STEALTH SANDBOX ANALYSIS OF MALWARE

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Ömer Sezgin Uğurlu

August, 2009

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Ali Aydin Selçuk(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Mustafa Akgül

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. İbrahim Körpeoğlu

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

STEALTH SANDBOX ANALYSIS OF MALWARE

Ömer Sezgin Uğurlu
M.S. in Computer Engineering
Supervisor: Assist. Prof. Dr. Ali Aydin Selçuk
August, 2009

Malware is one of the biggest problems of the world of bits and bytes. Generally malware does activities a user normally does not do, such as becoming part of a virtual army or submitting confidential data of the user to the malware author. There are publicly available analysis services for unknown binaries. Sandbox analysis is performed by execution of an untrusted binary in an isolated environment. It is a very common technique for malware research. Publicly available sandbox analysis platforms help users to see traces of the execution without harming their system. Also it helps owners of the sandbox to collect malware and makes the job of analysts easier. One major problem of the public sandbox testing is that malware authors can also benefit from analysis of sandboxes. If they can identify sandbox systems they can hide malicious behavior. This thesis presents the publicly used Anubis sandbox, detection mechanisms used against Anubis[3], further possible detection mechanisms and our efforts for hiding fingerprint of Anubis from malware and decreasing the resulting false negative rates for the malware detection.

Keywords: malware analysis, sandbox analysis, stealth analysis.

ÖZET

GIZLI KUM BAHÇESI ILE KÖTUCÜL YAZILIM ANALIZI

Ömer Sezgin Uğurlu
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Assoc. Prof. Dr. Ali Aydin Selçuk
Ağustos, 2009

Kötücül yazılım bit ve baytlar dünyasının en büyük problemlerinden birisidir. Genellikle kötücül yazılımlar bir kullanıcının normalde yapmayacağı, sanal bir orduya üye olmak, ya da kullanıcının kendi özel bilgilerini kötücül yazılımı yazanlara göndermek gibi eylemleri gerçekleştirirler. Bilinmeyen programlar için halka açık analiz servisleri bulunmaktadır. Kum bahçesi analizi güvenilmeyen programların izole ortamlarda çalıştırılması ile yapılır. Bu yöntem kötücül yazılım analizinde sık olarak kullanılan bir yöntemdir. Kum bahçesi analizi kötücül yazılım analizinde önemli bir tekniktir. Halka açık kum bahçesi analizi platformları, kullanıcıların kendi bilgisayarlarına zarar vermeden programın çalışması sırasında oluşan izleri görmelerine yardım eder. Halka açık kum bahçesi analizi ortamlarının ana problemlerinden biri de kötücül yazılım sahiplerinin de servisleri kullanabilmesidir. Eğer yazarlar kum bahçesi sistemlerini tanıyabilirlerse kötücül davranışları saklayabilirler. Bu tez halka açık olan Anubis[3] sistemi için, bilinen tespit sistemlerini, bilinmeyen olası tespit sistemlerini ve bizim Anubis'in parmak izlerini kötücül yazılımlardan saklama ve kötücül yazılım tespitindeki hatalı negatif oranlarını düşürme çabalarımızı sunmaktadır.

Anahtar sözcükler: kötücül yazılım, kum bahçesi analizi, gizli analiz.

Acknowledgement

I would specifically thank to my advisor Ali Aydın Selçuk for helping me through my whole M.Sc progress and expanding my vision in security.

This masters thesis was carried out in cooperation with Ulrich Bayer at International Secure Systems Lab in Technical University of Vienna.

Moreover, I would like to thank my advisor Engin Kırda for his patience and support.

I would like to thank Tübitak for financially sponsoring my M.Sc studies with scholarship.

Finally, thanks to my family and friends for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Each time a computer user executes a software he has a list of strong expectations. Generally these expectations are related with the functionality of the software. If it works, everything is perfect. Source or trustability of the software is not generally a concern.

Using digital signature mechanisms may establish trust on source of software. But still there is the risk of malicious behavior. We may take legal steps for software provider in case of malicious behavior. Since software is electronically signed we can be sure that software is authentic. It is not changed after the signature process. But conserved integrity does not imply security. Software may be created with malicious behavior in mind or it may be infected before signing process. Therefore the signed binary files are also may be secure and subject to future analysis.

Malware, the short form of "Malicious Software", is the type of software which is designed to perform malicious activities, without the owner's informed consent. Activities may involve information damage, information theft, zombie[1] in a DoS[2] attack.

[1] A zombie computer is a computer attached to the Internet that has been compromised by a hacker, a computer virus, or a Trojan horse

[2] Denial of Service

Anti-virus software may flag the binary as malicious by using signature detection or the heuristic approaches. Anti-virus vendors extract signatures from binary streams to identify malicious files. Scanning files for extracted signatures is the fastest way of detection. As a tradeoff it can fail to detect slightly modified malicious binaries.

While detection using signatures is fast, generation of malware signatures is not. It requires expert analyzers to work on subjects. It's a costly process. If tools gather high level information like a report the software classification may be done much faster. Human readable high level execution trace report gives better oral representation of software execution trace.

With high level report execution trace report, power users may want to see what action is taken by the unknown binary. Which files are modified, which registry values are read etc.. Analysis procedure may damage the the host environment. To prevent damage software may executed in a isolated environment which is called as sandbox. But not every expert may setup a sandbox for in house testing. Executing the software on a remotely hosted sandbox is probably the most sterile testing method. Code may behave and nuke the whole system. The harm will be done in sandbox and will not cause any damage on host environment.

Anubis [3] is a Web front end for TTanalyze [5] sandbox environment for Malware analysis. Automated bots or computer users may upload portable executable (PE) [21] files to obtain high-level reports of binary behavior. Uploaded binaries are executed in a virtual machine.

One major problem with Anubis environment is it's availability. Anubis is both available to malware analyzers, victims and malware authors. Usage of the system analyzers and victims is desired. But if malware authors can detect presence of Anubis, then they may hide their malicious activity. The idea is similar to malware signature. However instead of extracting signature from a binary file, signature is generated from running sandbox environment. MAC address of Ethernet device, or hard disk serial ID of sandbox image can be used to generate fingerprint. A signature of sandbox may lead to detection of sandbox by

malware. Malware may behave like normal software and hide malicious behavior. Therefore generated reports can be used to establish trust in malicious software, which may cause more damage. Level of trust is increased from “unknown” to “seems safe”. Traces seen in the reports seem totally normal.

My additions to Anubis will try to remove its fingerprints and provide an invisible shield for the viewpoint of subject analysis binaries. Ideas can be summarized as providing random execution environment for analysis of binaries. Trackable properties of system will be randomized using a key derived from the hash. Therefore execution of the same binary will lead to same execution trace. But with different binaries traceable system properties will be different. Fingerprint extracted for one binary won’t match for another one. Therefore there will be no fingerprint for Anubis and malware won’t be able to detect Anubis by looking at its fingerprint.

Chapter 2

Background

This chapter provides basic background information regarding malware, basic analysis of malware, tools used for malware analysis, Anubis, Anubis like platforms for malware analysis, Windows internals especially related to fingerprinting, and hardware components and fingerprint extraction. This chapter does not provide any solution for described problem.

2.1 Malware

Malware is a common name for all kinds of unwanted software such as viruses, worms, trojans and jokes. [10]

Being an unwanted piece of software most of the time, malware should be able to cover its traces to survive. Malware can be detected by software or by abnormal behavior of the system. A pop-up about our computer being in danger or crazy flashing network lights can be caused by malware. If malware somehow is detected, it can be further analyzed.

2.1.1 Malware Analysis

In order to understand structure and behavior of malware a sophisticated analyzer is needed. With strong skills in software, machine architecture, assembly and network. In terms of cost such sophisticated professionals cost too much. Therefore we need automated systems to decrease load in those professionals.

2.1.1.1 Static Analysis

Static analysis depends on observations on disassembled code. Analysis is done without executing the binary. By tracing the disassembled binary control and/or data flow graphs can be generated. However it can be problematic in case of self modifying binaries.

This sort of analysis does not cause any harm or infection on host computer. Guest binaries are opened but not executed. Therefore malicious behavior can't have any persistent effect on host computer. In one extreme case infection can occur is the case where analyzer software has bug in it's parser. Therefore execution of analysis might cause execution of arbitrary code. Example to this is tcpdump binary. It's a binary for network traffic inspection. Tcpdump had a bug which caused specially crafted packets to execute code on the machine running tcpdump.

2.1.1.2 Dynamic Analysis

Dynamic analysis is done by executing binary and collecting any traces that are created during execution. The behavior of the suspected binary is carefully watched. Any interaction between the malicious binary and the system are recorded.

One big major problem with dynamic analysis is trace-ability of only execution path. Alternate paths are not taken into account. So if malware does not activate itself, we won't be able trace malicious execution.

2.2 Tools

This section provides information about industry standard tools used in reverse engineering. Generally this tools are designed to be used against binary code. No source code is input is necessary while using this tools against binary files.

2.2.1 Detours

Detours [13] is a library for interception of Windows functions. It modifies execution redirecting Windows API call to `detour` function created by user.

An example scenario regarding use of detours is as follow: You have a binary file, which is modifying data on a block device. You want to know what is written or read exactly on to that block device. You want to hook `ReadFile` function. The execution in normal scenario is your program calls `ReadFile` function in kernel32.dll.

```
binary  -->  kernel32.dll  --> hard disk
```

When you create detours function for that file the execution is like

```
binary  -->  detours_function --> hard disk
```

Instead of `ReadFile` function `detours_function` you have created for that file is executed. What can you do with a detours function is completely up to you. You can redirect parameters to a file before calling kernel32's `ReadFile` function. Or you don't do anything at all. By detouring multiple calls even non existent components on a system.

In summary this tool can be used to change the communication between the binary and the libraries. This is a common approach in reverse engineering.

2.2.2 IDA

"IDA Pro is a Windows or Linux hosted multi-processor disassembler and debugger that offers so many features it is hard to describe them all."[1] For our work we used free version of `IDA Pro Freeware` [8]. It's a great tool for static analysis. It provides graphical navigation through binary components. Implements a few anti-anti-disassembly techniques. For example a common technique is using invalid opcodes in binary. Invalid locations are known by the author and they are never executed by binaries. When disassembler try to disassemble them they fail, becaue they don't execute code. Maps calls to human readable function names. For example a call to dll function is reflected to user.

For manual static analysis of suspect binaries we used IDA.

2.2.3 Objdump

We used objdump tool from mingw32 package. It displays information about object files. It's main targeted user base is people working on compilation tools, or low level binary inspection techniques. It can parse and display various resources on an object file. It can display most of the information displayed by IDA. Which includes, function imports, function exports, resource segments, etc.. It is a part of GNU suite.

In static analysis this tool is also used to gather information.

2.2.4 Sysinternals Tools

Sysinternals tools [26] work by hooking various functions on Windows kernel. Accesses to Windows Registry or file system can be monitored with this tools. Filemon shows Windows file system activity, and Regmon shows Windows registry activity item.

[1] `http://www.hex-rays.com/idapro/`

By providing filters this tools can monitor behavior of a specific binary. This tools belong to the category of dynamic analysis, since it requires code to be executed.

2.3 Analysis Platform

Our analysis platform is Anubis which is both a static and a dynamic analysis environment. It analyzes binary by dynamic execution. But in terms of being malware hostile to host system it has same properties with static analysis.

Major problem with dynamic analysis is also a problem of Anubis. Since Anubis only traces the execution it leaves a hole for malware authors. If malware detects Anubis it can simply not execute malicious behavior routines. Therefore system may not show real-world activity of the malicious binary.

2.3.1 QEMU

QEMU [6] is a fast machine emulator. It's a freely available open source project authored by Fabrice Bellard. Since it's an emulator, it can host unmodified guest operating systems. It can emulate various architectures on various hosts. QEMU can run on x86, PowerPC, ARM, Sparc, Alpha and MIPS platforms to host x86, PowerPC, ARM and Sparc CPUs. System consists of several components:

- CPU
- emulated VGA display
- serial port

QEMU runs fast using portable dynamic translator in blocks. In summary this technique converts executable code from guest instruction set to host instruction set, from current execution point to next jump point. Therefore instead of

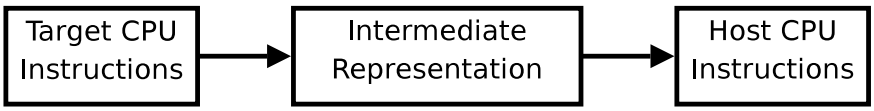

Figure 2.1: QEMU dynamic translation

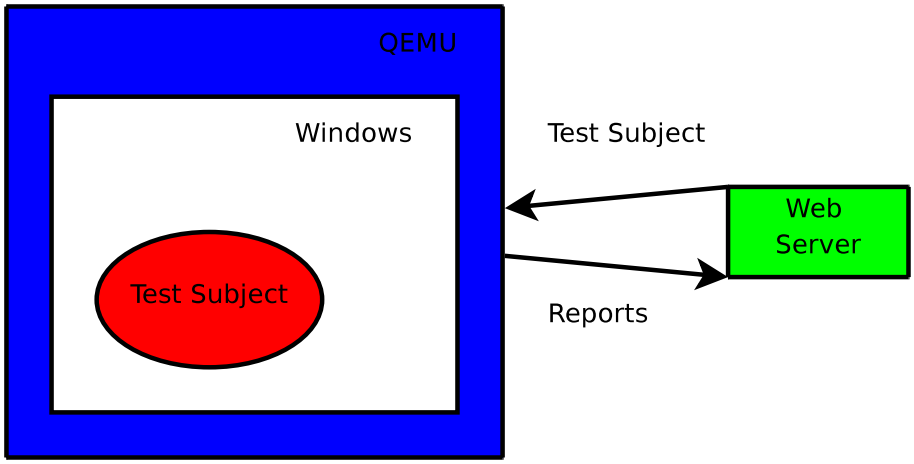

Figure 2.2: Anubis architecture

converting every single instruction one-by-one block translation technique compiles these instructions. This reduces context switching operations so the overall performance of emulated system increases.

