Detecting Remote Administration Trojans through Dynamic Analysis using Finite-State Machines

Kjetil Tangen Gardåsen

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2014

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Abstract

In this master's thesis we show how we can use the Application Programming Interface (API) calls a program makes to detect if it is a Remote Administration Trojan (RAT) or not. We have looked at which API calls that are performed when the webcam, microphone and keylogger are used by a program, and we use this information to determine if a program can be considered to be spying on its user by covertly capturing information from any of these sources. In addition, we have looked at the network traffic as seen through the API calls of the examined RAT, and we use this to better pinpoint which RAT that is present on the system.

To distinguish a RAT from a program with similar behavior (e.g. video chat vs. webcam spying) we have used user interaction with the program as a discriminator for whether or not it is malicious, with our theory being that a program such as a video chat program would interact with the user and have a user interface, while a RAT would not and would instead attempt to stay hidden. By observation we show that this is what is happening in real-life scenarios, and that the RATs do not interact with the user or provide a user interface.

We have subsequently modeled the API calls that denote the various forms of behavior and user interaction as finite-state machines that can be used to detect whether or not a RAT is present on the system and what it is doing.

Sammendrag

I denne masteroppgaven viser vi hvordan vi kan bruke Application Programming Interface (API) kallene et program utfører for å oppdage om det er en Remote Administration Trojan (RAT) eller ikke. Vi har undersøkt hvilke API-kall som blir gjort når webkamera, mikrofon og tastaturet brukes av et program, og vi bruker denne informasjonen til å finne ut av om et program spionerer på brukeren via noen av disse grensesnittene eller ikke. I tillegg har vi sett på nettverkstrafikken via API-kallene som den undersøkte RATen utførerer, og vi bruker denne informasjon til å mer nøyaktig finne ut hvilken RAT som befinner seg på systemet.

For å skille en RAT fra et program med lignende oppførsel (f.eks. video chat og wekamera-spionering) har vi brukt interaksjon med brukeren som et skille på hvorvidt programmet er skadelig eller ikke, hvor teorien vår har vært at et program som video chaten vil samhandle med brukeren og dermed ha et brukergrensesnitt, mens en RAT ikke vil det og heller forsøker å holde seg skjult. Ved observasjon har vi vist at det er dette som skjer, og at RATene ikke samhandler med brukeren eller har noe brukergrensesnitt.

Vi har til slutt modellert API-kallene som brukes for de forskjellige typene oppførsel og bruker-samhandling som tilstandsmakiner som kan brukes til å oppdage om en RAT finnes på systemet og hva den driver med.

Acknowledgments

I would like to thank my supervisor Lasse Øverlier for all the insightful help and guidance he has given me throughout the thesis. My dear friends André, Eirik, Espen and Ola have my eternal gratitude for enduring all the relevant academic and non-relevant academic discussions we have had during this thesis. I would also like to thank my colleagues at Nets who have been very supportive and provided me with encouragement along the way. Lasse Halaas and FLO/IKT deserves a thank-you for steering me onto this topic and their helpful sparring on related topics.

And last, but not least, I want to thank my parents and sister for their everlasting love and support.

Glossary

- **RAT** - Remote Administration Trojan/Tool
- **FSM** - Finite-State Machine
- **IDS** - Intrusion Detection System
- **SSDT** - System Service Descriptor Table
- **IAT** - Import Address Table
- **FUD** - Fully Undetectable
- **DDoS** - Distributed Denial of Service

Contents

List of Figures

List of Tables

1 Introduction

Malware has plagued computer users almost since the first computers were introduced. First invented as harmless pranks, malware has later evolved into an industry more profitable than the black market for marijuana, cocaine and heroin combined[6]. This means that there has become strong incentives for creating malware, and new attacks are as inventive as the techniques trying to stop them. This has led to a cat and mouse game between attackers and defenders, where new attacks appear as soon as new systems are developed.

Being infected by malware can range from being a small nuisance to a literally life-altering experience, depending on what sort of information that was compromised, deleted or altered, and to whom it was spread to. There are numerous stories about harmful malware attacks, with tragic stories of individuals having their privacy breached for fun and profit, corporations losing valuable business assets, and governments being subjected to complex intelligence operations from powerful foreign actors.

