European Commission
Directorate-General Home Affairs

Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme

HOME/2009/CIPS/AG/C2-050
i-Code: Real-time Malicious Code Identification

Deliverable D3: Integration and Pilot Operation

Workpackage:	WP3: Integration and Pilot Operation
Contractual delivery date:	June 2012
Actual delivery date:	July 2012
Deliverable Dissemination Level:	Public
Editor	Alessandro Frossi (POLIMI)
Contributors	All Partners
Internal Reviewers:	FORTH

Executive Summary: This deliverable is a technical report on the integration of the different detection and analysis subsystems, and the test operations of the unified i-Code system.

With the support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme. European Commission - Directorate-General Home Affairs[†].

[†]This project has been funded with the support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of European Commission - Directorate-General Home Affairs. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Contents

List of Figures

CHAPTER 1

Introduction

The i-Code project aims to detect and analyze malicious code and Internet attacks in real time. Its scope includes the detection of attacks in the network and on the host, the analysis of the malicious code, and post-attack forensics. Thus, the project takes on challenges from different aspects of operational security and proposes to address them with a number of novel detection and analysis tools. Due to the variety of the tasks they face, these tools are very diverse, but they can be interconnected to provide an enriched understanding of security incidents, by means of the i-Code console.

Deliverable *D1: System Design* described the design of each i-Code detection and analysis component. Furthermore, the potential synergies between these components were explored. In deliverable *D2: System Implementation*, then, we reported the actual implementation of the tools and the console carried out the project partners and the first steps taken towards the final system integration.

In this document we continue from there, going through all the single tools' integration processes to finally build a comprehensive system having the console as its only interface to the external world. We also describe the testing phase of the i-Code system, which represents its pilot operation.

Outline In this document we detail the integration process of all the components into a single system. In Chapter 2, we describe the general architecture of the system, going through a short overview of the components involved in the process and how they fit into the big picture. Chapter 3 dwells into the details of how each tool is integrated with the each other, showing how the information about security threats is collected, saved into events and dispatched to the manager, before being shown to the final user.

Chapter 4, finally, shows the results of this integration process with the detailed description of the architecture and procedures used in the pilot operation.

CHAPTER 2

Architectural Overview

2.1 General design

The i-Code project aims at detecting and analyzing malicious code and Internet attacks in real time. This is not, however, the only scope of the project: it also aims at creating an easy and centralized way to show the results of those tasks. Therefore, it's not all about the tools that represent the "core" of the system; they are, obviously, a big part of the system itself, but great importance has to be given to the integration of these components into a bigger picture: a comprehensive system capable of detecting threats in real-time and present them to the user on a single collector application. The advantage is clear: the user doesn't have to directly use the single tools and interpret their results (often presented in custom formats) but rather use the console application to have all the security alerts collected and shown in an easy-to-read format.

The general design is shown in Figure 2.1. All the sensors deployed within the target network (i.e., AccessMiner, Argos and Nemu) raise alerts whenever they detect a security threat and dispatch an event towards the console, giving detailed information about the threat itself. When the console receives an alert, it saves it to make it persistent and sends the attached shellcode to a remote installation of Anubis, which further analyzes it giving back a report identifier. The user is then presented via web interface the events (in tabular form) and, following a link to Anubis, the previously created report.

In Section 2.2 a more detailed view of the single components is presented, while Chapter 3 we will go in detail into the integration process.

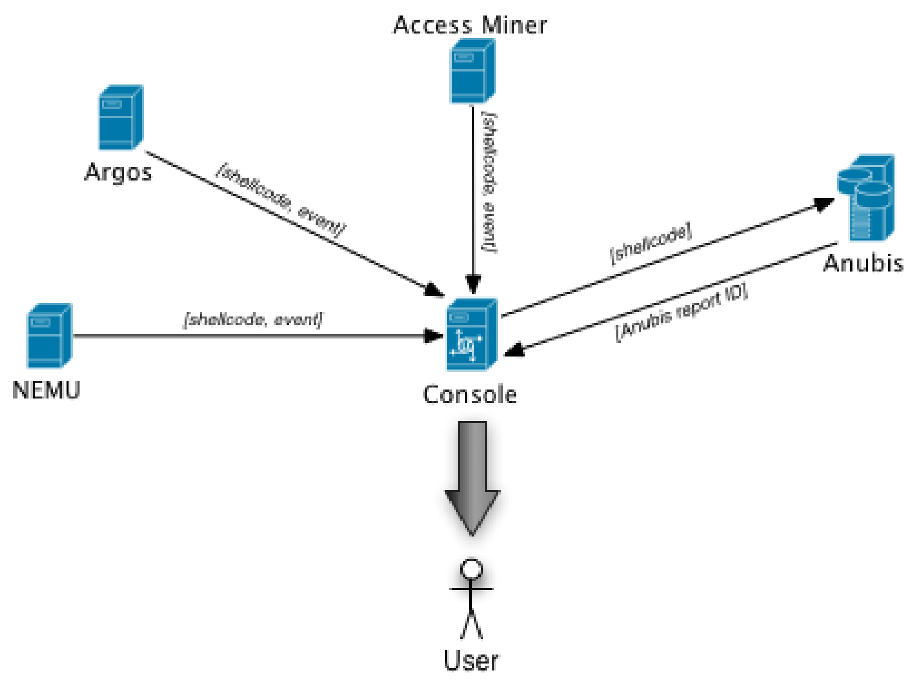


Figure 2.1: Design overview of the i-Code system.

2.2 Components

2.2.1 Console

This component, though not being “active” within the system (it doesn’t raise events or detect threats), has a fundamental role in i-Code project: it has to gather all the relevant events generated by the peripheral sensors and show them to the user in an understandable and usable way. It is, therefore, the main component in the integration process, since it is the end-point of any communication from and to the system: all the tools are supposed to send their alerts to the console which, in turn, is responsible of sending the shellcodes to Anubis, receiving the corresponding report id and showing the alerts to the final user (see Figure 2.1).

Figure 2.2 shows the overall architecture of the i-Code console: the front-end is a simple web application whose task is to simply show the user the result of the event analysis process performed by the back-end component. Here, in fact, the alerts are received and stored in a database; when the user accesses the web application, then, they are retrieved and sent to the front-end along with some useful statistics and additional information.

The console was built to be:

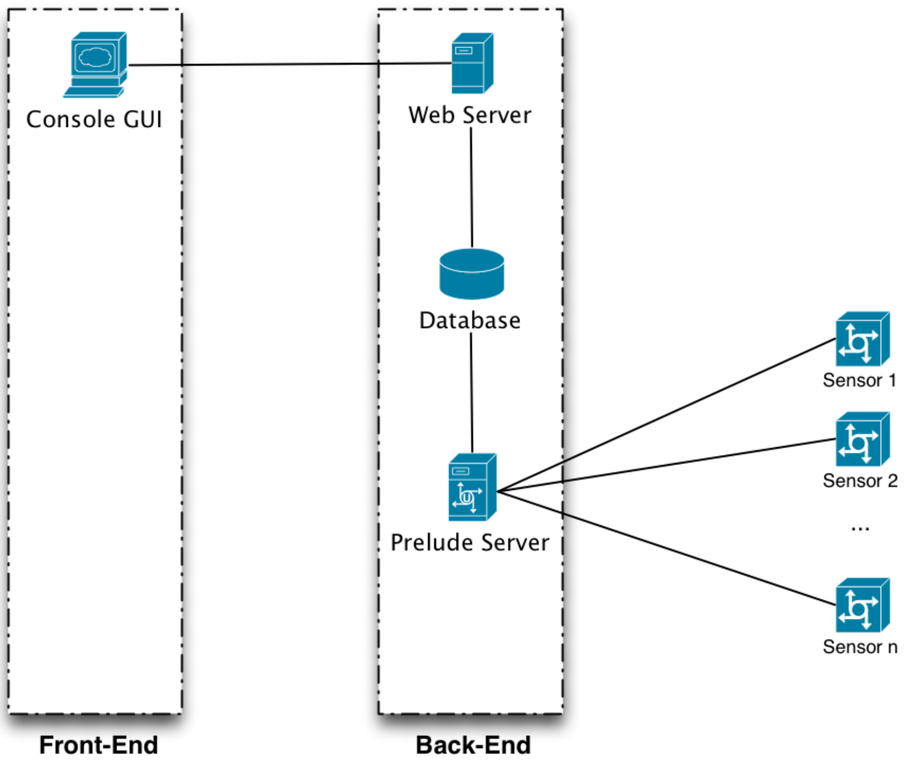


Figure 2.2: Overall architecture of i-Code console.