2.3.2 Anubis

Main idea of Anubis is tracking the calls to the operating system in the CPU level. Internal trace & execution operations are done without any memory fingerprint. But the process to launch suspect binary may leave some fingerprint.

Anubis can be considered as a modified machine emulator. It's basic engine is extended from well known machine emulator QEMU. For emulation environment it uses modified QEMU source. It has Windows operating system installed inside. It is slightly customized in sense of network IP address, wallpaper an execution daemon for observing `C:\` drive. When the analyze script is executed, subject

binary is copied into Windows instance running inside modified QEMU. Execution daemon executes the subject binary and the system collects the results, and presents them in multiple output formats.

2.3.2.1 Binary Tracing with Anubis

Anubis can trace execution of win32 binaries. Since it's using an emulator it has low level view of the system. For every executed instruction, we have the view of complete CPU. All registers are accessible by us. For matching this low level view with higher level application view, Anubis uses CPU registers. There are basically two problems solved by Anubis. Matching the application with CPU instruction, and matching the DLL call with CPU instruction.

Anubis makes application trace feasible by using the CR3 register a.k.a PDBR[2] on x86 architecture. It has unique value for each process. Since it has unique value for every process, it can be used to match CPU instructions with running process. It's reloaded on each context switch. Therefore it can always be trusted as a pointer for process.

The second problem solved by Anubis is matching DLL calls with CPU instructions. Problem of matching executed machine code with the Windows operating system DLLs. In win32 system functions resting in system DLLs has constant entry points. We can access this entry points using tools like nm, objdump. QEMU processes instructions in blocks. By comparing these entry points with EIP we can find out which function is called. This logic is executed in every block start. By tapping into execution start of each block, Anubis manages to intercept any DLL call. Also value returned by the call can be manipulated. Simple illustration can be seen in Figure 2.5

If we combine the values of CR3 & EIP registers, we can match the currently executed instruction with the corresponding process and win32 API call.

[2]page-directory base register is uniquely assigned by Windows operating system to each process

After determining the called function we might access in and out parameters with the help of stack information. In 32 bit Windows systems each parameter takes 32 bits of memory in stack. Knowing number and type of parameters we can access internals of API calls.

2.3.2.2 Components of Anubis

Anubis has several components. `generator.exe` is a tool to for code generation. It generates callback handlers for win32 functions. Given function names, parameter types and parameter names, it automatically generates Callback interfaces.

For each function generator creates 2 classes. An interface class and a decoder class. For example for `GetComputerNameA` method `GetComputerNameADecoder` and `GetComputerNameAInterface` classes are generated. The generated interface class declares the method interfaces for execution callback. Decoder class is used to obtain the physical addresses in the address space of QEMU. Since QEMU has an internal MMU, addresses in QEMU and host operating system aren't the same.

Interface class defines 2 methods for every function a `WasCalled` method and a `HasReturned` method. As the names imply former one is to be invoked before the call is started and the later one is invoked after the execution of function by the operating system installed inside the QEMU. Figure 2.3 is a sample interface class for GetComputerNameA method and Figure 2.4 is decoder class for GetComputerNameA method. Decoder class has members for each parameter. VirtualAddress member for each parameter and according to parameter is `in` and or `out` there are members accordingly. This duplicate members helps one class to hold both before and after data.

2.3.2.3 Anubis Public Interface

Anubis is malware analysis reports are publicly available. Users can upload any binary they are suspected. After upload any user may obtain detailed reports.

```
class GetComputerNameAInterface: public CCallbackObject
{
public:
    virtual ~GetComputerNameAInterface() {}

    virtual void GetComputerNameAWasCalled(
        int8_t *lpBuffer,
        uint32_t *nSize) {}

    virtual void GetComputerNameAHasReturned(
        int8_t *lpBuffer,
        uint32_t *nSize,
        int32_t  _functionResult) {}
};
```

Figure 2.3: Generated interface class for GetComputerNameA method

There major drawback of this openness. Anubis is also open to malware authors. Anubis leaks information about underlying infrastructure. Generated reports may be used as a side-channel to transfer fingerprint information to users. For example, value of a registry key is displayed in reports. If it carries unique information of that system, that will lead to disclosure of fingerprint information.

2.3.3 Automated Analysis Suite

Automated Analysis Suite[3] is an Anubis like system. The Automated Analysis Suite consists of two major components:

- Nepenthes [1]: Nepenthes' main aim is to collect malware. It achieve this task by emulating known vulnerabilities without giving full execution rights. Nepenthes emulates being exploited, therefore permits and targets automated download of malware. It's publicly available.
- CWSandbox: Main automated dynamic analysis component of the system is CWSandbox [31]. CWSandbox provides malware analysis environment

[3]http://www.sunbeltsoftware.com/Developer/Sunbelt-CWSandbox/

```
class GetComputerNameADecoder: public CallbackDecoder
{
private:
    int8_t *lpBuffer_out;
    VirtualAddress lpBuffer_addr;
    uint32_t *nSize_out;
    uint32_t *nSize_in;
    VirtualAddress nSize_addr;
    int32_t _functionResult;
public:
    GetComputerNameADecoder() :
        lpBuffer_out(NULL),
        nSize_out(NULL),nSize_in(NULL) {}
    virtual ~GetComputerNameADecoder();

    VirtualAddress getlpBuffer_addr() const {
        return lpBuffer_addr;}
    VirtualAddress getnSize_addr() const {
        return nSize_addr;}

    static CallbackDecoder* createObject() {
        return new GetComputerNameADecoder();}

    virtual std::string getFunctionName () const {
        return "GetComputerNameA";}
    virtual void functionWasCalled(
        uint32_t threadId,
        const std::vector<CCallbackObject*> &cbs);
    virtual void functionHasReturned(
        uint32_t threadId,
        const std::vector<CCallbackObject*> &cbs);
};
```

Figure 2.4: Generated decoder class for GetComputerNameA method

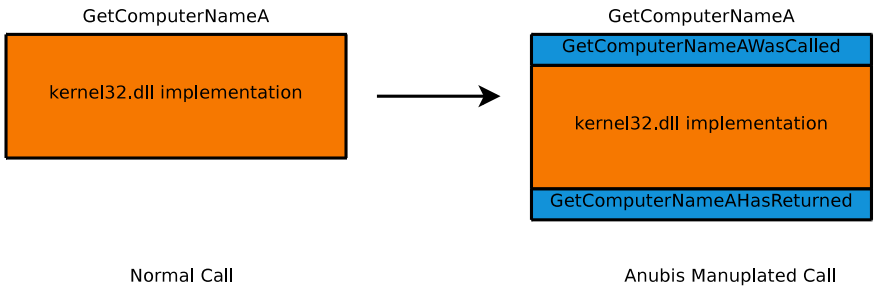

Figure 2.5: Anubis call manipulation and interception capability

using the same approach as Anubis. By recording Windows API calls the subject binary makes CWSandbox [31] can generate detailed reports. In addition to Anubis, CWSandbox can also analyze various forms of documents, including MS Office, Adobe Flash, malicious web sites. The only requirement is setup of a Windows sandbox environment.

2.4 Windows

This section provides information about Windows API. Ideas of possible fingerprinting locations are also discussed. But ideas are further discussed in Chapter 4. Internals of Windows API are obtained from Microsoft Developer Network [22].

2.4.1 Registry API

The Windows Registry is a special form of database to store various parameters. It can provide both information and storage. Microsoft Windows operating system, services, and applications.

Initial aim of the Windows Registry was providing a single source for configuration files. Instead of INI based configuration of pre 3.1 releases of Windows, the registry was introduced as a new approach.

```
#include <winreg.h>
#include <stdlib.h>

int main(){
    HKEY key;
    DWORD varType = REG_SZ;
    char buffer[256];
    RegOpenKeyEx(HKEY_CURRENT_USER,
        "HKLM\\System\\CurrentControlSet\\Control"
        "\\ComputerName\\ActiveComputerName",
        0, KEY_READ, key);
    printf("Computer Name is %s\n", key);
    RegCloseKey(key);
}
```

Figure 2.6: Sample code to read computer name from registry.

The Windows Registry provides INI like interface. Registry has two major components:

Registry Keys: Keys can have sub-keys, which can resemble folder structure. Key hierarchies can be denoted using backslashes. A\B\C means B is a subkey of A and C is a subkey of B.

Registry Values, are the various forms of data in the form of dictionary, that's stored in the keys.

Reading a registry key consists of three fundamental Windows calls. Opening a key reading value and closing the key. While opening a key, a handle is created. Values are queried over this handle. And then the handle is closed. Typical code is shown in Figure 2.6.

2.4.2 File API

All of the persistent information are organized in hard disk, and accessed by file system API. Files have contents and attributes. Creation, modification and

List of Registry Value Types		
0	REG_NONE	No type
1	REG_SZ	A string value
2	REG_EXPAND_SZ	An "expandable" string value that can contain environment variables
3	REG_BINARY	Binary data (any arbitrary data)
4	REG_DWORD REG_DWORD_LITTLE_ENDIAN	A DWORD value, a 32-bit unsigned integer (numbers between 0 and 4,294,967,29 5 [$2^{32} - 1$]) (little-endian)
5	REG_DWORD_BIG_ENDIAN	A DWORD value, a 32-bit unsigned integer (numbers between 0 and 4,294,967,295 [$2^{32} - 1$]) (big-endian)
6	REG_LINK	symbolic link (UNICODE)
7	REG_MULTI_SZ	A multi-string value, which is an array of unique strings
8	REG_RESOURCE_LIST	Resource list
9	REG_FULL_RESOURCE_DESCRIPTOR	Resource descriptor
10	REG_RESOURCE_REQUIREMENTS_LIST	Resource Requirements List
11	REG_QWORD REG_QWORD_LITTLE_ENDIAN	A QWORD value, a 64-bit integer (either big- or little-endian, or unspecified) (Introduced in Windows 2000)

Table 2.1: List of Registry Value Types

access times are stored as file attributes. Among different installations some files will probably differ. When we read a file we change last access time. When we modify a file we change last write time. Since accesses changes time information in attribute data, one may simply find a time-stamp on a file, which can be used as fingerprint. Also if any system wise unique file is created, or has unique content, this might also be used as a fingerprint. For Windows systems file attribute structure is shown in figure 2.7. If the topic is files we can choose to overwrite attribute data as well as contents of the file itself.

File attributes don't provide any evidence. They provide protection or comfort for the file. A hidden file takes more clicks to delete for example.

Data content on the files are also can be altered. But attention is needed. It should not invalidate the system integrity.

File I/O life chain is similar to registry life chain. Files are opened, processed and closed. Handles are created during opening process. For processing or closing files, these handles are used.

```
typedef struct _WIN32_FILE_ATTRIBUTE_DATA {
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
}WIN32_FILE_ATTRIBUTE_DATA, *LPWIN32_FILE_ATTRIBUTE_DATA;
```

Figure 2.7: Win32 File Attribute Data Structure

2.4.3 Screen

By looking at the desktop and the wallpaper, sometimes we're also able to identify systems. If a custom and non-standard wallpaper is used or if any part of the screen constantly holds the same unique image, this image may be used as a fingerprint.

2.4.4 Memory

Similar to screen scenario, memory parts if can be identified, can be used as a fingerprint. Memory also contains a wallpaper, handles, processes, etc.. .

There are two main points in memory. What is stored on a single block, and where that block is stored. The stored information can be overridden if it does not invalidate the consistency of system.

2.5 Hardware Identifiers

By hardware identifiers we mean the information embedded into hardware in manufacturing process. This parameters meant to be unique and unmodifiable. Therefore they can be used to identify a real system. Evaluation of these identifiers are explained below. The ideas regarding randomization will be further

discussed in Chapter 4.

2.5.1 CPU Serial

Executing a few instructions gives unique results in most CPUs. Therefore it can be considered as a strong fingerprint. `CPUID` instruction can be executed on CPUs supporting it. Some of the modern CPUs has this feature. But the CPU emulated by QEMU is `Pentium II` therefore it does not have this UniqueID feature.

Still after the theory we experimented with CPU-Z [4] and `wbemtest.exe` [5]. With both software we were not able to obtain CPU UniqueId parameter. CPU-Z recognized our CPU as `Intel Pentium II` with `Klamath` code name. WBEMTest is used to query Win32_Processor.DeviceID="CPU0". UniqueId string is reported as null.

2.5.2 Memory SPD

New memory modules has a chip called Serial Presence Detect (SPD). This tiny chip has 8 pins and assembled in to memory PCB. It stores information about the memory chips resting on the same memory module. The contained information is used by BIOS to tune timings for the performance and stability. It also has an additional space for storing serial numbers.

However, QEMU does not emulate SPD chip information. Again a hardware based fingerprint for a normal system is not applicable for Anubis.

2.5.3 Ethernet MAC address

On a normal computer system Ethernet MAC address is stored in eprom of Ethernet interface. These addresses are designed to be unique. This address is

[4] cpuz from `http://cpuid.com`

[5] "Windows Management Instrumentation (WMI) Tester, also called WBEMTest"

composed of 6 bytes; the first 3 bytes identifies the manufacturer and the later 3 bytes are random. A manufacturer may have more than one identifiers. It's nearly impossible to have two machines with the same MAC address. Therefore it's a strong fingerprint.

