CHAPTER -1

INTRODUCTION

For security of system resources and confidential data that can be misused and manipulated by various malware attacks, there is a need of malware detection and its analysis in modern operating system kernels for better and safe computing. Typical operating system kernels enforce minimal restrictions on the applications permitted to execute, resulting in the ability of malicious programs to abuse system resources. Malware running as stand-alone processes or alongside of a process once installed, may freely execute privileges provided to the user account running the process or the application. So in this system, malware detection and analysis is necessary.

Malware is a very generic term that encompasses different types of malicious software components. Traditionally, malware was detected by using syntax-based signatures that attempt to detect a specific part of the representation of a malware instance. However, these techniques are not able to detect previously unseen malware components, and, most notably, polymorphic malware and malware that hides themselves in kernel data structure. Therefore, this research has focused on how to characterize malware using its behavior or structure, which is independent of the malware's particular representation.

Traditional malware detection and analysis approaches have been focusing on code-centric aspects of malicious programs, such as detection of the injection of malicious code or matching malicious code sequences. However, modern malware has been employing advanced strategies, such as reusing legitimate code or obfuscating malware code to circumvent the detection. As a new perspective to complement code-centric approaches, they have proposed Data-Centric OS Kernel Malware Characterization architecture that detects and characterizes malware attacks based on the properties of data objects manipulated during the attacks.

In Data-Centric OS Kernel Malware Characterization architecture the detection and characterization of malware attack is based on the properties of data objects manipulated during the attacks [1]. This approach is effective at detecting a class of malware that hides dynamic data objects. This approach also has an extended coverage that detects not only the malware with the signatures, but also the malware variants that share the attack patterns by modeling the low level data access behaviors as signatures.

1.1. MOTIVATION

We motivate our work through discussing and distinguishing related aspects. We first discussed about various malware attacks related to operating system security. The basic problem was the unauthorized access to system resources, like the user's personal data, kernel level data structures such as system Stack. So there were various solutions to secure such resources like detecting the code patterns of the malicious logic used but the malware hiding inside the kernel data structures can misuse the system data by using the system resources. That's why we discussed about protecting the system and its resources by doing Code-Centric OS Malware Characterization. But later on we come up with the fact that this is not the efficient and very secure way to deal with hackers, because there are various methods by which the way of program execution can be changed and malicious logic present in the code and be made hidden inside the kernel data structures. Because of these issues we find out a way to deal with this problem and that was Data-Centric OS Kernel Malware Characterization i.e. detection based on data objects used during the execution of a process.

1.2. DATA CENTRIC OS KERNEL MALWARE CHARACTERIZATION BASED ON LIVE KERNEL OBJECT MAPPING SYSTEM

In this seminar, the Data Centric approach as the missing link in achieving system security is pointed out. This work addresses to show the detection of malware based on manipulation of data objects used during the attack. The demonstration of the Data Centric approach is a crucial step to prevent execution of a process containing malicious logic from accessing and abusing system resources. In this solution, which is referred to as Malware Characterization based on Live Kernel Object Mapping System, a live object map is created on each execution of a process under observation [1]. The changes in the behaviour of data object manipulation by a benign process and a malicious process can be detected by the kernel. It can be used in OS such as Linux.

In this seminar, we discuss about the detection of various malware attacks and various analysis approaches in preventing unauthorized use of kernel data structures like system stack. The prototype consists of two Linux kernel modules viz. Live Kernel Object Mapping System and generation of malware signatures based on the data access patterns specific to malware attacks [1]. A Live Kernel Object Mapping System generates the pattern of data objects used by a benign process and a malicious process using a virtual monitor placed over the hypervisor in Linux OS.

1.3. SCOPE

This Kernel Object Mapping system can be conveniently used easily with existing operating systems. Basically this approach is used for handling the processes in Linux based operating systems, but it can also be integrated with existing policy-based access control systems for strong system assurance. At this level this approach can only be used for Data Centric OS where the detection of malware attacks is based on the data access pattern behaviour rather than on code injection or malicious code sequences and patterns.

CHAPTER -2

LITERATURE SURVEY

2.1. Malware Attacks

2.1.1. Basic Concept

Malware, short for malicious software, is any software used to disrupt computer operation, gather sensitive information, or gain access to private computer systems. Malware is defined by its malicious intent, acting against the requirements of the computer user [8].

Traditional malicious programs such as computer viruses, worms, and exploits have been using code injection attacks which inject malicious code into a program to perform a nefarious function. Intrusion detection approaches based on such code properties effectively detect or prevent this class of malware attacks. In response to these techniques, alternate attack vectors were devised to avoid violation of code integrity and therefore elude such detection approaches. For instance, return-to-libc attacks, return-oriented programming, and jump-oriented programming, reuse existing code to create malicious logic [1].

TABLE 2.1

Various Malware Attacks

Malware Attack	**Description**
Return-to-libc Attack	Computer security attack usually starting with a buffer overflow. Subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process' executable memory.
Return Oriented Programming	Computer security exploit technique that allows an attacker to execute code in the presence of security defenses.
Jump Oriented Programming	This new attack eliminates the reliance on the stack and ret instructions seen in return-oriented programming without sacrificing expressive power.

