The Nemesis
A Tool for the Virus Analyst

“Computer Viruses are the first and the only form of artificial life to have had a measurable impact on society”
KepHard J. - IBM research center

Group Members:
1. Thabet Al Fishawi
2. Margaret Ghaly
3. Rania El Rouby
4. Peter Iskandar
5. Ahmed El Fakahany

Supervisor:
1. Dr. Ahmed Sameh
2. Dr. Mohy Mahmoud
3. Dr. Sherif El Kassas

Co-ordinator:
Dr. Hoda Hosny
The Nemesis - Proposal

Table of Contents

<table>
<thead>
<tr>
<th>1. Abstract of the Project</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Thesis Statement</td>
<td>6</td>
</tr>
<tr>
<td>3. Problem Definition</td>
<td>7</td>
</tr>
<tr>
<td>4. Background Information</td>
<td>9</td>
</tr>
<tr>
<td>5. Literature Survey of Similar Projects</td>
<td>12</td>
</tr>
<tr>
<td>5.1 The HoneyNet Project</td>
<td>12</td>
</tr>
<tr>
<td>5.2 The Digital Immune System</td>
<td>15</td>
</tr>
<tr>
<td>5.3 The Automatic and Controlled Virus Code Execution System.</td>
<td>21</td>
</tr>
<tr>
<td>6. Possible Technologies &amp; Choices</td>
<td>26</td>
</tr>
<tr>
<td>6.1 Contained Environment</td>
<td>26</td>
</tr>
<tr>
<td>6.2 Platform</td>
<td>45</td>
</tr>
<tr>
<td>6.3 Monitoring Techniques</td>
<td>45</td>
</tr>
<tr>
<td>7. Analysis of The Nemesis Project</td>
<td>53</td>
</tr>
<tr>
<td>8. Software and Hardware Requirements</td>
<td>57</td>
</tr>
<tr>
<td>9. 491 Time Plan</td>
<td>58</td>
</tr>
<tr>
<td>10. Conclusion</td>
<td>58</td>
</tr>
<tr>
<td>11. References</td>
<td>58</td>
</tr>
</tbody>
</table>
Outline

1. Abstract of the Project
2. Thesis Statement
3. Problem Definition
4. Background Information
   4.1. Malicious Codes
   4.2. Types Of Malicious Code
   4.3. Structure of Malicious Codes
   4.4. Third Generation Malicious Codes
   4.5. Malicious Code Statistics
5. Literature Survey of Similar Projects
   5.1. The HoneyNet Project
   5.2. The Digital Immune System
   5.3. The Automatic and Controlled Virus Code Execution System.
6. Possible Technologies & Choices
   6.1. Contained Environment
      6.1.1. Sandbox
      6.1.2. Emulators
      6.1.3. Virtual Machine
   6.2. Platform
      6.2.1. Linux
      6.2.2. Windows
   6.3. Monitoring Techniques
      6.3.1. Windows API
      6.3.2. Subclassing
7. Analysis of The Nemesis
   7.1.1. Monitoring
   7.1.2. Analysis
   7.1.3. Testing
   7.1.4. Learning
8. Software and Hardware Requirements
   8.1. Software Packages
   8.2. Hardware
9. 491 Plan
10. Conclusion
11. References
# Table of Figures

<table>
<thead>
<tr>
<th>Figure Name</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The HoneyNet Project</td>
<td>13</td>
</tr>
<tr>
<td>2. Second Generation HoneyNet</td>
<td>14</td>
</tr>
<tr>
<td>3. HoneyNet Statistics</td>
<td>15</td>
</tr>
<tr>
<td>4. Digital Immune System – Kill Signals</td>
<td>19</td>
</tr>
<tr>
<td>5. Digital Immune System – Overview</td>
<td>20</td>
</tr>
<tr>
<td>6. Automated &amp; Controlled Virus Code Execution System (ACVCES)</td>
<td>23</td>
</tr>
<tr>
<td>7. ACVCES FlowChart</td>
<td>25</td>
</tr>
<tr>
<td>8. VMware Structure</td>
<td>34</td>
</tr>
<tr>
<td>9. VMware Layers</td>
<td>37</td>
</tr>
<tr>
<td>10. VMware Network Architecture</td>
<td>39</td>
</tr>
<tr>
<td>11. Bridged Only Networking (VMware)</td>
<td>41</td>
</tr>
<tr>
<td>12. NAT Networking (VMware)</td>
<td>42</td>
</tr>
<tr>
<td>13. Host Only Networking (VMware)</td>
<td>43</td>
</tr>
<tr>
<td>14. VMware Snapshot</td>
<td>44</td>
</tr>
<tr>
<td>15. The Nemesis Proposed Structure</td>
<td>53</td>
</tr>
<tr>
<td>16. Analysis Subsystem</td>
<td>54</td>
</tr>
</tbody>
</table>
1. Abstract Of the Project

We get our name "The Nemesis" out of Greek mythology where Nemesis is the Goddess of Vengeance who punished all those who have broken the moral code. We are developing a virus analyst tool that an analyst will use to analyze new mutations of malicious codes or even totally new malicious codes. The idea of the project was inspired from several projects most important of which is a project still under research at the IBM labs called "The Digital Immune System". Other inspiring projects like "The HoneyNet Project" in the field of Intrusion Detection has given us lots of ideas.

The objective of The Nemesis is to help speed up and relive hundreds of dedicated virus analysts from the tedious job of going through the Hex structure of the malicious code so as to discover how it works and thus speed up the process of providing the signature of the virus to the end-users. An example of The Nemesis will work is that we submit a suspicious program to it and then The Nemesis will run it through a series of tests and finally will print out the results of the tests. Also The Nemesis will provide the human analyst with the Signature of the malicious code if the program was found to be infected, also a removal tool will be generated to disinfect the contained environment.
2. Thesis Statement

“The Nemesis will accelerate and automate the process of analyzing malicious codes; this will be achieved by triggering the code in a controlled environment where its behavior will be monitored and analyzed in order to produce its signature and disinfection information”
3. Problem Definition

Unique among all forms of artificial life, computer viruses have escaped their playpens and established themselves pervasively throughout the world's computing environment.

Computer viruses are serious business. They have engendered an entire anti-virus industry, consisting of hundreds of researchers and developers who are employed by dozens of companies around the world. At least one such company, devoted almost exclusively to anti-virus software, is traded on the Nasdaq stock exchange.

Currently, the arms race between virus authors and anti-virus developers is roughly even. During any particular moment, it is typical for a few viruses to be increasing in prevalence, and other formerly prevalent ones to be on the decline. However, two alarming trends threaten to turn the balance in favor of virus authors:

1. The rate at which new viruses are being written is quite high, and appears to be accelerating. Human experts who analyze and find cures for viruses are already swamped, and their ability to keep pace with the large influx of new viruses is being questioned.

2. The continuing increase in interconnectivity and interoperability among the world's computers enhances the ability of any particular virus to spread, and the rapidity with which it does so. The current strategy of periodically distributing updates to anti-virus software from a central source will be orders of magnitude too slow to keep up with the spread of a new virus.

Why current anti-virus techniques are doomed

Whenever a new virus is discovered, it is very quickly distributed among an informal, international group of virus collectors who exchange samples among themselves. Many such collectors are in the anti-virus software business, and they set out to obtain information about the virus which enables:
1. Detection of the virus whenever it is present in a host program, and
2. Restoration of an infected host program to its original uninfected state (which is usually possible.)

Typically, a human expert obtains this information by disassembling the virus and then analyzing the assembler code to determine the virus's behavior and the method that it uses to attach itself to host programs. Then, the expert selects a “signature” (a sequence of perhaps 16 to 32 bytes) that represents a sequence of instructions that is guaranteed to be found in each instance of the virus, and which (in the expert's estimation) is unlikely to be found in legitimate programs. This”signature” can then be encoded into the scanner, and the knowledge of the attachment method can be encoded into the repairer.

Such an analysis is tedious and time-consuming, sometimes taking several hours or days, and even the best experts have been known to select poor signatures -- ones that cause the scanner to report false positives on legitimate programs.
4. Background Information

4.1. Malicious Codes

Malicious code generally means any piece of code that tends to harm a victim’s machine or perform an illegal action on victim’s machine without the knowledge of the victim. It derives its name from the ignorance of what such code could do or help to do. Malicious code as a very general definition of what we are studying; the “malicious code” could be divided into many subcategories according to the mechanism this code follows or according to a special structure and the performed operations.

