernel data structures (as discussed by Carbone *et al*. [7]) that relate to well-known objects. They are also not effective when memory mapping and object reachability information is not available. Sometimes security experts need to make a high-level interpretation of a set of memory pages where the mapping information is not available *e.g.* system crash dumps. Incomplete subsets of memory pages cannot be traversed, and data that resides in the absent pages cannot be recovered. They have a high performance overhead because of poor spatial locality, as the problem with general-purpose OS allocators is that objects of the same type could scatter around in the memory address space. Thus traversal of the physical memory requires accessing several memory pages. Finally, they cannot follow generic pointer dereferencing as they leverage type definitions, thus can not know the target types of untyped pointers.

Value-invariants approaches such as DeepScanner [9], DIMSUM [10] and SigGraph [4], use the value invariants of certain fields or of a whole data structure as a signature to scan the memory for matching running instances. However, such a signature may not always exist for a data structure [4]. Moreover, many kernel data structures cannot be covered by such value-invariant schemes. For example, it is difficult to generate value-invariants for data structures that are part of linked lists (single, doubly and triply), because the actual running contents of these structures depend on the calling contexts at runtime. In addition, such approaches do not fully exploit the rich generic pointers of data structures' fields, and are not able to uncover the points-to relations between the different data structures. The performance overhead of these approaches is extremely high, as they must scan the whole kernel address space with large signatures and they typically include most data structure fields in the signature.

Motivated by the limitations of these current approaches and the need to accurately identify runtime kernel objects from a robust view that cannot be tampered with, we have developed a new approach called DIGGER. DIGGER is capable of systematically uncovering all system runtime objects without any prior knowledge of the operating system kernel data layout in memory. Unlike previous approaches, DIGGER is designed to address the challenges of indirect points-to relations between kernel data

structures. DIGGER employs a hybrid approach that combines new value-invariant and memory mapping approaches, in order to get accurate results with nearly complete coverage. The value-invariant approach is used to discover the kernel objects with no need of memory mapping information, while the memory mapping approach is used to retrieve the object's details in depth including *points-to* relations (direct and indirect) with the other running data structures without any prior knowledge of the kernel data layout in memory. DIGGER first performs offline static *points-to* analysis on the kernel's source code to construct a type-graph that summarizes the different data types located in the kernel along with their connectivity patterns, and the candidate target types and values of generic pointers. This type-graph is used to enable systematic memory traversal of the object details not to discover running object instances. Second, DIGGER uses the four byte pool memory tagging schema as a new value-invariant signature – that is not related the data structure layout – to uncover kernel runtime objects from the kernel address space.

DIGGER's approach has accurate results, low performance overhead, fast and nearly complete coverage, and zero rate of false alarms. We have implemented a prototype system of DIGGER and evaluated it on the Windows OS to prove its efficiency in discovering: *(i)* kernel runtime objects; *(ii)* terminated objects that still persist in the physical memory; and *(iii)* semantic data of interest in dead memory pages. To demonstrate the power of DIGGER, we also have developed and evaluated three OS security prototype tools based on it, namely, B-Force, CloudSec+ and D-Hide. B-Force is a brute force scanning tool. D-Hide is a tool that can systematically detect any hidden kernel object type not just limited to the well-known objects. CloudSec+, a virtual machine (VM) monitoring tool, is used in virtualization-aware security solutions to externally monitor and protect VM's kernel data.

Section 2 gives an overview on the kernel data problem and review of key related work. Section 3 presents our DIGGER approach and section 4 explores its implementation and evaluation. Finally we discuss results and draw key conclusions.

2 Background

In OSes we usually refer to a running instance of a data structure (or a data type) as an *object*. Locating dynamic kernel objects in memory is the most difficult step towards enabling implementing different OS security solutions, as discussed above. Efficient security solutions should not rely on the OS kernel memory or APIs to extract runtime objects, as they may be compromised and thus gives false information. On the other hand, the complex data layout of an OS's kernel makes it challenging to uncover all system objects. Previous solutions limit themselves to the kernel static data *e.g.* system call and descriptor tables [11], or can reach only a fraction of the dynamic kernel data [2,12], resulting in security holes and limited protection.