Advances in computer technology has also given access to new sensors that can capture the world around them, such as microphones, cameras, GPS, accelerometers, and fingerprint readers. As wonderful as these devices are, they have also raised the stakes for what kind of privacy related incidents that are now possible. Where as a computer from 20 years ago might have had a microphone, a camera was still unheard of for most consumers, and biometric readers were still the stuff of science fiction. But as computers became more powerful and got high-resolution screens and faster processors and cameras became sharper, smaller and cheaper, they soon found their way onto computers and handheld devices. First as external units, and later as integrated devices like the now ubiquitous built-in webcamera and microphone on all modern laptops. This "dawn of the webcams" that started around the beginning of the millennia did also give rise to a new kind of privacy threat, namely the possibility of someone spying on you through the camera now embedded in your computer. What was once only an Orwellian concept had suddenly become part of everyday life, without anyone seeming to oppose it or even take much notice.

The fact that cameras suddenly are everywhere have also created strong incentives for both governments and private citizens to get access to them and the information they yield access to. With a little hacking or social engineering anyone can suddenly get access to a bugging device in someone else's home, without ever entering it or even leaving a trace. This is something which has made the process of surveilling someone extremely cheap and easy, and it has not gone unnoticed by the worlds criminals, governments and teenage boys, all of whom have acquired the tools and capabilities to tap into these vast treasure troves of personal information.

The reasons for someone wanting to use and abuse such capabilities are manyfold. Governments are very interested in spying for reasons of counter-terrorism, intelligence gathering and crime investigation, while criminals on the other hand are usually in it for the money and wants to make a profit from the information they can gather. For them, this can be things like credit card numbers, email account passwords, and other easily tradeable commodity goods[1]. Individuals might have motives such as spying on a spouse, blackmailing an adversary, stalking a romantic interest or simply pure voyeurism.

This voyeurism aspect has also given rise to more "organized" Internet communities that share information and software for spying on other people through their computer and webcam. The tools of this trade are usually referred to as Remote Administration Trojans, or just RATs for short, and Nate Anderson, a journalist at the technology website Ars Technica, wrote an investigative piece about the online communities appearing around these RATs, what these communities do, and how they spy on other people[7]. These Internet cliques discusses techniques and exchanges software for taking control over a victims computers, and the RAT controllers, or "ratters" as they are called in the article, has nearly unlimited control of the computers they have infected, and can control every aspect of it, including the perhaps most sensitive - namely seeing their targets through their webcam. The reason for doing so is mainly twofold - monetary gain and pure "pleasure". The victims, or slaves as they are called in ratter-lingo, also gains the capabilities of the traditional botnet herder in that they can use their slaves to DDoS targets, send spam, and more recently, mine BitCoins or lock the victims computer and force them to do surveys for cash. The second motivation is voyeurism, namely watching their victims through their webcam. As one can imagine, young women are the most popular target for this kind of activity.

Victims are usually acquired by spreading the RATs through file sharing networks or social engineering, disguising them as games or movies, or sending the RAT to potential victims through Facebook. For monetary gain, spreading the RAT as wide as possible is clearly the most beneficial, but for spying, "high quality" targets are more important. The RAT communities are filled with discussions about how to spread their RATs to new victims, and even contains forum posts about trading and selling access to their slaves for money.

These sorts of past-times can however have severe consequences, as was experienced by the man who tricked Miss Teen USA 2013 into installing a RAT of his control and who was subsequently jailed for 18 months for doing so[8]. In May 2014 the U.S. Justice Department had a crackdown on purchasers of the RAT Blackshades followed by several indictments[9].

The leaked Snowden documents have shown that the NSA also have taken to these capabilities and are using them in their work. The NSA has been distributing malware on an unprecedented scale by infecting millions of targets worldwide automatically[10], and one such piece of malware is called UNITEDRAKE and allows the NSA to control the infected computer. Through plugins such as CAPTIVATEAUDIENCE, UNITEDRAKE's functionality can be expanded, and allow it to do such things as recording audio through the computers microphone or take snapshots through the webcam with the GUMFISH plugin. Other such plugins are GROK which is a keylogger, and FOGGYBOTTOM which collects passwords and web browser history.

As a quick summary, security journalist Brian Krebs has illustrated the types of valuable information that a hacked computer contains[1]. This can be seen in Figure 1. As we can see, a hacked computer is a treasure trove of personal and valuable information, and there is something for everyone in there, from criminals to governments to private citizens.

