Detecting Data Exfiltration by Integrating Information Across Layers

Puneet Sharma, Anupam Joshi and Tim Finin
Computer Science and Electrical Engineering
University of Maryland, Baltimore County
{tc56339, joshi, finin}@umbc.edu

Abstract

Data exfiltration is the unauthorized leakage of confidential data from a system. Unlike intrusions that seek to overtly disable or damage a system, it is particularly hard to detect because it uses a variety of low/slow vectors and advanced persistent threats (APTs). It is often assisted (intentionally or not) by an insider who might be an employee who downloads a trojan or uses a hardware component that has been tampered with or acquired from an unreliable source. Conventional scan and test based detection approaches work poorly, especially for hardware with embedded trojans. We describe a framework to detect potential exfiltration events that actively monitors of a set of key parameters that cover the entire stack, from hardware to the application layer. An attack alert is generated only if several monitors detect suspicious activity within a short temporal window. The cross-layer monitoring and integration helps ensure accurate alerts with fewer false positives and makes designing a successful attack more difficult.

1. Introduction

Data Exfiltration is a key target of many of the more sophisticated attacks today. It is typically engaged in by State actors and transnational crime syndicates and uses a variety of advanced persistent threats and low/slow vectors based on novel (zero day) exploits. Such attacks are much harder to detect than those seeking to bring down a system or deny access to it. Concerns have increased that such attacks can use trojans embedded in commodity hardware that is manufactured in a global supply chain with limited control.

Consider a scenario in which an employee uses an infected USB flash drive on a machine connected to his organization's network. The USB carries malware that automatically runs on insertion as a background process that gains root privileges by exploiting vulnerabilities in popular software installed on the computer. The malicious process then hides behind a legitimate one via code injection and evades detection. Once root privileges are gained, the attack designer has several options. One is to install the payload on the host followed by opening a remote shell on the attacker's machine. Another is to reduce its footprint by not installing any payload, but adding functionality that enables it to scan the victim's machine on its own to discover the information sought and relay it back to a remote machine before removing traces of the attack ands deleting itself. Many incidents have been reported (e.g. [20, 21]) in which data exfiltration took place in a manner similar to this scenario. There is concern that this is not limited to USB sticks, but can be done using compromised components and their firmware (e.g network cards, disks, ...).

Our scenario and many reported incidents have several common features: they are triggered by the use of a new hardware device; most are designed to communicate with a remote system after the host has been compromised; and a trusted insider is an unwitting contributor to the infiltration by providing the attacker with initial access to the host.

Ensuring tamper free hardware has become extremely difficult due to the increased use of international supply chains by vendors of commodity hardware components. Most have supply chains with components coming from several countries, assembled in others, and re-branded and marketed in dozens more. While this has lowered costs, it has also made implementing comprehensive security checks much more difficult. Since the security infrastructure has to be now deployed over a much more wider scale and over multiple organizational jurisdictions, it has become relatively easier for attackers to sabotage a small portion of the supply chain and tamper the final product with minimum chances of getting detected.

Since most of the current hardware testing tests only for cases where the hardware is supposed to perform a list of operations producing expected outputs, it fails to take into consideration cases where the hardware may be tampered so that it not only passes all the tests, but has a malicious circuit which executes additional sabotaging functionality on top of the expected activities [25]. Since most intrusion detection and prevention software try to protect their users by actively monitoring inbound data from the network or by looking for known attack signatures, very few of them can detect the aforementioned attack scenarios.

We describe a novel detection system that monitors a set of system and network level features of a host system and flags alerts based on temporally-related anomalous behavior detected in multiple monitored modules. It is well known that by building a behavioral model of the system under normal usage and detecting deviations from this model when under an attack can provide us with strong hints of an attack[3, 7, 26]. The individual alerts produced by each module are then expressed as resource description framework RDF assertions. These assertions when processed by semantic rules produce highly effective intrusion alerts that have a low false positive rate.

2. Related work

Fisk et al. [2] propose a global vault to prevent unauthorized data breaches by separating the employee machines from the ones that contain sensitive information. They implement this strict isolation between the user machines and the servers by placing limits such as a whitelist of allowed inter machine processes and a maximum allowed bandwidth. This is an impractical approach when applied for large organizations as it puts stringent conditions on what a user can or cannot do.