It is challenging to check the integrity of kernel dynamic data due to its volatile nature. Dynamic data structures change during system runtime in location, values and number of running instances. Moreover, modifications to kernel dynamic data violate integrity constraints that in most cases cannot be extracted from OS source code. This

is because the data structure syntax is controlled by the OS code while their semantic meaning is controlled by runtime calling contexts. Thus, exploiting dynamic data structures will not make the OS treat the exploited structure as an invalid instance of a given type, or even detect hidden or malicious objects. For example, Windows and Linux keep track of runtime objects with the help of linked lists. A major problem with these lists is use of C null pointers. Modifications to null pointers violate intended integrity constraints that cannot be extracted from source code as they depend on calling contexts at runtime. This makes it easy to unlink an active object by manipulating pointers and thus the object becomes invisible for the kernel and for monitoring tools that depend on kernel APIs *e.g.* HookFinder [6] or memory traversal such as KOP [7], CloudSec [5], and OSck [8] – in addition to the limitations discussed above.

DeepScanner [9], DIMSUM [10], Gilbraltar *et al.* [2], and Petroni *et al.* [12] – as value-invariant approaches – are limited in that their authors depend on their knowledge with the kernel data layout, making their approach limited to a few structures. Also these tools do not consider the generic pointer relations between structures, making their approach imprecise and vulnerable a wide range of attacks that can exploit the generic pointers, in addition to high performance overhead due to large signatures.

To the best of our knowledge all existing approaches, whether value-invariant or memory traversal – with the exception of KOP [7], and SigGraph [4] – depend on the OS expert knowledge to provide kernel data layout definition that resolves the points-to relations between structures. SigGraph follows a systematic approach to define the kernel data layout, in order to perform brute force scanning using the value-invariant approach. However, it only resolves the direct points-to relations between data structures without the ability to solve generic pointers ambiguities, making their approach unable to generate complete and robust signatures for the kernel. KOP is the first and only tool that employs a systematic approach to solve the indirect points-to relations of the kernel data. However, KOP is limited in that: the points-to sets of the void * objects are not precise and thus they use a set of OS specific constraint at runtime to find out the appropriate candidate for the objects. KOP assumes the ability to detect hidden objects based on the traditional memory traversal techniques which are vulnerable to object hiding. Moreover, both KOP and SigGraph have very high performance overhead to uncover kernel runtime objects in a memory snapshot.

Rhee *et al.* [13] propose an interesting approach to detect runtime objects by analysing object allocation and reallocation instructions executed. However their approach has quite high performance overhead and thus cannot be used in traditional OS security tools – only for advanced debugging tools. Also, despite the feature of detecting allocations and deallocations in near real time, they cannot identify the object type. They need to analyse executed instructions offline for object type and details.

3 DIGGER Architecture

DIGGER's goal is to systematically uncover all kernel running objects in a memory snapshot or from a running VM without any prior knowledge of the kernel data layout. The high-level process of DIGGER is shown in Fig. 1. DIGGER has

three main components: *Static Analysis Component, Signature Extraction Component and Dynamic Memory Analysis Component,* discussed below in detail.

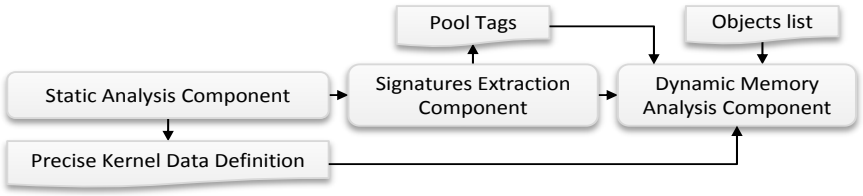

Fig. 1. The high-level process of DIGGER approach.

3.1 Static Analysis Component

Performing static analysis on the kernel source code is the key to automate the process of extracting kernel objects' details without any prior knowledge of the kernel data layout. DIGGER first performs static *points-to* analysis on the kernel's source code to systematically solve the ambiguous *points-to* relations between kernel data structures by inferring the candidate target types and values for generic pointers. The result of the *points-to* analysis step is a kernel data definition represented as a type-graph. This type-graph precisely models data structures and reflects accurately both direct and indirect relations that reflect the memory layout of the data structures. The type-graph is **not** used to uncover kernel objects - it is used to retrieve running objects' detailed type structure. The level of details is selected by tool users based on the required hierarchy depth. This controls the trade-off between details and performance overhead, as some object types have hundreds of hierarchically-organised fields.

