Virtual Machine Introspection

1. INTRODUCTION

By providing emulation of physical computing resources, virtualization enables multiple operating systems to run on the server in the form of virtual machines and share the underlying physical resources.
Virtualization platforms are becoming attractive targets of security attacks, ranging from data theft and denial-of-service attacks to the complete compromise of the virtualization infrastructure.
Amongst virtualization security solutions to protect the virtualization environment from security attacks, virtual ma- chine introspection (VMI) has become one of the most widely used security techniques.
This paper presents a review on VMI and its use in current virtualization security research. Section 2 discusses VMI rationale and its system components, while Section 3 provides a detailed analysis on the typical usages of how VMI is integrated with other security techniques. Limitations of VMI are discussed in Section 4, before the paper is summed up in Section 5.

2. VIRTUAL MACHINE INTROSPECTION

VMI inspects the VM memory and disk from the outside without intrusively injecting agents. Thus, one of its main benefits is to protect the VM monitoring tool from being compromised in the event of a successful security attack. To that end, the VM monitoring tool is placed outside of the target VM but in a trusted VM. The guest VM's internal behaviour is then inferred by using the VM state information obtained at the hardware level [8]. In a typical virtualization environment, the VM state information is obtained via the hypervisor using the application programming interface (APIs) specific to a particular virtualization platform such as the XenCtrl library for the Xen hypervisor.

2.1 Rationale behind VMI

VMI is aimed to address the shortcomings associated with previous threat detection solutions, which can be broadly classified into host-based threat detection and network-based threat detection.
Host-based threat detection monitors the run-time activities of a guest VM by placing the monitoring tool inside it, as illustrated in Figure 1. Analogous to how antivirus solutions are run on a native computer system, it periodically scans the guest VM and uses a signature database for threat detection.
While such an in-VM monitoring provides a complete and real-time view of the internal activities of a guest VM, it suffers from a number of shortcomings. The monitoring tool is susceptible to be corrupted in the event of a successful security attack. In addition, any software bug that exists within an in-VM monitoring solution can degrade the guest VM

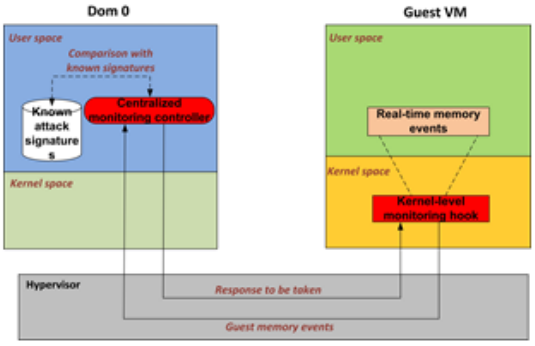

Figure 1: Host-based threat detection

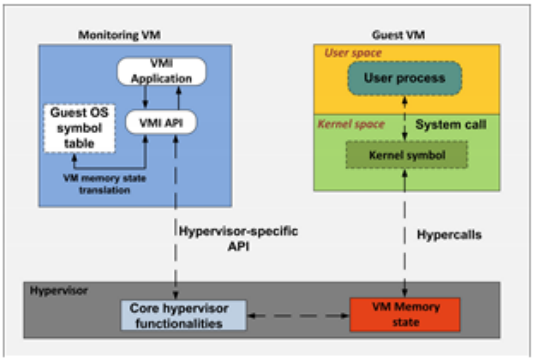

Figure 2: Architecture of virtual machine introspection

performance significantly, given the multi-tenancy nature of a typical virtualization environment.

On the other hand, network-based threat detection escapes these shortcomings as the monitoring tool is placed outside of the guest VMs. Running within a trusted VM, network-based threat detection monitors all the network traffic for any signs of possible threats. Network packets are intercepted and analysed for threats before forwarding them. By placing the monitoring tool outside of the guest VMs, it ensures that the monitoring tool can still function in the event of a VM compromise.

Despite being able to protect the monitoring tool from a malware corruption, network-based threat detection does not provide an accurate view of the internal behaviours of guest VMs. In addition, the information obtained from monitoring network packets may not be accurate in threat determination as attackers can launch attacks by exploiting legitimate ports (such as port 21, in the case of FTP).