4.2. Types Of Malicious Code

Malicious codes as we mentioned before are categorized according to mainly according to their characteristics. We will mention that the main three categories are the viruses, Worms and Trojan horses. We will talk about each type in order to show the differences and how can we tailor our system to adapt to the three types.

Staring by viruses we can define viruses, as Frederick Cohen in *A Short Course on Computer Viruses*: viruses are programs that can infect other programs by modifying them to include a, possibly evolved, copy of itself. With infected we mean that the virus can insert itself into the chain of commands so that attempting to execute the infected file would lead to the execution of the virus. However, we do not call every program that destroys or steals data a virus; it need not to do any kind of damages; all it has to do is replicate. Which viruses try to device unique ways of duplication and changing its appearance using polymorphism.

Worms are the second type of malicious code that we will be tackling in our project. Worms in definition are programs that usually spread across network and does not attach it self intentionally to other programs, they do not have to be viruses, though they sometimes are. Worms are said to infect OS and mail applications and not files as viruses.

Trojan horses are the third type of malicious code categories. We refer to a program as a Trojan horses if it is neither a worm nor a virus, because it does
not replicate. They do not infect files nor they spread automatically they are only transferred from machine to machine by the intention of the user, since they do not have a duplication code or section.

Worms and Trojan horses can be considered as a special case of viruses and so we will be calling all our suspected code viruses for simplicity. Viruses include all the unique features of malicious code, worms have some of them and Trojan horses have some of the some.

4.3. Structure of Malicious Codes

All three types of malicious code have a common structure thought with different stress on some parts in one type than others. We have three main sections or structure of the viruses they are as follows:

**Infection**: infection mechanism can be said to be the way that viruses spread, reproduce and infect files or OS. It differs from type to type. Basically the viruses search for an object to infect. The search could be active as the file infectors that use the directory listing to find appropriate programs to infect. On the other hand the search or infection could be passive as in the macro-viruses that tend to infect all the files when it is saved. Viruses are measured by their infection rate and how fast they spread, the faster they spread the more dangerous they are. However, some types of viruses tend to slow down there spread rate in order to avoid detection. A simple pseudo code for infection is:

```
BEGIN
  IF (infectable_object_found) AND (object_not_infected)
  THEN (infect_object)
END
```

Infection could be done through many different ways such as moving the boot sector into a new location, or addition of a jump codes in an infected program to point to an infected file or changes the system files etc.

**Payload**: is meant to be what the virus does apart from duplication, it is sometimes called “warhead”. A simple mechanism of the payload is as follows:

```
BEGIN
  IF (trigger_status_is_yes)
  THEN (execute_payload)
END
```
The Nemesis - Proposal

The payload could range from very simple one time message to complicated formatting hard drive commands.

**Trigger:** is the rule that decide when to activate the virus or execute it or deliver the payload. There are many ways of activating the virus. Some viruses are activated with a certain date, other with a special language support, others with a certain change in the registry or sometimes simply blow up as soon as patched.

**4.4 The 3G of Malicious code**

The “3G” stands for the new Third Generation of the malicious code. They are starting to evolve into a new generation of a more intelligent viruses that have knew all its enemies, AV, weak points and starts to attack them. The new generation of viruses is tending to use new techniques of attacks. The most highly expected is the collaborative way of attacks. Such viruses will be called collaborative codes; they will perform attacks. These attacks are done instead against many different users or systems without cooperation, now; they will cooperate and communicate with each other to attack one target. Resulting in a more devastating results and a more difficult to protect during the attack and will create sort of a mission impossible for antivirus analysts to track back or at least predict how would the following attack would look like. Moreover, new viruses are tending to concentrate on how to exploit Instant messaging tools that are world wide spread, which will give them, if exploited, a very powerful force to create a very wide range of attacks and a very high speed of virus spread.

**4. 5 Global Losses**

As we mentioned before, malicious code generally, since its start in 1981, started causing many damages on the individual basis as well as on the organizational basis. Viruses has grown exponentially during the last three years
and caused an enormous damage all over the world. In the following text we will concentrate only on statistics only in the last year to show where the threat of viruses stands now. About, 99.4% of the companies have encountered at least one virus attack during the last period. Mentioning that this was an increase of 20% compared to the last year. Such attack on one company, results on an average loss between 100,000 $ and $1 Million per company per virus attack. Giving another example of specific virus disaster was the love letter virus that caused damages around $10 Billion. Another astonishing statistic is that four out of the top ten viruses in the last month caused damages to around 14,000 computers worldwide in the first week of their outbreak. It took antivirus firms around 6 days on average for each virus to be analyzed and create a cure for it. Another example of the level of the threat of viruses that started to increase is that only 85% of the viruses are discovered, which shows how viruses are becoming more intelligent than to be discovered or tracked.

5. Literature Survey Of Similar Projects

5.1 The HoneyNet Project

A group of thirty security professionals dedicated to researching cyber threats. They are an all volunteer organization and the research is done on their own time with their resources. The group informally began in April, 1999 as the [Wargames] maillist. Over time the group has grown to thirty members, officially becoming the Honeynet Project in June, 2000.

**HoneyPot**

A security resource whose value lies in being probed, attacked or compromised which also has no production value, anything going to or from a honeypot is likely a probe, attack or compromise. HoneyPot is a system that is installed to capture a malicious hacker's attempts to gain unauthorized access to a system. By setting up a honey pot network, such as the Honeynet project, you can track and learn an enormous amount about how these malicious hackers gain access to systems; communicate with each other; what tools they use once they
get on, such as root kits; and where they attack after the current victim. This data lets the honest folks build countermeasures to their activities. It is important to keep in mind that the HoneyNet is mainly built for the purpose of research that is all the information gathered is used as an input for analysis.

**How It Works**

It is important to stress that a HoneyNet runs a resource (a machine connected to the internet) and leaves it open to be scanned, attacked or compromised. However, this system is not a simulation or emulation of a real system. HoneyNets use real production systems like Solaris, Linux and Windows NT. These systems are not special in any way nor are they less secure; they are exact duplicates of what many organizations run on their computers.

**Advantages**

- Reduce false negatives and false positives
- Collect little data, but data of high value
- Minimal resources
The Nemesis - Proposal

- Conceptually simple

Disadvantages
- Single point of failure
- Risk

Results of the HoneyNet Project

The figure below shows a graph that was published by the HoneyNet group, in this graph the down-spikes indicated a suspicious behavior to a HoneyPot and then the up-spikes indicate actual attacks where the system was actually compromised. The graph shows a number of “suspicious-behavior” before all the attacks which proves the theory of this system and also shows how useful it is in what it does, as some of these attacks where still new and where discovered by the HoneyNet.
5.2 Digital Immune System

It is a biologically inspired immune system that mimics the vertebrate immune system. It is developed by IBM but still under research in their Lab.

Its purpose is to analyze and cure Viruses faster than the viruses themselves can spread.

The Digital Immune System exhibits the following properties:

1. Recognition of known viruses.
2. Ability to learn about previously unknown viruses.

**Recognizing Known Viruses**

The vertebrate immune system recognizes particular antigens (viruses and other undesirable foreign substances) by means of antibodies and immune cell receptors which bind to epitopes (small portions of the antigen, consisting of at least 4 to 6 amino acids).
It is interesting to note that an *exact* match to the entire antigen is not attempted; in fact, it is almost certainly a physical impossibility.

Similarly, in the computer immune system, a particular virus is not recognized via an exact match; rather, it is recognized via an exact or fuzzy match to a relatively short sequence of bytes occurring in the virus (a “signature”). It has some important advantages. In particular,

1. It is more efficient in time and memory, and
2. It enables the system to recognize variants.

The issues of efficiency and variant recognition are relevant for biology as well.

For biological and computer immune systems, an ability to recognize variants is essential because viruses tend to mutate frequently. If an exact match were required, immunity to one variant of a virus would confer no protection against a slightly different variant. Similarly, vaccines would not work, because they rely on the biological immune system's ability to synthesize antibodies to tamed or killed viruses that are similar in form to the more virulent one that the individual is being immunized against.

**Recognize Unknown Viruses**

When the biological immune system encounters an intruder that it has never seen before, it can immediately recognize the intruder as non-self, and attack it on that basis.