Portable No constraints on operating system, platform, technology or network topology are imposed to the user: the console was designed to be a web application to make it usable on any browser, despite of the underlying software and without proprietary technologies. Also, it's not necessary for it to be accessed from the same network of the sensors, or even be in the same network, as long as the components can communicate with each other.

Lightweight No heavy JavaScript code is used, in order to make the console more responsive and faster even on less powerful computers or devices.

Usable The console shows the events in tabular form and allows the user to sort or narrow them using custom filters that can be composed using the built-in filtering system and modified as needed. Also, some useful statistics are computed and shown in graphs, so that the user can visually assess the situation of the system.

Complete All the events along with their details are presented to the user and additional information can be easily be retrieved by following the external link to Anubis associated to every alert.

2.2.2 AccessMiner

AccessMiner is a host-based behavioral malware detector designed to capture the activities of benign programs and to detect certain types of malware (those tampering with binaries or settings of other applications or the OS). Accessminer takes a system-centric angle and models the way in which a broad set of benign applications interact with OS resources. More precisely, our approach builds an access activity model that captures permissible read and write operations on files and registry entries.

Our experiments show that the access activity model is successful in identifying a large fraction of malware samples with a very low false positive rate. Of course, access activity models cannot detect all possible types of malware. They can only detect cases in which malicious code attempts to tamper with the binaries or the settings of other applications or the core OS itself. As our experiments show, this is true for a large fraction of malware - after all, malware often attempts to interfere with or modify the execution of legitimate programs or the OS, or, at the very least, establish a foothold on the system.

The Accessminer detection component is implemented as a lightweight hypervisor to prevent tampering from compromised operating systems.

2.2.3 Argos

Argos is a full and secure system emulator designed for use in honeypots. Argos extends Qemu to enable it to detect remote attempts to compromise the emulated guest operating system. Using dynamic taint analysis it tracks network data throughout execution and detects any attempts to use them in an illegal way. When an attack is detected, a footprint of the attack is logged that includes a possible payload injected by the attacker.

For the i-Code project, Argos is extended and deployed as a client-side honeypot that detects if websites, visited by clients inside the network, attempt to compromise visitors.

2.2.4 Nemu

Nemu is a network-level attack detector based on code emulation. The principle behind its detection approach is that the machine code interpretation of arbitrary data results to random code which, when it is attempted to run on an actual CPU, usually crashes soon, e.g., due to the execution of an illegal instruction. In contrast, if some input contains actual shellcode, then this code will run normally, exhibiting a potentially detectable behavior.

Nemu is built around a CPU emulator that executes valid instruction sequences found in the inspected input. Each input is mapped to an arbitrary location in the virtual address space of a supposed process, and a new execution begins from each and every byte of the input, as the position of the first instruction of the shellcode is unknown and can be easily obfuscated. The detection engine is based on multiple heuristics that match runtime patterns inherent in different types of shellcode. During execution, the system checks several conditions that should all be satisfied in order for a heuristic to match some shellcode.

All heuristics are evaluated in parallel and are orthogonal to each other, which means that more than one heuristic can match during the execution of some shellcode, giving increased detection confidence. For example, some heuristics match the decryption process of polymorphic shellcode, while others match operations found in plain shellcode. Polymorphic shellcode usually carries an encrypted version of a plain shellcode, so the execution of a polymorphic shellcode usually triggers both self-decrypting and plain shellcode heuristics.

2.2.5 Anubis

Anubis is a dynamic malware analysis system that is based on an instrumented Qemu emulator. It is offered as an open service through a public website, where users can submit binaries for analysis and receive reports that describe the system- and network-level behavior of the analyzed binaries in a human-readable way. For the i-Code project, Anubis was extended to support the analysis and classification of shellcode.

CHAPTER 3

Integration

3.1 Console

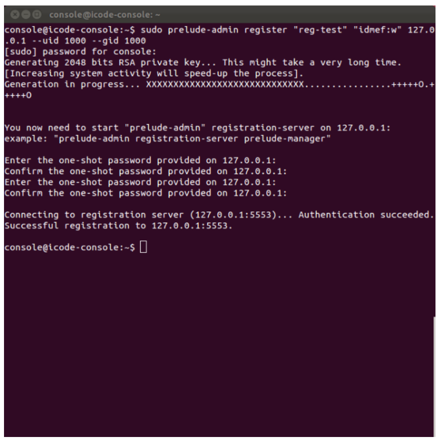
The console is the main integration point of the i-Code system since it's the only one that will be shown to the user and will take care of all the communication between the user and the peripheral components. Its architecture was accurately described in deliverable *D2: System Implementation* so, in this Section, only some components will be further detailed to give the reader an insight on how the single part of the tool cooperate to make the integration more smooth.

3.1.1 Prelude Back-end

The Prelude server is the real focus point of the whole integration; this component is responsible of:

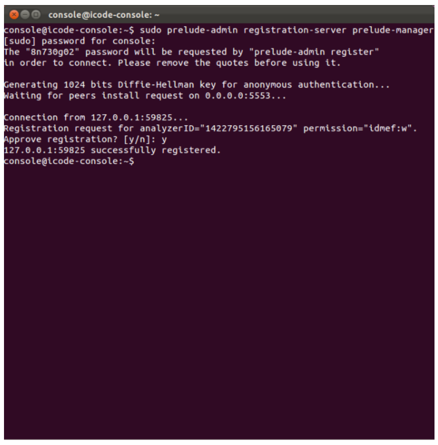
- receiving the alerts from the tools
- storing all the information in a local database
- attaching geographical information to each event
- sending the shellcodes (attached to the events) to Anubis

Since this server handles all the relevant information passing through the system, it is crucial that it is correctly built and configured keeping security as a first requisite. The database, in fact, is local and not accessible by processes not running on the console virtual machine; the password, also, is known to the Prelude instance and to the working plugins only. To increase security between the sensors and the server, the sensor registration



```
console@icode-console:~$ sudo prelude-admin register "reg-test" "idmef:w" 127.0
.0.1 --uid 1000 --gid 1000
[sudo] password for console:
Generating 2048 bits RSA private key... This might take a very long time.
[Increasing system activity will speed-up the process].
Generation in progress... XXXXXXXXXXXXXXXXXXXXXXXXXXXX...............+++++O.+
++++O


You now need to start "prelude-admin" registration-server on 127.0.0.1:
example: "prelude-admin registration-server prelude-manager"

Enter the one-shot password provided on 127.0.0.1:
Confirm the one-shot password provided on 127.0.0.1:
Enter the one-shot password provided on 127.0.0.1:
Confirm the one-shot password provided on 127.0.0.1:

Connecting to registration server (127.0.0.1:5553)... Authentication succeeded.
Successful registration to 127.0.0.1:5553.

console@icode-console:~$
```

(a) Sensor side



```
console@icode-console:~$ sudo prelude-admin registration-server prelude-manager
[sudo] password for console:
The "8n730g02" password will be requested by "prelude-admin register"
in order to connect. Please remove the quotes before using it.

Generating 1024 bits Diffie-Hellman key for anonymous authentication...
Waiting for peers install request on 0.0.0.0:5553...

Connection from 127.0.0.1:59825...
Registration request for analyzerID="1422795156165079" permission="idmef:w".
Approve registration? [y/n]: y
127.0.0.1:59825 successfully registered.
console@icode-console:~$
```

(b) Server side

Figure 3.1: Sensor registration procedure

process (shown in Figure 3.1) requires the interaction of both parties: the server listens for new registration requests and creates a *one-time password* to be used as confirmation; the client, instead, generates a 2048 bit RSA key to encrypt communications and requires the user to provide that same password given by the server; this ensures that a human operator is taking care of registering a known sensor.