2.1.2. Return-to-libc Attacks

A "return-to-libc" attack is a computer security attack usually starting with a buffer overflow in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process' executable memory, rendering the NX bit feature useless (if present) and ridding the attacker of the need to inject their own code [6].

On Unix and Unix-like operating systems the C standard library is commonly used to provide a standard runtime environment for programs written in the C programming language. Although the attacker could make the code return anywhere, libc is the most likely target, as it is almost always linked to the program, and it provides useful calls for an attacker (such as the system function used to execute shell commands) [6].

A non-executable stack can prevent some buffer overflow exploitation, however it cannot prevent a return-to-libc attack because in the return-to-libc attack only existing executable code is used [6]. On the other hand these attacks can only call preexisting functions. Stack-smashing protection can prevent or obstruct exploitation as it may detect the corruption of the stack and possibly flush out the compromised segment.

Return-to-libc attack does not need an executable stack; it does not even use shell code. Instead, it causes the vulnerable program to jump to some existing code, such as the system() function in the libc library, which is already loaded into the memory [6].

Ubuntu and several other Linux-based systems use address space randomization to randomize the starting address of heap and stack [7]. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. So disable this feature using the following command:

```
$ sudo -s
Password: (enter your password)
# sysctl -w kernel.randomize_va_space=0
# exit
```

The GCC compiler implements a security mechanism called “Stack Guard” to prevent buffer overflows [7].

In the presence of this protection, buffer overflow will not work. You can disable this protection when you are compiling the program using the switch *-fno-stack-protector*. For example, to compile a program example.c with Stack Guard disabled, you may use the following command [7]:

```
$ sudo gcc -fno-stack-protector example.c
```

2.1.3. Return Oriented Programming

Return-oriented programming (ROP) is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses such as non-executable memory and code signing [5].

In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences, called "gadgets". Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code [5]. Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine employing defenses that thwart simpler attacks.

Return-oriented programming is an effective code-reuse attack in which short code sequences ending in a ret instruction are found within existing binaries and executed in arbitrary order by taking control of the stack. This allows for Turingcomplete behavior in the target program without the need for injecting attack code, thus significantly negating current code injection defense efforts. On the other hand, its inherent characteristics, such as the reliance on the stack and the consecutive execution of return oriented gadgets, have prompted a variety of defenses to detect or prevent it from happening [2].

While return-into-libc is powerful, it does not allow arbitrary computation within the context of the exploited application. For this, the attacker may turn to *ROP*. As before, ROP overwrites the stack with return addresses and arguments [5]. However, the addresses supplied now point to arbitrary points within the existing code base, with the only requirement being that these snippets of code, or *gadgets*, end in a ret instruction to transfer the control to the next gadget. Return oriented programming is driven by the insight that return addresses on the stack can point *anywhere*, not just to the beginning of functions like in a classic return-into-libc attack [5]. Therefore, it can direct control flow through a series of small snippets of existing code, each ending in ret. These small snippets of code are called *gadgets*, and in a large enough codebase (such as libc), there is a massive selection of gadgets to choose from. On the x86 platform, the selection is made even larger because instructions are of variable length, so the CPU will interpret the raw bytes of an instruction differently if decoding is started from a different offset. Based on this, the return-oriented program is simply a sequence of gadget addresses and data values laid out in the vulnerable program's memory [5].

Return-oriented programming is an advanced version of a stack smashing attack. Generally, these types of attacks arise when an adversary manipulates the call stack by taking advantage of a bug in the program, often a buffer overrun. A return-oriented programming attack is superior to the other attack types discussed both in expressive power and in resistance to defensive measures [5]. None of the counter-exploitation techniques, including removing potentially dangerous functions from shared libraries altogether, are effective against a return-oriented programming attack.

As in ROP, a jump-oriented program consists of a set of gadget addresses and data values loaded into memory, with the gadget addresses being analogous to opcodes within a new jump oriented machine. In ROP, this data is stored in the stack, so the stack pointer esp serves as the “program counter" in a return-oriented program (Fig.). The “program counter" is any register that points into the dispatch table. Control flow is driven by a special *dispatcher gadget* that executes the sequence of gadgets. At each invocation, the dispatcher advances the virtual program counter, and launches the associated gadget [2].