Unfortunately, the notion of “self” in computers is somewhat problematic. We can not simply regard the “self” as the set of software that was pre-loaded when the computer was first purchased. Computer users are continually updating and adding new software. It would be unacceptable if the computer immune system were to reject all such modifications and additions out of hand on the basis that they were different from anything else that happened to be on the system already. While the biological immune system can usually get away with presuming the guilt of anything unfamiliar, the computer immune system must presume that new software is innocent until it can prove that it is guilty of containing a virus.
The process by which the proposed computer immune system establishes whether new software contains a virus has several stages. **Integrity monitors**, which use checksums to check for any changes to programs and data files, have a notion of “self” that is as restrictive as that of the vertebrate immune system: any differences between the original and current versions of any file are flagged, as are any new programs. However, evidence of a non-self entity is not by itself enough to trigger an immune response. Mechanisms that employ the complementary strategy of “know thine enemy” are also brought into play. Among these are **activity monitors**, which have a sense of what dynamic behaviors are typical of viruses, and various heuristics, which examine the static nature of any modifications that have occurred to see if they have a viral flavor.

In the computer immune system, integrity monitors and generic know-thine-enemy heuristics are periodically or continually on the lookout for any indications that a virus is present in the system. If one of the virus-detection heuristics is triggered, the immune system runs the scanner to determine whether the anomaly can be attributed to a known virus. If so, the virus is located and removed in the usual way. If the anomaly can not be attributed to a known virus, either the generic virus-detection heuristics yielded a false alarm, or a previously unknown virus is at large in the system.

**Eliminating Viruses**

If the computer immune system were to find an exact or fuzzy match to a signature for a known virus, it could take the analogous step of erasing or otherwise inactivating the executable file containing the virus.

If an unknown virus is encountered, it tries to disinfect it by means of **generic disinfection heuristic**.

**Generic disinfection heuristic** The underlying principle of the generic disinfection heuristic is that a program F' infected with a virus can generally be restored to its original, uninfected version F. The reason for this is simple. After a virus has carried out its own function, the best way for it to avoid immediate detection is to allow its host program to continue normally. This generally means that the virus must be able to
reconstruct the host in its original form, which in turn requires that none of the original host bytes be destroyed. In other words, for non-overwriting viruses, $F'$ can be regarded as a reversible transformation of $F$, i.e. the original host $F$ is in principle reconstructible from $F'$. On the other hand, legitimate changes such as updating a program to a new version are unlikely to be similarly reversible: too much will change, and some bytes that were contained in $F$ will no longer be present in $F'$.

Therefore, if a program has been modified from $F$ to $F'$ and the generic disinfection algorithm could recover $F$ from $F'$, it may be concluded with a high degree of confidence that $F'$ contains a virus. If so, the user is given the option to carry out the disinfection, and a sample of the program is captured.

**Generic disinfection performance** The generic disinfection heuristic has an extremely low false positive rate. Since its incorporation into IBM Anti-Virus in 1994, there have been no confirmed reports of a false positive. The false negative rate is also extremely low. In laboratory experiments, it caught over 99% of all non-overwriting virus infections. The proportion of viruses that overwrite their hosts is about 15%, but none of these are known to spread successfully because they are too harmful to go undetected for long. Thus, in practice generic disinfection is extraordinarily effective. Of course, there is one important drawback: generic disinfection can only catch viruses in programs that have been seen in their uninfected state. But if a file-infecting virus spreads appreciably on a machine, the generic disinfection algorithm will almost certainly catch it.

An additional feature, which the immune system developers refer to as the “kill signal”, would be used by a computer to inform neighboring computers on the network that was infected. The signal would also convey to the recipient any signature or repair information that might be of use in detecting and eradicating the virus. If the recipient finds that it is infected, it would send the signal to its neighbors, and so in. if the recipient is not infected, it does not pass along the signal, but at least it has received the database updates effectively immunizing it against that virus.
**Figure**: Fighting self-replication with self-replication. When a computer detects a virus, it eliminates the infection, immunizes itself against future infection, and sends a "kill signal" to its neighbors. Receipt of the kill signal results in the immunization of
uninfected neighbors; infected neighbors are both immunized and prompted to send kill signals to their neighbors. Thus detection of a virus by a single computer can trigger a wave of kill signals that propagates along the path taken by the virus, destroying the virus in its wake.
An immune system for computers is desirable and feasible. As suggested in the Fig., most of the necessary components are already in use in one form or another. Some already exist in IBM AntiVirus itself. Others are presently in use in the virus laboratory, for the purpose of updating the databases employed by IBM AntiVirus to recognize viruses and repair infected files.

5.3 Automatic and Controlled Virus Code Execution System

The system is not an emulation of PC environment. In contrast actual virus code is executed in a normal PC in conditions corresponding to the normal usage of a PC. Critical parts of the system such as victim PC's cold booting and keyboard control system are implemented by customized hardware solutions. The system is completely automatic and it can be left for working on its own. The system enables such tasks which were not previously possible or which at least required enormous work. The system enables an automated analysis of scanners, active monitoring and checksum calculation programs when viruses are memory resident and other tasks which require execution of virus code. The next improvement of the system could be a direct memory access of the victim PC via the monitoring PC. This enables to trace a virus and to read the victim PC's screen memory directly via the monitoring PC.

The components of the system

The automatic and controlled virus code execution system consists of the victim PC, the monitoring PC, the network server, the keyboard control device and the control probes for boot device switching and cold booting the victim PC (figure).
The monitoring PC and the victim PC

The monitoring PC controls critical operations of the victim PC such as resetting the victim PC whenever needed. The monitoring PC can also control the victim PC's keyboard by using the keyboard control device. When the victim PC is resetted, the monitoring PC can also select which boot device is used for booting the victim PC. The victim PC can be automatically booted from either 3.5" floppy drive, from 5.25" floppy drive, from fixed disk and from the network.

On the monitoring PC there is running a monitor program, which waits for a set of tasks from the program running on the victim PC. The program does what victim PC tells it to do and after all tasks have been executed it waits for another set of tasks. A typical task received from the victim PC can be something like “I am intending to execute virus code. If I am not alive after execution of the virus code, please cold boot me from the network.” The victim PC can send tasks to the monitoring PC only when it is in 'clean state' i.e. no virus code has been executed before the previous clean boot. Although the victim PC cannot send tasks when virus code has been executed it can send information that everything is working properly and the monitoring PC knows that so far there is no need for cold booting. The monitoring PC keeps also a record of the received tasks and changed storage areas of the victim PC.
The Network server

Processed suspect files are picked up one by one from a network directory containing source files (V:\SOURCE). Since suspect files may be originally stored on a large directory tree, the source directory can also include subdirectories. After process of a suspect file has been completed the file is moved from the source directory to a process directory (V:\PROCESS) and a subdirectory corresponding success of the process is created. Creation of directory V:\PROCESS\VIRUS means that executing suspect file caused change on files or boot sectors. Creation of directory V:\PROCESS\HANG means that execution of the file caused victim PC to hang. Creation of directory V:\PROCESS\HANG_2 means that execution of the suspect file did not hang PC but later at some point victim PC hanged. Creation of directory V:\PROCESS\NOHANG means that execution of the suspect file did not cause hanging at any point, but nothing was changed. V:\PROCESS\CVIR means that fixed disk boot sector or partition sector was changed, but the possible virus was not capable of spreading further. V:\PROCESS\FORMAT means that execution of the suspect file caused fixed disk malfunction. When analysing boot sector viruses, creation of directory V:\PROCESS\UNKNOWN means that boot sector image type could not be identified. If change was detected changed files and boot images are moved to a target directory (V:\TARGET) and a subdirectory corresponding the original path and file name of the suspect file is created. For example if the original suspect file was stored in path V:\SOURCE\FORM\A\FORM.BOO and change was detected...
on diskette in the floppy diskette drive A, changed diskette image would be written to a file V:\TARGET\FORM\A\FORM.BOO\DRIVE_A.ZIP and the original suspect file would moved to a directory V:\PROCESS\VIRUS\FORM\A.

The network is utilized also by storing clean fixed disk and floppy disk images on the network server. The victim PC's clean state can be automatically stored by writing the clean fixed disk image on the victim PC's fixed disk. The network is used also for clean booting the victim PC. There is a clean boot image stored on the network login directory and victim PC's boot ROM uses the boot image whenever clean boot is needed.