The Ethernet MAC address is defined is defined in Anubis configuration file. Since it is specified by the Anubis configuration we can change it in configuration. But the problem with this is that, we're not starting Windows from scratch. Instead we have a sandbox image that's executed up to a point. Changing parameters in configuration may not change active configuration. Also this information is probably stored in persistent storage and Windows Registry. Besides persistent storage, this address also might be cached in the memory.

2.5.4 Disk Information

Since all persistent information is stored on disk, emulated disk carries fingerprints for the system. Segments of the disk covered by the file system are changeable interceptable by the interception of file system I/O routines.

If a segment on the disk is used by a file, it can be perfectly manipulated. Values returned by I/O calls can be intercepted. However a raw reading of disk, without the view of file system can carry additional information.

The solution to this problem may be intercepting the file IO call in a read-only manner. By using external file system tools raw disk image may be manipulated.

Chapter 3

Stealth Malware Analysis

Observation without disturbing the environment is a must for a successful observation. Since the observed subject can change it's nature, any form of distribution should be avoided. A nature photographer taking pictures of wild animals, may not be able to take *natural* snapshots of the scene. This is the same case with the malware.

Some copy protection libraries also like privacy. They want to execute routines behind the doors. They prefer their action to stay hidden. If they detect a debugger installed, even worse an active debugger, they may politely refuse to execute validation routines. One of the software copy protection routine also asked me to close my Filemon instance. Malware may not only hide its activity. It can also disable protection software. An Agabot variant [9] for example can detect and remove 105 anti-virus processes.

3.1 Virtualization

Increasing capacity on modern computer systems lead to over-qualification for simple tasks. Also each service has possible bugs which may cause them to be exploited. On the other hand with the help of Internet a huge number of

clients are available as a client of a specific service. No specific single hardware is powerful enough to handle this scenario. Virtualization and clustering gained high importance because of this reasons.

Virtualization is gaining significant importance. Especially in server systems. Disabling malware in virtual environments also limits it's spread. So there is a tradeoff. If malware blocks itself under virtualization, it will also self-block it is habitat. Therefore malware authors might not check existence of any sort of VM in future. But if the target of the malware is customer terminals like ordinary desktop PCs, laptops, netbooks detection of virtualization is a good idea.

3.1.1 Virtualization Types

There are many types of virtualization which will be summarized below. Components of a basic virtualization scheme are a virtual machine monitor (VMM), which also can be referred as a hypervisor and a virtual machine (VM). VMMs are installed on host operating systems. VMs are created inside VMMs and guest operating systems are installed in VMs.

3.1.1.1 Emulation

This sort of systems emulates complete hardware, everything needed to run a complete unmodified operating system. Components include CPU, chipsets, VGA cards, disks, keyboards, etc., all components of a real system are emulated. They can emulate different architectures on the platform they are running QEMU[6] and Bochs[17] are examples of this kind.

3.1.1.2 Full Virtualization

Full virtualization systems known as virtual machine monitors (VMMs) emulate components other than CPU. They give us the ability to run various unmodified operating system for the same architecture in the same machine. VMware[30],

QEMU[6], Microsoft Virtual Server [20], Parallels[24] are examples of this kind. As of version 3.0 Xen[2] also supports this feature for x86 platform.

3.1.1.3 Paravirtualization

Instead of emulating a system they provide a special interface for guest systems. Therefore running guest should be modified in order to be executable in that hypervisor. Xen[2] is example of a such system. Linux kernel should be modified in order to boot under Xen environment. UML[7] is also another example.

3.1.1.4 Hardware-supported Virtualization

Emulation done in emulators are done on CPUs this time. Newer CPUs having virtualization support has this feature. In this scenario emulation of CPU is not needed. VMMs are able to execute full unmodified operating systems.

3.1.2 Virtualization Detection

Malware or in case of copy protection case legitimate software, may also detect and refuse to run in virtual environments. There are several virtualization techniques as there are multiple ways to detect if execution is occurring in a virtualized environment. Modern CPUs are quite complex. If they should be emulated without leaving any trail, their bugs should be also modelled. Non-existence of a bug in a certain CPU model clearly gives the being emulated idea.

One of the detection algorithms uses almost one instruction [27]. By execution of SIDT[1] instruction we are able to obtain IDTR[2]. According to obtained value we might obtain presence of virtual machines. There is only one IDTR register and this register is used by OS. If this instruction is executed in a VMM it does

[1]Store Interrupt Descriptor Table
[2]Interrupt Descriptor Table Register

```
int swallow_redpill () {
  unsigned char m[2+4], rpill[] =
                  "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
  *((unsigned*)&rpill[3]) = (unsigned)m;
  ((void(*)())&rpill)();
  return (m[5]>0xd0) ? 1 : 0;
}
```

Figure 3.1: redpill virtualization detection

not generate exception so the process gets relocated address. This does not work with QEMU because, QEMU also emulates CPU.

Another tool to check virtualization is ScoopyNG[15]. It performs 7 tests. Tests are as follows:

- test1: `IDT` test a.k.a redpill test we mentioned above.
- test2: `LDTR`[3] test
- test3: `GDT`[4] test
- test4: Alfredo Andrés Omella's (S21sec) STR technique
- test5: VMware version test
- test6: VMware special I/O port test
- test7: Derek Soeder's (eEye Digital Security) VMware emulation test

QEMU fails on test7 only. Test7 basically generates an exception and then it compares exception address with EndUserModeAddress[5] value.

More research yields [25, 11, 23] more interesting results. Instead of architecture specific patterns for detection they use hardware specific patterns.

[3]Load Descriptor Table Register
[4]Global Descriptor Table
[5](*(UINT_PTR*)0x7FFE02B4)

Ferrie [11] found several unrealistic behavior on QEMU. Mostly these finding are related to `CPUID` instruction. Execution of `CPUID` does not give easter-egg strings embedded into hardware AMD CPUs. A specific example from the paper is instead of "AMD [processor name] Processor", the string"QEMU Virtual CPU version x..x..". is returned.

Raffetseder [25] mostly tested if QEMU behaves like an emulated system by documents from providers. Limitations applied by manufacturers are not emulated by QEMU. Examples include Intel's 15 byte instruction limit. #GP [6] exceptions are not raised. MSR [7]s not implemented or reserved. Also another interesting finding is QEMU or VMware refuse to disable cache. On real hardware cache activation makes a difference, while on QEMU or VMware practically there's no difference between active cache and deactivated cache. They simply ignore the instruction.

3.1.3 More Than Detection

Ormandy [23] did more than detecting a VM. He was able to find coding flaws. A code running in guest machine is able to execute arbitrary code. Coding flaws in VGA emulator, NE2000 Ethernet emulator, socket interface, sound card, DMA controller, CPU emulator, BIOS, IDE controller. Basically most of the emulated peripherals in QEMU (version 0.8.2) had flaws.

Findings of Ormandy means a specially crafted malware could execute code on behalf of the Anubis process. Therefore malware would effect environment outside the emulation system.

[6] General Protection
[7] Model-Specific Registers

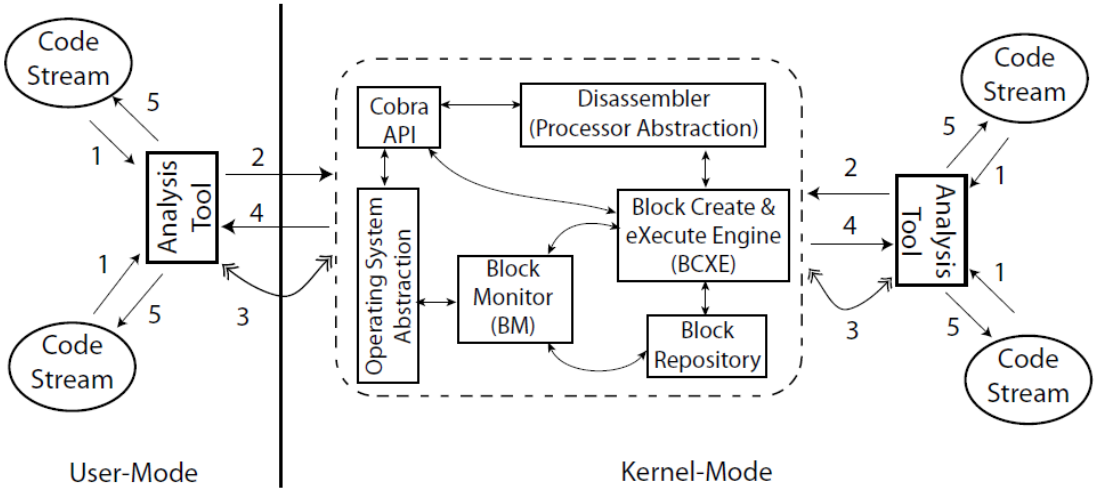

Figure 3.2: Cobra Architecture

3.2 Related Work

There are several framework for stealth malware analysis. They can be considered as related work since their aim is hiding the analysis routines from the malware. But their main focus is not hiding the publicly available, traceable, installation specific finger prints of their environment. They [29, 14, 19] don't hide or alter on the OS instance they're currently running on. Below there is summary of these malware analysis environments.

3.2.1 Cobra

Cobra [29] employs a lightweight hypervisor like component *Block Create and eXecute Engine* (BXCE). Blocks are code streams ending with a jump, or after certain amount of instructions. Architecture is shown in figure 3.2.

Cobra is designed to be running on OS. It has kernel level and user level components. It's similar to the approach in Figure 3.3.

Cobra employs stealth-implants technique. It's simply generating patches for privileged instructions. Instead of executing a privileged instruction

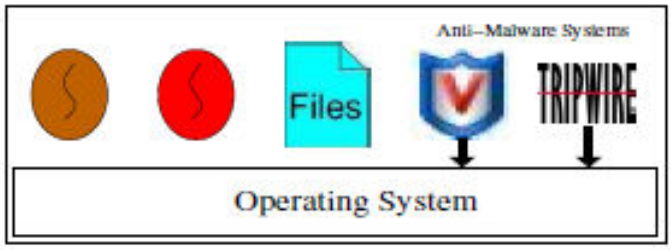

Figure 3.3: Traditional approach

(`SIDT,SGDT,SLDT..`), BXCE creates an implant. An implant is set of substitutions for an instruction. These implants are designed to return values stored in virtual registers. In QEMU this is already done, since QEMU is emulating a complete platform.

Another manipulation done by Cobra is related to the instruction counter. Cobra also maintains a clone register for RDTSC register to falsify amount of clock-cycles.

3.2.2 VMwatcher

WMwatcher [14] is a VMM based malware detection system like Anubis. It's main focus is constructing the semantic view of the guest OS in the host OS. They are trying to create in system like meaningful representation of the guest OS.

In traditional analysis, everything is in the same box. Everything is sharing the same kernel and hardware: Analyzed suspicious binary, files it's using, anti-malware software, analysis tools. This architecture has significant drawbacks. Malware can alter the execution of anti-malware and lead to false-negatives. This architecture is shown in Figure 3.3.

In VMwatcher approach, analysis software is separated from analysis environment. Like Anubis, system observes malware behavior outside the box. But

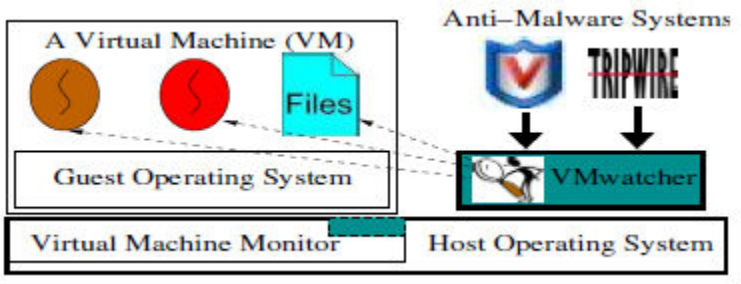

Figure 3.4: VMwatcher approach

the difference is it tries to construct a semantic view, to make *out of the box* deployment of anti-virus software. Their experiment with the Windows version Symantec anti-virus software is to detect malware instances in Linux VM honeypot.

The article [14] also mentioned fingerprinting problem. But instead of pointing single instance of VMs, they talk about risk of using VMs. If malware detects VM, it can shut itself. Increasing popularity of VMs makes decreases the increases the chance of malware not showing it's malicious behavior. Malware wants to create malicious activity.

3.2.3 Malyzer

"The key idea of Malyzer [19] is to unveil malware camouflaging at startup and runtime so that malware behavior can be accurately captured and distinguished."

Malyzer creates a copy of the malicious process that's mutually invisible to original process in the host. Malyzer is not addressed to general public. Therefore it does not need to eliminate it's fingerprint. Anyway anti-detection ideas from Malyzer is used. It's architecture is shown in figure 3.5. It tracks processes using `startup tracker`. Creates shadow process of the suspicious process. Manages shadow process through `shadow process manager`. Finally information is captured `using shadow process manager`.

Two types protection are provided by Malyzer. Internal anti-detection

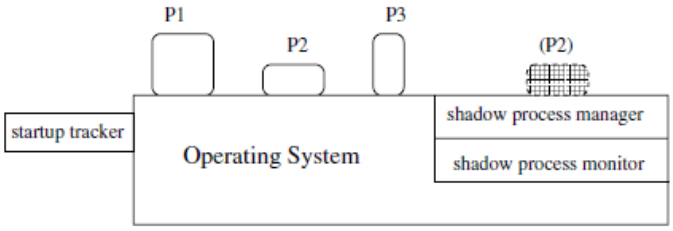

Figure 3.5: Malyzer Architecture, P2 is suspicious process

mechanism is checking only a single instance of malware is running. External anti-malware detection is checking of environment. For example if `trojan downloader` searches for Wireshark, ZoneAlarm and Olly Debug.

The external anti-detection mechanisms have similar properties with what we want to eliminate with eliminating fingerprint. This mechanisms are targeted to larger users, instead of identifying automated analysis systems or honeynets. We're concentrating on systems accessible by public. If this system made available to public, it will suffer from the same fingerprint issue with Anubis.