Liu et al. [9] describe a framework to actively monitor and react in cases of intrusions and their possible detection. Their proposed intrusion detection engine is placed at the network edge, scans outbound traffic, and decides if it should forward the data to the outside node or not. The main drawback in the system is the live monitoring and intrusion prevention approach that must mine a large amount of data and decide whether or not to forward it without affecting outbound bandwidth speeds. For even a medium sized corporation, a single module deployed at the lone egress point of a corporate network would require tremendous processing powers to monitor and analyze each outgoing packet at runtime.

Ramachandran et al. [23] claim that their behavior-based model can catch most network data exfiltration scenarios. They first learn the normal behavior of a system by using kernel density estimation methods on system features like memory consumption, CPU utilization and disk usage. The base values are used to derive correlation coefficients on test data which came from real attacks. This approach analyzes overall system features like the total memory and total CPU consumption level. A drawback of this detection technique is that it can be easily evaded by trojans with an extremely small memory and CPU footprint that can result in a significant deviation in the overall numbers for the host machine when observed as a whole. The false positive rate of their system is not documented.

There has also been some work building on the philosophy of using multiple sensing modules to detect attacks. However, these multiple sensors are typically all at the same level of the stack – just the host, or just the network. Such a narrow feature set can reduce the accuracy of alerts produced by increasing the false positive rates of the intrusion detection system. Kerschbaum et al. [5] discuss the use of multiple sensors embedded into the operating systems, but only describe in details sensors that specifically pertain to network based attacks.

Process profiling is proposed by Okazaki et al. [18], who derive a normal usage pattern based on system call sequences and compare this to the profile of a system under an attack. A similar approach based on a system calls profile is proposed by Eskin et al. [1]. Various machine learning approaches have been applied to selected system feature sets to classify attacks with good results, starting from the seminal work of Forrest et al. [3] and Lee et al. [7]. Undercoffer [26] created a model of a running system under normal usage, and then used that model to detect attacks in the future using machine learning algorithms. Mathews et al. [10] also took a machine learning approach in identifying a network-based feature set which was able to produce good classification results in identifying malicious network data.

3. System Design

We describe a prototype intrusion detection system (IDS) that is highly modular and has in place multiple sensing modules across multiple layers of the system. Each alert from the individual monitoring sensor is represented as a set of RDF assertions. Producing RDF assertions allows our system to fit into a larger semantic integration and reasoning framework being developed in our laboratory [10, 4] that uses traditional and non traditional sensors to form a collaborative approach to cybersecurity. The assertions from our system can be integrated with other information and the results augmented using various reasoners, including description-

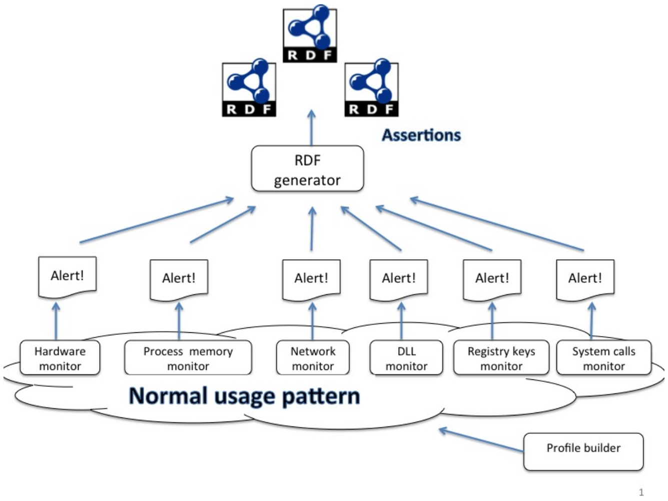

Figure 1. Our prototype system detects anomalous behavior at different layers, encode the events as RDF assertions which are integrated and reasoned over to recognize potential exfiltration events.

logic theorem provers and rule-based systems.

In this paper, we focus on a simple reasoning approach in which a collection of alerts denoting the quick succession of anomalous events across multiple layers indicates that an attack may have allowed data to be exfiltrated from a victim's machine. These multiple modules sensing different system parameters in tandem are crucial in reducing false positives, as a large number of system parameters behaving in a anomalous manner is most likely to be a strong indicator of an attack.