Conclusion

The automatic and controlled virus code execution system can be used as a powerful tool for reinfecting viruses and for other tasks which require execution of virus code. The system is completely automatic and it can be left for working on its own. The system can save enormous work efforts and it frees resources for other tasks. The drawbacks of the system are that it can infect viruses only under DOS and all possible viruses cannot be reinfected, because some viruses may spread only under some special conditions, which are not covered by the system.
Victim PC

- Get next file to process
- File type
  - Sys
  - Com/Exe
- Run process
- Completed normally

- Does drive C: exist?
  - Yes
    - Has CMOS changed?
      - Yes
        - Restore CMOS
      - No
        - No
          - Has image been restored previously?
            - Yes
              - Move changed objects to V:
            - No
              - Low level format hard disk
              - Move processed file to V:PROCESS
          - No
            - Move processed file to V:PROCESS
        - No
          - Move processed file to V:PROCESS
    - No
      - Has executable files changed?
        - Yes
          - Restore clean hard disk
          - Move processed file to V:PROCESS
        - No
          - Change hard disk settings

Monitoring PC

- Boot from hard disk
- Wait for victim PC to respond
- No response
  - Boot from hard disk
  - Wait for victim PC to respond
  - No response

- Has thorough process been executed?
  - Yes
  - Change hard disk settings
  - No
6.1 Contained Environment

6.1.1 Sandboxing

What is Sandbox?

Sandboxing technology is a security solution to protect workstations and networks against attacks from any kind of active content (ActiveX, Java and other executable code) received from the Internet, and to manage behavior of already installed applications residing on the PC.

Why Sandboxes?

It's this fear of the unknown that's starting to scare some IT managers into adding behavior-blocking, or "sandboxing," technology, as a last line of defense at the desktop. Behavior blocking prevents malicious code from doing something it's not authorized to do. If a downloaded executable program tries to erase the PC's hard drive or copy its address book, for instance, the software stops it cold. The desktop is the final frontier for many of these malicious attacks because that's where Internet code, such as macros, scripts, executables, screen savers, plug-ins, Java and ActiveX apps, typically run.

Theory of operation

With sandboxing technology you can create a closed environment (sandbox) around any application (known or unknown) and restrict its access to your computer's resources. Within this closed environment any code can run and access calls of the application to system resources. Drivers, the registry database (all configurations), and the file system are shielded and constantly monitored to protect the privacy and integrity of your system. Unlike traditional antivirus technology-which scans for known virus characteristics using its signature, database-behavior-blocking software uses policies to determine whether an application or a piece of code is acting out of line; if it is, it's terminated and its actions are blocked.

SafeTnet:
As an extra layer of protection to the traditional anti-virus systems, it is an idea to run (as a last Security level) the SafeTnet behavior-blocking software on the PC. Like other behavior tools, the SafeTnet software builds a sandbox around any applets or other code that is downloaded. It detects any attempts at harmful behavior, such as an executable attempting to modify a system registry, and alerts the end user and the administrator. This way, questionable code can be watched offline in the sandbox to determine its intentions. The SafeTnet application sits between the virus code and the operating system and intercepts any system calls. It checks its policy database to determine just what the code is allowed to do and then blocks any unauthorized activity. Then security managers can direct filters to block that subject from returning

**Control Mechanism**

**Resources Protection:** Sandbox Agent protects the computer resources against unwanted and suspicious accesses and changes; for example, the windows operating system saves the system and application configurations within the registry database. If a hostile applet changes settings within the registry database, it might leave applications or your entire system unusable. By changing the registry database, a hostile applet can also gain unwanted access to resources on your computer.

**Access to services:** Sandbox Agent monitors all access to system services issued from restricted applications. By changing the settings of particular services (stopping or accessing certain services), a hostile applet can make your computer unusable or gain unwanted access to resources and data.

**Dangerous calls to the system:** Certain functional interfaces of the system are intended for Windows internal use, or special application use only. There is no reason for these to be used in normal circumstances. Also, dangerous device-level access is protected. Sandbox Agent restricts the availability of these system entry points for restricted applications.
Access to the file system: By accessing the file system, an applet gains access to all your data and files. Sandbox Agent can restrict access to the file system depending on its configuration. In a typical scenario, you might want to set up a directory for saving information received from the Internet while blocking browser access to all other file areas.

Access and monitoring of IP ports and IP addresses: Sandbox Agent can monitor access to IP ports by restricted applications. By using certain IP ports an applet can e-mail information to the Internet or connect using any other protocol.

Spawning of processes control: Sandbox Agent can prevent a restricted application from running other applications or inheriting another application’s access to a secured environment. This can prevent misuse of trusted applications by hostile code.

Drawbacks

False alerts: One of the well known and irritating drawbacks of Sandboxes is that they have a high percentage of false alerts; whenever you attempt to install a new application, you will probably be faced by a false warning signaling that a malicious code is loose on the system.

Efficiency: Also Sandboxes by definition impose an overhead on the system due to the preliminary stage of testing the virus intentions.

Not easy to integrate: Obvious as this may look, but this poses a problem at many times, especially if one decides to resolve solely to behavioral-monitoring solution.
No “reliable” open-source solution: To us, this was the major impediment to our research and subject choice; we could not find a good developer's documentation for Sandboxes.

6.1.2 Emulators

An emulator is a software that simulates a running environment under a specific operating system. As it is the case with a lot of software, an emulator is written for a specific operating system; so we have emulators that simulate linux for windows, and more importantly emulators that simulate windows under linux. Thus, combining the benefits of having an open source OS while developing an application for a predominant OS (windows).

WINE

WINE or WINdows Emulator is an implementation of the Windows 3.x and Win32 APIs on top of X and Unix. Think of Wine as a Windows compatibility layer. Wine provides both a development toolkit (Winelib) for porting Windows sources to Unix and a program loader, allowing unmodified Windows 3.1/95/NT binaries to run under Intel Unixes. Wine is not quite an emulator; it is a free implementation of Windows API's on Linux. Allowing most Windows applications to run on Linux without modification

How it works

Wine is a single process which translates Windows calls, including undocumented calls that applications need (and a few DOS int21 calls as well) to X and Unix calls, respectively. It is also responsible for properly loading Windows applications. Wine reads the executable file, and correctly loads the code, data, and resources into memory.

Wine uses only one non-standard system call, which is required to be able to run 16-bit code instead of 32-bit code. Therefore, it is relatively simple to port Wine to operating systems which conform to POSIX (more or less), have X, and for which the source is free, or which provide an appropriate alternate non-
standard system call for setting up the LDT so that 16-bit code can run. Wine was originally developed for Linux, but the port to FreeBSD and NetBSD took less than a week. Wine actually translates the Windows API (Application Programming Interface, how Windows applications call Windows) into equivalent functionality available through the standard Unix and X interfaces. When a Windows program creates a window, Wine converts that into a call to create a window through the standard Xlib library. When DOS interrupts are called, for example to read a file, Wine translates them into Unix system calls.

Wine also implements other API's that are available for Windows. For instance, the WinSock API is becoming the standard way of accessing TCP/IP networking under Windows, and Wine provides the functionality of WinSock, mapping it to the standard Unix socket calls.

Finally, wine loads Windows binaries; it also loads the closely related DLLs (dynamically linked libraries) which most applications require. Wine itself uses a DLL called sysres.dll, in which are stored all the bitmaps for the standard buttons on the title bar.

**Drawbacks of WINE**

A very important to realize issue about WINE is that it does not emulate windows 100% accurately for all applications. This means that most windows applications will run in a similar manner under windows, however we cannot rely on WINE’s reliability whenever the situation demands high accuracy. Another major drawback about WINE is that the original version was written for Windows 3.11. Thus, it included support for the available API’s at the time. Even the updated versions found today which extend these API’s are old and not reliable.

**Disadvantages of Emulators**

*OS dependent:* as mentioned before, an emulator is written for a specific operating system.
Degraded performance: the overhead imposed by having to translate every CPU instruction, data transfer and function call is sometimes unacceptable.

Complexity: an emulator becomes usually one complex program that is hard to get a perfect functionality of it. Thus,

Imperfect: emulators simply do not offer what they promise to deliver; i.e. they are not hundred percent reliable.