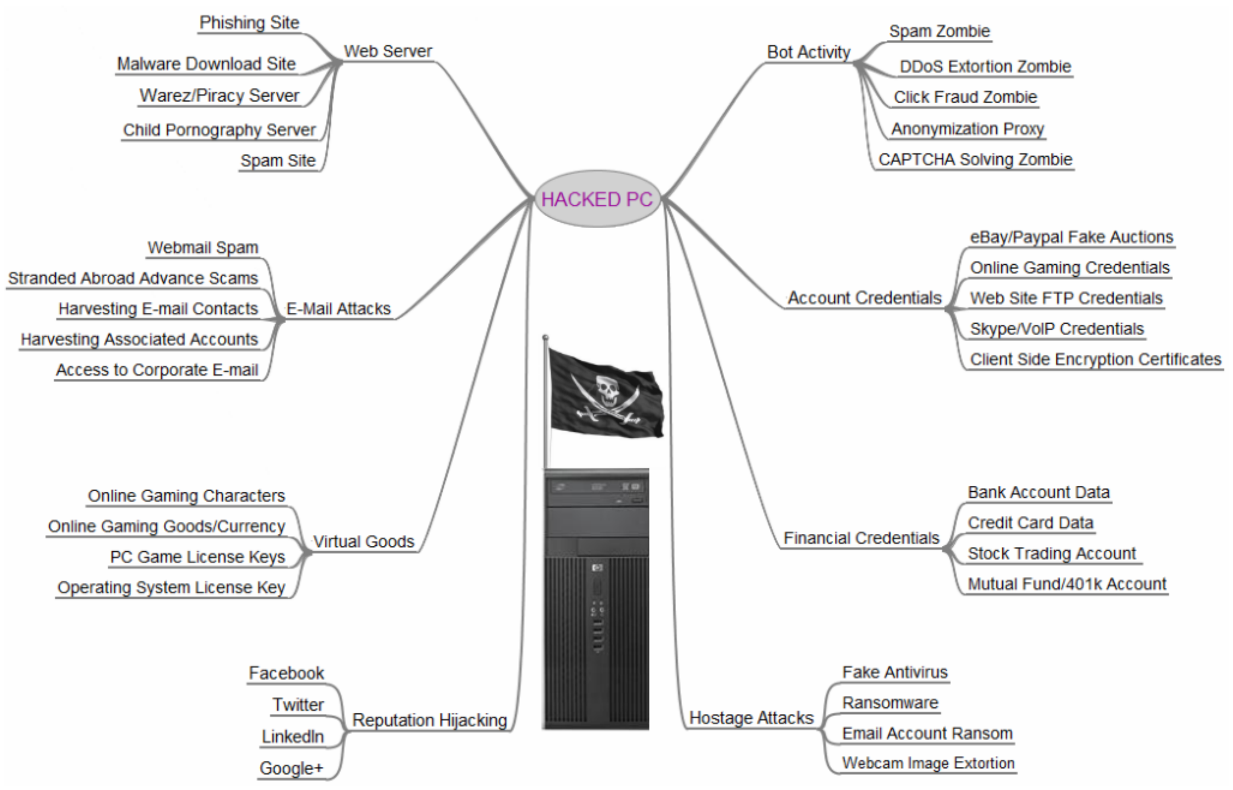


Figure 1: The different types of information a hacker can get from a hacked PC. Image from Brian Krebs[1].

1.1 Topics Covered by the Project

This master's thesis will look at how we can detect programs that spies on its users. We look for programs that surreptitiously monitors the webcam, microphone and keystrokes, and we use this behavior as an indicator for the presence of a RAT, because we theorize that a RAT has a unique behavior that is easily detected and can be distinguished from other programs, even those who also use the webcam, microphone and captures keyboard input.

We have examined six existing RATs to determine what they actually do when they spy, followed by a look at two video chat applications that have a behavior that is similar to a RAT in that they use the computer's webcam, microphone and keyboard, but who are not actually malicious because they are "spying" with the user's consent.

1.2 Keywords

Reverse engineering, application programming interfaces, malware, remote administration trojans, finite-state machines

1.3 Problem description

There are several problems related to detecting RATs. For one thing, it can be hard to distinguish harmful behavior from legitimate. Is it considered spying when a program uses the webcam? Or does the program have to ask the user for permission first for it to be OK? What if the program is a home monitoring system that starts recording video at given intervals without user intervention?