To build a normal profile of the system we ran the profiling module for a substantial amount of time logging the measured parameters such as memory patterns, DLLs called, and network usage. Once this profile is built, the data is stored and serves as a model of what our IDS defines as "normal behavior". Each sensor module is then run independently to monitor specific system level parameters. The monitoring module for that sensor continuously compares live data with the normal-behavior profile. In case that the module catches a deviation from the norm, it produces an alert that denotes a abnormalities for that particular system parameter. Next, an RDF generator module runs on top of all the sensing modules and takes as input the alerts produced by the earlier sensors and creates a graph of RDF assertions. These RDF assertions describe the situation using ontologies developed by our research group [27, 28, 4]. Figure 1 shows an overview of our system.

In the remainder of this section we describe some of the modules we have implemented and that are used in the example exfiltration scenario.

3.1 New hardware detection

This module produces an alert each time a new hardware device is inserted in the system. We maintain a host-based data resource of identifiers of devices that have been seen with the host to detect ones that are inserted for the first time. A sample rule that may take advantage of such classification could be for a case where a stronger alert is shown on instances when a completely new USB flash drive is inserted into the system rather than one which is frequently seen. We use the hardware device's UUID values as unique identifiers representing the device.

3.2 Memory usage by a process

As soon as an attacker is able to gain access to a victim's system the immediate next task in most attacks is to hide the malicious process from the user. A commonly used method to do this is via a code injection into an existing running process. In doing so, the memory usage of the process will likely change [26]. Based on this intuition we monitor the heap, stack and private data sections of a list of profiled processes and log alerts if the memory footprint of each memory type deviates significantly from the mean observed value.

3.3 Network data

Monitoring network data also can produce patterns that provide indicators of exfiltration attacks. A sudden burst of outgoing data or communication with a never-before seen IP address, especially one that is in the DHCP range of an ISP, can be a good hint of data being exfiltrated. While monitoring IP addresses to detect communication with a new IP address is straightforward, detecting "bursts" of outgoing data is more complex.

The primary difficulty arises in defining a "burst of data" compared to normal variability in traffic. Secondly, we must keep a track of all deviations and their occurrence patterns in order to avoid extremes results which could result in an unacceptable number of false positives (false alarms) and/or false negatives (missed attacks). Our model of an abnormal "burst of data" is derived from an analysis of data collected from multiple TCP sessions of every outbound communication from the host over a significant period. Our decision to analyze TCP sessions is based on earlier work done in our group [10] that had shown good results in detecting malicious network traffic by analyzing TCP sessions on inbound data.

We use two features to model the outbound network flow characteristics: the mean inter-departure packet times and the number of packets in a single TCP session. The first feature denotes the rate of packets flying out and the second feature denotes the pure quantity of outbound data. These two features, when taken together, give a good picture of sufficient data going out of a system in a short span of time. Both characteristics are expected to be high when an attacker infiltrates a victim and tries to maximize his information theft by exfiltrating the data as quickly as possible.

3.4 Dynamic link libraries

We profile the list of dynamic link library (DLL) calls a process makes during its normal execution. It is a fair assumption that for an extensively profiled process, one can gather a finite list of all DLL files that the process typically opens for its regular use. A process making a DLL call that is not among its normal set may indicate that it has been compromised and an alert is generated.

3.5 Registry keys

Similar to the list of DLLs, we maintain a list of all registry keys a Windows process usually accesses. Any new registry key being accessed is another indicator on our list that gets flagged as a possible process executing maliciously. A trojaned process can have multiple reasons to access to registry keys it has never accessed before. A simple process like notepad, for example, should not have to access a network configuration registry entry. If it does, there is high probability that a malicious process pretending to be notepad is accessing network information in order to connect to a remote server.

3.6 System calls

There is sufficient past work [18, 1, 8] that proves that system call monitoring can produce good indicators of an attack. One of the process characteristics that we monitor to detect any deviations from the norm is the system calls being made by that process. We assume that a trojan hiding underneath an existing process is likely to call a distinct set of system calls which if monitored, can be used to raise an alert. We use a fairly simple approach, essentially only looking the the number of system calls made, not their pattern.

4. Profile building and live monitoring

Most of our development and testing was on systems running the Windows operating system due to the high number of publicly available attacks specifically targeted towards these. Our profiling and process monitoring module is currently limited to Windows-based processes, though similar routines can be easily written (and in some cases already exist) for Linux. We successfully profiled and monitored a list of nine common Windows processes: *calc.exe*, *conhost.exe*, *explorer.exe*, *firefox.exe*, *msinfo32.exe*, *mspaint.exe*, *notepad.exe*, *powershell.exe* and *wmplayer.exe*.