6.1.3 Virtual Machines

History of Virtual Machines

The idea of a virtual machine is not new--its roots go back almost to the beginning of computing itself. Initially, the concept of a virtual machine came about in the 1960's on mainframes as a way to create less complex multi user time-share environments. As described by Melinda Varian's canonical paper "VM and the VM Community: Past, Present, and Future", a time sharing system was developed by a group of MIT programmers, working with equipment donated by IBM. The system, "Compatible Time Share System" or CTSS, was initially developed in 1961 and evolved over the years as the example of how to do time-sharing. CTSS was designed much like current multitasking systems, doling out processing time in scheduled slices. The system provided a supervisory program that controlled resources, and scheduled time shares for foreground and background tasks. The key to its operation was the supervisor program's control of Trap Interrupts. By trapping interrupts, the control program was able to isolate the users or processes from each other.

As the systems developed, changes in the hardware were made to support relocation of memory, a key process to facilitating a virtual memory system. Without the ability to relocate (page) memory, entire programs would need to be swapped in and out to active memory address space. With virtual memory, a big performance boost could be realized.
In the 1960's, the concept of a "virtualizable machine" was developed, and virtual machine technology became a very popular subject of study, and the key focus of user organizations and conferences in the late 60's and 70's. For some deeper background, in late 1964, a project at MIT called CP-40 (Control Program for IBM System/360 Model 40 mainframes) really turned the corner on virtual machines. The idea was to create an operating system that would let each mainframe user have their own IBM System/360 virtual machine (which was originally called a pseudo machine). The subsequent release of a single user virtual environment running atop CP-40 called CMS (Cambridge Monitor System) was the beginning of a long line of IBM VM operating system products. Later CMS would be called Conversational Monitor System, and it worked in conjunction with CP on IBM System/370 systems. Though IBM was reluctant to invest in VM technology in the 60's and 70's, it became fairly successful in the 80's, and they still sell VM systems today.

Idea of Virtual Machine Monitor

The idea behind running a VMM under the Microsoft operating systems is to have a virtual machine without requiring core OS modifications. As an extension to that concept, Mendel and his graduate students started experimenting with creating a virtual machine monitor that can also run other commodity operating systems within a VM for use on single or multiprocessor systems.

Virtual Machine Definition

A Virtual Machine (VM) is defined by Popek and Goldberg (in their paper "Formal requirements for virtualizable third generation architectures," Communications of the ACM, Vol 17, July 1974) as an "efficient, isolated duplicate of a real machine". A real machine has a number of systems that it provides to an operating system and applications for use. Starting at the core, the CPU and motherboard chipset provides a set of instructions and other foundational elements for processing data, memory allocation, and I/O handling. On top of that are hardware devices and resources such as memory, video, audio,
disk drives, CDROMs, and ports (USB, Parallel, serial). In a "real machine", the
BIOS provides a low level interface that an operating system can use to access
various motherboard and I/O resources. With a real machine, when an operating
system accesses a hardware device, it typically communicates through a low-level
device driver that interfaces directly to physical hardware device memory or I/O
ports.

A VMM manages one or more VMs, with each VM providing facilities
for an application or "guest OS" to believe it's running in a normal environment
with access to physical hardware. Instead, when applications or guest OSs execute
low-level instructions that inspect or modify hardware state, they appear to the
OS to be directly executing on the hardware, but are instead virtualized by the
VM and passed to the VMM. For traps or interrupts occurring at the application
level, they can pass directly to the VMM, which in turn interacts with the
hardware.

A virtual machine is an environment created by a Virtual Machine
Monitor (VMM). The VMM can create one or more VM's on a single machine.

Definition of VMware

VMware Workstation is virtualization software for software developers
and other technical professionals. VMware Workstation runs multiple operating
systems in secure, transportable and high-performance virtual machines on
physical computers.
Hardware Virtualization

As mainframe hardware and software evolved, a concept of virtualization of the hardware developed. A protected processor has two basic modes (and associated instructions)--Privileged (aka System, Supervisor, or Kernel mode) and Non-privileged (aka User mode). Popek and Goldberg's rules of virtualization state that a virtualizable processor architecture allows any instruction that inspects or modifies machine state to be trapped when executed in any but the most privileged mode. When a trap occurs, rather than generating an exception and crashing, the instruction is sent to the VMM. This allows the VMM to take complete control of the machine. The VMM then either executes the instruction on the processor, or emulates the results and returns them to the VM or app. Some instruction results may be cached by the VMM, so the instruction doesn't need to be reissued. For example, an instruction that gets the state of a port can be cached if the port hasn't changed and the VMM knows it.

In a typical mainframe configuration, the VMM runs on top of the hardware in privileged mode, and the VMs run in user mode. When the VM issues non-privileged instructions to the processor, the VMM passes them through to be
executed directly. For example, an instruction such as Add or Sub, doesn't affect the state of the hardware so the VMM executes it directly on the hardware. Most instructions fall into this category, so the performance of many applications running within a VM vs. the Real system is better than running under emulation. Privileged instructions can only be executed by a process running in a privileged mode of the same or higher level. Privileged instructions issued by processes running in user mode cause a trap. This provides the basis for the isolation of a VM from the rest of the machine. Processors include instructions that can affect the state of the machine, such as I/O instructions, or instructions to modify or manipulate segment registers, processor control registers, flags, etc. VMware calls these "sensitive" instructions. Such sensitive instructions can affect the VMM and other processes if executed by a given process without regard for other running processes. In a virtual machine, sensitive instructions cannot be executed on the CPU directly by applications or guest OSs for that reason. A primary job of a VMM is to trap and handle all such machine-specific sensitive instructions.

For a processor to be fully virtualized, all sensitive instructions are privileged and are able to be trapped. On traditional virtualized mainframes, when privileged instructions are issued from the VM, they are trapped and passed to the VMM for handling. A specific example of a sensitive instruction would be PushF (push flags) and PopF (pop flags) instructions that save and retrieve the current state flags of the CPU.

As mentioned earlier, one or more processes running in a "real machine" would rely on the OS to manage sensitive instructions. With multiple programs running and issuing sensitive instructions, the OS multiplexes among them and issues one protected instruction at a time to the CPU.

With a VMM environment, multiple VMs may each have an OS running that wants to issue sensitive instructions and get the CPU's attention. This is why the hardware needs to be virtualizable to allow a VMM to keep each VM isolated. On a mainframe, the hardware, and more specifically, the processor, is designed to be virtualizable.
VMware Products

VMware offers three virtual machine systems—Workstation, GSX Server, and ESX Server. Workstation is hosted on Windows NT, Windows 2000, or Windows XP Professional. A Linux-hosted version is also available. GSX Server is similar to Workstation in looks, and is hosted on Windows NT or Windows 2000 Server, or Linux. ESX Server is a stand alone system, running directly on the hardware itself, without the need for a host operating system.

VMware Installation

When Workstation installs, it creates three main components—the VMX driver, Virtual Machine Monitor (VMM), and the VMware application (VMapp). The VMX driver and VMM operate at Ring 0. The Workstation application runs in Ring 3 and looks like any other application to the host operating system. During installation, the VMX driver is installed within the operating system to gain the high privilege levels that drivers are permitted and that the VMM requires. When executed, the application (through the VMX driver), loads the VMM into Kernel memory, giving it Ring 0 privileges. At this point, the host operating system only knows about the VMX driver and the Workstation application, and is completely ignorant of the VMM. Also at this point, your machine contains two "worlds"—the host world and the VMM world. The VMM world can communicate directly with the processor hardware or through the VMX driver to the host world. When a "world switch" is done between the VMM and host worlds, all user and system CPU state needs to be saved, which takes a toll in terms of performance as compared with a real system, or a mainframe VMM.
Persistent, Non-persistent, and Undoable Drives

One common usage for Workstation is to create multiple instances of a single operating system so you can change various settings, but start with the same base. There are several ways to do this--make separate VMs and install fresh copies of the OS, make copies of an original VM file, or create a single file and reuse it without saving changes.

Workstation gives the users of working in Persistent or Non-persistent or Undoable drives. A persistent drive is the default mode and is the basic virtual drive, where any software additions, configuration changes, or file deletions are immediately applied. A persistent drive acts like any physical hard drive. You would use persistent mode to install applications and configure an operating system in a VM.