We build this type-graph using our tool KDD [14,15]. KDD performs interprocedural, context-sensitive, field-sensitive and inclusion-based *points-to* analysis on the kernel source code. KDD is able to perform highly precise and scalable *points-to* analysis for large C programs that contain millions lines of code *e.g.* OS kernel, without any prior knowledge of the OS structure.

3.2 Signature Extraction Component

It is difficult to obtain robust signatures for kernel data structures for the following reasons: **First**, data structure sizes are not small. From our analysis for Windows and Linux, we found that a single data structure could be several hundreds of bytes. Such big signatures increase the discovery cost and the performance overhead. **Second**, it is difficult to identify which fields of a target data structure can be used as scanning signatures to effectively detect stealthy malware and be difficult to be evaded. **Third**, the OS kernel contains thousands of data structures, making the process of generating "unique" signatures for this huge number of structures very challenging.

DIGGER makes use of the pool memory tagging schema of the kernel object manager to overcome the first two problems, and is motivated by the below paragraph from Windows internals book [16] (we call it WI-note) to overcome the third problem – details discussed below: "*Not all data structures in the OS are objects. Only data*

that needs to be shared or made visible to user is placed in objects. Structures used by one component of the OS to implement internal functions are not objects".

Windows kernels use pool memory to allocate the kernel objects. The pool memory can be thought of as a kernel-mode equivalent of the user-mode heap memory. When the object manager allocates a memory pool block, it associates the allocation with a pool tag – a pool tag is a unique four byte tag for each object type. We use this tag as a value-invariant signature to uncover the kernel objects running instances. The pool tag list for the Windows OS can be extracted from the symbol information *e.g.* Microsoft Symbols. However, the pool tag is not enough to be an object signature. For instance, if we have a pool tag "Proc" and we scan the memory using the ASCII code of this pool tag, any word that has the same ASCII string will be detected as an object instance from that object type. Thus we need to add another checking signature that guarantees accurate results. We make use of the object dispatcher header (each allocated object starts with a dispatcher header that is used by the OS to provide synchronization access to resources). The first three bytes of the dispatcher header is unique for each object type, as they describe an object's type and size. These three bytes can be calculated from the generated type-graph (from our static analysis component). From our experiments, we found that those three bytes are static and cannot be changed during object runtime. Key features of using pool tags as signatures are: *(i)* not being tied to data structure layout and thus effective in different OS kernel versions where data structure layout change may occur; and *(ii)* the very small size of the signature decreases performance overhead significantly.

To the best of our knowledge, all current OS security research for Windows and Linux treat all data structures as objects and do not consider the WI-note. This WI-note enables filtering of the list of data structures extracted at the static analysis step to obtain a list of actual objects. Each data structure that has a pool tag used by the Windows allocators is considered as an object and the other data structures are not. This massively reduces the number of object types from thousands to dozens. This solves the problem of generating unique signatures for such a huge kernel data structures size (the third obstacle), and also frees resources for analysis of the most important data structures. For the other data structures (non-objects that are less important than objects), we use the type-graph to uncover these data structures using the *points-to* relations of these data structures with the uncovered objects.

3.3 Dynamic Memory Analysis Component

The output of the memory analysis component is an object-graph whose nodes are instances of data structures and objects – in the memory snapshot – and edges are the relations between these objects. Using the pool tags and the additional checking signature, the dynamic memory component scans the kernel address space with eight byte granularity (default size of the pool header) to extract the runtime instances of the different kernel object types. However, until this step we can just identify that there is a running object instance of type *T* but we cannot know any details about the object itself or even the object name. When an object is being allocated by the object

manager it is prefixed by an object header and the whole object (including the object header) is prefixed with a pool header data structure, as shown in Fig. 2.

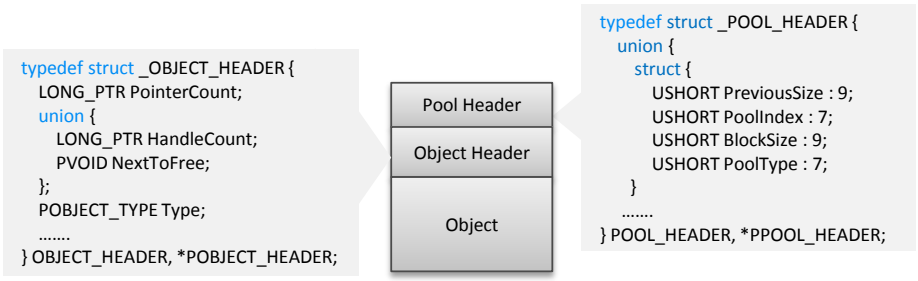

Fig. 2. The memory layout of allocated objects in the pool memory.