When a sensor raises an alert (see tools' related Sections to know how an event is created and sent), it is sent to the *Prelude Manager process* running on the console machine. This process saves the alert in the database in a fully normalized form (which also results difficult to be accessed by a human operator) and, at the same time, it forwards it to the registered and enabled plugins. While there are many built-in plugins bundled with Prelude, none of them was used since they provide functionalities not relevant to the i-Code system; two custom scripts, instead, were created and registered (see Section 3.1.2 for details). Each script receives as input every alert received by the manager and performs on them the small task it is designed for: this can be the modification of the alert (or any field) or the creation of a new correlated alert, which will be sent to the manager and treated as any other new event.

When the front-end needs to show the events to the user, it uses a custom script to retrieve them directly from the database, without creating unnecessary load on the manager.

3.1.2 Correlation Rules

Correlation rules (or *plugins*) are scripts that are registered on the manager to be eligible to have all the events forwarded and made accessible to them.

Their task is to perform some minor operations on such events (e.g., checking if the source address is in some specified networks) and, if necessary, raise additional events to the manager. This is particularly useful when dealing with contexts: if, for example, 10 login failed events are detected in a 3 minutes period, a new *Possible Brute Force* event can be raised by a rule.

As said in Section 3.1.1, no built-in rules were used since they implemented functionalities that were not needed by the i-Code system. Two additional custom rules, instead, were created and registered. These plugins, however, deviate from the standard plugin design: they do not create additional events but rather modify incoming events; this is not usually allowed, anyway, because in a standard Prelude deployment all the events have to be securely stored in the database and cannot be modified.

In our case, however, the goal of the two plugins is to *augment* the information contained in the event and it is, therefore, necessary to modify it after being stored in the database. Since there's no API provided by Prelude for this (while there is for context manipulation and event creation), we had to do it manually: when a tool creates an alert, it stores a *beacon* in each additional field: this beacon is simply a random string that will be substituted by the new information once the alert is delivered to the correlation rule. This trick was made necessary because the information stored in the database is fully normalized and fragmented over many tables: retrieving the entire field set for an event is not a simple matter of joining some tables but requires a much greater effort; the use of random beacons (long enough to avoid unfortunate collisions) allows to pin-point the interesting data in the database and modify it in a completely transparent way.

Source IP Geolocation Plugin

The *Source IP Geolocation Plugin* uses the event source IP to geographically locate the origin of the security threat.

The address contained in the *alert.source(0).node.address(0).address* field of the IDMEF event (see Section 3.6) translated into a country code: the country from where the packet was sent. Once this information is available, the plugin looks for the corresponding beacon in the *additional_data* table and substitutes it with the correct information.

In this way, the information is *added* to the alert, instead of creating a brand new one.

Anubis Submitter Plugin

The *Anubis Submitter Plugin* is the real integration point between the i-Code system and the remote Anubis application. It works just like the *Source IP Geolocation Plugin*: it retrieves the relevant information from the alert, uses it to collect new additional data and stores these new information in the alert itself by modifying the beacon in the database.

The relevant piece of data, however, is not the source address, but rather the shellcode associated to the detected security threat; as described in Section 3.6, in fact, each tools attaches the attack shellcode (encoded in base64) to the IDMEF alert in the *alert.additional_data(2).data* field. This is then sent to Anubis via the Python script provided on the website[1] and the resulting report ID is written on the database in place of the beacon saved in the *alert.additional_data(1).data* field.

The result is that this plugin allows to further analyze the shellcode and retrieve the corresponding report directly from the Anubis website.

3.1.3 Console Virtual Machine

All the softwares that are part of the console component in the i-Code system are hosted on the same virtual machine. This choice was made for two main reasons:

- keeping all the components on the same machine does not require them to be accessible by processes running on other machines. Obviously, both the Prelude server and the web server have to be accessible from outside, but at least there is no need to expose the database and open a possible security hole. It would be, however, possible to host the database or any other component (e.g. the Prelude Correlator tool) on other machines to balance the load on the console

- for testing purposes the system was not stressed enough to make a strong load-balancing a necessity: the frequency of incoming alerts was low enough to keep the Prelude server and the database far under their capacity limits.

There were no particular hardware requisites so a single core machine with 1 Gb RAM was instantiated. The operating system is the one that offers both the maximum compatibility with the Prelude software (with pre-compiled packages offered by standard repositories) and easiness of use: Ubuntu Desktop 11.10 64-bit. The only packages needed to make the console work were the following:

Prelude Libraries The libraries necessary to make the Prelude architecture work: they handle all the communications between the manager and the sensors, including cryptography and heartbeats.

Prelude Manager This is the software component that receives all the events from the peripheral sensors. It relies on the above libraries

[1] `http://shellcode.iseclab.org/Resources/submit_to_anubis.py`

and a storage connector, which, in our case, is a MySQL database connector; the latter is in charge of writing and reading data from the non-volatile storage.

MySQL Database The DBMS used to save all the alerts: it does not have to be exposed to external connections because it is always accessed by local processes: the manager, the web application back-end and the correlation rules.

Prelude Correlator This software registers and runs the Geolocation and Anubis Submitter plugins: it is registered as a sensors with "forward" permissions on the manager, which means that every alert received by the Prelude system is sent immediately to the rule chain in the correlator. This component has also direct access to the database to modify the events adding useful information.

Python + Flask The back-end logic and the front-end web application are both written using the Flask framework for Python, which is light enough for the console application.

3.2 AccessMiner

The model enforcement component of AccessMiner was redesigned to increase its security by moving it at the hypervisor level. In addition, AccessMiner was extended with a new module to communicate to the prelude system. These modifications are explained in the rest of the section.

3.2.1 Model Enforcement

Our new enforcement model exploits the hardware virtualization support available in commodity x86 CPU. By using the VMM extensions we designed a tamper-resistant detector that is able to control OS operations and verify the policies derived from the AccessMiner system.

3.2.2 Threat Model

The threat model under which our enforcement model runs considers a very powerful attacker. The attacker can operate with kernel-level privileges. On the other side the attacker cannot perform hardware-based attacks (e.g., a DMA-based attack) and he cannot tamper the hypervisor operations. We assume that our hypervisor starts during the boot process of the machine and it is the privilegest hypervisor on the system.

3.2.3 Technology Overview

Before describing how our detector works we give a brief introduction about Intel virtualization.

The main characteristic of Intel VT-x technology is supporting new VMX mode of operation. When in VMX mode, the processor can be either VMX root or VMX non-root operation. The behavior of the processor in VMX root operation is similar to the one that operates in normal mode expect of using a new set of instructions called VMX instructions. The behavior of the processor in non-root operation is limited in terms of controlling the access to the resources even when the CPU is running in ring 0 (highest privilege).

The VMM can monitor operations on critical resources without modifying the code of the guest OS. Moreover because VMX non-root mode operation includes all four IA-32 privileges levels (rings) guest software can run in the original rings in where designed to be run.

A processor which has been turned on in a normal mode can be made to enter VMX root operation by executing *vmxon* operation. The virtual machine monitor VMM running in root operation sets up the environment and initiates the virtual machine by executing *vmlaunch* instruction.

When a VMM is running, the CPU switches back and forth between non-root and root mode: the execution of the virtual machine might be interrupted by an exit (*VMexit*) to root mode and subsequently resumed by an enter to non-root mode (*VMentry*).