A non-persistent drive is the antithesis of the persistent drive, where anything the user does in a session is discarded when you close the session. This is great for testing truly unstable software which can corrupt an operating system. Workstation maintains a Redo file during your session, and when the user exits, it is deleted. The name "Redo" is somewhat of a misnomer, because the file is only used during the current session, and it contains the session changes to the VM, rather than having changes written directly to the VM file itself. The Redo file doesn't exist beyond the current session.
The compromise between the two previous modes, and probably the best of both worlds is the Undoable mode. Undoable mode allows you to work in a session, and when exiting gives you the option to commit, discard, or save the changes that were made. During an Undoable mode session, Workstation keeps a Redo file like the non-persistent mode, but gives the user the option of committing, discarding, or saving. If you commit, all your changes are applied to the base VM. If you discard, they are thrown away. However, if you save, the file itself is preserved and can be used in another session. When the user power-up a VM session later, it gives you the option to commit, discard, or append the earlier change file. This allows you to do continue a previous day's work without committing, or losing your changes.

**Peripherals and VMware**

For high throughput devices such as a network controller, the processing overhead of the switching between the host made and the VMM mode can become a problem (World Switch). To overcome some of the overhead problem with world switches, the device emulation in VMware's VMM understands the context in which I/O requests are made, and uses that knowledge to reduce world switches. In the case of the virtual network card, the VM will make lots of I/O requests, but only a percentage of them will actually require moving packets. The rest are either getting or setting the state of the I/O port. Since the VMapp translates raw I/O requests into higher level network requests, the VM only really needs to pass packet requests with a world switch. A large part of a network packet transaction consists of In/Out requests that are just getting the state of the ports. For these requests, the VMM emulates the data that the VM is looking for. This technique is uses on other I/O devices as well, minimizing the performance penalty of going to the host OS. The following diagram shows the steps that a packet send and receive goes through in VMware Workstation.
Networking

Networking is one of the most interesting areas in the VM. Networking is installed whether or not your host OS is connected to a network. Networking can be configured in several ways—Bridged, NAT, and Host Only, as well as a Custom mode that gives more low level control. The configuration editor provides full control of configuring networking for a virtual machine.

VMware’s network architecture (courtesy of VMware)
Up to three virtual network cards can be configured in each VM, and each appears to the Guest OS as an AMD PCNet II card-- your average, everyday generic Ethernet card. There are also nine virtual Ethernet switches, designated as VMnet0 - VMnet8, that exist within the PC. Workstation 2.0 only supported four virtual switches. Each switch can connect to one or more virtual network cards. By default, some of the VMnet switches have specific functions: VMnet0 is dedicated to bridged mode, VMnet1 is dedicated to Host Only mode, and VMnet8 is for NAT (Network Address Translation) mode. The others, VMnet2 - VMnet7, are available to be used by the VM when using Custom networking mode.

Bridged networking is the default (unless you opt for no networking), and it allows your VM to talk to the outside world via your host machine's network card. VMnet0 connects your virtual network card to the physical network card. While the VM shares the host's Ethernet connection, it appears as an entirely separate machine on the local Ethernet with its own MAC and TCP/IP address. For TCP/IP connections you can either set an IP address, or get a dynamic address from an outside DHCP server. If you can't allot more than one IP address for a physical machine (your IT guy may have something to say about this), then skip ahead to the NAT mode. Bridged mode by default connects the VM with VMnet0 which maps to the first available NIC automatically. If you have more than one physical network card in your Host machine, it is recommended to manually map your virtual switches to physical adapters.
In bridged mode, the packets are created in the VM, and sent through the switch to the physical network card. Since a virtual machine has its own MAC address for each virtual NIC, the wireless adapter will not send packets other than those generated on the physical wireless card. However, the workaround for using a wireless network card is to use NAT mode.

NAT mode is new with Workstation 3.0. It allows the virtual machines to share the IP address of the host system. The Workstation DHCP server assigns IP addresses to the virtual network cards, and the NAT system translates packets between the host's network card and the guest OS. VMs using NAT connect via the VMnet8 switch. Workstation configures a separate NAT service that runs on the host OS and connects between the VMnet8 switch and a TCP/IP socket on the host OS. The host routes traffic from that socket to wherever needed, such as an Ethernet, Token Ring, or dialup connection. With the NAT service, the wireless Ethernet adapter just sees normal TCP/IP traffic from the host OS and no special
bridge software trying to send packets with other MAC addresses.

For Host Only mode, VMware installs a virtual Ethernet adapter in the host OS that communicates with the VMnet1 switch. The host OS believes this is just another Ethernet adapter, and can be viewed from the standard Windows (host) network properties panel. Workstation also runs a virtual DHCP server connected to the VMnet1 switch. In this mode, the DHCP service will assign addresses to the VM’s virtual Ethernet adapters (and actually the host OS’s virtual Ethernet adapter as well) that are connected to the VMnet1 switch. This allows communication between a virtual machine and the host operating system, but it is not routed to the outside world. Multiple virtual machines can talk to each other as well. Host Only networking also allows connection of virtual machines to outside networks through the host OS. This is done by configuring network routing, or even a firewall, on the host OS between a physical network and the Host Only virtual Ethernet adapter in the host OS.

Using Custom mode, the user can directly specify the connection between a virtual switch and virtual Ethernet adapter. Virtual machines connected though the unused virtual Ethernet switches (VMnet2 through VMnet7) are completely isolated from the host OS or any local physical networks. This mode can be helpful in testing the malicious code that we want to isolate.
The Workstation DHCP capability is a modified DHCP server that can only serve IP addresses to the VMnet switches. The DHCP server assigns IP addresses from a class C non-routable subnet. This is configured automatically at Workstation install time, and the installer checks to make sure the subnet is not already used by the host OS. The DHCP server normally only serves IP addresses to virtual machines connected to VMnet1 (Host Only) or VMnet8 (NAT), but can also be configured to serve IP addresses to the other virtual switches.

Deciding which networking mode is basically a matter of whether the user wants the physical network to see the virtual network, or not. If the user wants the physical network to see the virtual network, then the Bridged mode would be the way to achieve that. Else if the user wants an isolated virtual network without outside connectivity, then the Host Only mode would be the way to achieve that. Else if the user needs outside connectivity, but wants the virtual network invisible to the physical world, use NAT mode.
Advantages of VMware

- Ensuring the isolation of the virus.
- Near native performance which is proportional to the speed and memory of the native pc.
- Support for virtual networking.
- less time installing and recovering operating systems.

Disadvantages of VMware

- For I/O intensive operations, the performance degradation is evident in comparison to a real system.
- The performance of the VMM and ultimately any guest OS or application running within a VM is dependent on the scheduling and allocation decisions that the host OS makes.
Host OS scheduling of priority levels for any or all VMs can be set in the Workstation preferences panel.

- Overhead problem as a result of context switch between the two modes of the VMware to support networking.

6.2 Platform

We faced the problem of choosing the platform that we will work on. We had to choose between Linux or Windows operating systems. Linux of course had the advantage of being an open source operating system while Windows is not. This criterion of Linux being an open source was a great advantage for us as it would have eased our job in implementing our monitoring subsystem. While through our research in our problem domain we found out that the number of malicious codes targeting Windows operating systems exceeds fifty thousand ones. Which induces a far greater threat that induced by Linux’ malicious codes. Also Linux operating systems is not as widespread as Windows.

So in order to address the proliferation of malicious codes in an effective way we settled down on using Windows operating system or at least we will use it as our guest operating system.

6.3 Monitoring Techniques

6.3.1 Windows API

**Base Services**

The base services functions give applications access to the resources of the computer and the features of the underlying operating system, such as memory, file systems, devices, processes, and threads.

**Performance Monitoring Techniques**

**Event Tracing**

Each operating system provides events that can be logged using event tracing, such as disk I/O and page faults. Applications can also define their own event types for use with event tracing. Examples of application events
are the start and end of a certain transaction, such as a search operation. Events are delivered in real time or recorded to log files.

**Tool Help Library**

The functions provided by the tool help library make it easier to obtain information about currently executing applications.

**Snapshots of the System**

Snapshots are at the core of the tool help functions. A snapshot is a read-only copy of the current state of one or more of the following lists that reside in system memory: processes, threads, modules, and heaps.