The pool header is a data structure used by the Windows object manager to keep track of memory allocations. The most important fields in the pool header are the pool tag and the block size. These fields help in our algorithm to extract the object details as follows: **First**, Pool Tag; by subtracting the offset *f* of the pool tag field from the address *x* where an object has been detected (using the pool tag and the additional checking signature), we can get the pool block start memory address *y*. By adding the size of the pool header and the object header to *y*, we can calculate the object's start address. Then we retrieve the object's details based to our generated kernel type-graph – from the static component – by traversing the kernel memory. The size of the pool and object headers are calculated from the kernel type-graph. **Second**, Block Size; the block size field indicates the pool block size *s* that has been allocated for an object *O*. This field helps to speed up the scan process, by skipping *s* bytes of memory starting from the *y* address to reach the start address of next pool block or a kernel memory address.

We have two strategies for uncovering running kernel objects. **First**, for memory images; the size of a complete memory image is quite big and the kernel address space ranges from 1GB to 2GB in 32bit OSs and up to 8TB in 64bit OSs according to the memory layout used by the hardware and the available hardware memory. Scanning such a huge number of memory pages is too expensive. To solve this problem and get the fastest coverage for the kernel address space, we scan only the pool memory instead of the whole kernel address space. There are two distinct types of pool memory in Windows OS: paged pool and nonpaged pool. Both are used by the kernel address space to store the kernel and executive objects. The nonpaged pool consists of virtual memory addresses that are guaranteed to reside in physical memory as long as the corresponding kernel objects are allocated. The kernel uses the nonpaged pool memory to store the runtime objects that may be accessed when the system cannot handle page faults *e.g.* processes, threads and tokens. The paged pool consists of virtual memory that can be paged in and out of the system. This means that scanning the nonpaged pool memory is a trusted source to get all the running objects instances that are potential target for hackers as it always resides in physical memory. On uniprocessor or multiprocessor systems there exists only one nonpaged pool and

this number can be confirmed using the global variable *nt!ExpNumberOfNonPagedPools*. The OS maintains a number of global variables that define the start and end addresses of the paged and nonpaged pool memory: *MmPagedPoolStart*, *MmPagedPoolEnd*, *MmNonPagedPoolStart* and *MmNon-PagedPoolEnd*. These pointers can be used to speed up the scanning by limiting the scanned area. From our observations, we found pool memory size is around 4.5% and 8% from the kernel address space in 32-bit and 64-bit OS, respectively. **Second**, for un-mappable memory pages; in this case, the size of pages set is relatively small. We perform a scan on the whole set of the memory pages using the pool tag and the additional checking signature. However, as the memory mapping information may not be available in such un-mappable memory pages, not all of the details for the discovered objects can be retrieved as we depend on the memory traversal technique according to the generated type-graph.

4 Implementation and Evaluation

We have developed a prototype of DIGGER. The static analysis component was built using our previously developed tool, KDD [1,14]. The signatures and runtime components are standalone programs and all components are implemented in C#. The runtime component works: *(i)* offline on memory snapshot, raw dumps (*e.g.* dumps in the Memory Analysis Challenge and Windows crash dumps), and VMware suspended sessions. *(ii)* Online in a virtualized environment by scanning VMs' physical memory from the hypervisor level. We have evaluated the basic functionality of DIGGER with respect to the identification of kernel runtime objects and the performance overhead of uncovering these objects. We performed different experiments and implemented different OS security prototype tools to demonstrate DIGGER's efficiency. In section 4.1 we evaluate the static and runtime components, and their performance overhead. In section 4.2, we explore the implemented OS prototype tools, and finally in section 4.3 we discuss the main features and limitations of DIGGER.

4.1 Uncovering Objects

For the static analysis component, we applied KDD's static analysis to the source code of the Windows Research Kernel (WRK[1]) (a total of 3.5 million lines of code), and found 4747 type definitions, 1858 global variables, 1691 void pointers, 2345 null pointers, 1316 doubly linked list and 64 single linked lists. KDD took around 28 hours to complete the static analysis on a 2.5 GHz core i5 processor with 12 GB RAM. As our analysis was performed offline and just once on each kernel version, the performance overhead of analyzing kernels is acceptable and does not present any problem for any security application using KDD. The performance overhead of KDD could be decreased by increasing the hardware processing capabilities, as such types of analysis usually run with at least 32 GB RAM.