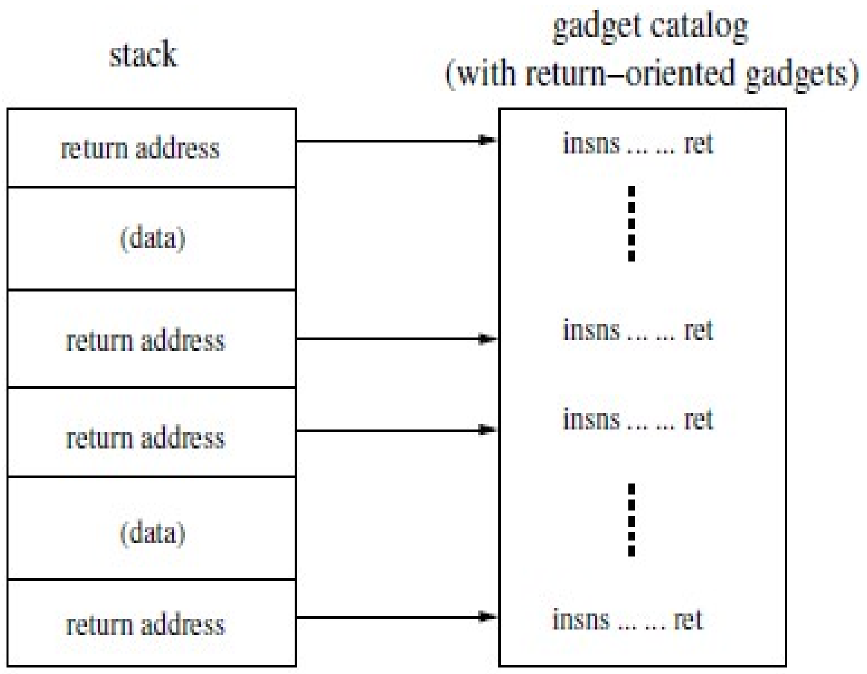

Fig 2.1 The ROP Model [2]

2.1.4 Jump Oriented Programming

Jump Oriented Programming is a new class of code-reuse attack. Jump oriented programming is one of the most up-to-date form of the memory corruption attacks. During this kind of attack the attacker tries to achieve his goal by using library files linked to the binary, without the placing of any own code [2]. This new attack eliminates the reliance on the stack and ret instructions (including ret-like instructions such as pop+jmp) seen in return-oriented programming without sacrificing expressive power. This attack still builds and chains *functional gadgets*, each performing certain primitive operations, except these gadgets end in an indirect branch rather than ret [2]. Without the convenience of using ret to unify them, the attack relies on a *dispatcher gadget* to dispatch and execute the functional gadgets.

In a JOP-based attack, the attacker abandons all reliance on the stack for control flow and ret for gadget discovery and chaining, instead using nothing more than a sequence of indirect jump instructions. Because almost all known techniques to defend against ROP depend on its reliance on the stack or ret, none of them are capable of detecting or defending against this new approach. The one exceptions are systems that enforce full control-flow integrity; unfortunately, such systems are not widely deployed, likely due to concerns over their complexity and negative performance impact [2].

JOP is not limited to using esp to reference its gadget addresses, and control flow is not driven by the ret instruction. Instead, JOP uses a dispatch table to hold gadget addresses and data [2]. The “program counter" is any register that points into the dispatch table. Control flow is driven by a special *dispatcher gadget* that executes the sequence of gadgets. At each invocation, the dispatcher advances the virtual program counter, and launches the associated gadget.

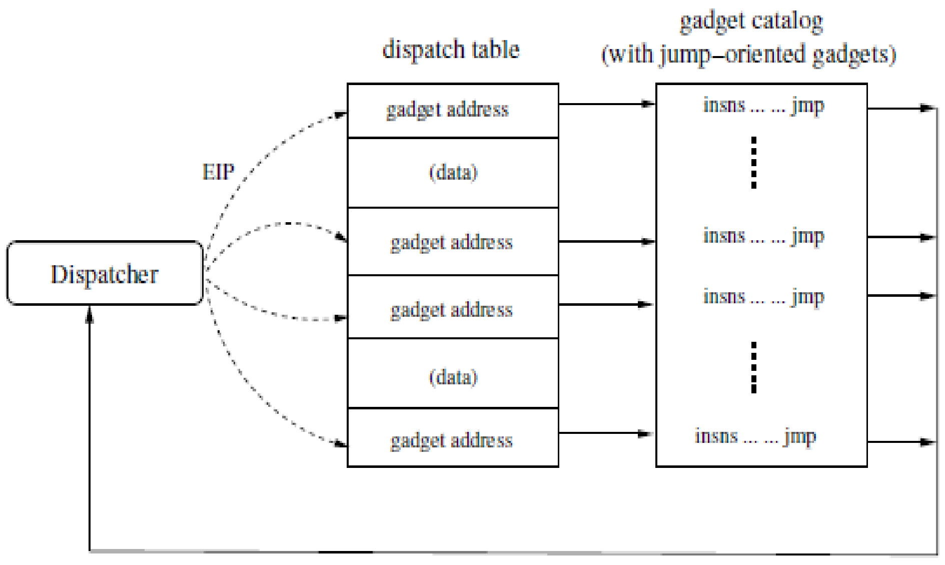

Fig.2.2 The JOP Model [2]

2.2. Malware Detection and Analysis

2.2.1. Basic Concept

Malware detection and analysis is the study of a malware by dissecting its different components and studying its behavior on the host computer's operating system.

There are two main techniques of malware analysis:

1) Static Malware Analysis

2) Dynamic Malware Analysis

2.2.2. Code Centric Approach

Traditional malware detection and analysis approaches have been focusing on code-centric aspects of malicious programs, such as detection of the injection of malicious code or matching malicious code sequences [1]. However, modern malware has been employing advanced strategies, such as reusing legitimate code or obfuscating malware code to circumvent the detection.

TABLE 2.2
Code Centric Approaches

CC Technique	**Description**
Detection of injection of malicious code	Malicious Code is any part of software system or script that is intended to cause undesired effects, security breaches or damage to the system. Such malicious codes are injected within a benign code (software script) whose detection is carried out at the time of injection by the operating system.
Matching malicious code sequences	Malicious codes have a specific sequence that is present in a benign code where it is injected. By matching these sequences presence of malicious logic can be detected.
System Call pattern generation	All the malicious code uses various system calls to damage the system. Every malicious logic generates a pattern of instruction sequences or system call sequences. By detecting these patterns presence of malicious code can be found out.