Malyzer intercepts `Process32Next` API call. Generally process list checking is done by calling `CreateToolhelp32Snapshot` function. Then the process list is enumerated by `Process32Next` function. Therefore overriding Process32Next seems enough to hide running processes. Similarly `ShowWindow` call is intercepted to activate only original process. While shadow process executes this call `nCmdShow` is replaced with SW_HIDE.

Malyzer interceptions are made through Detours Express. [13] Therefore interception can be detected by the malware itself. Detours changes addresses of the Windows API calls to itself. Therefore a self-checking malware might be able to detect the modification.

Chapter 4

Stealth Anubis

Anubis is a public service. It gives automated results established by execution of submitted binary. However this information can reveal information about Anubis itself. The whole work in this thesis is finding and eliminating this type of unique information. Unique information can be a *product id* as an example. Retail license keys are unique for a computer under normal conditions.

During my research I haven't encountered any mechanism designed to overcome this issue. There exists mechanism to hide analysis programs, employed by Anubis, Cobra, VMwatcher and Malyzer. They hide their analysis programs from the running binary. However they are open to fingerprinting techniques discussed.

Mainly two fingerprinting elimination schemes are proposed in this thesis. On-the-fly call interception and off-line image modification. This techniques will be discussed in this chapter with the experiments to extract randomizable data points.

We start This chapter with an overview of known methods to detect Anubis fingerprint. This is followed by additional information about known methods are given. Then we discuess the procedure to extract more fingerprint information and the results. Finally we present the methods proposed to eliminate these

#	Observer Comparison with	Number of Samples	Number of Clusters
1	Windows Product Id of Anubis	55	28
2	Windows Product Id of CWSandbox	32	14
3	Windows Product Id of Joebox	32	14
4	Executable name of sample.exe	35	17
5	Computer name of Anubis	4	4
6	Qemu's HD name	2	2
7	VMWare's HD name	1	1
8	Windows user name of 'user'	2	2
A	Any Anti-Anubis comparison	99	54
B	Any Anti-Sandbox comparison	100	55

Table 4.1: Anti-Sandbox detection mechanism encountered in sample database fingerprints.

4.1 Known Sandbox Detection Mechanisms

As a starting point we might investigate work of Bayer et al. [4]. A study conducted on 901,294 malware samples which is all samples submitted to Anubis between February 7th 2007 and January 14th 2009. This malware samples are unique according to their MD5 hashes.

Table 4.1 shows know anti-sandbox mechanisms. Some of the mechanisms employed are in the category of detecting system emulators. Some other are specifically designed to detect a specific sandbox. As compared to whole submissions ratio is only 0.03

Mechanism 1, 2 and 3 work by querying a simple registry location.

```
HKLM\Software\Microsoft\Windows NT\Current Version\ProductID
```

This location provides `ProductID` for active Windows XP installation. Details related to internals of the relation of this id with the product key used to install Windows XP is discussed in [12]. Simple summary regarding ProductID is below.

Format of a product key is in the form of *AAAAA-BBB-CCCCCCC-DDEEE*. *AAAAA* part is nearly identical for a provider in Windows XP. Part BBB is part of raw product key or string `OEM` for OEM installations. CCCCCCC is a part of raw product key. DD is public key index and EEE is just a random value. Minimalist approach is only changing EEE part. Changing both last 4 digits of CCCCCCC and all digits of EEE will probably both preserve string format and keep id validatable. However performing additional ProductID format checks is beyond the scope this thesis.

Mechanism 4 checks if the name of executed binary matches with `sample.exe` string. This check is done with `GetModuleFileName` function. This function has 3 parameters. If it's called with first parameter having NULL value, process returns the location of the binary that is calling the function. It's easy to override this function but if additional checks are made with file name such as controlling if file exists, former checks would fail. This check does not work with Anubis. Anubis can execute any file name. This check exists to check former version.

Mechanism 5 checks the name of ComputerName. The name for Anubis is `user`. This is obtained by `GetComputerName` method. This method internally reads associated registry key. Manipulation can be done in both registry record, or by intercepting the function.

Mechanism 6,7 determine the name associated with emulated disks. For QEMU it is QEMU. Since it is sort of detection for QEMU it outside our scope. Because there exists more advanced probing techniques for virtual machine detection.

Mechanism 8 checks the name of process owner. In old versions of Anubis it was `user` later it has been changed to `Administrator` which is standard administrator account that exists in every Windows XP operating system.

4.2 Possible Detection Points

`HKLM\Software\Microsoft\Windows NT\Current Version`

Location contains additional information than queried with known detection mechanisms. Additional fields include

- *DigitalProductId* binary coded product id with additional information.
- *InstallDate* DWORD value storing data of installation
- *LicenseInfo* binary license information

Modification of these values would make Windows XP activation scheme to deactivate Windows. That would lead to blocking of Windows XP.

4.3 Differences Between Installations

For achieving successful results we should now what kind of differences exist between various installations. This differences can be taken while:

- When installation is done on different times
- When installation is done with different product key
- When additional software is installed
- When different programs are executed

Analyzing this snapshots can lead us to possible fingerprints.

4.4 Defining Differences / Finding Fingerprints

We're trying to provide a framework for randomization. A randomizable resource, randomization quantity and method. And we'll integrate it into Anubis to eliminate signature. This section discusses the methodology used to find differences between two installations. We will try to install identical OSes to identical virtual hardware. The only initial difference will be serial numbers. And we will try to obtain differences.

For eliminating the fingerprint of an installation we should have a destination. Differences, which can be considered as fingerprint should be clearly specified. For this purpose we'll try to generate identical installations on hardware with identical serial numbers. We observed the differences between these installations.

Anubis launches suspect binaries with the same snapshot. This makes analysis quite fast. This method has the disadvantage of having the same memory image everywhere.

A partial solution to this problem might be defining randomizable bytes in the snapshot image. If we are able to find such magical bytes in the image, we can randomize VM before loading snapshot. This will lead to dynamically fingerprinted snapshots. This approach would perfectly work if the given conditions are satisfied:

- Positioning of data: If data is uncompressed and stored using without any modification, it is clearly accessible. For example, modification of a string is easy if we can clearly find in raw disk image using a normal search operation, we can change raw disk image. But if it's compressed or divided, we should look from the higher API. We can execute grep function in files of file system. By using this approach we can override a complete image file for example.
- Preserving consistency: If any single bit we change on the image should match within data in another location of the snapshot, we loose consistency.

Algorithm	MiB / Second	Cycles Per Byte	Microseconds to Setup	Cycles to Setup
VMAC(AES)-64	1519	1.1	3.133	5734
VMAC(AES)-128	779	2.2	3.738	6841
CRC32	253	6.9	-	-
Adler32	920	1.9	-	-
MD5	255	6.8	-	-
SHA-1	153	11.4	-	-
SHA-256	111	15.8	-	-
SHA-512	99	17.7	-	-
Tiger	214	8.1	-	-

Table 4.2: Selected Hash Function Benchmarks

This might lead to malfunctioning system. Also changes made in license fields may invalidate installation.

For file comparison we need a fast hash function. Speed is more important than security on this project. A comparison of functions from cryptopp[1] leads to use of VMAC (AES) - 64[16] it has nearly 6 times better performance than MD5. Table 4.2 provides brief summary of benchmark.

However simple code implemented to utilize the library had worse performance than md5sum utility on a Debian system. Figures for a 650M image 2.7s average for MD5, 3.3s average for VMAC. Due to this reason we prefer using md5sum.

4.4.1 Virtual Machine Setup

For performing our comparisons and experiments three Windows XP Professional instances are installed. Virtual machines, and in case of QEMU only virtual disks are need to be created first. [18]

First and second installations share the same product key. Second and third share same `ComputerName` and `Organization`.

We start by creating persistent storage units. 3 qcow2 disk images are created

[1] `http://www.cryptopp.com/benchmarks.html`, retrieved on 21.07.2009

with command:

```
$ for i in {0..2}; do \
qemu-img create -f qcow2 img$i.qcow2 4G; \
done
```

The produced image files have size of 28672Bytes and md5sum of `65287344ab282042a3f9cafd30b37f5d`. Files are totally identical in this stage. qcow2 format supports expanding. It does not have to be initially 4GB to support a 4GB file system.

Then we used Microsoft Windows XP Professional image sp3 integrated with hash of `f424a52153e6e5ed4c0d44235cf545d5`.

Installation is done in virtual machine with 256MB of memory with command:

```
$ qemu -m 256 -cdrom \
en_windows_xp_professional_with_service_pack_3_x86_cd_x14-80428.iso \
-hda img0.qcow2 -boot d
```

Everything is left as default. Installation is done with user names, `user0, user1, user2`. Organization and owner names are specified as `SETUP_NAME, SET_NAME1, SET_NAME2`. Timezone is set to Turkey's timezone. ComputerName is left as proposed by installation application. Formatting is used as NTFS.After the installation procedure, we start analyzing file systems and registry contents.

4.4.2 File Comparison

We compared information obtained from installed system. Also we extracted parameters including time stamps and MD5 hashes. For each file disk image we created a flat CSV [2] file consisting of, file type (file or directory), md5hash,

[2]Comma separated values, Comma delimited

Directory	0 vs 1	0 vs 2	1 vs 2
WINDOWS	857	873	888
Documents and Settings	251	250	248
Program Files	2	2	2
System Volume Restore	51	59	59
Total	1161	1185	1197

Table 4.3: Number of different for files in nearly identical systems.

file list	atime	mtime	ctime
list0	3	16	2
list1	2	15	1
list2	2	15	1
list3	23	722	6
list4	58	602	118

Table 4.4: Comparison of unique dates (in days) in a system.

atime [3], mtime [4], ctime [5] and file name. Figures are extracted while file system is mounted as read-only using ntfs-3g driver for Linux.

Table 4.4 summarizes unique dates found in different sets. list0, list1, list2 correspond to fresh installs mentioned above. list3 set is extracted from Anubis image and finally list4 is extracted from a random PC. According to the result we can clearly conclude that all machines except 4th were running for only few days. Table also shows us the image creation process for Anubis took 6 different days. On the other hand 4th machine has been running for at least 118 days.

Surprisingly nearly 10% of files found to have different hashes in fresh nearly identical installations. Number of different files are shown in table 4.3. Most common extensions of different files are as follows: 56 log, 90 lnk, 31 pf, 692 PNF, 21 dat, 15 LOG. Locations of these files are in table 4.3.

Figures clearly conclude that a system that's being in use has values of ctime's spread across the dates. Figure 4.4 is a clear example. Limited usage of Anubis