The technology provides a data structure called the VMCS (virtual machine control structure) that embeds all the information needed to capture the state of the virtual machine or resume the virtual machine. The various control fields determine the conditions under which control leaves the virtual machine (*VMexit*) and returns to the VMM, and define the actions that need to be performed during VM entry and VM exit.

Various events cause the processor to leave control to VMM in root operation. The processor can also exit from the virtual machine explicitly by executing *vmcall* instruction. In particular, for our detector model we are interested to exploit the *vmcall* instruction for intercepting the system call event. More details about the technique are explained in the next sections.

3.2.4 Hypervisor Architecture

The new Detector model is composed by three components: System call Interceptor, Process Revelear and Policies Checker. The outputs of all the components are combined together to check the policies derived by Access-Miner system.

3.2.5 System Call Tracer

The core of the system is represented by System Call Tracer. In order to intercept the system call we monitor the sysenter/systexit instructions. Whenever a system call is issued by a process a sysenter instruction is invoked. The sysenter instruction refers to the MSR register that contains the SYSENTER_EIP, the instruction pointer that will point to a small stub that handles the invocation of the appropriate system call handler. When the system call handler is terminated the control flow execution returns into the stub, at this point sysexit instruction is issued and the control-flow returns to the user space application.

In order to bring the execution flow inside the hypervisor we need to switch from VMX non-root mode to VMX root mode. To this end we substituted the SYSENTER_EIP value into the MSR register in order to point to the *vmcall* instruction. By using this hooking technique the hypervisor is able to intercept every sysenter and sysexit performed on the system. When a sysenter is invoked the hooking code intercepts the control-flow and parses the parameters according to the sort of the operation. Before sending the information to the Policies Checker, the system needs to check the successful execution of the operation and the return values (sysexit interception). In case the operation fails, the component does not produce any output. In the other case, the System Call Tracer invokes the Policies Checker component and it provides all the system call information: system call type, parameters, and return values.

3.2.6 Process Revealer

Another important information that we need to provide to the Policies Checker is the name of the process. To this end we utilize a cache system that is able to associated the CR3 value to the process's name. In particular every time a process is created or destroyed the system updates the process cache with the new information (CR3, process). From technical point of view we utilize the same interception technique deployed for System Call Tracer. In particular the system intercepts the sysexit for create/terminate process operation and queries the EPROCESS structure in order to obtain the association between the process's name and the CR3 value. The Process Revealer component will update the cache information accordingly. It is important to note the cache memory system can speed up our detection mechanism since the Policies Checker needs the information CR3, process name for applying the policies rules. Without cache system, every time a system call is invoked, the Policies Checker should query the EPROCESS structure and retrieves this information. In case of using the cache system the information can be obtained in constant time (e.g, hash table).

3.2.7 Policies Checker

The main goal of this component is to check the policies derived from the AccessMiner system and creates an alert in case some of them are violated. In particular, in order to check the policies we deployed an hash table memory structure where the resources name (files pathname, registries pathname) is the key of the hash table and the element is represented by the name of the processes with their own permission on the resources. We recognize two main phases for the Policies Checker: Initialization and Detection Phase. The initialization phase occurs after the loading of the hypervisor kernel module. The main task of such a phase is to initialize the memory structures that will be used for the detection phase. The phase works as follows. First the Hypervisor Kernel Module reads the signatures from a configuration file. Then, whenever a signature is loaded the full-pathname of the resources is extracted and utilize as a key of the hash table in order to store the following information: the list of the processes that can get access to the resources and their own access permissions on that resources.

After the initialization of the hash memory structure a memory handles structure is created. This memory structure is used in order to track down the different operations on the resources (e.g., files and registries) by looking up the full-pathname and the resource handle. In particular every time a resource is opened we track down the handle associated to the resource full-pathname and we store it in the memory structure. After wards when some operations occurs on the resource we link the resource full-pathname to the handle and we pass this information to the detection module. Every time an handle is closed we remove it from the list.

During the detection phase, The System Call Tracer invokes the Policies Checker and sends the system call information. At this point the Policies Checker by using the full pathname as the key of the hash table it retrieves the list of the processes and the resource access permissions. It also queries the process_cache in order to get the process's name that performed the system call. When obtained all the information it scans the list of the processes for searching the actual process name. If the process name is found, the Policies Checker checks the permission associated to it and if there's a mismatch with the permission it reports a warning. If the process is not present in the process list the Policies Checker reports a warning to the system as well. In the other cases it does nothing.

3.2.8 Prelude Integration

Whenever a warning is produced the detector system sends the output of the warning to the serial port where a receiver (python server) is in charge to decode the message according to the Prelude standard format and send it back to the console. The Prelude standard format contains 8 fields with different information related to the type of warning. In the following we describe the meaning of each field for AccessMiner System:

- *alert.source(0).node.address(0).address*: the IP of the machine where the attack is performed. For AccessMiner the source/destination nodes are the same.
- *alert.target(0).node.address(0).address*: same as the source IP.
- *alert.analyzer.name*: “AccessMiner” string.
- *alert.additional_data(3).data*: process Name that gets access to the resource.
- *alert.additional_data(4).data*: name of the resources involved in the operation.
- *alert.additional_data(5).data*: system call that operates on the resource.
- *alert.additional_data(6).data*: permissions required by the operation.
- *alert.additional_data(7).data*: permissions for the resource according to AccessMiner’s policies.

3.3 Argos

The integration of Argos with the i-Code system required three extensions. The first extension, the HTTP proxy module, enables Argos to verify if clients have been compromised by visiting a malicious website. The second extension, the payload collector, allows Argos to extract an injected payload from the memory footprint logged after detection of an attack. The last extension, the Prelude proxy module, enables Argos to transparently communicate its findings to the Prelude Server. All extensions will be briefly discussed in the following sections.

3.3.1 Client Integration

The role of Argos in the i-Code system is to detect, a posteriori, if clients inside the network have been compromised by visiting a web page. Argos requires integration with clients inside the network to collect the URLs of the visited websites.

The HTTP proxy acts as an intermediary for request from clients inside the network and stores the requested URLs inside a queue. Argos monitors this queue and schedules a client-side honeypot instance for each URL in the queue.

3.3.2 Payload Analysis Integration

The i-Code system has the capability of submitting payloads to Anubis for in-depth threat analysis. This capability of the i-Code system requires a payload to be a base64 encoded binary blob.

The memory footprint logged by Argos contains the required information, but in a different format. The payload extractor module is able to find a payload inside this footprint, by using information collected during the attack, and extract it. Finally, after a successful extraction the payload module returns a base64 encoded payload.

3.3.3 Prelude Integration

For the integration with Prelude, Argos is extended with a Prelude proxy. The proxy model allows for transparent scaling of capacity if required by an increase or decrease of client requests. The Prelude proxy acts a client to the Prelude Manager and is responsible for transparently forwarding events from Argos using the IDMEF format, as defined in deliverable *D2: System Implementation.*

3.3.4 Argos Virtual Machine

The virtual machine is configured to run the aforementioned HTTP proxy and Prelude proxy, as well as a Argos driver and a single Argos instance.

The Argos driver is responsible for orchestrating the website verification process. For each URL collected by the HTTP proxy, the Argos driver schedules an Argos instance, configured as a client-side honeypot, and directs it to the website that requires verification. In the test phase the Argos driver schedules only one Argos instance, but the pool of Argos instances can be scaled linearly by distributing multiple instances over multiple servers.

In case of an attack detection, the Argos driver obtains the collected information from the Argos instance and invokes the payload collector to find an injected payload. When the payload collector finishes, the Argos driver submits everything to the Prelude proxy server, which sends an event to the Prelude Manager.