The main drawback of this traditional code centric approach is that some malware employ techniques that obfuscate or vary the patterns of code execution. Such as

- Code Obfuscation [1]
- Code Emulation [1]

These techniques can confuse behavior-based malware detectors and hence avoid detection. To overcome this drawback there are modern technique discussed further.

2.2.3. Data Centric Approach

As a new perspective to complement code-centric approaches, a data-centric architecture that detects and characterizes malware attacks based on the properties of data objects manipulated during the attacks is discussed.

Data-centric approaches require neither the detection of code injection nor malicious code patterns. Therefore they are not directly subvertible using code reuse or obfuscation techniques. Detects and characterizes malware attacks based on the properties of data objects manipulated during the attacks. However, detecting malware based on data modifications has a unique challenge that makes it distinct from code based approaches [1].

Unlike code, which is typically expected to be invariant, data status can be dynamic. Correspondingly, conventional integrity checking cannot be applied to data properties. In addition, monitoring data objects of an operating system (OS) kernel has additional challenges because an OS may be the lowest software layer in conventional computing environments, meaning that there is no monitoring layer below it.

2.2.4. Task Structure of Linux Processes

A novel framework that uses the information in kernel structures of a process to do run-time analysis of the behavior of an executing program. Classifying a process as malicious or benign using the information in the kernel structures of a process is not only accurate but also has low processing overheads; as a result, this lightweight framework can be incorporated within the kernel of an operating system [3].

The run-time analysis of the behavior of an executing program is successfully used to discriminate between a benign and malicious process using the information in the Kernel structures of a process [3]. The task structure maintained in the kernel of an operating system contains records of every action and resource usage of a process; therefore, intuitively speaking the actions and resource usage patterns of a malicious process are expected to be different from that of a benign process [3].

In order to provide a proof-of-concept of the theory, take the example of Linux operating system as a case study.

The framework consists of four modules [3]:

1. Features Logger
2. Features Analyzer
3. Features Preprocessor
4. Classifier

Time series analysis of 118 parameters of Linux task structures and preprocess them to come up with a minimal features' set of 11 features. The analysis show that these features have remarkably different values for benign and malicious processes; as a result, a number of classifiers operating on these features provide 93% detection accuracy with 0% false alarm rate within 100 milliseconds [3].

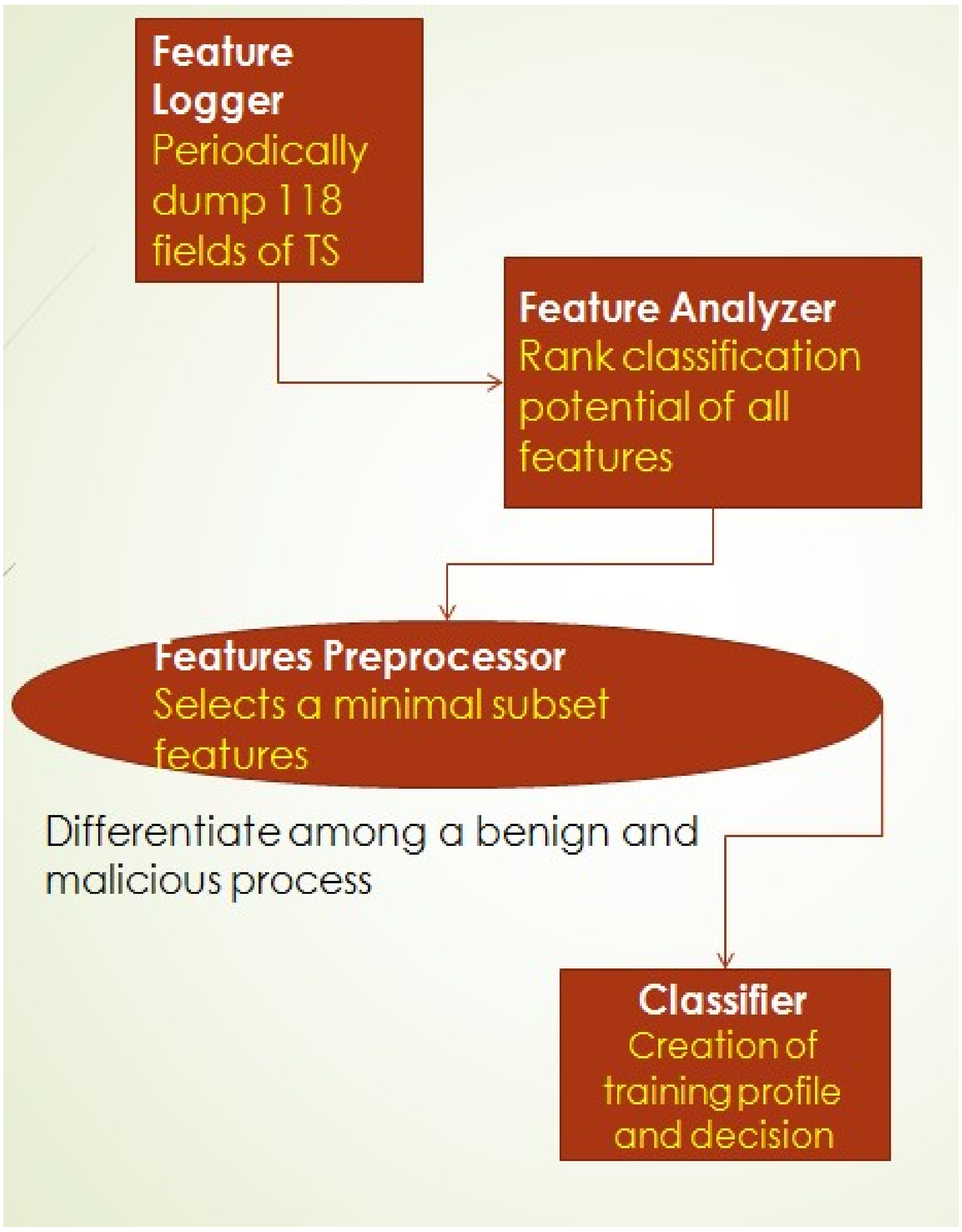
Feature Logger
Periodically dump 118 fields of TS
Feature Analyzer
Rank classification potential of all features
Features Preprocessor
Selects a minimal subset features
Differentiate among a benign and malicious process
Classifier
Creation of training profile and decision