[1] WRK is the only available source code for Windows.

To enable efficient evaluation for the runtime component, we need a ground truth that specifies the exact object layout in kernel memory so that we can compare it with the results of DIGGER to measure false positive rates. We build the ground truth as follows: we extracted all data structure instances of the running Windows OS memory image via program instrumentation using the Windows Debugger (WD). We instrumented the kernel to log every pool allocation and deallocation, along with the address using the WD. In particular, we modified the GFlags (Global Flags Editor) to enable advanced debugging and troubleshooting features of the pool memory. We then measured DIGGER efficiency by the fraction of the total allocated objects for which DIGGER was able to identify the correct object type. We performed experiments on 3 different versions of the Windows OS on a 2.8 GHz CPU with 2GB RAM. Each memory snapshot size was 4GB. Table 1 shows the results of DIGGER and WD in discovering the allocated instances for specific object types in two of the three Windows versions (not showing all objects, for brevity).

Table 1. Experimental results of DIGGER and WD on Windows XP 32 bit and 64bit. Memory, paged and nonpaged columns reprsent the size in pages (0x1000 graunrality) of the kernel address space, paged pool and nonpaged pool, repectively. WD and DIG refer to WD's and DIGGER results. FN, FP and FP* denote the false negative, reported false positive and the actual false poitive rates, repectively.

Object	Windows XP 32bit					Windows XP 64bit				
	Memory		Paged		Nonpaged	Memory		Paged		Nonpaged
	915255		27493		11741	1830000		35093		17231
	WD	DIG.	FN %	FP %	FP*%	WD	DIG.	FN %	FP %	FP*%
Process	119	121	0.00	1.65	0.00	125	125	0.00	0.00	0.00
Thread	2032	2041	0.00	0.44	0.00	2120	2121	0.00	0.04	0.00
Driver	243	243	0.00	0.0	0.00	211	211	0.00	0.00	0.00
Mutant	1582	1582	0.00	0.0	0.00	1609	1609	0.00	0.00	0.00
Port	500	501	0.00	0.19	0.00	542	542	0.00	0.00	0.00

From table 1 we can see that DIGGER achieves zero false negative rates, and a low false positive rate. However, from our manual analysis of the results, we found that this reported false positive rate is not an actual false positive. This difference represents deallocated objects that still persist in the physical memory after termination; we call these "dead memory pages objects – DMAO". They are present because the Windows OS does not clear the contents of memory pages to avoid the overhead of writing zeroes to the physical memory. However, we noticed from our analysis that the pointer and handle count of the DMAO is always zero. This enables differentiating the active objects from the DMAO, and thus our actual false positive rate becomes zero (FP*). We argue this finding thus: whenever the kernel has to allocate a new object it will return the pool block address from the pool free list head. For example, the EPROCESS structure of a newly created process will overwrite the object data of a process that has been terminated previously. This because when a block is freed using the `free` function call, the allocator just adds the block to the list of free blocks without overwriting memory. These DMAO can provide forensic information about an attacker's activity. Imagine an attacker runs stealthy malware and then terminates

it on a victim machine. After the termination there may still exist for a non-trivial period of time some forensic data of interest in the dead memory pages. To prove our assumption, we analyzed the dead memory pages in order to uncover semantic data of interest for the some terminated processes. However, our approach can work for any other object type. We used some benchmark programs to run in three memory images and then analyzed the dead memory pages to uncover some data of interest: user login information (GroupWise email client), chat sessions (Yahoo messenger), FTP sessions (FileZilla). We created 9 processes (three of these are the benchmark programs) and performed some CPU-intensive operations using these processes. We terminated these processes after 5 hours, 2 hour and 15 minutes in three different memory images – identified L, M and S, respectively. Then we created 4 different (new) processes 5 minutes after termination. The memory images were then scanned for runtime objects using DIGGER's runtime component. We found that 3 from the terminated processes' physical addresses were overwritten by EPROCESS structure for new processes, while another three processes (from the terminated ones) still persisted in memory (at the same address in the memory). We made the following observations. First, for the email client we were not able to identify the login information (user name and password) for all of the memory images. For the ftp client we were able to identify the server name, and the server and client connection ports for the S image only, without any ability to locate the login credentials in all of the three images. For the chat benchmark application, we were able to locate the username, the connection port and some recent chat sessions in the S image only. This data recovery approach is not effective if the program zeros its memory pages before termination.