3.4 Nemu

3.4.1 Prelude Integration

As part of the i-Code system, Nemu was extended with a new module that acts as a client for the Prelude manager and submits to it all detected events. For each attack, Nemu stores in a local SQLite database all the relevant information, such as the date and time of the incident, the source and destination IP address and port, the type of the identified shellcode and its execution trace, MD5 sums of the shellcode and the stream chunk that contained it, and other details. It also extracts the raw shellcode from the input stream and stores it into a separate file for further analysis.

For performance and modularity, Nemu's Prelude client module runs as a separate process, and does not require any form of communication with the actual Nemu detector. This is achieved by relying only on Nemu's local database for retrieving new incident alerts. The client module monitors for changes in Nemu's database using Pyinotify, a Python module for monitoring filesystem events. Whenever a new record is written in the main "alerts" table in the database, the module is triggered and pulls the necessary information for submission to the Prelude manager. It then constructs an IDMEF message according to the format specified in deliverable *D2: System Implementation*, and transmits it to the Prelude manager. All the relevant info is retrieved from the database, except the extracted shellcode, which is read directly from its own file. The shellcode is included in the field `alert.additional_data(2).data` of the IDMEF message in base64 encoding for subsequent analysis by Anubis, as described in Section 3.6.

3.4.2 Nemu Virtual Machine

Nemu is running on a separate virtual machine, and monitors all traffic reaching the main network interface, which is set to promiscuous mode. Nemu has been configured to scan both directions of each connection, so as to detect both client-side and server-side attacks. All attack incidents are logged in the local database, and are then transmitted to the Prelude manager by Nemu's Prelude client module, which also runs on the same virtual machine.

3.5 Anubis

The Anubis malware analysis sandbox for analyzing shellcodes is deployed at TUV and is accessed through a public website[2]. Additionally, a Python script facilitates the automatic submission of shellcodes. For each successful submission of a shellcode, Anubis returns a unique report ID, with which the analysis report can be retrieved, once the analysis is finished.

3.6 Events

The events are the only mean for the sensors to communicate with the manager and pass it the information about security threats they detected; they comply to the *Intrusion Detection Message Exchange Format*[3] (IDMEF)

[2] http://shellcode.iseclab.org

[3] http://www.ietf.org/rfc/rfc4765.txt

which defines both data formats and exchange procedures for intrusion detection and response systems. The data format offers a skeleton structure for events but also allows some degrees of freedom in customizing it; in the i-Code system we took this liberty to customize alerts to our need, which resulted in the structure described in deliverable *D2: System Implementation*.

The modification that most impacts on the integration process is the addition of three custom fields besides the standard ones:

alert.additional_data(0).data This field will contain the country the source IP belongs to. This information, however, is not immediately available but will be filled after the alert will be sent to the *Source IP Geolocation Plugin* (see Section 3.1.2); to ensure that the data will be written in the correct event in the database, the sensors save a random string in this field, leaving it behind as a *beacon* to allow the correlation rule to point to the correct record when the source country is available.

alert.additional_data(1).data This field is also filled with a random beacon but the information it is reserved to is different: the *Anubis Submitter Plugin*, in fact, will use this slot to save the report ID obtained after sending the shellcode to the remote installation of Anubis.

alert.additional_data(2).data This field contains the base 64 encoded shellcode that was part of the attack and that was detected and saved by the tools. This is the data that will be sent to Anubis for further analysis.

CHAPTER 4

Testing phase

This chapter is entirely dedicated to i-Code system testing phase. Section 4.1 describes the architecture build to verify if the tools correctly detect and signal security threats and if the console is able to gather such events and show them to the user in a fast and easy way.

Section 4.2, instead, shows how the testing phase was conducted in every aspect. Each step, in fact, is the result of a careful planning aimed at reducing the possible external factors that could compromise the experiment and, at the same time, keeping the attack as close to reality as possible: this means that the vulnerable software, the exploit and the deployed shellcode are not custom and were not coded purposely for this test.

4.1 Testbed Architecture

The testbed for the i-Code system is a purely virtual environment made of the machines described in Chapter 3. All the tools, exception made for Anubis, are hosted on dedicated machines and connected to each other via a dedicated LAN network: in fact, while Argos, AccessMiner and Nemu do not rely on external data and computational power, Anubis takes advantage of remotely deployed workers performing the relevant shellcode and malware analysis and from previously analyzed samples. As a consequence we decided to keep Anubis the way it was meant to be since the beginning: a remote web service.

This type of architecture was chosen because it has some advantages:

Manageability Keeping the virtual machines in the same environment made it faster to deploy them and easier to manage them; the presence of a single entry point for all the configuration needs was also

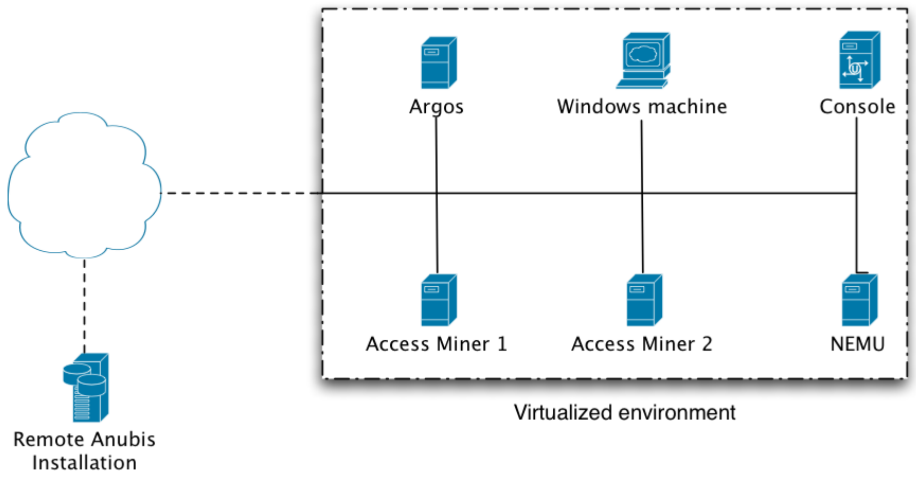


Figure 4.1: i-Code testbed architecture.

beneficial, since the partners don't have to deal with access and OS permission issues and could instead focus on integrating the tools.

Network problems resilience During the integration of the entire system into i-Code the main focus had to be maintained on the development and debugging of the tools: network problems not directly related to the development phase would have been a source of delays and would have need additional efforts to be solved. This architecture partly solves the issue by having a single virtual network to be managed and troubleshooted, thus relieving the partners from this burden.

Locality Some of the tools need to work in promiscuous mode and be therefore able to read the traffic originating from and going to other machines in the system. On a distributed testing environment this constraint would have posed some issues on the placement of the machines. This, however, wouldn't be a problem in a *real* deployment since the sensors would likely be in the same network of the monitored machines and not remotely located.

The overall architecture is described in Figure 4.1: there's a total of six virtual machines hosted on a single physical machine, including a VM created for testing purposes only (described in Section 4.1.1).

The host is has the following configuration:

Model: HP Z210 Workstation

CPU: Intel® Xeon(R) CPU E31245 @ 3.30Ghz x 4

RAM: 16 Gb

Disk: 1 Tb

OS: Ubuntu Desktop 11.10 64-bit

Environment: VMware® Workstation 8.0.2

Because of some architectural constraints, *VMWare Workstation 8* was used to host the virtual machines: AccessMiner, in fact, relies on nested virtualization to run and therefore has to be able to access particular CPU technologies even in a virtual environment. For this same reason we also had to be sure that the host CPU provided two key features: *Extended Page Tables* (EPT) and *Intel Virtualization Technology* (VT-x)

Virtual LAN. As said, tools like Nemu had to be able to enter promiscuous mode to read traffic originating from and going to other virtual machines. Also, to avoid network noise from external machines, the network had to be restricted to those guests only. For this reason a virtual LAN was created and used for the entire environment and VMWare was set to allow VMs to access promiscuous mode and get traffic not addressed to them.