[8] proposed VMI in order to overcome the limitations of host-based and network-based threat detection solutions. Similar to network-based threat detection, VMI places the monitoring tool outside of the guest VMs. Different from network-based threat detection, however, VMI monitors the internal behaviour of the guest VM using its state and event information obtained at the hardware-level.

2.2 Architecture of VMI

A typical VMI system is composed of three main components, namely userspace VMI application, VMI API (application programming interface) and guest OS symbol table as depicted in Figure 2.

2.2.1 VMI Application

Running within the monitoring VM, VMI application is responsible for the external monitoring of the guest VM. The application is typically run on a trusted VM (i.e., Dom0), and may have the access privileges necessary to access the underlying hypervisor using hypervisor-specific APIs.

The userspace VMI application uses the functions provided by the VMI API to introspect the various operations of the guest VM. While the application is primarily used to monitor memory events, its functionality can be extended to monitor other aspects of the guest VM such as network flow and storage activities.

2.2.2 VMI API

Installed into the monitoring VM and running as a library module, the VMI API provides an interface between the VMI application and the underlying hypervisor.

The userspace VMI application obtains different aspects of a VM's state by using the VMI API, which in turn uses the API specific to the virtualization platform on which it runs.

2.2.3 Guest OS Symbol Table

While VMI uses hypervisor-specific APIs to obtain information about the guest VM state, it is impossible to interpret the acquired data without knowledge of the guest OS since it only has access to the hardware-level VM state information[8] [16]. This is commonly referred to as a semantic gap, which is the knowledge gap between the internal workings of a guest VM and the low-level information obtained externally. A VMI implementation typically uses the guest OS kernel symbol table (System.map in Linux, ntdll.dll in Windows) to make sense of the low-level state information. Initialized during the kernel compilation process, the symbol table contains the virtual addresses of important kernel data structures such as the system call table as well as the Interrupt Descriptor Table (IDT). Figure 3 shows a snippet of the kernel symbol table.

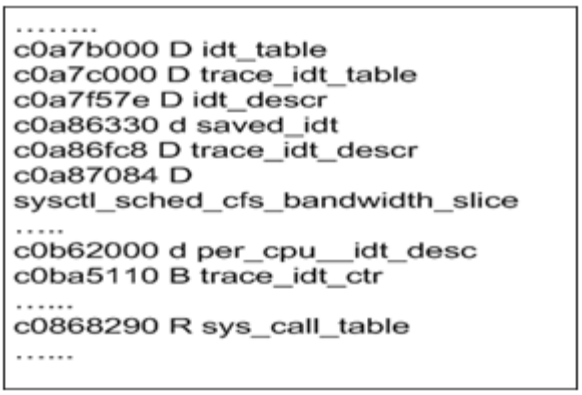

Figure 3: Sample entries in the System.map symbol table of CentOS 6 (32-bit) OS

The kernel symbol table helps narrow this semantic gap by enabling the VMI API to translate the hardware information with the virtual addresses of important guest kernel structures. It also provides the virtual addresses of structures such as the system call table, which enables the VMI API to traverse it and manipulate its entries.

3. TYPICAL USAGES OF INTEGRATING VMI WITH OTHER VIRTUALIZATION SECURITY TECHNIQUES

3.1 Signature-based Detection Using VMI

One of the applications of VMI in virtualization security is integrating VMI with signature-based threat detection. This scheme will use an attack signature database to identify the presence of threats within the guest VM, as depicted in Figure 4. Given it is done externally, however, the monitoring tool is protected from being compromised in the event of a successful VM attack.