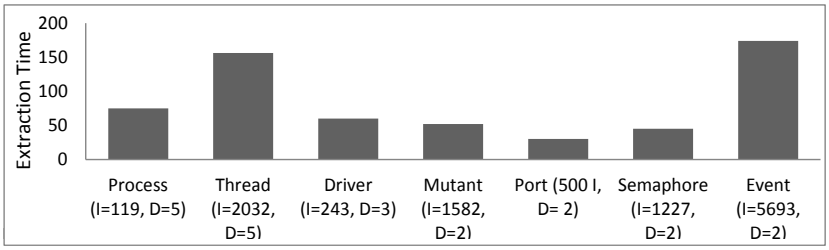

Fig. 3. Object details extraction normalized time.

We have evaluated DIGGER's runtime performance to demonstrate that it can perform its memory analysis in a reasonable amount of time. We measured DIGGER running time when analyzing the memory snapshots used in our experiments. The median running time was around 0.8 minutes to uncover 12 different object types from the nonpaged pool, and 1.6 minutes to uncover another 15 object type from the paged pool. This time included the time of loading the memory snapshot from the disk to the runtime analysis component. We consider this running time to be acceptable for offline analysis and even for online analysis in virtualized environments. This is because DIGGER is able to detect the DMAO that could be created and terminated between the scan time intervals. However, we cannot argue that it would be 100% accurate. Comparing DIGGER with SigGraph [4], DIMSUM [10], KOP [7],

CloudSec [5]: DIGGER is the fastest with highest coverage and lowest performance overhead. The performance overhead of extracting object details based on our generated type-graph differs according the required details-depth. Fig. 3 shows the time consumed (in seconds) to extract object details with different depths for all of the running instances from a specific object type. "I" denotes the number of the running objects from the object, and "D" denotes the depth of the extracted details.

4.2 Security Applications

We have also evaluated DIGGER by developing three prototype OS security tools to demonstrate its efficiency. We chose these applications because they address common important OS security tools. Our experiments with these tools have demonstrated DIGGER's efficiency and the false alarms rate is similar to what discussed in table 1.

Object Hiding Detection. Previous efforts have focused on detecting specific types of hidden objects by hardcoding OS expert knowledge of the kernel data layout [12]. Other approaches rely on value-invariants such as the matching of process list with the thread scheduler [17]. Other approaches are based on logging malware memory accesses [18,19] and provide temporal information. However they can only cover known attacks and cannot properly handle zero-day threats. All of these approaches are time-consuming, and require a human expert with deep knowledge of the system to create the rules and thus cannot cover all system objects. There are some approaches such as Antfarm [20] that track the value of the CR3 register. Although this approach is useful in a live environment, it cannot be used for memory forensics applications and the performance overhead of such an approach is very high. Given DIGGER's ability to uncover kernel objects, we developed a tool called D-Hide that can systematically uncover all kinds of stealthy malware (not just limited to specific object type, as done to date), by detecting their presence in the physical memory. We used DIGGER's approach to uncover the runtime kernel objects and then perform "external" cross-view comparisons with the information retrieved from mapping the physical memory using our generated type-graph. In other words, the first view is DIGGER's view and the other view is the memory traversal view (we start from the OS global variables and then follow pointer dereferencing until we cover all memory objects). Discrepancies in this comparison reveal hidden kernel objects. We implemented a memory traversal add-on for the runtime component that takes our generated type-graph and based on that graph, we traverse the kernel address space. We evaluated D-Hide ability to identify hidden objects with four real-world kernel rootkits samples: FURootkit, FuToRootkit, AFX Rootkit and HideToolz. We used WinObj (windows internal tool) to compare the results with D-hide. D-hide correctly identified all hidden objects with zero false alarms. D-Hide has two key advantages: *(i)* No need for deep knowledge of the kernel data layout, as it depends on DIGGER static component to get an accurate kernel data layout. *(ii)* D-Hide can perform cross-view comparison without the need for any internal tools *e.g.* task manager or WinObj that gets the internal view, as done in the current cross-view researches. This feature enables deploying D-hide in VMs hosted in the cloud platform where the cloud providers do not have any control over VMs, as discussed in [11]. *(iii)* D-Hide is unlike

previous tools [21,17] that rely on authors' knowledge of the kernel data and thus is not limited to specific objects.