The only other architectural constraint to fulfill was letting the system have access to the Internet to send shellcodes to Anubis and get back their report id. In addition, for testing purposes, the vulnerable virtual machine (Exposure) needed Internet access to allow droppers or particular shellcodes to download data from the web to the machine and trigger some specific rules in the monitoring tools. Therefore both the Console and Exposure machines were allowed to be *NAT*ted through the host network connection.

4.1.1 Exposure Virtual Machine

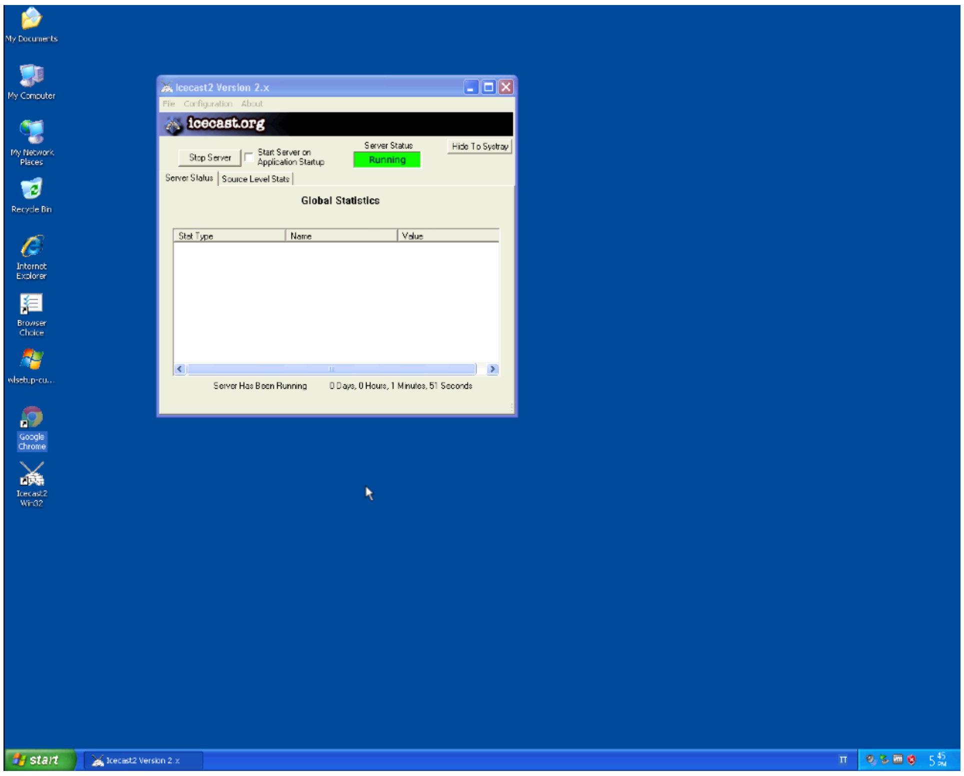
In order to fully test the i-Code system including the peripheral tools, the best choice would be to deploy the system and wait for some random attackers to drop a shellcode onto the vulnerable decoy machine. This, however, would be a time-consuming task, so the next best choice was taken: simulate a *real* attack on a target machine in the virtual environment.

For this purpose a dedicated VM was created with *Windows XP Service Pack 3* as operating system. Instead of exploiting an OS vulnerability, however, we chose a vulnerable software as the target of our testing phase: *IceCast2 2.0*[1] (see Figure 4.2), a free software for streaming multimedia was installed on Exposure machine. No other software was installed.

The vulnerability that was exploited in IceCast server was a buffer overflow in the request header, described in *CVE-2004-1561*[2]. Since there's an

[1] http://www.icecast.org/

[2] http://www.cvedetails.com/cve/CVE-2004-1561/



Server Status: Running

Server Has Been Running 0 Days, 0 Hours, 1 Minutes, 51 Seconds

Figure 4.2: Exposure virtual machine with IceCast2 server running.

existing *Metasploit*[3] module to exploit this vulnerability and drop a shell-code into the vulnerable machine, this vulnerability was the best candidate to test the i-Code system.

In order to test AccessMiner functionalities, its client was installed on this machine: this is the component responsible of detecting policy violations and notify them. Also, to avoid noise on the console reducing false positives, a very minimum set of policies was configured; in particular a folder called *vulnfolder* was created in the root level of the hard drive and AccessMiner was configured to grant read permissions and revoke write permissions to *all* processes for that directory.

Finally, for Argos to work properly, the Argos virtual machine was set as Internet proxy to allow it to intercept all the traffic going from and to the Exposure VM and identify possible threats.

4.2 Event Triggering

The real testing phase takes place in three different steps:

Shellcode choice The shellcode to be used for the test attack must be chosen carefully since it has to trigger all the tools involved in the process, namely Argos, Nemu and AccessMiner. While the first two tools should not have problems in detecting any kind of shellcode going through the network, allowing us to use Metasploit built-in shellcode payloads, AccessMiner raises alerts on predefined system policy violations. For this reason, we couldn't use any shellcode but we had to find one capable of violating such policies.

Vulnerability exploitation Once a suitable shellcode is chosen, the vulnerability described in Section 4.1.1 has to be exploited in order to be able to deploy the payload and let the tools react to the situation.

Console verification If all the tools correctly react to the shellcode sent to the vulnerable machine, the console must correctly and timely show the generated alarms.

Test scenario This scenario evaluates how the system reacts to attack launched against hosted application: on the Exposure VM is installed a vulnerable streaming server (IceCast server) answering HTTP requests coming from clients outside the network. For this test we pretended the host machine to be an external client and launched the attack against the server, dropped a shellcode and tried to manipulate data on the vulnerable system: both Nemu and AccessMiner reacted to the attack.

[3] http://www.metasploit.com

In the following sections the three phases previously defined will be described.

4.2.1 Shellcode Choice

The shellcode that was chosen was the one that allowed the highest degree of freedom on the vulnerable machine and allowed us to know exactly when the AccessMiner policies were violated. Therefore, for testing, we used a simple *shell_bind_tcp* shellcode, which makes the vulnerable machine spawn a shell and connect it to a socket purposely created on the attacker's side.

As a result, if the exploit was successfully landed, we had full remote control over the Exposure VM and could easily write a file in the directory configured to have only read-permissions, thus trigger AccessMiner.

4.2.2 Vulnerability Exploitation

The *icecast_header* in Metasploit was built to exploit the vulnerability described in Section 4.1.1 and it was therefore the best choice to deploy a shellcode into the vulnerable machine.

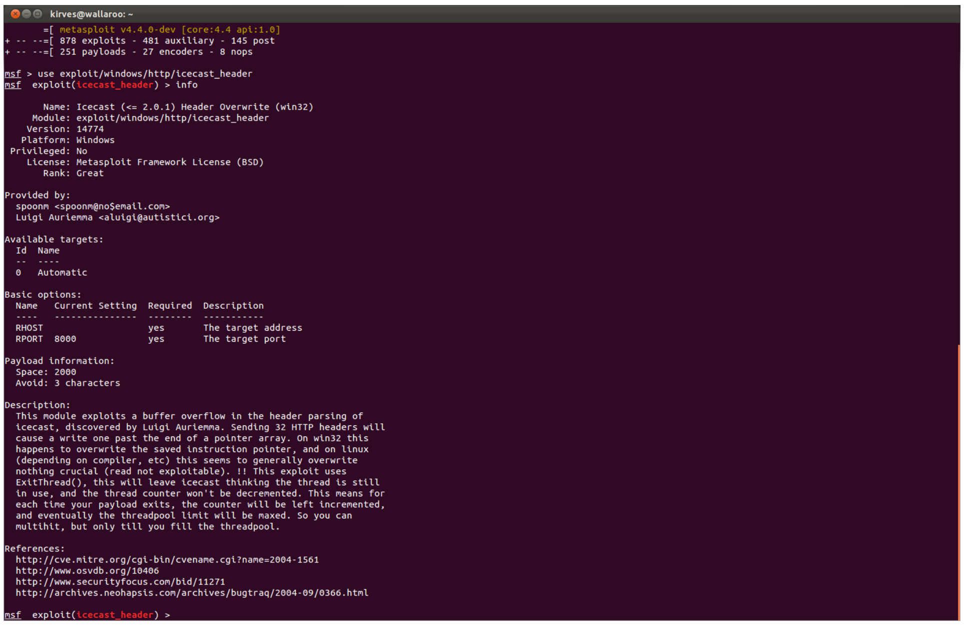
Since the idea was to conduct the attack in realistic condition, as far as possible, the Metasploit software was not executed on a virtual machine in the virtual environment but rather on the host machine. No other external machines were used, so that no exposure was given to the test environment allowing external factors to possibly compromise the experiment.