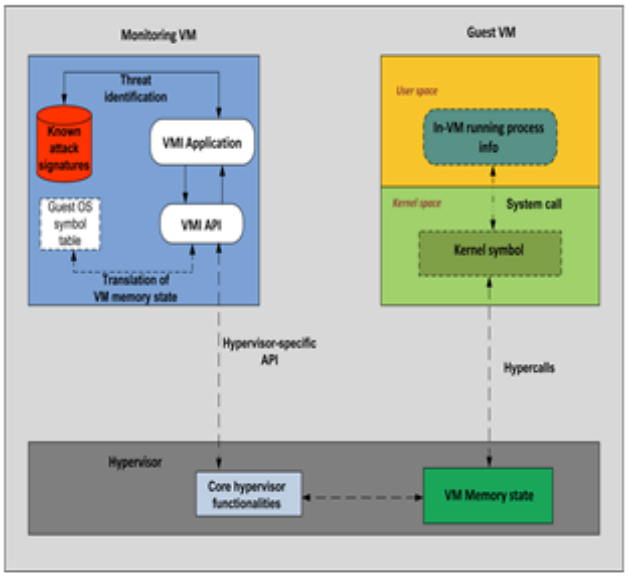

Figure 4: Signature-based detection using VMI

[8] proposed a VMI-based approach to monitoring guest VMs and implemented LiveWire which monitors guest VM activities via the management VM (i.e., Dom 0). The scheme consists of two parts, that is, an OS interface library which provides an interface for the monitoring VM to interact with the guests; and a Policy Engine containing the security policies for monitoring different aspects of the virtualized environment. The security policies detect malware by calculating the hash values of in-memory programs, and using a database of attack signatures. The isolation of the monitoring tool from the guest VMs greatly reduces the threat of malware compromise.

This approach was extended in the VmiIDS intrusion detection system [12]. Implemented on the QEMU-KVM virtualization platform, VmiIDS monitors the memory activities by comparing the results obtained from the VMI module to the process list obtained from an in-VM tool such as the ps command. The consistency of the VM file system was monitored by calculating the hash value of a file which is assumed to be consistent during the VM's lifetime.

[15] used pre-obtained system information to detect threats in a virtualization environment in its Patagonix rootkit detection system. The scheme uses hardware information and a database containing hashes of legitimate application binaries to determine the presense of rootkits. Specifically it uses the Non-Executable (NX) bit present in CPUs to mark the memory pages of the processes prior to their execution. Thus any attempts to change the page attributes can be trapped and checked by the "identity oracles" against the database. Implemented on the Xen 3.0.3 platform, Patagonix was able to identify the rootkits tested in the guest VM with additional 3% overhead.

[6] incorporated Function-call Injection (FCI) and localized shepherding in its SYRINGE virtualization security solution. FCI periodically suspends the guest VM operation, and places the address of the remote monitoring tool in the EIP register of its virtual CPU (vCPU). This enables the monitoring tool to obtain a complete view of the VM's internal operation without the threat of being compromised. Localized shepherding protect the monitoring tool by dissembling its code execution and comparing its flow to a whitelist of control flows deemed to be safe. Implemented on the VMWare virtualization platform, it incurred an over- head of 8% during its execution on a Windows XP guest VM. VMI was used in [5] to detect malware which uses multi-layer packing to hide its presence in the guest VM. Based on the observation that a packed malware must be unpacked prior to its execution, Maitland tracks for changes in the guest VM's memory pages by hooking the hypercall which updates its Memory Management Unit (MMU). The Non-Executable (NX) bit of this

hypercall is examined to determine whether a non-executable memory page has been changed to executable. A callback is then made to determine if the CR2 register contains the stack pointer address of the malicious process.

VMWatcher, a rootkit detection scheme proposed in [11], incorporated VM state reconstruction with VMI. A technique called guest view casting was used in this scheme which reconstructs the storage and memory contents of a guest VM based on its external observations. The storage contents are reconstructed based on the knowledge of the OS's file system structure, while the memory contents are reconstructed by traversing its physical memory using its kernel symbol table. Implemented on the Xen hypervisor platform using QEMU for guest VM access, VMWatcher used ten commercially available antivirus solutions to externally monitor a Windows XP guest VM. While this approach represents an attempt to bridge the semantic gap, the po- tential reconstruction overhead in a multi-VM environment will hinder the overall performance of the virtualization en- vironment.