Fig. 2.3 TS of Linux Processes Framework

Thus the above flow diagram shows the functionality and working of all the four modules of the Task Structure of Linux processes. This basic modulo structure deals with the 118 fields of the Task Structure and out of them 11 are processed by the feature preprocessor.

2.3. Data-Centric OS Kernel Malware Characterization

2.3.1. Basic Concept

A novel scheme, *data-centric OS kernel malware characterization* which enables the detection and characterization of OS kernel malware based on the properties of kernel data structures [1]. This system consists of two essential components to monitor and analyze data properties of OS kernels.

The first component is a *kernel object mapping system* that externally identifies dynamic kernel objects of the monitored OS at runtime [1]. This component enables an external monitor to recognize the access behavior to data objects. We make use of memory allocation events to build the object map. Some malware hides itself by manipulating data structures, and our experiments show that this map can reliably detect such attacks since its view is not manipulated by malware. With this map in place, we then present a *malware characterization approach based on kernel object access patterns* [1]. This approach can generate a signature of a malware's unique data access behavior. By matching data behavior signatures, it can detect classes of kernel malware that share common attack patterns on kernel data structures.

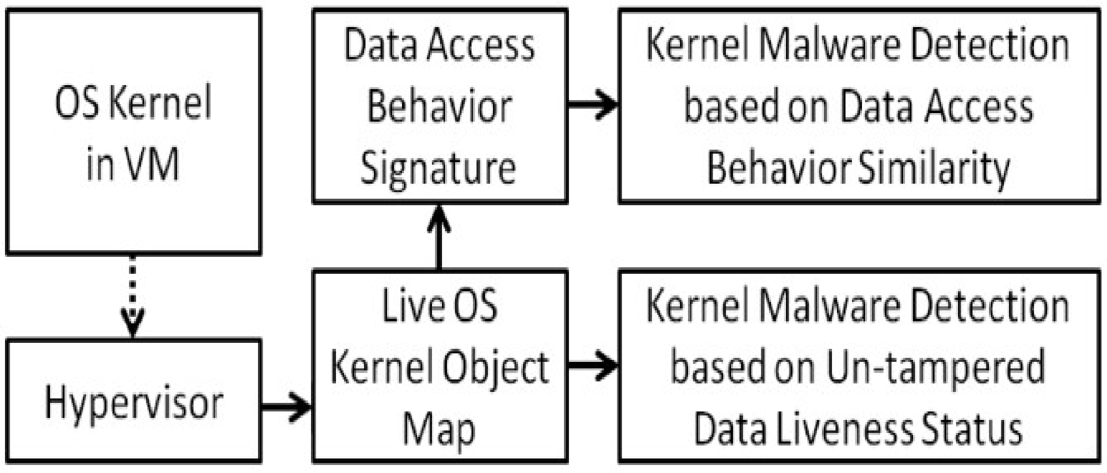
OS Kernel in VM
Hypervisor
Data Access Behavior Signature
Live OS Kernel Object Map
Kernel Malware Detection based on Data Access Behavior Similarity
Kernel Malware Detection based on Un-tampered Data Liveness Status

Fig. 2.4 DC-OS Kernel Malware Characterization [1]

The overall design of data centric kernel malware characterization approach is illustrated in Fig.2.4.

Tracking OS data allocations and uses is difficult because the OS is traditionally the lowest software layer in a conventional computer system. To overcome this challenge, we make use of virtualization technology. A guest OS runs on top of a hypervisor which transparently and efficiently captures memory related OS events to generate a kernel object map. This map is able to provide the live status of dynamic kernel objects. Many kernel rootkits are stealthy and attempt to hide themselves. Many of these attacks are implemented by manipulating data structures and making them appear dead (freed) to the OS when they are in fact alive (allocated) [1].