The decision to select these nine was based on three factors. First, we wanted a list of processes that are either pre-installed in a standard configurations or are part of very popular software packages. Second, we wanted a wide range of processes in terms of their memory consumption pattern to avoid biasing our results. The final selection criterion was the amount of user interaction each of the monitored processes witness in their lifetime. We wanted a broad variety of processes which would include background processes such as *explorer.exe* or *conhost.exe* that do not involve user interaction to processes like *firefox.exe* and *wmplayer.exe* that do.

We ran the profiling module for three to four days with intermittent use of each process to produce average values of memory consumed by the heap, stack and private data sections. We also calculated the standard deviation of these three respective mean values for each process. Once the profile was built, we monitored these process live and raised alerts if the memory consumption for any of the three memory types went over three standard deviations of the averaged value. We implemented a simplistic non-statistical approach to profile the list of DLLs, windows registry keys and system calls that the processes called under normal usage. During the profiling phase of these processes, a whitelist of all DLLs, system calls and registry keys was prepared which was essentially a list of all calls the processes made under normal use. If any new DLL, system call or registry key is called outside the earlier built whitelist, an alert is raised.

For our networking module we used the libpcap [15] libraries to implement packet sniffing for all outbound traffic. The splitcap [14] tool was used to extract TCP session based information from the network packets being monitored. The system was run for a few days and all IP addresses that the host communicated with logged. This list of IP addresses served as a whitelist of all destinations that were deemed safe to be communicating with. A network packet sent to any IP address outside this list would throw an alert. Packet sniffing sessions were initiated on five machines in our lab used by multiple users who had volunteered. The data collected from these volunteers was aggregated to produce overall network flow characteristics. These characteristics collected and aggregated produced an average value of the inter packet departure time per TCP session and the average number of packets sent in a single TCP session.

The hardware monitoring module had a simple implementation. All connected hardware devices were profiled using their manufacturer UUID as their identification number and alerts were raised for any new hardware introduced in the system. In case of USB flash drives, an additional information informing us whether the USB device was seen in the past or not was added in the produced alerts. This allows the possibility of highly flexible rules running on our RDF assertions such a sample rule which called for no alerts to be raised if the USB drive inserted in the system had been frequently used in the past. This approach can be extended to other devices – for instance logging the MAC address of a network card or a disk serial number.

5. Testing our system

We used the Metasploit [17] open source penetration-testing framework to create and apply attacks in order to test our intrusion detection system. Within Metasploit, we extensively used the social engineering toolkit (SET) [22]. Social engineering based attacks are among the most common forms used today for data exfiltration. SET is popular, with over two million downloads, for two reasons: (1) it offers a large number of easy to run attacks that do not require much experience or background knowledge, and (2) it is tightly integrated with Metasploit, allowing pen-testers and white hat hackers to develop custom exploits by combining SET based attack options with custom payloads. The list of past attacks that used social engineering to infiltrate their victims includes highly sophisticated APTs like Stuxnet [6], which was spread using USB drives, and the Aurora attack on Google [19], which is believed to have been initiated by sending malicious URLs to Google employees. The social engineering toolkit under Metasploit allows us to test our system against similar attacks that can be launched by using malicious hardware to directly transfer the Trojan payloads on to a known system.

We ran the following five attacks available in Metasploit's SET:

1. *PowerShell attack using shellcode injection*
2. *Metasploit executable*
3. *Applet based attack*
4. *Remote Administration using HTTP tunneling (RATTE)*
5. *Tab nabbing attack*

Once the victim's machine was successfully compromised and complete access gained, we tried to mimic a real attack resulting in data exfiltration. The first step was to hide our malicious process behind an existing one using code injection. We then downloaded files from the victim's machine, took screen shots of the victim's screen, and captured key strokes. We also executed remote processes and extracted network configuration information from the victim.

We ran the same set of attacks against six different commercially available security software systems. These covered traditional anti-virus systems, firewalls and pure intrusion detection systems. The list included *Microsoft forefront endpoint*, *Spyware terminator*, *Windows defender*, *Snort*, *AVG* and *Comodo firewall.*

6. Results

Every time a new USB flash drive was inserted, our hardware monitoring module was able to produce an alert with the additional information of whether the flash drive had been seen before or not. Results from the memory monitoring module 1 show that all three memory types can potentially be good features to be monitored to detect an attack. For the nine sample processes however, heap and stack turned out to be less accurate indicators when compared to private data memory type.