Questions like these blurs the line between malware and useful software and makes it harder to distinguish when behavior can be considered legitimate and when it is not.

Also, many RATs does actually serve a (purported) dual-purpose as tools for remotely administrating computers for system administrators, and for this simple reason they might not be detected by anti-virus programs. But a user would probably classify such a program as malicious if they had it installed on their computer without their consent, even if a anti-virus solution views it as legitimate.

Another point is that the RATs can be quite targeted and only be spread to certain individuals or sub-groups, without having the wide coverage of a replicating worm or a spam campaign. And malware authors can even create completely new RATs for their own personal use, which makes it highly unlikely that any malware researcher ever will get their hands on it.

Obfuscation technologies is another neat trick for preventing analysis and detection of the RATs, and when a new and fully undetectable RAT can be created from an old one in mere seconds, it has become trivial to fool virtually all existing commercial anti-virus solutions. Analyzing malware by statically looking at its source code can easily be fooled by obfuscation techniques such as encryption and insertion of extra operations. And current tools for dynamic analysis can often be stopped by some of the same techniques, namely inserting bogus operations and weaving through instructions and code in the program when running[11].

Of course, Intrusion Detection Systems (IDS) tries to remedy some of these problems by attempting to detect malicious behavior by looking at network traffic and system usage, either by making a baseline of normal system behavior and detecting everything that deviates from this, or else by having specific signatures of known attacks that it looks for[11]. IDSes are however often plagued by high false-positive rates and problems with keeping up with the pace of change in deployed software and new usage patterns. Trend Micro has also described two challenges relating to network traffic analysis, namely encryption and cloud services[12]. The cloud makes it impossible to block access to certain IP ranges, and encryption thwarts attempts at looking at URL parameters and other traffic content.

1.4 Justification, Motivation and Benefits

Based on the aforementioned problems, finding new ways to detect RATs is essential for unveiling them in the future. Bishop states that new detection profiles for detecting unknown attacks would also improve the state of intrusion detection[13].

As described above, current techniques for detecting RATs are easy to circumvent for even the most ham-fisted attacker, yet RATs have a very unique way of behaving and should thus be possible to detect with new techniques that leverage the fact that they have a unique behavior, rather than using the static signatures that we already know from existing anti-virus solutions.

Creating new solutions that leverage behavior detection rather than signature-based detection is beneficial because it makes it possible to detect new pieces of malware that is both obfuscated or completely new, provided that it has the same behavior as we expect a RAT to have. This would be beneficial for all computer users, seen as malware is still a huge problem that plagues everyone from corporations to private citizens. Being spied on through your webcam might also be one of the most privacy invading things that can happen, seen as people expect to have privacy in their own home and not be watched by total strangers over the Internet.

1.5 Research Questions

The process of detecting a RAT can be divided into several research questions. First, we need to find a suitable way to examine an application and collect information about its behavior so that we can infer enough knowledge about it to realistically decide if it is a RAT or not. This will be done using the information that can be gathered from looking at the Application Programming Interface (API) calls that the examined application makes.

Secondly, we need a set of rules for determining what kind of programs might be malicious and which might be not. These rules need to be unique for the RATs behavior, and not catch other programs that also uses the webcam, microphone and keyboard, such as video conferencing or home-monitoring systems. A detection system is obviously useless if it gives too many false alarms.

A distinctive trait of a RAT is that it usually performs actions automatically that a user normally would start manually. E.g. starting the webcam. For most users this would be a manual process, something done when starting a video chat or taking a picture, and not something a background task should do by its own will without notifying the user.

This can be summarized into these four research questions:

1. What kind of behavior is unique for a RAT?

2. Which API calls can be used to discover this behavior?
3. How can we use this information to discriminate a RAT from a benign program?
4. Is it possible to model the RATs behavior with a finite-state machine and detect them based on that?

1.6 Contributions

Based on the research questions above, this master's thesis will make the following contributions:

1. An analysis of different RATs and the API calls they make and how they can be used to determine which actions the RAT performs.
2. A model for determining wheter a program is a RAT or not based on the captured API calls of the examined program.

Contribution 1 will be a novel contribution that looks at which API calls that are performed when different types of RAT behavior is occurring. Our goal has been to capture the behavior of the RAT, e.g. detecting when the webcamera has started recording, and then using the calls that appear when this happens to create a model for the RAT behavior. This has been done by studying a selection of carefully curated RATs which we feel represent a cross section of the RATs available today.