[3]time of most recent access
[4]most recent modification
[5]time of creation

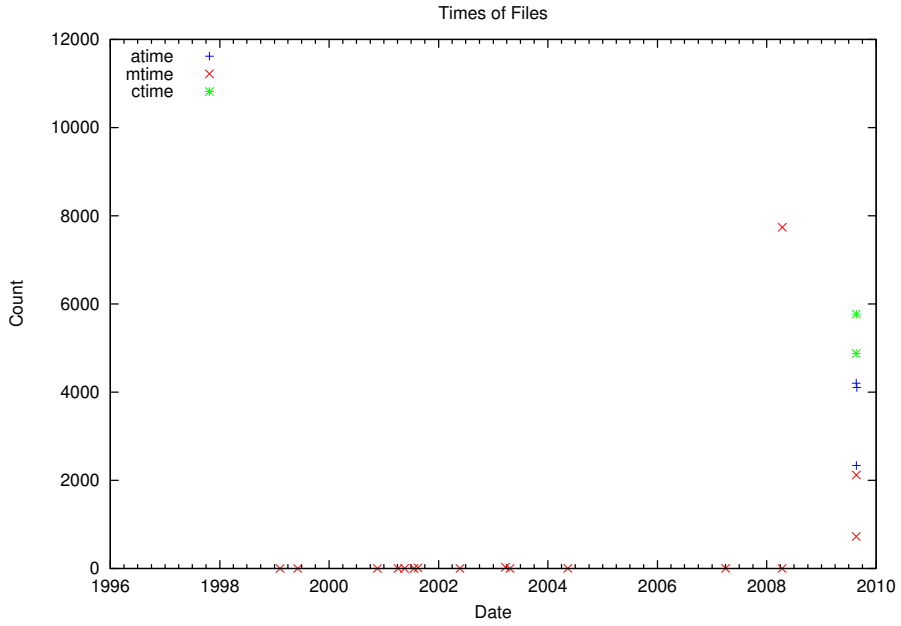

Figure 4.1: Dates in a new Installation 1

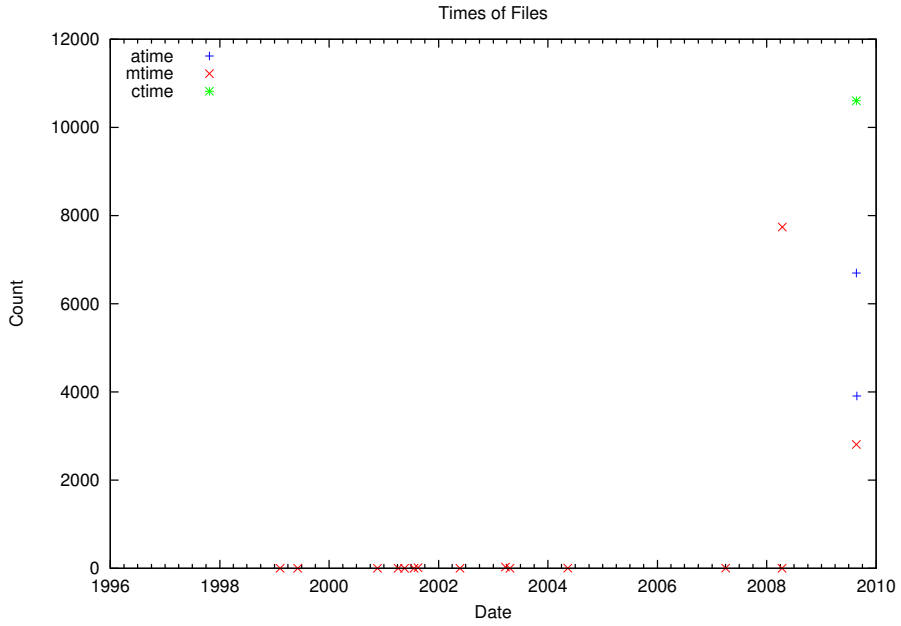

Figure 4.2: Dates in a new Installation 2

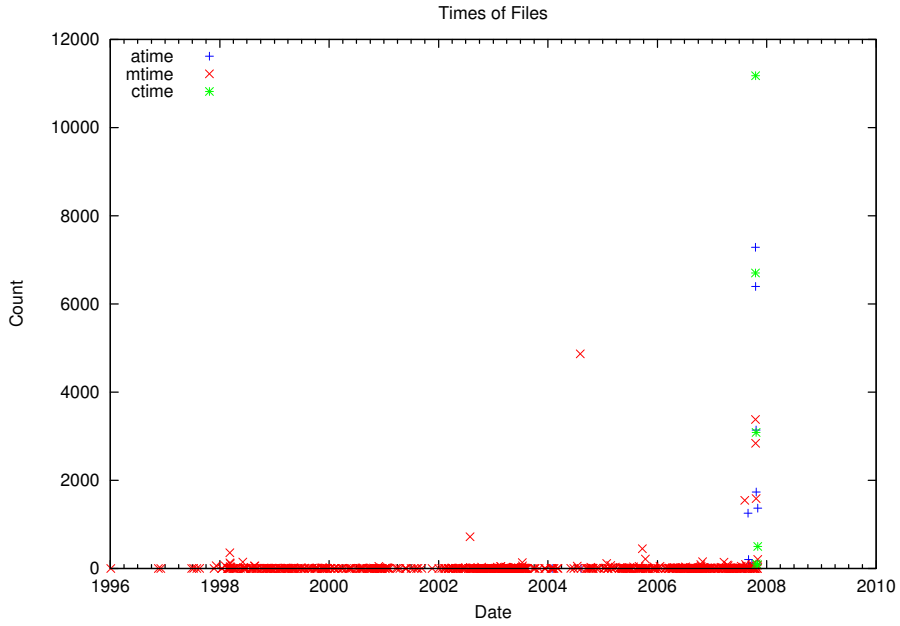

Figure 4.3: Dates in current Anubis image

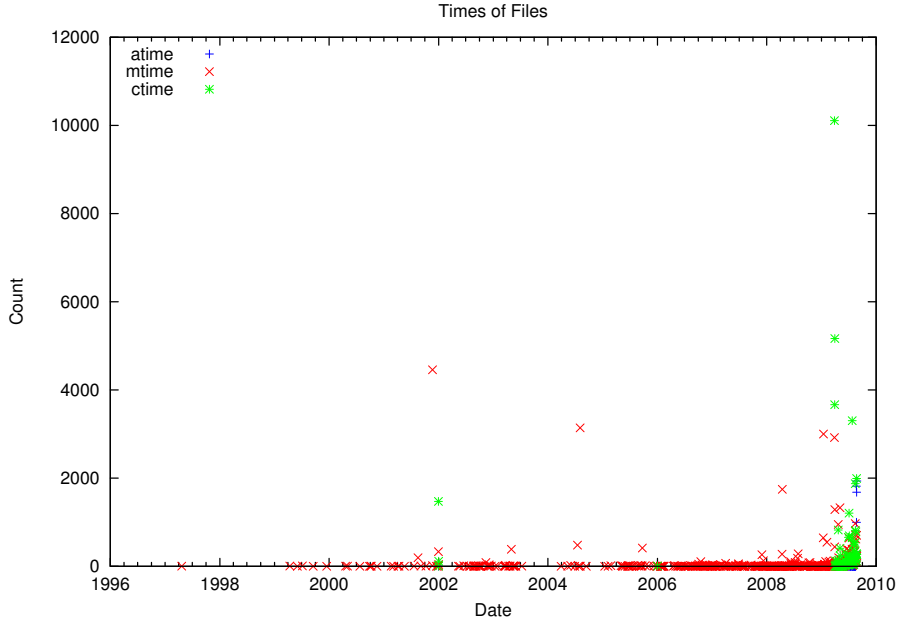

Figure 4.4: Dates from a running system

image is shown in figure 4.3, while fresh installed systems has nearly all files created in same day. Histograms of clearly installed systems are in Figures 4.1 and 4.2.

Any file can be used as a fingerprint for the Anubis system. Access times cannot be trusted since they can easily change, but a specific file with a specific date is a strong fingerprint. Also spectrum of the dates is another fingerprint. If a system has only 6 unique modification or creation days one may conclude that system is not actively used. And even if malware alters the system configuration malware can simply ignore to run on this system.

Particular solution to file system dates may be modifying dates of file in a random manner. However this would lead to long run times. We can randomize file system dates by intercepting file related functions including `FindFirstFile, FindFirstFileEx, GetFileAttributes, GetFileAttributesEx`. Further details are discussed in Section 4.5.4.

4.4.3 Registry Comparison

Windows Registry stores vast amount of system information. For comparison of registry, system wide Windows Registry has been dumped with [28] to a CSV file. Since nearly every application modifies registry, it is not preferable to compare registry dumps of two running machines. Instead we focus on the differences just after the initial installation. Therefore only registries of 3 generated virtual machines img0, img1, img2 are compared.

After detection of unique registry values for these 3 images, these values are compared with img3. The keys found are different in such a small set. We can clearly conclude that the key values found in Anubis image will likely to be strong fingerprint.

However we cannot change these values in a random manner. For example the case with ProductID defined in Section4.1 has strict formatting rules.

Unique values can have various forms. We started with keys. Registry keys are similar to folders of file system. Several forms of differences have been observed. Comparisons are done between img0 and img2.

Example 1:

```
/Software/Microsoft/NetDDE/DDE Trusted Shares/
D0178F2AB - D0F490750
```

Key in the chain has possible form of `D || a random DWORD`

Example2:

```
/Software/Microsoft/Windows/CurrentVersion
/Internet Settings/5.0/Cache/Extensible Cache/

MSHist012009082120090822  -  MSHist012009082220090823
```

Example in this key has time information encoded.

Example3:

```
/ControlSet001/Control/DeviceClasses/{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}/

##?#STORAGE#Volume#1&30a96598&0&Signature22F722F6Offset7E00 \
LengthFF6D1400#{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}

##?#STORAGE#Volume#1&30a96598&0&Signature32143214Offset7E00 \
LengthFF6D1400#{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}
```

String after Signature is a random DWORD.

Example 4:

```
/ControlSet001/Control/DeviceClasses/{ad498944-762f-11d0-8dcb-00c04fc3358c}
/##?#PCI#VEN_10EC&DEV_8029&SUBSYS_11001AF4&REV_00#3&13c0b0c5&0&18
#{ad498944-762f-11d0-8dcb-00c04fc3358c}/

#{70C404A4-E712-467A-9740-318247AA21A2}
#{A6E2D95A-74E9-428D-8F89-214E8BABCC24}
```

Observed pattern here is actually universally unique identifier. Which has 2^{128} bits. Details are provided in Section 4.5.

4.5 Proposed Solution: On-The-Fly Fingerprint elimination

In a specific time instance of operating system's execution, there are many points that can be considered as a fingerprint. Even after complete reboot of the operating system, there are custom points remain same, and unique to that installation only.

The idea of eliminating fingerprint depends on simple randomization. By applying substitution or XOR'ing a bit mask to unique values, without reinstalling the complete operating system, or restarting it we might have a system behaving like another one.

Every I/O routine may be used for fingerprint extraction. E.g serial numbers for hardware (BIOS, CPU, Memory), installation specific values, Ethernet MAC id. An installation specific value can be created in various ways. A randomly generated 128 bit string has 3.4 x 10^{38} possible values. Which means if we can create 1 trillion UUIDs every nanosecond, it will take a little more than 10 billion years to consume all possible UUIDs. If the seeds used to feed algorithm is chosen carefully while generating UUID, even one instance can be a strong fingerprint.

While changing the signature preserving consistency is important. The change

in data must be minimal, while satisfying the consistency. There is a tradeoff between the number of bits changed, and the established randomness.

4.5.1 Pseudo Randomness

System unique points should be randomized, but in order to track this randomization procedure should be carefully recorded and for every source it should return the same unique output. If any binary reads a static registry record, it should return the same values all the time. Reading of a constant source should always return the same results. If the binary modifies the value, binary should obtain the same results.

For generating such deterministic random bits (rb) we can use hash functions.

$$rb = HASH(secret|HASH(binary)|unique\_resource\_id) \tag{4.1}$$

This function helps us to achieve easily traceable randomness. We can determine random bits needed to randomize the string.

- *secret* is an organization wide secret
- $HASH(binary)$ is hash of the binary being analyzed
- *unique_resource_id* is a value that is unique for a given query. For a registry value, it can be full key path.

For every different value type, we should plan randomization carefully. Plain usage of rb is not enough, and probably will lead to a malfunctioning system.

4.5.2 String Randomizer

For any resource randomizer component works the following way. It reads the input, applies randomization mask described in table 4.5. The length of the mask

Code	Meaning
.	preserve character
H	place a random hex-digit (upper-case)
h	place a random hex-digit (lower-case)
D	place a digit
A	place a random letter [A-Z]
a	place a random letter [a-z]
1	XOR 1 random bit
2	XOR 2 random bit
3	XOR 3 random bit
4	XOR 4 random bit
5	XOR 5 random bit
6	XOR 6 random bit

Table 4.5: Value randomization table

and value should be same. As a parameter randomizer takes random bits from calculated *rb* with the help of registry or file randomizer classes.

4.5.3 Anubis Call Interception

Basic information of call interception is given in section 2.3.2.2. Detours[13] like call manipulation is implemented in functions regarding Windows Registry operations, Windows file system functions and few other functions. Implemented manipulation works as follows

Before a Windows API function is called, Anubis redirects this call into `...WasCalled` method. This function is the point where we can modify parameters going into Microsoft Windows implementation of routines. For example if it's a file open function called with parameter `fileA`, and we change the parameter to `fileB` in this stage, calling binary will think it's opening `fileA`, but it is going to open `fileB`. The size of the strings should be identical to overcome issues regarding memory allocation of the guest OS.

After Microsoft Windows implemented execution of the specific kernel function `...HasReturned` is executed. If we continue with previous example, we can

restore value `fileB` to `fileA` again.

If it's a function that's only filling a buffer, then we only need to intercept `...HasReturned` function. This case will lead to execution of the function perfectly by the guest OS. However, returned value will be modified by our mechanisms. This would perfectly work, in places where the associated value is not vital, or can be used as a parameter. For example ProductID is not associated with any other file, or it's not a key for another registry value.

If any manipulated value, is a key for another value this scenario should be handled accordingly. We cannot simply randomize all values. For example, network adapters in Windows are stored with keys in the form of UUID. This values, later used by other functions to access corresponding network adapter. Same rules apply for the system users and groups. We may not simply alter these values randomly, or obeying a specific format.

UUIDs or UUID like resources stored in registry and have the role of being a key to another resource should be tracked by a separate unit. They should be randomized, and when they're used as a parameter they should return into their original value. Another point to consider is obeying formatting rules. We should conserve properties of the original value.

Finally these manipulation is only done if the calling binary is under analysis. Normal Windows routines do not interfere with randomized behavior of native functions.

4.5.4 File Dates

Figures regarding dates defined section 4.4.2 should be scattered over a time space in Anubis. Modification dates of the files and creation dates should be randomized.

We can make changes regarding installation date, and a total randomization.

Date	Number of Files created
2007-11-01	82
2007-11-02	500
2007-10-22	3086
2007-10-23	105
2007-10-18	6704
2007-10-19	11177

Table 4.6: Creation dates in Anubis

Installation date can be calculated using creation times of files in `WINDOWS` directory. They can be shifted.

For example, from the table 4.6 we can clearly conclude installation is started on 2007-10-18. We can use this as a base for our randomization. Adding a specific amount of time we can shift installation date. However this will preserve time delta between files. Which is another fingerprint. A solution to this problem can be defining two time deltas. Δ_{days} defines shift in installation date in days and $\Delta_{seconds}$ defines shift in seconds. Resulting new file date. rb in equation 4.3 was defined in 4.1.

$$\Delta_{seconds} = \frac{HASH(filename)}{HASH_{MAX}} * MAX\_SECONDS \tag{4.2}$$

$$\Delta_{days} = \frac{rb}{HASH_{MAX}} * (DATE_{Current} - DATE_{Initial}) \tag{4.3}$$

$$time_{file} = \Delta_{days} + \Delta_{seconds} + DATE_{Initial} \tag{4.4}$$

Application of these methods over the ones defined in section 4.5.3 can be summarized as follows.

File operations are done with `HANDLES` in Windows operating systems. File name is associated with handles, which are 32bit words. Handles to files are obtained using `NtCreateFile` function. Later versions of Windows has additional `NtCreateFileTransacted` function also.

Invoking this function registers handle on the `DateRandomizer`. Queries regarding file information are done with `GetFileInformationByHandle` and

Id	Key	Mask
1	HKLM\Software\...xxx\Randomizable_Key1	...DDD
2	HKLM\Software\...xxx\Randomizable_Key2	HH.3

Table 4.7: Sample randomizable registry keys and randomization masks.