We observed that for most of the profiled processes, the private data memory type witnessed a significant jump whenever we tried to hide our malware behind a particular process using code injection. The three processes for which the jump was less than one standard deviation (¡1σ) were Microsoft paint (mspaint.exe), Windows media player (wmplayer.exe), and Firefox (firefox.exe). This was largely due to these processes having a highly variable memory consumption pattern dependent on their usage which leads to a high standard deviation value. Firefox, for example, can start as a small process with a memory footprint of a few hundred kilobytes, but can reach a value more than ten times that due to heavy graphic content of the websites being viewed or simply by the number of concurrent tabs opened by the user. In case of Windows media player, we found surges in the memory usage when the player was used to stream high definition videos when

Process	Priv data	Stack	Heap
calc.exe	554σ	11.14σ	3.72σ
conhost.exe	1964σ	32σ	428σ
esplorer.exe	30.8σ	0.96σ	2.32σ
firefox.exe	0.47σ	2.1σ	15.6σ
msinfo.exe	31σ	0.047σ	$0.89\ \sigma$
mspaint.exe	1.08σ	0.38σ	0.24σ
notepad.exe	42.58σ	0.01σ	2σ
powershell.exe	1972σ	21σ	15.9σ
wmplayer.exe	0.65σ	0.9σ	0.82σ

Table 1. Memory deviations for attacked processes

Process	DLL	Registry	System call
calc.exe	17	31	4
conhost.exe	27	233	3
esplorer.exe	22	34	3
firefox.exe	5	40	0
msinfo.exe	21	45	0
mspaint.exe	14	280	0
notepad.exe	16	31	0
powershell.exe	34	310	0
wmplayer.exe	84	2175	9

Table 2. The number of new DLLs calls, registry keys accessed and system calls are indicators of compromised processes.

compared to simple music playing or image viewing operations.

Although the memory monitoring module was ineffective for a small set of our profiled processes, it worked extremely well for processes that have a low memory footprint and run in the background without much user interaction. These background processes are generally the first choice for most attackers. The Explorer process, for example, is one of the most popular choices for hiding malware and is recommended in many hacking tutorials on the Web [12, 11, 16, 13]. Processes like Firefox and Wmplayer are poor choices because they have short lifetimes since they are often killed by users after their use.

Table 2 shows that during an attack, effected process accessed a number of new DLLs and registry keys that they had never accessed during normal operation. Our alert sensitivity for this sub-module was such that a single new DLL or registry key access produced an alert. While monitoring system calls, however, we observed that six out of nine processes did not show any new system call being accessed during an attack. We believe that this is largely due to the simplistic model of system call access that we used, as prior work has shown that detecting complex usage patterns of system calls will detect subverted processes[3].

For the networking module, the network data before and after an attack was not varied by a scale of tens or hundreds, as was the case with the memory monitoring module. Therefore, we needed to come up with an allowed standard deviation number to use as a benchmark when differentiating between normal network data flow and data exfiltration due to an attack. After running our module over test data and analyzing the results, we selected $+4\sigma$ as the maximum allowed deviation for the number of packets sent in a TCP session. The mean inter-departure time of packets is more varied in terms of its sample set with a large standard deviation. After more analysis of the sample test data, we chose $0.01\times\gamma$, where γ is the *mean inter departure time per packet per TCP session*, as the minimum allowable value for a TCP session to not be suspected of belonging to an attack.

We monitored over 1154 TCP sessions out of which twelve were part of the illegitimate intrusions leading to data exfiltration. Since multiple TCP sessions are often created for a short, one-time communication between two nodes in a network, the number of actual attacks run was much less than twelve. Out of these twelve malicious sessions, only three were sessions that involved the attacker exfiltrating small files (<1 Mb) from the victim's machine. The inter departure packet time sub-module detected 114 TCP sessions, which was a high number of false positive alerts. The packet count sub-module was relatively better but still generated a substantial number of false positives. However, requiring a conjunction of both significantly increased the alert accuracy with the overall network module detecting all three exfiltration sessions apart from one other false positive (Table 3).

When the six commercial security software mentioned earlier were run over our sample attacks, none performed well. The majority could not detect most of our attacks, although each one of them found some success for a few attacks. Table 4 compares the performance of our intrusion detection system and the others for each of the five attacks tested. For all, our modules were able to trigger alerts warning us of anomalous behavior.