This map, which accurately identifies static and dynamic kernel data objects, enables the monitoring and analysis of kernel memory access patterns. Using this information we propose a new approach to characterize and detect kernel malware.

2.3.2 Contributions

1) Reliable Detection of Kernel Object Hiding Attacks [1].

Kernel object hiding attacks attempt to hide data objects by manipulating pointers reaching such objects. The kernel object mapping approach recognizes data objects based on memory allocation events, not inter-memory pointers. Therefore, such attacks do not tamper with the identification of data objects in our mapping scheme. Experiments show that our approach successfully detects kernel data hiding rootkits that manipulate data object pointers in order to evade traditional rootkit detectors.

2) Conception of Malware Signature Based on Data Access Behavior During Attacks [1].

We have discussed a new malware signature based on the unique patterns of kernel data accesses that occur during an attack. This technique can complement code-based malware signatures.

3) Detection of Malware Variants Having Similar Data Access Patterns [1].

This approach determines malware attacks by extracting and matching data access patterns specific to malware attacks. Kernel malware aiming at similar malicious features often manipulates common data structures. This mechanism can detect such malware attacks [2]. Kernel malware aiming at similar malicious features often manipulates common data structures. This mechanism can detect such malware variants having similar data access patterns.

CHAPTER -3

DISCUSSION AND ANALYSIS ON LIVE KERNEL OBJECT MAPPING SYSTEM

3.1. Overview

Traditional malware detection and analysis approaches have been focusing on code-centric aspects of malicious programs, such as detection of the injection of malicious code or matching malicious code sequences. However, modern malware has been employing advanced strategies, such as reusing legitimate code or obfuscating malware code to circumvent the detection. As a new perspective to complement code-centric approaches, in this seminar we discuss a data-centric OS kernel malware characterization architecture that detects and characterizes malware attacks based on the properties of data objects manipulated during the attacks.

This framework consists of two system components with novel features. First a runtime kernel object mapping system which has an un-tampered view of kernel data objects resistant to manipulation by malware [1]. Second, this framework consists of a new kernel malware detection approach that generates malware signatures based on the data access patterns specific to malware attacks. This view is effective at detecting a class of malware that hides dynamic data objects.

3.2. Kernel Object Mapping System

A novel scheme, *data-centric OS kernel malware characterization* which enables the detection and characterization of OS kernel malware based on the properties of kernel data structures. This system consists of two essential components to monitor and analyze data properties of OS kernels. The first component is a *kernel object mapping system* that externally identifies dynamic kernel objects of the monitored OS at runtime. This component enables an external monitor to recognize the access behavior to data objects. We make use of memory allocation events to build the object map.

Here we are introducing an allocation-driven mapping scheme which enables the creation of a live, dynamic map of kernel data object.

3.2.1. Allocation-Driven Mapping System

Allocation-driven mapping is a kernel memory mapping scheme that generates a synchronous map of kernel objects by *capturing the kernel object allocation and deallocation events* of the monitored OS kernel [1]. Fig.3.2 illustrates how this scheme works. Whenever a kernel object is allocated or deallocated, the virtual machine monitor (VMM) intercedes and captures its address range and the information to derive the data type of the object subject to the event in order to update kernel object map [1].

This approach does not rely on any content of the kernel memory which can potentially be manipulated by kernel malware. Therefore, the kernel object map provides an *un-tampered view* of kernel memory wherein the identification of kernel data is not affected by the manipulation of memory contents by kernel malware. This tamper-resistant property is especially effective to detect sophisticated kernel attacks that directly manipulate kernel memory to hide kernel objects.

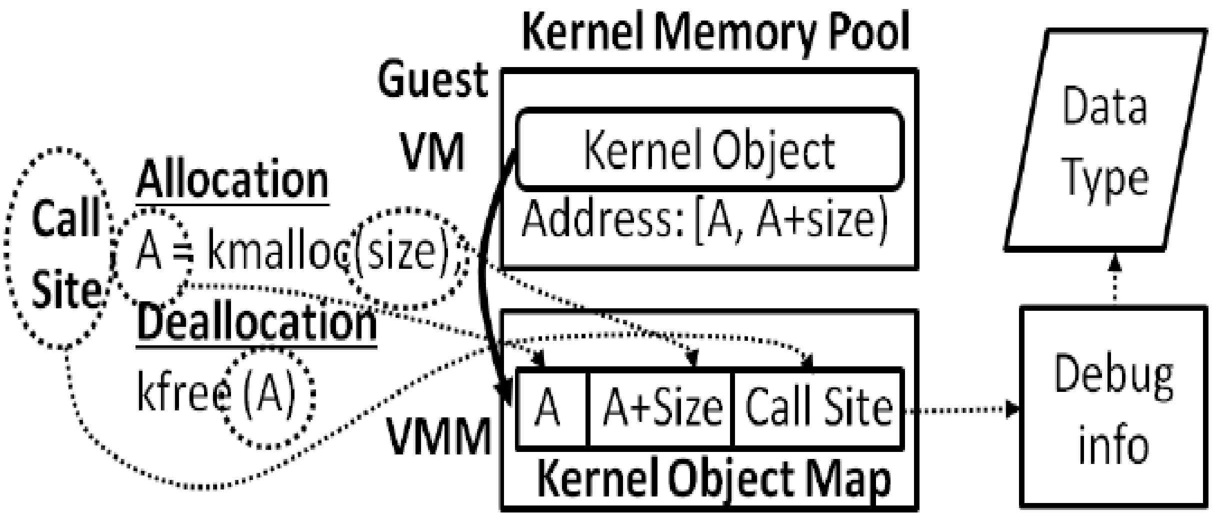

Fig. 3.2 Live Kernel Object Mapping System [1]

The key observation is that allocation-driven mapping captures the *liveness status* of the allocated dynamic kernel objects. For malware writers, this property makes it significantly more difficult to manipulate this view [1]. This mapping can be used to automatically detect data hiding attacks without using any knowledge specific to a kernel data structure.