Processes that use tool help functions access these lists from snapshots instead of directly from the operating system. The lists in system memory change when processes are started and ended, threads are created and destroyed, executable modules are loaded and unloaded from system memory, and heaps are created and destroyed. The use of information from a snapshot prevents inconsistencies. Otherwise, changes to a list could possibly cause a thread to incorrectly traverse the list or cause an access violation (a GP fault). For example, if an application traverses the thread list while other threads are created or terminated, information that the application is using to traverse the thread list might become outdated and could cause an error for the application traversing the list.

A snapshot of the system memory can be taken by using the CreateToolhelp32Snapshot function. The content of a snapshot can be controlled by specifying one or more of the following values when calling this function:

- `TH32CS_SNAPHEAPLIST`
- `TH32CS_SNAPMODULE`
- `TH32CS_SNAPPROCESS`
- `TH32CS_SNAPTHREAD`
The TH32CS_SNAPHEAPLIST and TH32CS_SNAPMODULE values are process specific. When these values are specified, the heap and module lists of the specified process are included in the snapshot. If a zero is specifies as the process identifier, the current process is used. The TH32CS_SNAPTHREAD value always creates a system-wide snapshot even if a process identifier is passed to `CreateToolhelp32Snapshot`.

Enumerating the heap or module state for all processes can be done by specifying the TH32CS_SNAPALL value and the current process. Then, for each process in the snapshot that is not the current process, you can call `CreateToolhelp32Snapshot` again, specifying the process identifier and the TH32CS_SNAPHEAPLIST or TH32CS_SNAPMODULE value.

You can retrieve an extended error status code for `CreateToolhelp32Snapshot` by using the `GetLastError` function.

When your process finishes using a snapshot, you should destroy it by using the `CloseHandle` function. Not destroying a snapshot causes the process to leak memory until the process exits, at which time the system reclaims the memory.

**Note** The snapshot handle acts like a file handle and is subject to the same rules regarding which processes and threads it is valid in.

**Process Walking**

A snapshot that includes the process list contains information about each currently executing process. Information can be retrieved for the first process in the list by using the `Process32First` function. After retrieving the first process in the list, traversing the process list for subsequent entries is done by using the `Process32Next` function. `Process32First` and `Process32Next` fill a `PROCESSENTRY32` structure with information about a process in the snapshot.

6.3.2 Subclassing
API spying utilities are among the most powerful tools for exploring the inner structure of applications and operating systems. Unfortunately, neither the SDK nor the DDK provide any documentation or examples demonstrating a way for implementing such a utility.

Spying upon a single application or set up a system-wide API interceptor can be done through multiple ways.

One way to do that would be to establish a system-wide API interceptor that monitors calls made to the CreateProcess function (actually, to both the Ansi and Unicode versions of this function). Whenever an application issues a call to one of these functions in order to create a new process, the interceptor will gain control and perform whatever processing is necessary.

**Several API hooking techniques:**

**Proxy DLL**

As an example for a use of this technique, consider an anti-virus application that scans incoming email messages for viruses. An obvious requirement for such an application is the ability to hook Winsock's I/O functions in order to analyze data transfers between email clients and remote mail servers.

This can be easily accomplished by creating a proxy DLL, which contains a stub for each of the functions exported from the Winsock library. If the proxy DLL is named exactly like the original Winsock library (i.e. wsock32.dll) and placed in the same directory where the target email application resides, then the interception occurs automatically. When the target application attempts to load the original Winsock library, the proxy DLL is loaded instead. All the calls made to Winsock's functions are routed to the exported stubs in the proxy. After performing the necessary processing, the proxy DLL simply routes the calls to Winsock and returns control back to its caller.
This technique has one major drawback - hooking a single function located in a DLL that exports 200 functions would require creating a stub for each of these functions in your proxy DLL. This could be rather tedious and also impossible at times, when some of the functions are not fully documented.

**Patch those calls**

When thinking about ways for intercepting an API call, there are two locations where one can intervene - either at the source of the call (the application code) or at the destination (the target function). This technique relies on the first option.

For intercepting each API function, patching all the locations in the target application where calls to this function are issued. The modification can be done either on disk (to the executable file itself) or in memory (after the executable is already loaded). The tough part is pinpointing the exact locations where patching is necessary. In order to accomplish that, a disassembler should be implemented that is capable of analyzing assembly instructions. Obviously, writing a disassembler is far from being a trivial task, making this API interception technique one of the least popular among the group.

**IAT Patching**

The foundation of this technique relies on the fact that 32-bit Windows executable files and DLLs are built upon the Portable Executable (PE) file format. Files based on this specification are composed of several logical chunks known as sections. Each section contains a specific type of content. For example, the .text section holds the compiled code of the application while the .rsrc section serves as a repository for resources such as dialog boxes, bitmaps and toolbars. Among all of the sections present in a Windows executable file, the .idata section is particularly useful for those who wish to implement an API
interceptor. A special table located in this section (known as the Import Address Table) holds file-relative offsets to the names of imported functions referenced by the executable's code. Whenever Windows loads an executable into memory, it patches these offsets with the correct addresses of the imported functions.

In the current implementation of Windows executables and DLLs, calls made to imported functions are routed through the IAT using an indirect JMP instruction. The fact that imported function calls are "drained" through one location saves Windows the trouble of traversing the executable image in memory, looking for call instructions that are destined for patching.

By overwriting a specific IAT entry with the address of a logging routine, an API interceptor can gain control before the original function gets a chance to be executed by the processor.

Obviously, there are other issues involved in the implementation of this technique, such as the requirement for the logging code to be executed in the memory context of the intercepted application.

**Patch the API**

It has the inherent advantage of being able to trace API calls issued from different parts of an application while requiring a modification only to a single location - the API function itself.

There are several approaches that can be used here. One option is to replace the first byte of target API with a breakpoint instruction (Int 3). Any call issued to that function would generate an exception, which would be reported to your API interceptor in case it serves as a debugger of the target process. Unfortunately, there are several problems with this approach. First, the poor performance of Windows exception handling mechanism would considerably slow down the system. A second problem is related to the implementation of Windows debugging API. As soon as a
debugger shuts down, it terminates all the applications that were under its control. Obviously, such a behavior is completely unacceptable in case you're implementing a system-wide interceptor, which must be able to terminate itself before its target applications cease executing.

Another possibility is to patch the target function with one of the control-transfer instructions of the CPU (i.e. either a CALL or a JMP). Once again, there are several problems with this approach. First, it is possible that the patching would overrun the end of the intercepted function. This can occur in cases where the target API is shorter than 5 bytes (CALL and JMP are each 5 bytes long). Another issue is the need to constantly switch between the patched and "unpatched" versions of the intercepted function. This means that once your logging routine receives control from the CPU, it must restore the intercepted function to its previous unhooked state. This is required to allow the API interceptor to route the call to the original function without generating an infinite loop of calls back to the logging routine. Note that during the time the CPU is executing the original function, other calls to that function might be issued from different parts of the system. Since the function is in the unhooked state at that stage, the API interceptor will miss those calls.

In this case, the API interceptor places a JMP instruction at the beginning of the target function, but not before saving the first 5 bytes of the function to a pre-allocated buffer in memory (a stub). The exact number of bytes to be copied to the stub may change depending on the instructions present at the head of the function. It cases where 5 bytes do not fall within instruction boundary, it is necessary to copy additional bytes until there is enough space for the JMP instruction to be inserted. Note that relative control-transfer instructions (i.e. JMPs and CALLs) need to be modified during copying to ensure that they transfer control to the right location in memory when executed from the stub. Obviously, performing such an analysis of assembly instructions requires the assistance of a disassembler, which, is not very easy to implement.
Breaking address space barriers

After having an API interceptor that is capable of redirecting API calls to a logging code. The problem of ensuring that the logging code is executed in the right address space arises.