For all of the attacks studied except tab nabbing, all of our modules succeeded in flagging anomalous system behavior, thus providing enough information to produce an alert with reasonably high confidence. Since the tab nabbing attack involved neither remote code execution, process migration, nor substantial network

Total TCP sessions monitored	**1154**
Malicious sessions	12
MIDPT module alerts	114
Packet count alerts	34
Combined alerts	**4**
True positives	**3**
False positives	**1**

Table 3. Combined detection rate for attacks

traffic, some of our modules were unable to detect any anomalous behavior and raise alerts.

Our IDS was able to detect and log a sequence of two major events as they happened over the course of tab nabbing attacks: the insertion of a foreign device followed by a connection to a never seen before IP address. Even though the individual alerts raised was low in confidence, our system was still able to produce some alert for an attack that was missed by all of the commercial software systems we tested.

We also tested our system to see how prevalent false positives were. It was run under normal usage for about six hours without executing any attacks on it in order to observe the number of false alarms our individual modules produced. One of the original goals of our work was to reduce the false positive rate while detecting attacks. We aimed to achieve this by integrating information from multiple detection modules and integrating them as events that are temporarily close. We vary the number of individual modules that need to declare an intrusion before the system as a whole would. Figure 2 shows the results. They confirm that our approach performs well since there were no false positives when three or more of our monitoring modules raised alerts concurrently. Also, the high number of false positives where a single module raised an alert were primarily due to the networking module that flagged an alert every time we communicated to a new IP address. We believe that with a longer profiling duration than ours, this number can go down in the future.

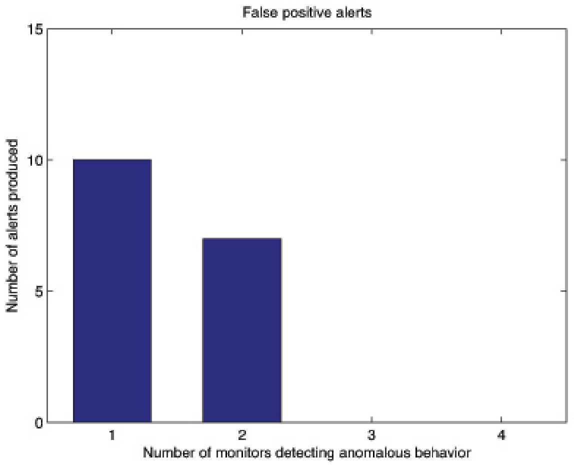

Figure 2. False positive rate

7. Conclusion

We implemented and evaluated a prototype system that is effective in detecting attacks leading to data exfiltration from a compromised computer running the Windows operating system. In our evaluation six popular commercial software products performed poorly on this task. Our approach is based on integrating information from a collection of monitoring systems that operate at different conceptual layers of a computing environment, from hardware up to applications. The information is encoded as RDF assertions supported by several ontologies designed to support representing and reasoning over information about security-related entities, relations, events and concepts. Our cross-layer and temporally-aware approach was able to minimize the number of false positive alerts which plagues most of the current security solutions. We plan to build on our prototype by adding additional modules, incorporating additional background knowledge, using more sophisticated techniques to model and discriminate normal and malicious behavior, incorporate machine learning algorithms where appropriate and conduct a more comprehensive and larger-scale evaluation.

Acknowledgment

This research was partially supported by AFOSR award FA9550-08-1-0265 and a gift from Northrop Grumman. Joshi's work was supported by funds from the Oros Professorship endowment.

References

[1] E. Eskin, W. Lee, and S. J. Stolfo. Modeling system calls for intrusion detection with dynamic window sizes. In *DARPA Information Survivability Conf. & Expo. II*, volume 1, pages 165–175, 2001.

[2] M. Fisk, S. Miller, and A. Kent. Global virtual vault: Preventing unauthorized physical disclosure by the insider. In *Military Communications Conf.*, pages 1–7. IEEE, 2008.

[3] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Symposium on Security ad Privacy*. IEEE, 1996.

	PowerShell	**Metasploit**	**Applet**	**tunneling**	**Tab nabbing**
Microsoft forefront endpoint	Missed	Caught	Caught	Missed	Missed
Spyware terminator	Missed	Missed	Missed	Missed	Missed
Windows defender	Missed	Missed	Missed	Missed	Missed
Snort	Caught	Caught	Missed	Missed	Missed
AVG	Missed	Caught	Missed	Caught	Missed
Comodo firewall	Missed	Caught	Missed	Caught	Missed
Our system	**Caught**	**Caught**	**Caught**	**Caught**	**Caught**

Table 4. Our system performed well compared to others on experiments with several common types of attacks.