There are a number of challenges in implementing a live kernel object map based on allocation-driven mapping. For example, kernel memory allocation functions do not provide a simple way to determine the type of the object being allocated. One solution is to use static analysis to rewrite the kernel code to deliver the allocation types to the VMM, but this would require the construction of a new type-enabled kernel, which is not readily applicable to off-the-shelf systems. Instead, we use a technique that derives data types by using runtime context (i.e., call stack information) [2]. Specifically, this technique systematically captures code positions for memory allocation calls and translates them into data types so that OS kernels can be transparently supported without any change in the source code.

3.3. Techniques for Allocation Driven Mapping

We employ a number of techniques to implement allocation driven mapping. First, a set of kernel functions (such as+ kmalloc) are designated as kernel memory allocation functions. If one of these functions is called, we say that an allocation event has occurred [1]. Next, whenever this event occurs at runtime, the VMM intercedes and captures the allocated memory address range and the code location calling the memory allocation function. This code location is referred to as an *allocation call site* and we use it as a unique identifier for the allocated object's type at runtime. Finally, the source code around each allocation call site is analyzed offline to determine the type of the kernel object being allocated [1].

3.3.1 Runtime Kernel Object Map Generation

At runtime, the VMM captures all allocation and deallocation events by interceding whenever one of the allocation/deallocation functions is called. There are three things that need to be determined at runtime:

1. The call site [1]
2. The address of the object allocated or deallocated [1]
3. The size of the allocated objects [1].

To determine the call site, we use the return address of the call to the allocation function. In the instruction stream, the return address is the address of the instruction after the call instruction. The captured call site is stored in the kernel object map so that the type can be determined during offline source code analysis.

The address and size of objects being allocated or deallocated can be derived from the arguments and return value. For an allocation function, the size is typically given as a function argument and the memory address as the return value. For a deallocation function, the address is typically given as a function argument. These values can be determined by the VMM by leveraging function call conventions [1]. Function arguments are delivered through the stack or registers, and they are captured by inspecting these locations at the entry of memory allocation/deallocation calls. To capture the return value, we need to determine where the return value is stored and when it is stored there. Integers up to 32-bits as well as 32-bit pointers are delivered via the EAX register and all values that we would like to capture are either of those types. The return value is available in this register when the allocation function returns to the caller. In order to capture the return values at the correct time the VMM uses a virtual stack. When a memory allocation function is called, the return address is extracted and pushed on to this stack. When the address of the code to be executed matches the return address on the stack, the VMM intercedes and captures the return value from the EAX register [1].

3.3.2 Dynamic Data Type Interface

The object type information related to kernel memory allocation events is determined using static analysis of the kernel source code offline. First, the allocation call site of a dynamic object is mapped to the source code using debugging information found in the kernel binary. This code assigns the address of the allocated memory to a pointer variable at the left-hand side of the assignment statement. Since this variable's type can represent the type of the allocated memory, it is derived by traversing the declaration of this pointer and the definition of its type. Specifically, during the compilation of kernel source code, a parser sets the dependencies among the internal representations (IRs) of such code elements. Therefore, the type can be found by following the dependencies of the generated IRs. There are several patterns regarding how these components are related in the source code [1].

3.3. DISCUSSION

Since the framework operates in the VMM beneath the hardware interface, we assume that kernel malware cannot directly access the framework's code or data. However, it can exhibit potentially obfuscating behavior to confuse the view seen by the framework. Here we describe several scenarios in which malware can affect this framework and the counter-strategies to detect them.

First, malware can implement its own custom memory allocators to bypass framework observation. This attack behavior can be detected based on the observation that any memory allocator must use internal kernel data structures to manage memory regions or its memory may be accidentally re-allocated by the legitimate memory allocator. Therefore, we can detect unverified memory allocations by comparing the resource usage described in the kernel data structures with the amount of memory being tracked by the framework. Any deviance may indicate the presence of a custom memory allocator.

The Kernel Object Mapping System framework is a signature-based approach that detects known and unknown rootkits based on kernel data access patterns similar to the signatures of previously analyzed rootkits [2]. If a rootkit's attack behavior is not similar to any behavior in existing signatures or it does not involve kernel data accesses, such malware is out of coverage of this framework since such behavior does not match the signature produced by the mapping system.

Many existing rootkits that share common attack goals often exhibit similar data access patterns because essentially these malicious programs generate a false view by manipulating legitimate kernel data structures relevant to the goals. This approach can detect rootkits by focusing on the common attack targets described in the malware signatures even though such rootkits have different functionalities.