After starting the IceCast2 vulnerable server on the target machine, we run the Metasploit console on the host machine and load the icecast_header module, as shown in Figure 4.3; we then set the required options for the module: the only one needed at this point is the *RHOST*, set to the Exposure VM network address (172.0.0.50) (Figure 4.4).

After loading the payload chosen in Section 4.2.1, the attack is successfully launched to the vulnerable machine. Nemu should have now detected the shellcode going through the network while AccessMiner should have seen some policy violations and therefore raised an alarm. Therefore both Nemu and AccessMiner should have sent security events to the console, which can now display them on the web application.

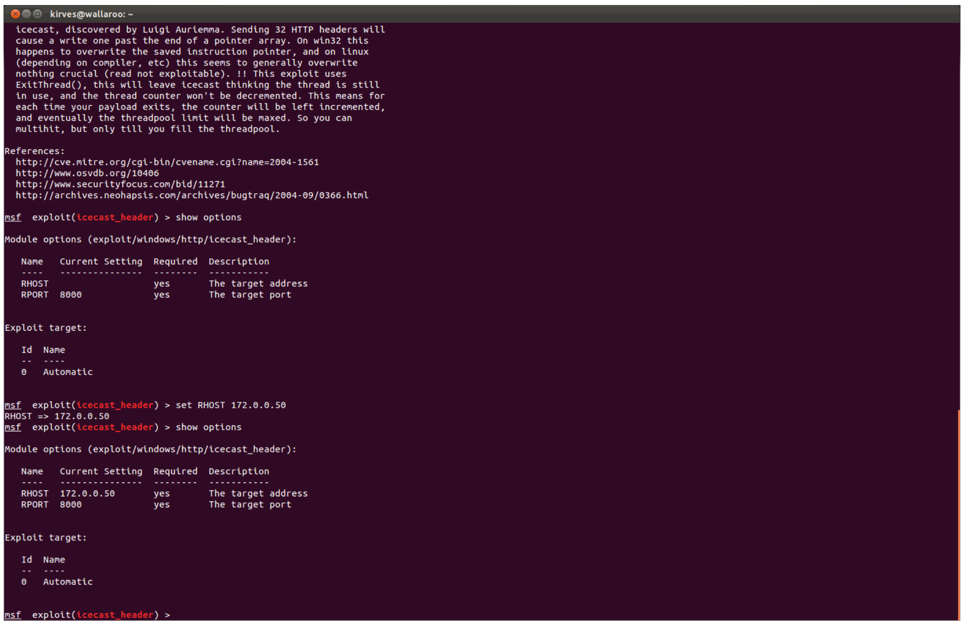
4.2.2.1 i-Code Console reaction

Right after launching the attack, Nemu correctly detects an attack originating from the host machine client (with IP 172.0.0.1 – mapped as "Reserved" – and random port higher than 1024) and aimed at the Exposure VM server (IP 172.0.0.50 and port 8000), as shown in Figure 4.5. Also, the shellcode is sent to the Anubis server, where the in–depth analysis is immediately started (see Figure 4.6).



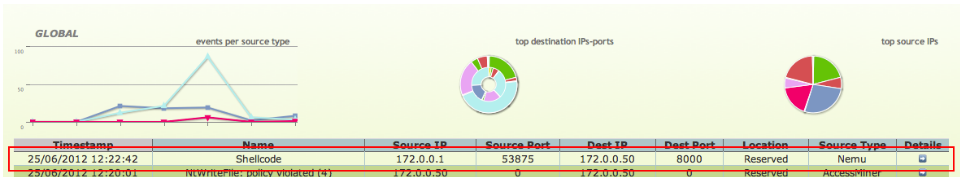
```
kirves@wallaroo: ~
       =[ metasploit v4.4.0-dev [core:4.4 api:1.0]
+ -- --=[ 878 exploits - 481 auxiliary - 145 post
+ -- --=[ 251 payloads - 27 encoders - 8 nops

msf > use exploit/windows/http/icecast_header
msf  exploit(icecast_header) > info

       Name: Icecast (<= 2.0.1) Header Overwrite (win32)
     Module: exploit/windows/http/icecast_header
    Version: 14774
   Platform: Windows
 Privileged: No
    License: Metasploit Framework License (BSD)
       Rank: Great

Provided by:
  spoonm <spoonm@no$email.com>
  Luigi Auriemma <aluigi@autistici.org>

Available targets:
  Id  Name
  --  ----
  0   Automatic

Basic options:
  Name   Current Setting  Required  Description
  ----   ---------------  --------  -----------
  RHOST                   yes       The target address
  RPORT  8000             yes       The target port

Payload information:
  Space: 2000
  Avoid: 3 characters

Description:
  This module exploits a buffer overflow in the header parsing of
  icecast, discovered by Luigi Auriemma. Sending 32 HTTP headers will
  cause a write one past the end of a pointer array. On win32 this
  happens to overwrite the saved instruction pointer, and on linux
  (depending on compiler, etc) this seems to generally overwrite
  nothing crucial (read not exploitable). !! This exploit uses
  ExitThread(), this will leave icecast thinking the thread is still
  in use, and the thread counter won't be decremented. This means for
  each time your payload exits, the counter will be left incremented,
  and eventually the threadpool limit will be maxed. So you can
  multihit, but only till you fill the threadpool.

References:
  http://cve.mitre.org/cgi-bin/cvename.cgi?name=2004-1561
  http://www.osvdb.org/10406
  http://www.securityfocus.com/bid/11271
  http://archives.neohapsis.com/archives/bugtraq/2004-09/0366.html

msf  exploit(icecast_header) >
```