Brute Force Scanning Tool. Given a range of memory addresses and a signature for a data structure or object, brute force scanning tools can decide if an instance of the corresponding data structure exists in the memory range or not [4]. Brute force scanning of kernel memory images is an important function in many operating system security and forensics applications, used to uncover semantic information of interest *e.g.* passwords, hidden processes and browsing history from raw memory. Given DIGGER's ability to uncover kernel objects, we developed B-Force – a brute force tool. From our experiments with five different small crash dumps of small sizes ranging from 12MB to 800MB, we found out that this method is highly effective with zero rates of alarms. However, this method could reveal false positives if the memory page set does not contain the pool header (that contains the pool tag) of the pool block along with the first three bytes of the object itself (to perform the additional signature checking). However, from our point of view this is unlikely, as the single memory page size is big enough to contain tens of pool blocks.

Virtual Machines Monitoring. We modified our earlier-developed VM monitoring tool, *CloudSec* [5], to use DIGGER to online analyze a Virtual Machine's (VM) memory of a running OS and extract all kernel running objects. *CloudSec* is a security appliance that monitors a VMs memory from outside the VM itself, without installing any security code inside the VM. *CloudSec* successfully uncovered and correctly identified the running kernel objects, with zero rate false alarms. The performance overhead of *CloudSec* to uncover the entire kernel running objects was around 1.1, 1.9 and 2.8 minutes with 0-level, 1-level and 2-level depths, respectively for a VM with a 2.8 GHz CPU and 4GB RAM. VM was executed under normal workload (50 processes, 912 threads, etc.). We can see that the performance overhead of scanning a VM's memory online is less that scanning a memory image, as access to VM's memory *via* hypervisors is faster than uploading a memory image to the analysis tool. As CloudSec runs in its own separate VM and potentially on a separate host machine, it can do this analysis concurrently with the target VM continuing to run normally.

4.3 Discussion

DIGGER's approach provides a robust view of OS kernel objects not affected by the manipulation of actual kernel memory content. This enables development of different OS security applications as discussed in section 4.3, in addition to enabling systematic kernel data integrity checks based on the resultant object-graph. The key features of DIGGER include: **first**, the systematic approach it follows to extract OS kernel data layout and to disambiguate the points-to relations between data structures, all without any prior knowledge of the OS kernel data layout. **Second**, the robust and quite small signature size to uncover runtime objects, enhancing performance.

As the pool memory concept is related to Windows OSes, the current approach used in DIGGER's runtime component can only be used to analyze Windows OSes. DIGGER's runtime component is not related to a specific version of the Windows OS kernel and can work on either 32bit or 64 bit layout – SigGraph, DIMSUM and KOP

are also limited to a specific OS. However, the same approach can be used in Linux using the slab allocation concept. Slab allocation can be thought of as a pool memory equivalent of the Windows OS. Slab allocation is a memory management mechanism for Linux and UNIX OSes for allocating kernel runtime objects efficiently. The basic idea behind the slab allocator is having caches (similar to the pool blocks in Windows OS) of commonly used objects kept in an initialized state. The slab allocator caches the freed object so that the basic structure is preserved between uses to be used by a newly allocated object of the same type. The slab allocator consists of caches that are linked together on a doubly linked list called a *cache chain* that is similar to the list head of the pool memory used in Windows kernel. The static analysis component of DIGGER (KDD) can be applied on any C-based OS *e.g.* Linux, BSD and UNIX to perform highly detailed and accurate points-to analysis for the kernel data layout.

5 Summary

Current state-of-the-art tools are limited in accurately uncovering the running instances of kernel objects. We presented DIGGER, a new approach that enables uncovering dynamic kernel objects with nearly complete coverage and accurate results by leveraging a set of new techniques in both static and runtime components. Our evaluation of DIGGER has shown its effectiveness in uncovering system objects and in supporting the development of several OS security solutions.

6 Acknowledgement

The authors are grateful to Swinburne University of Technology and FRST Software Process and Product Improvement project for support for this research.