[13] incorporated VMI with signature based detection. Implemented on the VMWare ESXi virtualization platform, it records the details of all the processes running on a guest VM such as their PID and their respective user names. Sys- tem calls made by these processes are also recorded and broken up into sequences of length three. These sequences are stored in a database on a trusted VM, which uses VMI to monitor the activities of the guests. System calls from the guest VM are compared with the stored sequences, and any calls which deviate from the stored sequence are deemed suspicious.

3.2 Training-based Detection Using VMI

[7] proposed Virtuoso, which uses the traces of in-VM pro- grams to introspect the guest VM. It uses a program which, given a Process Identifer (PID), identifies the name of the process as well as its location in the guest VM's memory. It is executed a number of times with different PIDs to obtain a collection of trace logs. These trace logs are then used to create an introspection program which is run in a security VM to introspect the guest VM's activities. While this approach to VMI helps to reduce the ambiguity associated with introspecting guest VMs, it can prove to be difficult for operating systems which uses Address Space Layout Randomization (ASLR) to make its memory mapping hard for the hacker to predict. In addition, it can be difficult to introspect processes which are mapped across different locations in memory.

[9] incorporated VMI with forensic memory analysis (FMA) and machine learning to implement a malware detection scheme. It monitors the guest VM's memory activities over a period of time. The memory data is then fed into a ma- chine learning algorithm to generate a classifier for malware detection. It uses LibVMI to obtain the memory contents and the FUSE library to obtain the virtual drive contents. This malware detection scheme fails to prevent hardware- based attacks.

[1] used VMI in HookLocator to monitor attempts in altering the function pointers stored in the kernel pool, which is a dynamically allocated region used for storing function pointers . Implemented on the Xen virtualization platform using LibVMI, it first identifies the kernel function pointers from the physical memory to build a function pointer list. The kernel pool data of the guest VM is then searched for any references to the pointers identified in the list and cross-checked with the genuine function pointer list, before being placed into the learning pointer list. The entries in the latter are then put into the genuine pointer list after deter- mining their changes throughout their execution life-span. It is deemed to be a threat if the fuction pointer points to an address beyond the kernel code during its execution.

[2] designed a hypervisor-based host intrusion detection sys- tem for IaaS cloud environments. The scheme uses KVM and treats each VM as a single process. The aim is to mon- itor system calls for anomalies using VMI. The system calls are collected from the virtual machines by so-called 'bags of system calls' which is a frequency-based representation method that requires no probability calculation. Then the classifier is trained

with the normal behaviour so that it will be able to detect anomalies. The classifier generated very strong anomaly signals with a very high detection rate.

3.3 Host-based Detection Using VMI

Availability is one of the main pillars to information security. Because most cloud providers employ automatic load balancing mechanism to move virtual machines across physical servers for utilization purposes, an attacker could lure the system into migrating VMs without a real need. This could result in wasting hardware resources or disturbing quality of services. Thus, there is a need to detect Denial of Service (DoS) attacks coming from virtual machines especially attacks which will not normally be detected by intrusion detection systems. In [2], a VMI approach was used to detect DoS attacks in IaaS environments. Their method relies on monitoring system calls using training techniques.

The Psycho-Virt security solution scheme in [4] incorporated VMI with existing host-based intrusion detection and rootkit detection approaches. Using VMI to monitor the guest VM activities, the scheme runs either the chkroot rootkit or SNORT intrusion detection tools. These tools constantly check the integrity of the guest VM memory by calculating the hash values of the running processes' TEXT area, which is marked read-only. The calculated hash value is the compared with previous values to identify any dis-crepencies.

The VMI-Honeymoon intrusion detection scheme in [14] integrated VMI with high interaction honeypots (HIH). Us- ing a network of HIHs (Honeynet), it monitors abnormalities within the incoming network packets while using the Volatility framework for memory reconstruction. The memory overhead caused by the multiple deployments of HIHs is mitigated by employing a Copy-on-Write (CoW) memory sharing approach between the parent VM and the other interconnected VMs.

3.4 System Call Interception Using VMI

[17] incorporated the key VMI concepts with system call interception in its XenFIT. Implemented on Xen, it inserts breakpoints within the guest VM's kernel code and inter-cepts the system calls using the XenCtrl library. The modification of a file is determined based on its location in the filesystem.