This approach is clearly distinguished from traditional behavior-based methods. Traditional code behavior based approaches use code sequences as patterns. Since code execution follows a program control flow specified in the program semantics, this approach is intuitively understandable [2]. Unlike the program control flow; however, data accesses are not a single continuous flow. From the data point of view, the accesses from various codes can be interleaved making a sequence not stable as a consistent pattern for a behavior signature [1]. This problem is solved by using a different aspect of program behavior. Instead of simply using the code to create malware signatures, we model data accesses with two entities:

1. The subject (the accessing code)
2. The object (the accessed data).

This allows us to determine the patterns of relationship between subjects and objects, and hence provides more robust signatures [1].

Regarding the effectiveness of the framework when compared to code behavior-based approaches, there are more constraints a malware author must consider when designing an evasion technique. For example, one evasion technique for a standard code behavior-based approach would be to find a functionally similar code sequence from the existing code and use that instead of including your own code. Return-oriented and jump oriented programming would be such examples [2]. In contrast, data access behavior has multiple dimensions to consider: accessing code, specific field of data, and the source of data (allocation). First of all, regarding the accessing code, this approach has an advantage since this framework normalizes accessing code to detect malware variants. Second, specific fields being accessed should be preserved for the data object to be valid so that legitimate code can also properly use them. Third, using a custom allocator could be a feasible attack, but such an unknown memory allocation would be trackable by the OS as discussed. By checking the allocation code of data objects in kernel data structures, foreign objects could be detected.

3.4. ADVANATGES

- **Malware hiding inside kernel data structures can be detected**
- **Detection of code injection and malicious code pattern is not required**
- **More robust malware signature generation**

  Instead of simply using the code to create malware signatures, the modeling of data accesses with two entities (accessing code and access data) is there. This allows determining the patterns of relationship between subjects and objects and hence more robust signatures.

3.5. LIMITATIONS

- **Need of VMM interfaced over the hypervisor**
- **Not all rootkits are detectable**

  If the attack behavior of some rootkits is not similar to any behavior in existing signatures or it does not involve kernel data accesses, such malware is out of coverage since such behavior does not match the framework's signature.

CHAPTER -4

4.1. CONCLUSION

In this seminar we have learnt two different aspects of securing the operating system. One by optimizing Process Authentication and the other by detecting and analyzing the malware dynamically at run time using Data Centric OS kernel malware characterization.

The operating system can be affected either by manipulating the kernel resources by unauthorized processes or by malware that hides themselves inside the kernel data structures. In this seminar both of these aspects of violation of operating system security are handled. First aspect is handled by adding Secure Process Authentication to the trivial Process Identification mechanism in which user level applications are required to present their identities at run time to the kernel. The other aspect is taken care of by using Data Centric approach over the traditional Code Centric approach so as to detect the malware attacks at run time by creating a live kernel object mapping system and monitoring the data access pattern behavior of malware.

4.2. FUTURE SCOPE

<u>Problem Definition:</u>

Implementation of more secure Kernel to detect and analyze malware present inside it along with the process authentication modules to filter out the unauthorized processes.

4.2.1. Inter-compatibility issues of the two aspects discussed.

A more secured Kernel devised for intrusion avoidance can be implemented using both the aspects discussed.

The main challenge will be the inter-compatibility issues of both the Secure Process Authentication System and the Data Centric OS Kernel Malware Characterization Modules in the same Kernel.

REFERENCES

[1] **Data-Centric OS Kernel Malware Characterization**

Junghwan Rhee, Member, IEEE, Ryan Riley, Member, IEEE, Zhiqiang Lin, Member, IEEE, Xuxian Jiang, and Dongyan Xu, Member, IEEE(2014,Jan. 1).

[2] **Jump-Oriented Programming: A New Class of Code-Reuse Attack**

Tyler Bletsch, Xuxian Jiang, Vince W. Freeh Zhenkai Liang Department of Computer Science School of Computing North Carolina State University National University of Singapore.

[3] **In-Execution Malware Detection using Task Structures of Linux Processes**
Farrukh Shahzad, Sohail Bhatti, Muhammad Shahzad and Muddassar Farooq Next Generation Intelligent Networks Research Center (nexGIN RC) National University of Computer & Emerging Sciences (FAST-NUCES) Islamabad, 44000, Pakistan.

[4] **Characterizing Kernel Malware Behavior with Kernel Data Access Patterns**
Junghwan Rhee, Zhiqiang Lin, Dongyan Xu Department of Computer Science and CERIAS, Purdue University West Lafayette, IN 47907.

[5] Shacham, Hovav; Buchanan, Erik; Roemer, Ryan; Savage, Stefan. "Return-Oriented Programming: Exploits Without Code Injection". Retrieved 2009-08-12.

[6] Sickness (13 May 2011). "Linux exploit development part 4 - ASCII armor bypass + return-to-plt".

[7] https://lasithh.wordpress.com/2013/06/23/how-to-carry-out-a-return-to-libc-attack/

[8] "Malware definition". techterms.com. Retrieved 26 August 2013.

[9] "Evolution of Malware-Malware Trends". *Microsoft Security Intelligence Report-Featured Articles*. Microsoft.com. Retrieved 28 April 2013.

[10] www.stackoverflow.com