7 References

1. Ibrahim, A.S., Hamlyn-Harris, J., Grundy, J., Almorsy, M.: Supporting Virtualization-Aware Security Solutions using a Systematic Approach to Overcome the Semantic Gap. In: Proc. of 5th IEEE International Conference on Cloud Computing, Hawaii, USA 2012
2. Baliga, A., Ganapathy, V., Iftode, L.: Automatic Inference and Enforcement of Kernel Data Structure Invariants. In: Proc of 2008 Annual Computer Security Applications Conference 2008, pp. 77-86. IEEE Computer Society, 1468197
3. Andreas, S.: Searching for processes and threads in Microsoft Windows memory dumps. Digital Investigation **3**(1), 10-16 (2006).
4. Lin, Z., Rhee, J., Zhang, X.: SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In: Proc. of 18th Network and Distributed System Security Symposium, San Diego, CA 2011
5. Ibrahim, A.S., Hamlyn-Harris, J., Grundy, J., Almorsy, M.: CloudSec: A Security Monitoring Appliance for Virtual Machines in the IaaS Cloud Model. In: Proc. of 2011 International Conference on Network and System Security (NSS 2011), Milan, Italy 2011.

6. H. Yin, Z. Liang, and D. Song: HookFinder: Identifying and understanding malware hooking behaviors. In: Network and Distributed Systems Security Symposium (NDSS) 2008
7. Carbone, M., Cui, W., Lu, L., Lee, W.: Mapping kernel objects to enable systematic integrity checking. In: Proc of 16th ACM conference on Computer and communications security, Chicago, USA 2009, pp. 555-565. ACM, 1653729
8. Hofmann, O.S., Dunn, A.M., Kim, S.: Ensuring operating system kernel integrity with OSck. In: Proc. of 16th international conference on Architectural support for programming languages and operating systems, California, USA 2011, pp. 279-290. ACM, 1950398
9. Liang, B., You, W., Shi, W., Liang, Z.: Detecting stealthy malware with inter-structure and imported signatures. In: Proc. of the 6th ACM Symposium on Information, Computer and Communications Security, Hong Kong, China 2011, pp. 217-227. ACM, 1966941
10. Lin, Z., Rhee, J., Wu, C., Zhang, X., Xu, D.: Discovering Semantic Data of Interest from Un-mappable Memory with Confidence. In: Proc. of the 19th Network and Distributed System Security Symposium, San Diego, CA 2012
11. Ibrahim, A.S., Hamlyn-Harris, J., Grundy, J.: Emerging Security Challenges of Cloud Virtual Infrastructure. In: Proc. of 2010 Asia Pacific Cloud Workshop co-located with APSEC2010, Sydney, Australia 2010
12. Petroni, N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: Proc. of 15th conference on USENIX Security Symposium - Vol 15, Canada 2006.
13. Rhee, J., Riley, R., Xu, D.: Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In: Proc. of 13th international conference on Recent advances in intrusion detection, Ontario, Canada 2010, pp. 178-197. Springer-Verlag, 1894179
14. Ibrahim, A.S., Grundy, J.C., Hamlyn-Harris, J., Almorsy, M.: Supporting Operating System Kernel Data Disambiguation using Points-to Analysis. In: Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany 2012
15. Ibrahim, A.S., Hamlyn-Harris, J., Grundy, J., Almorsy, M.: Operating System Kernel Data Disambiguation to Support Security Analysis". In: Proc. of 6th International Conference on Network and System Security (NSS 2012), Fujian, China 2012. Springer
16. Russinovich, M., Solomon, D., Ionescu, A.: Windows Internals, 5th Edition. Microsoft Press, (2009)
17. Nanavati, M., Kothari, B.: Hidden Processes Detection using the PspCidTable. In. MIEL Labs, (2010, Accessed November 2010)
18. Riley, R., Jiang, X., Xu, D.: Multi-aspect profiling of kernel rootkit behavior. In: Proc. of the 4th ACM European conference on Computer systems, Germany 2009, pp. 47-60.
19. Xuan, C., Copeland, J., Beyah, R.: Toward Revealing Kernel Malware Behavior in Virtual Execution Environments. In: Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection, Saint-Malo, France 2009, pp. 304-325.
20. Jones, S., Arpaci-Dusseau, A., Arpaci, R.: Antfarm: tracking processes in a virtual machine environment. In: Proc. of USENIX '06 Annual Technical Conference, Boston, 2006.
21. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: VMM-based hidden process detection and identification using Lycosid. In: Proc. of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, USA 2008, pp. 91-100.