Figure 4.3: Metasploit console with icecast_header module



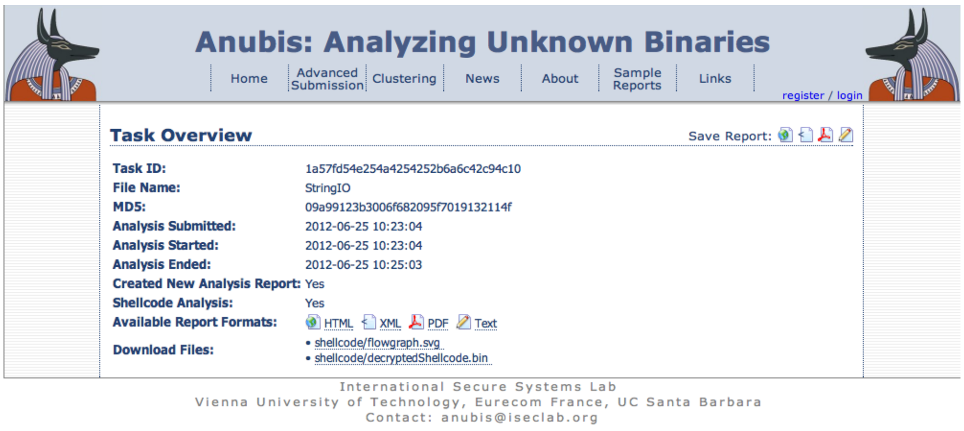
```
kirves@wallaroo: ~
  icecast, discovered by Luigi Auriemma. Sending 32 HTTP headers will
  cause a write one past the end of a pointer array. On win32 this
  happens to overwrite the saved instruction pointer, and on linux
  (depending on compiler, etc) this seems to generally overwrite
  nothing crucial (read not exploitable). !! This exploit uses
  ExitThread(), this will leave icecast thinking the thread is still
  in use, and the thread counter won't be decremented. This means for
  each time your payload exits, the counter will be left incremented,
  and eventually the threadpool limit will be maxed. So you can
  multihit, but only till you fill the threadpool.

References:
  http://cve.mitre.org/cgi-bin/cvename.cgi?name=2004-1561
  http://www.osvdb.org/10406
  http://www.securityfocus.com/bid/11271
  http://archives.neohapsis.com/archives/bugtraq/2004-09/0366.html

msf  exploit(icecast_header) > show options

Module options (exploit/windows/http/icecast_header):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST                   yes       The target address
   RPORT  8000             yes       The target port


Exploit target:

   Id  Name
   --  ----
   0   Automatic


msf  exploit(icecast_header) > set RHOST 172.0.0.50
RHOST => 172.0.0.50
msf  exploit(icecast_header) > show options

Module options (exploit/windows/http/icecast_header):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST  172.0.0.50       yes       The target address
   RPORT  8000             yes       The target port


Exploit target:

   Id  Name
   --  ----
   0   Automatic


msf  exploit(icecast_header) >
```

Figure 4.4: Metasploit icecast_header options



Timestamp	Name	Source IP	Source Port	Dest IP	Dest Port	Location	Source Type	Details
25/06/2012 12:22:42	Shellcode	172.0.0.1	53875	172.0.0.50	8000	Reserved	Nemu	➡
25/06/2012 12:20:01	NtWriteFile: policy violated (4)	172.0.0.50	0	172.0.0.50	0	Reserved	AccessMiner	➡

Figure 4.5: i-Code console reacting to Nemu alert in the first testing scenario.



Task Overview

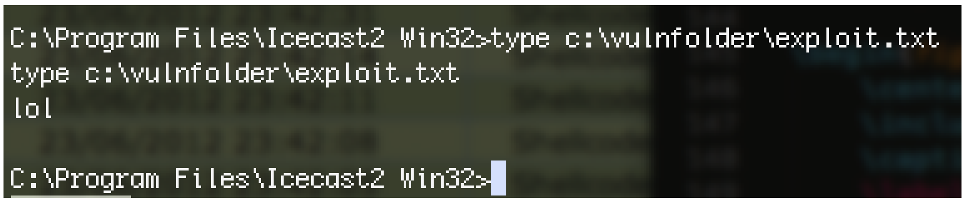
Task ID:	1a57fd54e254a4254252b6a6c42c94c10
File Name:	StringIO
MD5:	09a99123b3006f682095f7019132114f
Analysis Submitted:	2012-06-25 10:23:04
Analysis Started:	2012-06-25 10:23:04
Analysis Ended:	2012-06-25 10:25:03
Created New Analysis Report:	Yes
Shellcode Analysis:	Yes
Available Report Formats:	HTML XML PDF Text
Download Files:	• shellcode/flowgraph.svg • shellcode/decryptedShellcode.bin

International Secure Systems Lab
Vienna University of Technology, Eurecom France, UC Santa Barbara
Contact: anubis@iseclab.org

Figure 4.6: Shellcode submission to Anubis in the first scenario.

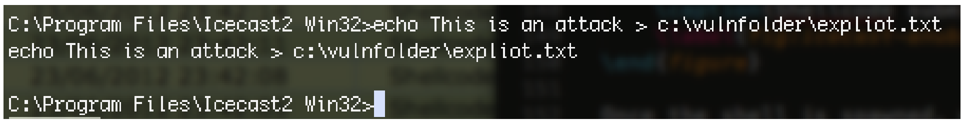
Once the shell is spawned, we have to test AccessMiner's capability of detecting the policy violation without raising false positives; therefore, as a first test, we issue a *type* command on a file in the monitored folder and see if this action is erroneously flagged as not allowed (see Figure 4.7). The console does not report any policy violations on this action.

When, instead, the same file is written (Figure 4.8), AccessMiner raises an event, as shown in Figure 4.9. In this case the "detail" icon opens a popup with detailed information about the violation (Figure 4.10).



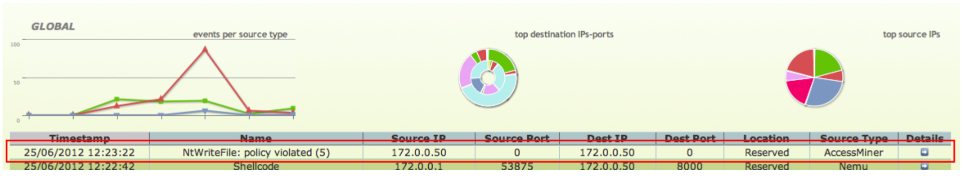
```
C:\Program Files\Icecast2 Win32>type c:\vulnfolder\exploit.txt
type c:\vulnfolder\exploit.txt
lol

C:\Program Files\Icecast2 Win32>
```

Figure 4.7: Issuing a *read* operation on monitored folder.



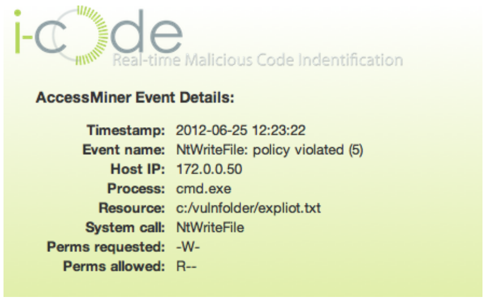
```
C:\Program Files\Icecast2 Win32>echo This is an attack > c:\vulnfolder\expliot.txt
echo This is an attack > c:\vulnfolder\expliot.txt

C:\Program Files\Icecast2 Win32>
```

Figure 4.8: Issuing a *write* operation on monitored folder.



Timestamp	Name	Source IP	Source Port	Dest IP	Dest Port	Location	Source Type	Details
25/06/2012 12:23:22	NtWriteFile: policy violated (5)	172.0.0.50	0	172.0.0.50	0	Reserved	AccessMiner	➡
25/06/2012 12:22:42	Shellcode	172.0.0.1	53875	172.0.0.50	8000	Reserved	Nemu	➡

Figure 4.9: i-Code console reacting to AccessMiner alert in the first testing scenario.



AccessMiner Event Details:

Timestamp:	2012-06-25 12:23:22
Event name:	NtWriteFile: policy violated (5)
Host IP:	172.0.0.50
Process:	cmd.exe
Resource:	c:/vulnfolder/expliot.txt
System call:	NtWriteFile
Perms requested:	-W-
Perms allowed:	R--

Figure 4.10: Detailed report popup for AccessMiner's events.

4.3 Deployment in Real-World Networks

The diverse set of detection approaches that comprise the i-Code architecture allows it to be adopted to fit different operational needs and deployment scenarios. As part of the testing phase, two instances of the i-Code system have been installed and are operational in real networks.

The first instance has been installed at the University of Crete, and the second one at the FORTH-ICS. In both deployments, Nemu is used to scan the traffic that goes through the main gateway that connects each organization's campus to the Internet. Nemu has been configured to scan the client-to-server data of all established TCP connections, irrespectively of the destination port (service). In both cases, all detected attacks are submitted to Anubis for further analysis.

Due to the more strict firewall configuration policies that are in place at FORTH-ICS, the number of detected incidents for the past month is about three per week, and are mostly related to non-managed guest devices that connect to the local network through the provided public WiFi. In contrast, the more open-access nature of the University network results to services being exposed to the public Internet, and consequently to a higher number of attacks against local servers. The number of detected incidents at the University of Crete for the last three months is about two incidents per day.