This approach was extended in the Gateway intrusion detection system [20]. Designed to monitor malware attacks against the guest OS's kernel space, Gateway monitors all kernel-level API requests made by the drivers. It places the kernel drivers in a separate memory space of the hypervisor. VMI is applied to trap the guest kernel driver request, so as to determine if its virtual address belongs to a pre- determined entry point. If the request is legitimate, the guest VM's CR3 register is changed to point to the kernel page table (KPT) stored in the hypervisor memory. Tested on a guest VM running a PAE (Physical Address Extension) Linux kernel, it incurred an operational overhead of approximately 10%.

3.5 VM State Reconstruction Using VMI

Implemented on the VMWare ESXi platform, the Cloud- Sec virtualization security scheme in [10] incorporated VMI with VM memory state reconstruction. Designed to monitor the memory events within the guest VM, CloudSec obtains the physical representation of the guest VM from the hypervisor and uses OS-specifc kernel structure information (System.map in Linux, and Microsoft Symbols in the case of Windows) for semantic reconstruction.

[18] introduced a mechanism called Distributed Streaming VM introspection (DS-VMI) which can infer file system modifications from sector-level disk updates in real-time and stream them to a central monitoring point. The experiments showed that the overhead

of this approach is modest except for write-intensive workloads.

4. LIMITATIONS OF VMI

4.1 Semantic Gap

Semantic gap refers to the knowledge gap between the guest OS's internal state and the information obtained externally at the hypervisor-level. This knowledge gap is usually compensated in a typical VMI implementation by using the guest OS's symbol table to interpret the low-level guest VM information obtained from the hypervisor.
While this approach helps close the semantic gap, it has a number of limitations. First of all, it is based on the assumption that it will not be changed throughout the guest OS's execution, an assumption which does not tend to hold true in real-time environments. In addition, the need to store the symbol tables for every guest VM to be monitored also makes this approach inefficient in a typical virtualization environment.
A possible approach to closing the semantic gap efficiently may be running the userspace applications (eg. ps) in the guest VM and collecting the output obtained. Another approach may use machine learning techniques to automate the process. However, they can be unreliable in the face of a successful rootkit attack.

4.2 Being Susceptible to Attacks which Manipulate the Kernel Data Structures

While VMI uses the guest symbol table to access the internal state of the guest OS, it does not provide any mechanism to determine the integrity of these kernel structures. This leaves VMI vulnerable to attacks which alter the guest OS state through the manipulation of these kernel data structures, as well as return-oriented programming (ROP) at- tacks. Direct Kernel Structure Manipulation (DKSM), proposed by [3], is designed to defeat the VMI monitoring by manipulating kernel structures such as the system call table and the Interrupt Descriptor Table (IDT). DKSM manipulates the kernel structures through two different techniques. In the direct scheme, the kernel code which accesses the kernel structures is manipulated by identifying the kernel code pointers and redirecting their access to the address contain- ing the attack code. In the shadow scheme, redirection of kernel code execution is achieved by redirecting the function pointers instead of the kernel code. This is achieved by using a split-memory approach, by which the Translate Lookaside Buffer (TLB) is manipulated to point to the attack code. All accesses to the original code are then redirected to an- other memory location during the execution of the attack code containing logic to transfer control back to it after its execution. While DKSM can be used to compromise a guest VM by providing a false view of its internal state to a VMI monitoring tool its attack tends to fail when attempting to manipulate kernel invariant structures such as the IDT and the GDT, which could not be updated without raising suspicion. In addition, DKMS cannot function in a Control Flow Integrity (CFI)-enforced kernel.
In addition, VMI is of limited ability against attacks such as return-oriented programming (ROP) and return-to-libc attacks. Designed to get around the non-execution (NX) re- strictions placed by modern processors on memory pages by operating at the CPU register level, these forms of attacks aim to hijack the control flow of a process. That is, they try to gain control of the process stack, and identify the machine instructions which can be exploited and executing them [19]. As these forms of attacks take place in the stack layer, it is not visible to a VMI monitor which only has access to the overall memory layout.