Contribution 2 is about using this captured behavior to create models that generalize RAT behavior and which can be used to detect the presence of new RATs that fit into this observed behavior pattern. By using the previously gathered information we should then be able to reasonably infer if a program is a RAT or not.

2 Background

This chapter explains some background information related to the RATs and our experiments. It contains the following sections:

- A definition of what a RAT is.
- A brief history of some notable RATs.
- A technical description of the RATs properties and how to detect them.
- How finite-state machines can be used to model the RATs behavior.

2.1 What is a RAT?

Before we look at the technical details of how a RAT works we should begin by looking at what a RAT actually *is*. The answer is sort of given by its name, Remote Administration Trojan, which reveals that it is a trojan that remotely controls a computer[14]. But let us try to be more precise.

Modern computer security literature contains an abundant taxonomy for classifying malware and computer programs into categories. And since these are the established notion in our "community" we should apply them to RATs as well. As with most things in life, categories are seldom clear cut, and the modern RAT fits into several malware groups, based both on how it enters the system and what it does when it arrives there.

2.1.1 Backdoors and Zombies

Since a RAT allows someone to remotely control the system and send it commands it fits well within the definition of a backdoor. A backdoored computer is often referred to as a zombie because it allows its master to perform operations through it, and thus becomes zombie-like in its state of behavior[15].

2.1.2 Trojan Horses

This backdoor might be distributed to its victim in a way that makes it a trojan horse, meaning that it will be disguised as being new song, a popular game, or a video showing war time atrocities in your home country. By disguising its real identity as some other tantalizing piece of software, it will probably make it more likely to be executed and distributed by new potential victims[15].

Thus, an informal definition of a trojan horse would be that it is a program that disguises itself as being something other than what it really is. A more stringent definition is given by Bishop who simply states that "*A Trojan horse is a program with an overt (documented or known) effect* *and a covert (undocumented or unexpected) effect.*"[13].

Trouble usually start when this file is executed by the receiver who thinks that she has downloaded a new song or movie, but in reality has received a program *disguised* as that. When executed, the trojan will usually attempt to execute its payload and start deleting files, stealing information, connecting back to its master, and everything else that it has been programmed to[15]. And depending on the cleverness of the trojan creator, a song might also play or a game might run, making it less obvious for the victim that she received a trojan instead of what she actually wanted[15].

The trojan would usually also be attempting to maintain a foothold on the computer it was executed on, so as to able survive reboots and process termination, and thus achieving persistence. This is usually done by setting itself to launch when the operating system starts or when a special process, such as a web browser, is run[15].

The clever part here is, as Tanenbaum points out, that the trojan horse does not require the attacker to break into the victims computer. It simply makes the victim do all the work for him by executing the trojan, and thus bypassing any barriers to entering the victims machine[15]. So it makes the cliché that humans are the weakest link in the security chain seem all the more true.

Trojan Variations

A subset of the trojan horse is the propagating (or replicating) trojan horse, which creates copies of itself[13]. The example given by Bishop was a game called "ANIMAL" which cloned itself every time the game was played. A later version was modified to delete the older version of itself, and instead create two more copies of its new self, effectively killing of and replacing the older version.

2.1.3 Spyware

So far we have described what the RAT is and how it enters the system. But it usually has more tricks up its sleeves. For instance, it spies on you.

Tanenbaum describes spyware as software that is installed on the user's computer without them knowing about it, and doing things that they are not informed about happening. He does however point out that this does not provide a sufficient definition, because something like Windows Update, which downloads updates without user interaction, fits this description but the users would probably not classify it as spyware. Tanenbaum does however paraphrase U.S. Supreme Court justice Potter Stewart, who was famed for saying that he was unable to define pornography but "*I know it when I see it*", into the expression "*I can't define spyware, but I know it when I see it.*"[15].

On a more serious note, Tanenbaum further refers to Barwinski et al. who claims spyware has four characteristics:

1. It hides so that it is difficult to detect

2. It gathers information about the user
3. It sends this information back to its creator or controller
4. It tries to avoid being removed

Further referring to Barwinski, the spyware was divided into three main categories based on the reason for its existence[15]:

- **Marketing** - Spyware that collects information about the user so as to better serve targeted ads that fits the interests of the target machine.
- **Surveillance** - Employers might install spyware to keep track of what the employees uses the computer for, which websites they visit, etc.
- **Botnet** - Spyware that makes the computer part of a botnet, ready to take orders from a remote master.