`GetFileInformationByHandleEx`. Functions. By creating a `GetFileAttributesHasReturned` function we can randomize date information stored in `LPBY_HANDLE_FILE_INFORMATION` structure.

However, if a setter method is invoked on active file handle, we can safely drop the file from the list of `DateRandomizer`.

4.5.5 Windows Registry

Very similarly to file system functions, registry functions work on handle basis. This time handles are created using `NtCreateKey` `NtOpenKey` or `NtOpenKeyEx` calls. While obtaining handles, key names are checked against randomization list. If key is marked as randomizable, method defined in section 4.5.3 can be applied to value obtaining functions in registry.

Functions to be intercepted in registry are `NtQueryKey` and `NtQueryValueKey`. While keys are known to store values only close look at the Registry API lead to storage of modification date associated with the key. So date randomization algorithm also can be applied to registry keys defined in equation 4.4.

`NtQueryKey` on the other hand actually retrieves data. The information retrieved by this function should be randomized according to it's data type. This marker is one of `KEY_VALUE_INFORMATION_CLASS`'s members. Which are defined in 2.1. Attention should be paid to only changing successful calls. Key corresponds to $unique\_resource\_identifier$.

Additions we propose to registry system is as follows. We collect randomizable entries in database. Key and value names are combined in a single string with randomization masks.

Below there are some proposals for randomization regarding value types.

4.5.5.1 String, Multi-String

Substation or randomization of last 4 bits may be used. ASCII characters are is 7bit. We don't want to change the type of character. $A \rightarrow B$ can be pretty normal but $3 \rightarrow A$ conversation may cause to failures. Since an integer might be saved as ASCII in it's decimal notation. According to this scenario this would also cause for hex base representations. Possible mappings may include ($[0-F] \rightarrow [0-F], [0-9] \rightarrow [0-9]$). For every byte we must design a byte class preserving mask. For multi-string same rules might be applied.

4.5.5.2 Unicode String

Same rules as string may be applied. Unicode strings may have double bytes for defining a single character. Since we don't want to change family of characters we may alter only last byte.

4.5.5.3 DWORD, QWORD

Changing last 4 bits may be an option. Position of bits may depends whether it's little endian (REG_DWORD, REG_DWORD_LITTLE_ENDIAN) or it's big endian (REG_DWORD_BIG_ENDIAN). For QWORD we can also use this.

4.5.5.4 Blob & Binary Data

Changing last bit of every byte may be an option. But possibly there exists formatting rules for any sort of blob.

4.5.6 Network

QEMU has embedded QEMU MAC hard coded is `52-54-00-12-34-56` router MAC. Since it's QEMU specific and hard coded it can only be used as a fingerprint of QEMU. But the MAC address of router or computers can be used to track environment. However MAC addresses can be instantly changed. If guest ARP cache is flushed before analysis, there will be no trace of MAC addresses.

Anubis VM can be set to re-obtain MAC address from a DHCP server. This way we eliminate IP level blocking. However, if any connection to outside world is permitted network paths may be blacklisted by malware authors.

4.5.7 File Randomizer

A good idea is randomizing file blocks also. It's not implemented but for changing hashes of files, few bytes can be changed. This can be done with the same call interception approach. Instead of using a pattern, we may define offsets for randomizable data points. Even a single bit change will change file hash.

This assumption is based on malware being not able to verify file contents. It's clearly unfeasible for us to detect such points. But a generic construct that changes few bytes will resolve a big problem.

Implementation of this can be done by intercepting `NtFileRead` function. Getting file handle is by hooking of `NtCreateFile` or `NtOpenFile`. We can check if the currently read block is containing any randomizable byte. And apply a randomization policy to a specific byte sequence.

Chapter 5

Experiments & Evaluation

While experimenting and evaluating our work, we used two malware samples captured by Anubis installation hosted by International Secure Systems Lab. The samples are known to detect Anubis.

The procedure followed is as follows: First we analyzed the samples with Anubis and generated initial reports. After the initial analysis we used reverse engineering techniques for static analysis. Procedure steps include extracting strings from samples, extracting imported functions, disassembling samples, determining detection policies and re-analyzing samples with modified Anubis.

Main tools used for static analysis are objdump from mingw32 port of binutils package, and IDA-Free. These tools were previously introduced in Section 2.2. Objdump is used is used to display information from object files. Including headers, section headers, disassembled code, relocation entries. Objdump as compared to nm gives more possibilities and information about the binary. IDA has a nice GUI and a better disassembly engine. IDA also makes navigation through the assembly code of the binary. It has visual aids to support navigation.

Windows binaries are in PE format. This format consists of several headers and sections. For example `.text` holds executable code. Another section is `IAT` where the information of imported functions from external DLLs are stored.

Offsets of entry points and function names are stored in these tables. These sections can be viewed with either IDA or objdump.

For searching strings in binary `strings` program is used. It searches and prints minimum of 4 consecutive printable characters in a file.

For obtaining information from binary files we used objdump. To see the imports we invoked binary as follows.

```
$ i586-mingw32msvc-objdump -p sample1.exe
```

Generated import lists for `sample1.exe` are in Table 5.1, Table 5.2, Table 5.3. `sample2.exe` has imports from a single file only which is shown in Table5.4.

5.1 Sample 1

Sample execution is clearly terminated in Anubis. The output to console is `Anubis detected...` This output clearly indicates the detection of Anubis.

Hash information for sample1 is as follows:

```
MD5: 722ec327d827ca34f51b7073ca7c23c2
SHA1: 4c6acc804e89edfb0053a5a105a41617f2a9ddfe
```

Imported function list at Table 5.1, Table 5.2 and Table 5.3 gives us clues. For example there's an enumeration of processes [1], a window search [2], obtaining username [3], and time related operations [4]

Additionally and more surprisingly we see 25 functions having names with `Is` prefix in table 5.5. Simple execution of strings[5] command reveals few strings

[1] Process32First, Process32Next functions are used to enumerate process list
[2] FindWindowA
[3] GetUserNameA
[4] GetTickCount
[5] member of binutils package to extract strings from a given file

vma	Hint/Ord	Member-Name Bound-To
4e408	9	AllocConsole
4e418	53	CopyFileA
4e424	87	CreateSemaphoreA
4e438	93	CreateToolhelp32Snapshot
4e454	155	ExitProcess
4e464	175	FindAtomA
4e470	213	FreeLibrary
4e480	220	GetAtomNameA
4e490	281	GetCurrentProcess
4e4a4	323	GetLastError
4e4b4	333	GetModuleFileNameA
4e4cc	335	GetModuleHandleA
4e4e0	362	GetProcAddress
4e4f4	421	GetTickCount
4e504	494	InterlockedDecrement
4e51c	498	InterlockedIncrement
4e534	522	LoadLibraryA
4e544	583	Process32First
4e558	585	Process32Next
4e568	617	ReadProcessMemory
4e57c	626	ReleaseSemaphore
4e590	709	SetLastError
4e5a0	735	SetUnhandledExceptionFilter
4e5c0	747	Sleep
4e5c8	759	TlsAlloc
4e5d4	760	TlsFree
4e5e0	761	TlsGetValue
4e5f0	762	TlsSetValue
4e600	805	WaitForSingleObject

Table 5.1: Imports of sample1.exe from kernel32.dll

vma	Hint/Ord	Member-Name Bound-To
4e3c8	245	GetUserNameA
4e3d8	389	RegCreateKeyA
4e3e8	432	RegSetValueExA

Table 5.2: Imports of sample1.exe from advapi32.dll

vma	Hint/Ord	Member-Name Bound-To
4e8e8	209	FindWindowA
4e8f8	222	GetAsyncKeyState
4e90c	577	ShowWindow

Table 5.3: Imports of sample1.exe from user32.dll

containing `detected`. Full list is: `Anubis, Threat Expert, JoeBox, Norman, Wireshark, Kaspersky, iDEFENSE sysAnalyzer, Sunbelt, Sandboxie, Virtual PC, Virtual Box, WPE Pro` and `some others`.

This instance of malware simply detects, defense systems, detection systems, virtualization systems and honey net systems. Since our focus is on Anubis stealth execution we'll focus on Anubis detection methods used by this malware instance.

`IsAnubis` function entry is located at offset `0x00401662` of text segment. It performs several checks for Anubis installation fingerprints. If any of the symptoms matches then function is terminated, and binary stops execution. But for authors of this malware preferred to print `Anubis Detected...` to stdout.

Malware first checks whether the binary is running inside the directory `C:\InsideTM\`. This is comes from the design of TTanalyze. This check returns false since Anubis does not need and use this folder.

The second check used for detection by this malware is file name. Also this check belongs to historical reasons. Initial TTanalyze design was capable of executing files with name `sample.exe` since it's not mandatory with new version this check is also useless.

Third control is current username. `GetUserNameA` function is called to retrieve name of the user executing current process in guest operating system. Default Anubis installation executes suspect binaries with user account named *"user"*. In Anubis architecture we can achieve overwriting this value by manipulating `GetUserNameAHasReturned` function Code can be seen in Figure 5.1. This function is not implemented normally, we extend our `Randomizer` class to implement `GetUserNameAInterface`.

vma	Hint/Ord	Member-Name Bound-To
65a0	276	GetComputerNameA
65c2	272	GetCommandLineA
65d4	488	GetVersion
65e2	185	ExitProcess
65f0	862	TerminateProcess
6604	322	GetCurrentProcess
6618	878	UnhandledExceptionFilter
6634	381	GetModuleFileNameA
664a	246	FreeEnvironmentStringsA
6664	247	FreeEnvironmentStringsW
667e	916	WideCharToMultiByte
6694	341	GetEnvironmentStrings
66ac	343	GetEnvironmentStringsW
66c6	804	SetHandleCount
66d8	441	GetStdHandle
66e8	358	GetFileType
66f6	439	GetStartupInfoA
6708	383	GetModuleHandleA
671c	344	GetEnvironmentVariableA
6736	489	GetVersionExA
6746	532	HeapDestroy
6754	530	HeapCreate
6762	899	VirtualFree
6770	534	HeapFree
677c	727	RtlUnwind
6788	932	WriteFile
6794	528	HeapAlloc
67a0	260	GetCPInfo
67ac	253	GetACP
67b6	403	GetOEMCP
67c2	897	VirtualAlloc
67d2	538	HeapReAlloc
67e0	416	GetProcAddress
67f2	594	LoadLibraryA
6802	369	GetLastError
6812	238	FlushFileBuffers
6826	795	SetFilePointer
6838	629	MultiByteToWideChar
684e	580	LCMapStringA
685e	581	LCMapStringW
686e	442	GetStringTypeA
6880	445	GetStringTypeW
6892	823	SetStdHandle
68a2	52	CloseHandle

Table 5.4: Imports of sample2.exe from kernel32.dll

position	name
004014b2	Z10IsUsernameSs
0040455c	Z11IsAutostartPKc
00401614	Z11IsFileExistSs
004021dc	Z11IsKasperskyv
004028ac	Z11IsSandboxiev
004020ae	Z11IsWireSharkv
0040156e	Z13IsFolderExistSs
0040150c	Z14IsFileInFolderPKc
004013e8	Z16IsProcessRunningSs
004015a0	Z19IsFileNameEqualThisSs
0040230a	Z4IsIDv
00401b86	Z4IsJBv
0040192a	Z4IsTEv
00402a48	Z4IsVBv
00402d02	Z5IsAllv
004029f4	Z5IsEmuv
00402ff2	Z5IsLogv
004028e4	Z5IsVPCv
00402b76	Z5IsWPEv
00402974	Z7IsOtherv
00401662	Z8IsAnubisv
00401e1a	Z8IsNormanv
00401a58	Z9IsSandboxv
004025e6	Z9IsSunbeltv

Table 5.5: Functions with possible analysis names.

```
target::GetUserNameADecoder *dec =
    dynamic_cast<target::GetUserNameADecoder*>(
        fCoordinator->getCurrentDecoderObject());
target::VirtualAddress va;
va = dec->getlpBuffer_addr();
uint8_t *tPtr1 = (uint8_t *)"test";
vSys->writeMemory(va, 4, tPtr1);
```

Figure 5.1: Manipulation of GetUserNameA function

This code block in Figure 5.1 returns `test` instead of `user`. For sample1 manipulating only `GetUserNameA` function defeats Anubis detection mechanism.

Initially the activities tag in the analysis report is empty:

```
<activities>    </activities>
```

After modification report contains following entries:

```
<activities>

<registry_activities>
<reg_value_modified count="1" description="auto_start"
   key="HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run"
   value_data="C:\WINDOWS\z_antivir.exe" value_name="z_antivir"/>
</registry_activities>

<file_activities>
<file_created name="C:\WINDOWS\z_antivir.exe"/>
<file_created name="C:\WINDOWS\z_antivir.log"/>
<file_read name="C:\WINDOWS\z_antivir.log"/>
<file_read name="C:\sample1.exe"/>
</file_activities>

<misc_activities>
<mutex_created name="CTF.TimListCache.FMPDefaultS-1-5-21-
   1229272821-1004336348-527237240-1003MUTEX.
   DefaultS-1-5-21-1229272821-1004336348-527237240-1003"/>
<key_was_checked count="7166" key="VK_BACK (8)"/>
<key_was_checked count="7166" key="VK_TAB (9)"/>
<key_was_checked count="7166" key="undefined (10)"/>
<key_was_checked count="7166" key="undefined (11)"/>
<key_was_checked count="7166" key="VK_CLEAR (12)"/>
<key_was_checked count="7166" key="VK_RETURN (13)"/>
```

```
Computername USER
Honeypot [ANUBIS]
```

Figure 5.2: Initial stdout output of sample2

```
<key_was_checked count="7166" key="undefined (14)"/>
<key_was_checked count="7165" key="VK_MULTIPLY (106)"/>
⋮
</misc_activities>
</activities>
```

Full report is available in Appendix.