5. SUMMARY

This paper provides a review on virtual machine introspection (VMI) and its usages of being integrated with other virtualization security techniques. VMI employs a clear ap- proach to separate the security scheme outside of the VMs being inspected upon, which

lays an effective framework for its usages of being integrated alongside a broad range of existing security techniques. VMI can be considered a popular monitoring technique to protect the virtualization environment from compromise in the event of a successful VM security attack. If the limitations are addressed, then VMI has the potential to become one of the most effective virtualization security techniques.

6. REFERENCES

[1] I. Ahmed, G. G. Richard III, A. Zoranic, and V. Roussev. Integrity checking of function pointers in kernel pools via virtual machine introspection.

[2] S. S. Alarifi and S. D. Wolthusen. Detecting anomalies in iaas environments through virtual machine host system call analysis. In Internet Technology And Secured Transactions, 2012 International Conferece For, pages 211–218. IEEE, 2012.

[3] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. Dksm: Subverting virtual machine introspection for fun and profit. In Reliable Distributed Systems, 2010 29th IEEE Symposium on, pages 82–91. IEEE, 2010.

[4] F. Baiardi and D. Sgandurra. Building trustworthy intrusion detection through vm introspection. In Information Assurance and Security, 2007. IAS 2007. Third International Symposium on, pages 209–214. IEEE, 2007.

[5] C. Benninger, S. W. Neville, Y. O. Yazir, C. Matthews, and Y. Coady. Maitland: Lighter-weight vm introspection to support cyber-security in the cloud. In Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, pages 471–478. IEEE, 2012.

[6] M. Carbone, M. Conover, B. Montague, and W. Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In Research in Attacks, Intrusions, and Defenses, pages 22–41. Springer, 2012.

[7] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In Security and Privacy (SP), 2011 IEEE Symposium on, pages 297–312. IEEE, 2011.

[8] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In Proceedings of Network and Distributed Systems Security Symposium, pages 191–206, 2003.

[9] C. Harrison, D. Cook, R. McGraw, and J. Hamilton. Constructing a cloud-based ids by merging vmi with fma. In Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on, pages 163–169. IEEE, 2012.

[10] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy. Cloudsec: a security monitoring appliance for virtual machines in the iaas cloud model. In Network and System Security (NSS), 2011 5th International Conference on, pages 113–120. IEEE, 2011.

[11] 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, pages 128–138. ACM, 2007.

[12] T. Kittel. Design and implementation of a virtual machine introspection based intrusion detection system. diploma thesis, Technische Universit¨at Mu¨nchen (TUM), 2010.

[13] M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In Euromicro Conference, 2004. Proceedings. 30th, pages 520–525. IEEE, 2004.

[14] T. K. Lengyel, J. Neumann, S. Maresca, and A. Kiayias. Towards hybrid honeynets via virtual machine introspection and cloning. In Network and System Security, pages 164–177. Springer, 2013.

[15] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In USENIX Security Symposium, pages 243–258, 2008.

[16] B. D. Payne, M. De Carbone, and W. Lee. Secure and flexible monitoring of virtual

machines. In Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, pages 385–397. IEEE, 2007.
[17] N. A. Quynh and Y. Takefuji. A novel approach for a file-system integrity monitor tool of xen virtual machine. In Proceedings of the 2nd ACM symposium on Information, computer and communications security, pages 194–202. ACM, 2007.
[18] W. Richter, C. Isci, B. Gilbert, J. Harkes, V. Bala, and M. Satyanarayanan. Agentless cloud-wide streaming of guest file system updates. In Proceedings of the 2014 IEEE International Conference on Cloud Engineering, IC2E '14, pages 7–16, Washington, DC, USA, 2014. IEEE Computer Society.
[19] R. Roemer, E. Buchanan, H. Shacham, and S. Savage.
Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC), 15(1):2, 2012.
[20] A. Srivastava and J. T. Giffin. Efficient monitoring of untrusted kernel-mode execution. In Proceedings of the 18th Annual Network and Distributed Security Symposium, 2011.