[4] A. Joshi, R. Lal, T. Finin, and A. Joshi. Extracting cybersecurity related linked data from text. In *Seventh IEEE International Conference on Semantic Computing*. IEEE Computer Society, September 2013.

[5] F. Kerschbaum, E. Spafford, and D. Zamboni. Using embedded sensors for detecting network attacks. In *ACM Workshop on Intrusion Detection Systems*, 2000.

[6] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy*, 9(3):49–51, 2011.

[7] W. Lee and S. J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions Infformation Systems Security*, 3(4):227–261, Nov. 2000.

[8] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[9] Y. Liu, C. Corbett, K. Chiang, R. Archibald, B. Mukherjee, and D. Ghosal. Sidd: A framework for detecting sensitive data exfiltration by an insider attack. In *42nd Hawaii Int. Conf. on System Sciences*, pages 1–10. IEEE, 2009.

[10] M. L. Mathews, P. Halvorsen, A. Joshi, and T. Finin. A collaborative approach to situational awareness for cybersecurity. In *8th Int. Conf. on Collaborative Computing: Networking, Applications and Worksharing*, pages 216–222. IEEE, 2012.

[11] Metasploit Commands. http://hacking-tutorial.com-/tips-and-trick/7-metasploit-meterpreter-core-commands-you-should-know/. (accessed 2013-05-29).

[12] Metasploit Tutorial. http://offensive-security.com-/metasploit-unleashed/Meterpreter_Basics. (accessed 2013-05-29).

[13] MeterpreterClient. http://wikibooks.org/wiki/Metasploit/MeterpreterClient. (accessed 2013-05-29).

[14] SplitCap. http://netresec.com/?page=SplitCap. (accessed 2013-05-29).

[15] TcpDump and LibPcap. http://tcpdump.org/. (accessed 2013-05-29).

[16] Using Metasploit Meterpreter Keylogger. http://hacking-tutorial.com/hacking-tutorial/5-step-using-metasploit-meterpreter-keylogger-keylogging/. (accessed 2013-05-29).

[17] J. O'Gorman, D. Kearns, and M. Aharoni. *Metasploit: The Penetration Tester's Guide*. No Starch Press, 2011.

[18] Y. Okazaki, I. Sato, and S. Goto. A new intrusion detection method based on process profiling. In *Symposium on Applications and the Internet*, pages 82–90. IEEE, 2002.

[19] Operation aurora. http://wikipedia.org/wiki/Operation_Aurora. (accessed 2013-05-29).

[20] IBM distributes infected USB drives at conference. http://scmagazine.com/ibm-distributed-infected-usb-drives-at-conference/article/170862/. (accessed 2013-05-29).

[21] Netbook comes with factory-sealed malware. http://scmagazine.com/netbook-comes-with-factory-sealed-malware/article/137147/. (accessed 2013-05-29).

[22] N. Pavkovic and L. Perkov. Social Engineering Toolkita systematic approach to social engineering. In *MIPRO 2011, 34th International Convention*, pages 1485–1489. IEEE, 2011.

[23] R. Ramachandran, S. Neelakantan, and A. Bidyarthy. Behavior model for detecting data exfiltration in network environment. In *Conf. on Internet Multimedia Systems Architecture and Application*. IEEE, 2011.

[24] P. Sharma. A multilayer framework to catch data exfiltration. Master's thesis, University of Maryland, Baltimore County, August 2013.

[25] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *Design & Test of Computers, IEEE*, 27(1):10–25, 2010.

[26] J. Undercoffer. *Intrusion Detection: Modeling System State to Detect and Classify Aberrant Behavior*. PhD thesis, University of Maryland, Baltimore County, Feb. 2004.

[27] J. Undercoffer, A. Joshi, T. Finin, and J. Pinkston. Using DAML+OIL to classify intrusive behaviours. *Knowledge Engineering Review*, 18(3):221–241, 2003.

[28] J. Undercoffer, A. Joshi, and J. Pinkston. Modeling computer attacks: An ontology for intrusion detection. In *6th Int. Symp. on Recent Advances in Intrusion Detection*, pages 113–135. Springer, 2003.