5.2 Sample 2

Initial output can be seen in Figure 5.2. This output indicates detection of Anubis. Analysis report also shows us binary just reads a single registry key. Which holds ComputerName [6].

File hash information is as follows

```
MD5: ada52b41dab750dab2d057b7f2a2117d
SHA1: 9fd6089a6f63a607770945ebf1d85b656d344bf1
```

Imported functions from kernel32 are shown in Table 5.4. If we associate what we obtained from stdout output with the imported calls, we may agree that `GetComputerNameA` function is called to obtain computer name for the Anubis. This also gives us the clue that `GetComputerNameA` function uses native API calls to extract computer name from Windows Registry.

[6] HKLM\System\CurrentControlSet\Control\ComputerName\ActiveComputerName

String
not Honeypot
Honeypot [ANUBIS]
USER
Honeypot [THREAT EXPERT]
COMPUTERNAME
Computername %s

Table 5.6: Interesting strings in sample2

Extracted strings are seen in Table 5.6. We see that Binary is able to detect `Threat Expert`[7] and `Anubis`. It achieves this task by using ComputerName. From the reversed code we can say that ThreatExpert system uses `COMPUTERNAME` as computer name and Anubis uses `USER`. However the accuracy of comparing the name of the computer with a string is questionable. Full disassembly of `main` function is in Appendix.

[7] "ThreatExpert (patent pending) is an advanced automated threat analysis system (ATAS) designed to analyze and report the behavior of computer viruses, worms, trojans, ad-ware, spy-ware, and other security-related risks in a fully automated mode." http://www.threatexpert.com

Chapter 6

Conclusion

In this thesis we studied fingerprinting and detection methods for Anubis. We proposed mechanism to eliminate fingerprints on the fly, by modifying values returned by Windows operating system.

Our efforts regarding mapping differences in identical installations of Windows lead to more distinction than we initially thought. Nearly 10% of files are different in installations with identical setup parameters. Even registry key has a special date stored, which makes every key a possible fingerprint.

Additionally every piece of installed software or even every day the computer has been running leaves trails that can be considered as a fingerprint. A file located in a directory which is not a part of any installation is also a fingerprint.

Malware has limitless options for fingerprinting. It may even trace the days the computer is active by looking at file dates. If malware chooses not to run on inactive computers, computers that have been running for only few days, it only a looses a limited amount of active computers. For malware it is very important that it is active and can spread itself. If it applies a non-greedy approach and carefully inspects systems, it will take much longer to include it in anti-virus vendors' databases.

According to our studies full elimination of fingerprint is not feasible. This

work eliminates known detection schemes and probably a few further ones. But in a public installation possible fingerprinting options are limitless. Only if a public system has a private installation, one with a different fingerprint, malware will probably fail hide itself.

Bibliography

[1] P. Baecher, M. Koetter, M. Dornseif, and F. Freiling. The nepenthes platform: An efficient approach to collect malware. In *In Proceedings of the 9 th International Symposium on Recent Advances in Intrusion Detection (RAID*, pages 165–184. Springer, 2006.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[3] U. Bayer. Anubis: Analyzing unknown binaries. http://anubis.iseclab.org/, 2009.

[4] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. *Usenix Workshop on Large-scale Exploits and Emergent Threats (LEET).*, 2009.

[5] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[6] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, page 41, Berkeley, CA, USA, 2005. USENIX Association.

[7] J. Dike. User-mode linux. In *ALS '01: Proceedings of the 5th annual conference on Linux Showcase & Conference*, page 2, Berkeley, CA, USA, 2001. USENIX Association.

[8] C. Eagle. IDA Pro Freeware. `http://www.hex-rays.com/idapro/`.

[9] F-Secure. F-secure virus descriptions : Agobot. `http://www.f-secure.com/v-descs/agobot.shtml`, 2004.

[10] F-Secure. Virus glossary. http://www.f-secure.com/glossary/eng/malware-code-glossary.shtml, 2009.

[11] P. Ferrie. Attacks on more virtual machine emulators. `http://pferrie.tripod.com/papers/attacks2.pdf`, 2006.

[12] Fully Licensed GmbH. Inside windows product activation. `http://www.licenturion.com/xp/fully-licensed-wpa.txt`, 2001.

[13] G. Hunt, , G. Hunt, and D. Brubacher. Detours: Binary interception of win32 functions. In *In Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1998.

[14] X. Jiang, X. Wang, and D. Xu. D.: Stealthy malware detection through vmm-based "out-of-the-box"semanticview reconstruction. In *In:Proceedings of theACM Conference on Computer and Communications Security (CCS*, 2007.

[15] T. Klein. ScoopyNG. `http://www.trapkit.de/research/vmm/scoopyng/`, 2008.

[16] T. Krovetz and W. Dai. VMAC implementation. `http://www.fastcrypto.org/`, 2007.

[17] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, 1996:7, September 1996.

[18] A. Liguori. QEMU Manual. `http://www.qemu.org/qemu-doc.html`, 2009.

[19] L. Liu and S. Chen. Malyzer: Defeating anti-detection for application-level malware analysis. In M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud, editors, *ACNS*, volume 5536 of *Lecture Notes in Computer Science*, pages 201–218, 2009.

[20] Microsoft. Microsoft Virtual Server. `http://www.microsoft.com/windowsserversystem/virtualserver/`, 2005.

[21] Microsoft. Microsoft portable executable and common object file format specification. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, 2008.

[22] Microsoft. Windows api reference. `http://msdn.microsoft.com`, 2009.

[23] T. Ormandy. An emprical study into the security exposure to hosts of hostile virtualized environments. `http://taviso.decsystem.org/virtsec.pdf`, 2008.

[24] Parallels. SWSoft Parallels. `http://www.parallels.com/`, 2008.

[25] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In J. A. Garay, A. K. Lenstra, M. Mambo, and R. Peralta, editors, *ISC*, volume 4779 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2007.

[26] M. Russinovich and B. Cogswell. Freeware sysinternals. http://www.sysinternals. com, 2006.

[27] J. Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. http://invisiblethings.org/papers/redpill.html, 2006.

[28] Sentinel Chicken Networks. RegLookup. http://projects.sentinelchicken.org/reglookup/, 2009.

[29] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *IEEE Symposium on Security and Privacy*, 2006.

[30] VMware. VMware: Virtualization, Virtual Machine and Virtual Server Consolidation. `http://www.vmware.com/`.

[31] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, March 2007.