Under Windows NT/2000. There is no documented way of loading a DLL into an area shared by all processes, thus the only way to ensure that the logging code is accessible by the target process is to inject the spying DLL into its address space. One of the ways to accomplish this is by adding the name of the DLL to the following registry key:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\Windows\AppInit_DLLs
```

This key causes Windows to load the desired DLL into every address space in the system. Unfortunately, this technique can only be used to inject a DLL into processes that link with user32.dll, meaning that console applications, which do not usually link with this DLL, are not included.

File system monitoring techniques can be done through a number of techniques. There are a number of available tools that can perform this task. Filmon is an example of such tools. Filemon monitors and displays file system activity on a system in real-time. Filemon's timestamping feature will show every read, write or delete, happens, and its status column tells you the outcome. Its output window can be saved to a file for off-line viewing.
7. Analysis of The Nemesis

7.1 Monitoring

The main idea is that, the system tries to lure any virus that might be present in the system to infect a diverse suite of "decoy" programs. A decoy program's sole purpose in life is to become infected. To increase the chances of their success, decoys are designed to be as attractive as possible to those types of viruses that spread most successfully. And since it is a good strategy for a virus to follow is to infect programs that are touched by the operating system in some way. Such programs are most likely to be executed by the user, and thus serve as the most successful vehicle for further spread. The immune system entices a putative virus to infect the decoy programs by executing, reading, writing to, copying, or otherwise manipulating each of them. Such activity tends to attract the attention of many viruses that remain active in memory even after they have returned control to their host. To catch viruses that do not remain active in memory, the decoys are placed in places where the most commonly used
programs in the system are typically located, such as the root directory, the current directory, and other directories in the path. The next time the infected file is run, it is very likely to select one of the decoys as its victim.

Since viruses can be particular about the conditions under which they replicate, the decoys are run under several different environmental settings if necessary. The date may be changed a few times, as may the operating system version and other parameters. This helps to increase the chances of replication, and also helps expose differences in viral progeny that depend upon such parameters.

From time to time, each of the decoy programs is examined to see if it has been modified. If one or more have been modified, it is almost certain that an unknown virus is loose in the system, and each of the modified decoys contains a sample of that virus. These virus samples are stored in such a way that they will not be executed accidentally.

7.2 Analysis
From time to time, the snapshots of the environment is examined and compared to see whether modifications have occurred or not. The presence of any modification, informs the system of the presence of an unknown virus on the loose, and that each of the modified decoy files has captured a sample of that virus.

The infected decoys are then processed by another module of the analysis subsystem -- the signature extractor -- so as to develop a recognizer for the virus. The signature extractor must select a virus signature from among the byte sequences produced by the attachment derivation step. The signature must be well-chosen, such that it avoids both false negatives and false positives. In other words, the signature must be found in each instance of the virus, and it must be very unlikely to be found in uninfected programs.

First, consider the false negative problem. The samples captured by the decoys may not represent the full range of variable appearance of which the virus is capable. As a general rule, non-executable “data” portions of programs, which can include representations of numerical constants, character strings, work areas for computations, etc. are inherently more likely to vary from one instance of the virus to another than are “code” portions, which represent machine instructions. The origin of the variation may be internal to the virus (e.g. it could depend on a date). Alternatively, a virus hacker might deliberately change a few data bytes in an effort to elude virus scanners. To be conservative, “data” areas are excluded from consideration as possible signatures. Although the task of separating code from data is in principle somewhat ill-defined, there are a variety of methods, such as running the virus through a debugger or virtual interpreter, which perform reasonably well.

Briefly, the automatic signature extractor examines each sequence of \( S \) contiguous bytes (referred to as \``candidate signatures\'') in the set of invariant-code byte sequences that have presented to it, and for each it estimates the probability for that \( S \)-byte sequence to be found in the collection of normal, uninfected \``self\'' programs. Typically, \( S \) is chosen to be 16 or 36. The probability estimate is made by forming a list of all \( n \)-grams (sequences of \( n \) bytes; \( 1 \leq n \leq n_{\text{max}} \)) contained in the input data ( \( n_{\text{max}} \) is typically 5 or 8), and calculating the frequency of each such \( n \)-gram in the uninfected programs, using
a simple formula to combine the \( n \)-gram frequencies into a probability estimate for each candidate signature to be found in a set of programs similar in size and statistical character to the corpus, and selecting the signature with the lowest estimated false-positive probability. Thus, satisfying the two conflicting criteria of a signature which are capturing a broad variety of conceivable mutations for a particular virus, but with no low probability of false-positives.

Moreover, another module within the analysis subsystem attempts to extract from the decoys information about how the virus attaches to its host, so that infected hosts can be repaired (if possible).

7.3 Testing

It mainly performs two functions, testing the candidate signature and the removal tool generated.

The automatically-extracted signatures and repair information are then subjected to a variety of independent tests. The signatures are run against a half-gigabyte corpus of legitimate programs to make sure that they do not cause false positives. The generated removal tool is tested against the infected decoy files, to make sure that by using this tool the infected decoy can be returned to its original state.

7.4 Learning

Mainly the system must learn from each case it scans, we will achieve this by adding the learning subsystem to The Nemesis which read the logs from the Database and try to find relations. We aim at consulting the learning system in future analysis so as to predict the viral behavior and prevent fatal instructions from executing. This part has lots of difficulties associated with it like:

- Designing the appropriate input scheme for training the learning subsystem (parameters).
- Scarcity of training sets (viruses).
- Finding a trade-off between false positives and false negatives.
An example that we have found about a neural-net based learning subsystem that was designed by IBM to detect unknown boot-sector viruses, was that due to the constraints mentioned above the learning sub has caught approximately 75% of new boot sector viruses that have come out since the product was released. Most of the viruses that escaped detection did so not because they failed to contain the feature trigrams but because the code sections containing them were obscured in various ways.

This shows that the learning subsystem is hard to implement due to the uncertain and unpredictable nature of malicious codes in general.

8. Software and Hardware Requirements

8.1 Software Packages

- VMWare Workstation 3.0 for Windows.
- All the Antivirus packages that are available in our department so as to use them to learn how they discover new and "custom-built" viruses.
- Also any commercial emulators/sandboxes available in our department.

8.2 Hardware Requirements

We URGENTLY need 2 other computers because we deploy LIVE viruses to test their effect and latter on we will need to deploy the viruses against our system to see how our system classify it and this means we can NOT deploy it on our dedicated 491 computer.
9. 491 Time Plan

<table>
<thead>
<tr>
<th>ID</th>
<th>Task</th>
<th>Apr 2002</th>
<th>May 2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Organize a Library of Tools &amp; Malicious Code</td>
<td>3/17</td>
<td>3/24</td>
</tr>
<tr>
<td>2</td>
<td>Setup the Environment</td>
<td>3/31</td>
<td>4/7</td>
</tr>
<tr>
<td>3</td>
<td>Understand Analysis Techniques</td>
<td>4/14</td>
<td>4/21</td>
</tr>
<tr>
<td>4</td>
<td>Understand Monitoring Techniques</td>
<td>4/28</td>
<td>5/5</td>
</tr>
<tr>
<td>5</td>
<td>Classify Viral Behavior</td>
<td>5/12</td>
<td>5/19</td>
</tr>
<tr>
<td>6</td>
<td>Analyze Software Requirements (SRS)</td>
<td>5/29</td>
<td>5/29</td>
</tr>
<tr>
<td>7</td>
<td>Implement a Prototype</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10. Conclusion

We will design a tool for Virus and malicious code analysts, this tool will automate and accelerate the analysis and discovery process which is mainly manual nowadays.

Our aim is to reduce the time gap between the out-break of the virus and the availability of the cure by a considerable amount of time thus drastically reducing losses.

11. References

- http://portal.acm.org
- http://www.internals.com
- http://www.wd-mag.com
- http://msdn.microsoft.com
- http://www.security-focus.com
- http://www.bocklabs.wisc.edu/~janda/
- http://rr.sans.org/malicious/anti-virus.php
http://www.neowisdom.com/home.htm

http://ourworld.compuserve.com/homepages/kenbechtel/edu.htm

**Windows SDK questions**

http://www.mvps.org/vcfaq/sdk/

**Windows API**


**Performance Monitoring**


**Tool Help Library**


**Process Status Helper**


**Process Information**


**VMWare**

http://www.ch-open.ch/html/events/Event_3.5.2001/1

http://www.vmware.com

http://www.extremetech.com
**Top 10 viruses**

http://www.eweek.com/article/0,3658,s=712&a=20526,00.asp

*Virus Information Center*

http://www3.ca.com/virus/

**Subclassing**

http://www.internals.com/articles/apispy/apispy.htm


http://embedded.centurysoftware.com/docs/nx/

http://www.ch-open.ch/html/events/Event_3.5.2001/