Appendix A

Data

A.1 Analysis Report

Analysis report of sample1.exe after the modification:

```
<activities>

<registry_activities>
<reg_value_modified count="1" description="auto_start"
   key="HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run"
   value_data="C:\WINDOWS\z_antivir.exe" value_name="z_antivir"/>
</registry_activities>

<file_activities>
<file_created name="C:\WINDOWS\z_antivir.exe"/>
<file_created name="C:\WINDOWS\z_antivir.log"/>
<file_read name="C:\WINDOWS\z_antivir.log"/>
<file_read name="C:\sample1.exe"/>
</file_activities>

<misc_activities>
```

```
<mutex_created name="CTF.TimListCache.FMPDefaultS-1-5-21-
   1229272821-1004336348-527237240-1003MUTEX.
   DefaultS-1-5-21-1229272821-1004336348-527237240-1003"/>
<key_was_checked count="7166" key="VK_BACK (8)"/>
<key_was_checked count="7166" key="VK_TAB (9)"/>
<key_was_checked count="7166" key="undefined (10)"/>
<key_was_checked count="7166" key="undefined (11)"/>
<key_was_checked count="7166" key="VK_CLEAR (12)"/>
<key_was_checked count="7166" key="VK_RETURN (13)"/>
<key_was_checked count="7166" key="undefined (14)"/>
<key_was_checked count="7166" key="undefined (15)"/>
<key_was_checked count="7166" key="VK_SHIFT (16)"/>
<key_was_checked count="7166" key="VK_CONTROL (17)"/>
<key_was_checked count="7166" key="VK_MENU (18)"/>
<key_was_checked count="7166" key="VK_PAUSE (19)"/>
<key_was_checked count="7166" key="VK_CAPITAL (20)"/>
<key_was_checked count="7166" key="VK_HANGUL (21)"/>
<key_was_checked count="7166" key="undefined (22)"/>
<key_was_checked count="7166" key="VK_JUNJA (23)"/>
<key_was_checked count="7166" key="VK_FINAL (24)"/>
<key_was_checked count="7166" key="VK_KANJI (25)"/>
<key_was_checked count="7166" key="undefined (26)"/>
<key_was_checked count="7166" key="VK_ESCAPE (27)"/>
<key_was_checked count="7166" key="VK_CONVERT (28)"/>
<key_was_checked count="7166" key="VK_NONCONVERT (29)"/>
<key_was_checked count="7166" key="VK_ACCEPT (30)"/>
<key_was_checked count="7166" key="VK_MODECHANGE (31)"/>
<key_was_checked count="7166" key="VK_SPACE (32)"/>
<key_was_checked count="7166" key="VK_PRIOR (33)"/>
<key_was_checked count="7166" key="VK_NEXT (34)"/>
<key_was_checked count="7166" key="VK_END (35)"/>
<key_was_checked count="7166" key="VK_HOME (36)"/>
<key_was_checked count="7166" key="VK_LEFT (37)"/>
```

```
<key_was_checked count="7166" key="VK_UP (38)"/>
<key_was_checked count="7166" key="VK_RIGHT (39)"/>
<key_was_checked count="7166" key="VK_DOWN (40)"/>
<key_was_checked count="7166" key="VK_SELECT (41)"/>
<key_was_checked count="7166" key="VK_PRINT (42)"/>
<key_was_checked count="7166" key="VK_EXECUTE (43)"/>
<key_was_checked count="7166" key="VK_SNAPSHOT (44)"/>
<key_was_checked count="7166" key="VK_INSERT (45)"/>
<key_was_checked count="7166" key="VK_DELETE (46)"/>
<key_was_checked count="7166" key="VK_HELP (47)"/>
<key_was_checked count="7166" key="VK_0 (48)"/>
<key_was_checked count="7166" key="VK_1 (49)"/>
<key_was_checked count="7166" key="VK_2 (50)"/>
<key_was_checked count="7165" key="VK_3 (51)"/>
<key_was_checked count="7165" key="VK_4 (52)"/>
<key_was_checked count="7165" key="VK_5 (53)"/>
<key_was_checked count="7165" key="VK_6 (54)"/>
<key_was_checked count="7165" key="VK_7 (55)"/>
<key_was_checked count="7165" key="VK_8 (56)"/>
<key_was_checked count="7165" key="VK_9 (57)"/>
<key_was_checked count="7165" key="undefined (58)"/>
<key_was_checked count="7165" key="undefined (59)"/>
<key_was_checked count="7165" key="undefined (60)"/>
<key_was_checked count="7165" key="undefined (61)"/>
<key_was_checked count="7165" key="undefined (62)"/>
<key_was_checked count="7165" key="undefined (63)"/>
<key_was_checked count="7165" key="undefined (64)"/>
<key_was_checked count="7165" key="VK_A (65)"/>
<key_was_checked count="7165" key="VK_B (66)"/>
<key_was_checked count="7165" key="VK_C (67)"/>
<key_was_checked count="7165" key="VK_D (68)"/>
<key_was_checked count="7165" key="VK_E (69)"/>
<key_was_checked count="7165" key="VK_F (70)"/>
```

```
<key_was_checked count="7165" key="VK_G (71)"/>
<key_was_checked count="7165" key="VK_H (72)"/>
<key_was_checked count="7165" key="VK_I (73)"/>
<key_was_checked count="7165" key="VK_J (74)"/>
<key_was_checked count="7165" key="VK_K (75)"/>
<key_was_checked count="7165" key="VK_L (76)"/>
<key_was_checked count="7165" key="VK_M (77)"/>
<key_was_checked count="7165" key="VK_N (78)"/>
<key_was_checked count="7165" key="VK_O (79)"/>
<key_was_checked count="7165" key="VK_P (80)"/>
<key_was_checked count="7165" key="VK_Q (81)"/>
<key_was_checked count="7165" key="VK_R (82)"/>
<key_was_checked count="7165" key="VK_S (83)"/>
<key_was_checked count="7165" key="VK_T (84)"/>
<key_was_checked count="7165" key="VK_U (85)"/>
<key_was_checked count="7165" key="VK_V (86)"/>
<key_was_checked count="7165" key="VK_W (87)"/>
<key_was_checked count="7165" key="VK_X (88)"/>
<key_was_checked count="7165" key="VK_Y (89)"/>
<key_was_checked count="7165" key="VK_Z (90)"/>
<key_was_checked count="7165" key="VK_LWIN (91)"/>
<key_was_checked count="7165" key="VK_RWIN (92)"/>
<key_was_checked count="7165" key="VK_APPS (93)"/>
<key_was_checked count="7165" key="undefined (94)"/>
<key_was_checked count="7165" key="VK_SLEEP (95)"/>
<key_was_checked count="7165" key="VK_NUMPAD0 (96)"/>
<key_was_checked count="7165" key="VK_NUMPAD1 (97)"/>
<key_was_checked count="7165" key="VK_NUMPAD2 (98)"/>
<key_was_checked count="7165" key="VK_NUMPAD3 (99)"/>
<key_was_checked count="7165" key="VK_NUMPAD4 (100)"/>
<key_was_checked count="7165" key="VK_NUMPAD5 (101)"/>
<key_was_checked count="7165" key="VK_NUMPAD6 (102)"/>
<key_was_checked count="7165" key="VK_NUMPAD7 (103)"/>
```

```
<key_was_checked count="7165" key="VK_NUMPAD8 (104)"/>
<key_was_checked count="7165" key="VK_NUMPAD9 (105)"/>
<key_was_checked count="7165" key="VK_MULTIPLY (106)"/>
<key_was_checked count="7165" key="VK_ADD (107)"/>
<key_was_checked count="7165" key="VK_SEPARATOR (108)"/>
<key_was_checked count="7165" key="VK_SUBTRACT (109)"/>
<key_was_checked count="7165" key="VK_DECIMAL (110)"/>
<key_was_checked count="7165" key="VK_DIVIDE (111)"/>
<key_was_checked count="7165" key="VK_F1 (112)"/>
<key_was_checked count="7165" key="VK_F2 (113)"/>
<key_was_checked count="7165" key="VK_F3 (114)"/>
<key_was_checked count="7165" key="VK_F4 (115)"/>
<key_was_checked count="7165" key="VK_F5 (116)"/>
<key_was_checked count="7165" key="VK_F6 (117)"/>
<key_was_checked count="7165" key="VK_F7 (118)"/>
<key_was_checked count="7165" key="VK_F8 (119)"/>
<key_was_checked count="7165" key="VK_F9 (120)"/>
<key_was_checked count="7165" key="VK_F10 (121)"/>
<key_was_checked count="7165" key="VK_F11 (122)"/>
<key_was_checked count="7165" key="VK_F12 (123)"/>
<key_was_checked count="7165" key="VK_F13 (124)"/>
<key_was_checked count="7165" key="VK_F14 (125)"/>
<key_was_checked count="7165" key="VK_F15 (126)"/>
<key_was_checked count="7165" key="VK_F16 (127)"/>
<key_was_checked count="7165" key="VK_F17 (128)"/>
<key_was_checked count="7165" key="VK_F18 (129)"/>
<key_was_checked count="7165" key="VK_F19 (130)"/>
<key_was_checked count="7165" key="VK_F20 (131)"/>
<key_was_checked count="7165" key="VK_F21 (132)"/>
<key_was_checked count="7165" key="VK_F22 (133)"/>
<key_was_checked count="7165" key="VK_F23 (134)"/>
<key_was_checked count="7165" key="VK_F24 (135)"/>
<key_was_checked count="7165" key="undefined (136)"/>
```

```
<key_was_checked count="7165" key="undefined (137)"/>
<key_was_checked count="7165" key="undefined (138)"/>
<key_was_checked count="7165" key="undefined (139)"/>
<key_was_checked count="7165" key="undefined (140)"/>
<key_was_checked count="7165" key="undefined (141)"/>
<key_was_checked count="7165" key="undefined (142)"/>
<key_was_checked count="7165" key="undefined (143)"/>
<key_was_checked count="7165" key="VK_NUMLOCK (144)"/>
<key_was_checked count="7165" key="VK_SCROLL (145)"/>
<key_was_checked count="7165" key="undefined (146)"/>
<key_was_checked count="7165" key="undefined (147)"/>
<key_was_checked count="7165" key="undefined (148)"/>
<key_was_checked count="7165" key="undefined (149)"/>
<key_was_checked count="7165" key="undefined (150)"/>
<key_was_checked count="7165" key="undefined (151)"/>
<key_was_checked count="7165" key="undefined (152)"/>
<key_was_checked count="7165" key="undefined (153)"/>
<key_was_checked count="7165" key="undefined (154)"/>
<key_was_checked count="7165" key="undefined (155)"/>
<key_was_checked count="7165" key="undefined (156)"/>
<key_was_checked count="7165" key="undefined (157)"/>
<key_was_checked count="7165" key="undefined (158)"/>
<key_was_checked count="7165" key="undefined (159)"/>
<key_was_checked count="7165" key="VK_LSHIFT (160)"/>
<key_was_checked count="7165" key="VK_RSHIFT (161)"/>
<key_was_checked count="7165" key="VK_LCONTROL (162)"/>
<key_was_checked count="7165" key="VK_RCONTROL (163)"/>
<key_was_checked count="7165" key="VK_LMENU (164)"/>
<key_was_checked count="7165" key="VK_RMENU (165)"/>
<key_was_checked count="7165" key="VK_BROWSER_BACK (166)"/>
<key_was_checked count="7165" key="VK_BROWSER_FORWARD (167)"/>
<key_was_checked count="7165" key="VK_BROWSER_REFRESH (168)"/>
<key_was_checked count="7165" key="VK_BROWSER_STOP (169)"/>
```

```
<key_was_checked count="7165" key="VK_BROWSER_SEARCH (170)"/>
<key_was_checked count="7165" key="VK_BROWSER_FAVORITES (171)"/>
<key_was_checked count="7165" key="VK_BROWSER_HOME (172)"/>
<key_was_checked count="7165" key="VK_VOLUME_MUTE (173)"/>
<key_was_checked count="7165" key="VK_VOLUME_DOWN (174)"/>
<key_was_checked count="7165" key="VK_VOLUME_UP (175)"/>
<key_was_checked count="7165" key="VK_MEDIA_NEXT_TRACK (176)"/>
<key_was_checked count="7165" key="VK_MEDIA_PREV_TRACK (177)"/>
<key_was_checked count="7165" key="VK_MEDIA_STOP (178)"/>
<key_was_checked count="7165" key="VK_MEDIA_PLAY_PAUSE (179)"/>
<key_was_checked count="7165" key="VK_LAUNCH_MAIL (180)"/>
<key_was_checked count="7165" key="VK_LAUNCH_MEDIA_SELECT (181)"/>
<key_was_checked count="7165" key="VK_LAUNCH_APP1 (182)"/>
<key_was_checked count="7165" key="VK_LAUNCH_APP2 (183)"/>
<key_was_checked count="7165" key="undefined (184)"/>
<key_was_checked count="7165" key="undefined (185)"/>
<key_was_checked count="7165" key="VK_OEM_1 (186)"/>
<key_was_checked count="7165" key="VK_OEM_PLUS (187)"/>
<key_was_checked count="7165" key="VK_OEM_COMMA (188)"/>
<key_was_checked count="7165" key="VK_OEM_MINUS (189)"/>
<key_was_checked count="7165" key="VK_OEM_PERIOD (190)"/>
<key_was_checked count="7165" key="VK_OEM_2 (191)"/>
<key_was_checked count="7165" key="VK_OEM_3 (192)"/>
<key_was_checked count="7165" key="undefined (193)"/>
<key_was_checked count="7165" key="undefined (194)"/>
<key_was_checked count="7165" key="undefined (195)"/>
<key_was_checked count="7165" key="undefined (196)"/>
<key_was_checked count="7165" key="undefined (197)"/>
<key_was_checked count="7165" key="undefined (198)"/>
<key_was_checked count="7165" key="undefined (199)"/>
<key_was_checked count="7165" key="undefined (200)"/>
<key_was_checked count="7165" key="undefined (201)"/>
<key_was_checked count="7165" key="undefined (202)"/>
```

```
<key_was_checked count="7165" key="undefined (203)"/>
<key_was_checked count="7165" key="undefined (204)"/>
<key_was_checked count="7165" key="undefined (205)"/>
<key_was_checked count="7165" key="undefined (206)"/>
<key_was_checked count="7165" key="undefined (207)"/>
<key_was_checked count="7165" key="undefined (208)"/>
<key_was_checked count="7165" key="undefined (209)"/>
<key_was_checked count="7165" key="undefined (210)"/>
<key_was_checked count="7165" key="undefined (211)"/>
<key_was_checked count="7165" key="undefined (212)"/>
<key_was_checked count="7165" key="undefined (213)"/>
<key_was_checked count="7165" key="undefined (214)"/>
<key_was_checked count="7165" key="undefined (215)"/>
<key_was_checked count="7165" key="undefined (216)"/>
<key_was_checked count="7165" key="undefined (217)"/>
<key_was_checked count="7165" key="undefined (218)"/>
<key_was_checked count="7165" key="VK_OEM_4 (219)"/>
<key_was_checked count="7165" key="VK_OEM_5 (220)"/>
<key_was_checked count="7165" key="VK_OEM_6 (221)"/>
<key_was_checked count="7165" key="VK_OEM_7 (222)"/>
</misc_activities>
</activities>
```

A.2 Disassembled Code

Disassembly of `main` function of sample2.exe:

```
.text:00401000 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401000 _main           proc near               ; CODE XREF: start+AFp
.text:00401000
.text:00401000 nSize           = dword ptr -104h
.text:00401000 Buffer          = byte ptr -100h
```

```
.text:00401000 argc            = dword ptr  4
.text:00401000 argv            = dword ptr  8
.text:00401000 envp            = dword ptr  0Ch
.text:00401000
.text:00401000    sub     esp, 104h
.text:00401006    push    ebx
.text:00401007    lea     eax, [esp+108h+nSize]
.text:0040100B    push    esi
.text:0040100C    lea     ecx, [esp+10Ch+Buffer]
.text:00401010    push    eax             ; nSize
.text:00401011    push    ecx             ; lpBuffer
.text:00401012    mov     [esp+114h+nSize], 100h
.text:0040101A    call    ds:GetComputerNameA
.text:00401020    lea     edx, [esp+10Ch+Buffer]
.text:00401024    push    edx
.text:00401025    push    offset aComputernameS ; "Computername %s\r\n"
.text:0040102A    call    _printf
.text:0040102F    add     esp, 8
.text:00401032    mov     esi, offset aComputername ; "COMPUTERNAME"
.text:00401037    lea     eax, [esp+10Ch+Buffer]
.text:0040103B
.text:0040103B loc_40103B:                ; CODE XREF: _main+5Dj
.text:0040103B    mov     dl, [eax]
.text:0040103D    mov     bl, [esi]
.text:0040103F    mov     cl, dl
.text:00401041    cmp     dl, bl
.text:00401043    jnz     short loc_401063
.text:00401045    test    cl, cl
.text:00401047    jz      short loc_40105F
.text:00401049    mov     dl, [eax+1]
.text:0040104C    mov     bl, [esi+1]
.text:0040104F    mov     cl, dl
.text:00401051    cmp     dl, bl
```

```
.text:00401053    jnz     short loc_401063
.text:00401055    add     eax, 2
.text:00401058    add     esi, 2
.text:0040105B    test    cl, cl
.text:0040105D    jnz     short loc_40103B
.text:0040105F
.text:0040105F loc_40105F:                ; CODE XREF: _main+47j
.text:0040105F    xor     eax, eax
.text:00401061    jmp     short loc_401068
.text:00401063 ; ---------------------------------------------
.text:00401063
.text:00401063 loc_401063:                ; CODE XREF: _main+43j
.text:00401063                            ; _main+53j
.text:00401063    sbb     eax, eax
.text:00401065    sbb     eax, 0FFFFFFFFh
.text:00401068
.text:00401068 loc_401068:                ; CODE XREF: _main+61j
.text:00401068    test    eax, eax
.text:0040106A    jnz     short loc_401082
.text:0040106C    push    offset aHoneypotThreat ; "Honeypot [THREAT EXPERT]"
.text:00401071    call    _printf
.text:00401076    add     esp, 4
.text:00401079    pop     esi
.text:0040107A    pop     ebx
.text:0040107B    add     esp, 104h
.text:00401081    retn
.text:00401082 ; ---------------------------------------------
.text:00401082
.text:00401082 loc_401082:                ; CODE XREF: _main+6Aj
.text:00401082    mov     esi, offset aUser ; "USER"
.text:00401087    lea     eax, [esp+10Ch+Buffer]
.text:0040108B
.text:0040108B loc_40108B:                ; CODE XREF: _main+ADj
```

```
.text:0040108B    mov     dl, [eax]
.text:0040108D    mov     bl, [esi]
.text:0040108F    mov     cl, dl
.text:00401091    cmp     dl, bl
.text:00401093    jnz     short loc_4010B3
.text:00401095    test    cl, cl
.text:00401097    jz      short loc_4010AF
.text:00401099    mov     dl, [eax+1]
.text:0040109C    mov     bl, [esi+1]
.text:0040109F    mov     cl, dl
.text:004010A1    cmp     dl, bl
.text:004010A3    jnz     short loc_4010B3
.text:004010A5    add     eax, 2
.text:004010A8    add     esi, 2
.text:004010AB    test    cl, cl
.text:004010AD    jnz     short loc_40108B
.text:004010AF
.text:004010AF loc_4010AF:                ; CODE XREF: _main+97j
.text:004010AF    xor     eax, eax
.text:004010B1    jmp     short loc_4010B8
.text:004010B3 ; ---------------------------------------------
.text:004010B3
.text:004010B3 loc_4010B3:                ; CODE XREF: _main+93j
.text:004010B3                            ; _main+A3j
.text:004010B3    sbb     eax, eax
.text:004010B5    sbb     eax, 0FFFFFFFFh
.text:004010B8
.text:004010B8 loc_4010B8:                ; CODE XREF: _main+B1j
.text:004010B8    test    eax, eax
.text:004010BA    jnz     short loc_4010D2
.text:004010BC    push    offset aHoneypotAnubis ; "Honeypot [ANUBIS]"
.text:004010C1    call    _printf
.text:004010C6    add     esp, 4
```

```
.text:004010C9    pop     esi
.text:004010CA    pop     ebx
.text:004010CB    add     esp, 104h
.text:004010D1    retn
.text:004010D2 ; ---------------------------------------------
.text:004010D2
.text:004010D2 loc_4010D2:                ; CODE XREF: _main+BAj
.text:004010D2    push    offset aNotHoneypot ; "not Honeypot"
.text:004010D7    call    _printf
.text:004010DC    add     esp, 4
.text:004010DF    pop     esi
.text:004010E0    pop     ebx
.text:004010E1    add     esp, 104h
.text:004010E7    retn
.text:004010E7 _main          endp
.text:004010E7
.text:004010E7 ; ---------------------------------------------
```