Thus, we can say that a RAT also fits into the definition of spyware, based on the fact that it does harvest information about its users. Especially the surveillance and botnet definitions appear to be the main motivations for creating RATs.

2.1.4 Summary

If we apply all the definitions above we could reasonably infer that a RAT is a backdoored spying horse zombie. This is probably an accurate definition, but it is still very broad and not of much use for detection. To pinpoint the RATs more accurately we need to look at them specifically and see how they perform the actions that puts them into the just mentioned categories. The sections below will look closer at their details and technicalities so that we can better understand how to capture their behavior.

2.2 A Brief History

But first it is time for a history lesson. The sections below will give a brief tour of some famous RATs so that we can see how this "field" has started and evolved.

2.2.1 Back Orifice and Back Orifice 2000

One of the first RATs to appear was Back Orifice, created by Sir Dystic of Cult of the Dead Cow. First released at DEF CON 6 in 1998, Back Orifice can probably be said to be one of the first and most prolific RATs out there. Its release even lead to Microsoft publishing a security bulletin trying to downplay the potential harm of BackOrifice[16, 17, 18]. Originally it was created as a demonstration of Microsoft's "*Swiss cheese approach to security*" as the press release described it[17], but BackOrifice soon became one of the best known hacking tools of the late 90's and early 2000's, gaining lots of public attention and notoriety[19].

Functionality in BackOrifice included spawning and killing processes, playing .WAV files and capturing video from connected video input devices. Back Orifice worked over a TCP/IP connection, and included a GUI control panel. It was later succeeded by Back Orifice 2000 which was last updated in 2013[20].

2.2.2 Poison Ivy

Poison Ivy[21] has been the pièce de résistance of numerous well-published attacks after it was first released in 2005[22]. One of the most well-published attacks were the so-called Nitro Attacks that happened in 2011. These were targeted attacks against various chemical, motor companies and human rights organizations[12].

Another well-published event was the SecurID attacks against RSA, which also happened in 2011. Here, a zero-day vulnerability in Adobe Flash Player was used to deliver Poison Ivy to its targets[22]. The exploit was served through a Microsoft Excel spreadsheet which used a Flash Player zero-day exploit to install Poison Ivy.

Poison Ivy was last updated in 2008, so it is interesting to note that the RAT was already three years old when it was used in these attacks. This should serve as a reminder that current tools have not been effective in detecting current threats, even those that were released three years ago.

2.2.3 Today

Today there is an abundance of different RATs, ranging from completely free ones to full commercial solutions with professional support[9]. Security journalist Brian Krebs says that commercial RATs have become so simple to use that they are designed for people who "*who wouldn't know how to hack their way out of a paper bag*". This marks a shift from RATs being used by novice hackers and people with at least a sliver of computer knowledge, to becoming a service used by people who would rather pay for someone else to do the work for them than learn how to do it themselves.

2.3 Detection Mechanisms

Now that we have seen what a RAT is and how they have evolved, it is time to look at the current solutions used for detecting them. This section will describe the current anti-malware solutions so that we can compare them with our proposed system.

2.3.1 Static Anti-Virus Detection

First, we will look at static anti-virus detection systems. Dang et al. [23] defines static analysis as "*the discipline of automatically inferring information about computer programs without running them*".

Signature-Based Anti-Virus Programs

Traditional signature-based anti-virus scanners work by looking at the files on the computer as sequences of bytes. Malware is detected by searching for specific byte patterns that are known to occur in a given piece of malware, and if this sequence is detected, the file can be flagged as being malicious and infected with the specific piece of malware that owns that byte signature[24][25][p. 55]. Wildcard characters can also be insert to ignore certain byte sequences to prevent some obfuscation attempts.

Heuristic Anti-Virus Detection

Heuristic scanning looks for code that might be suspicious instead of specific virus byte signatures. Such code might be code that does not do anything, code that modifies itself, uses undocumented APIs, and specific strings such as profanity[25][p. 69]. It is also important to look for things that malware usually does not do, such as creating pop-up dialogs[25].

Integrity Checkers

Integrity checkers compute checksums of files and detects changes by re-computing the checksum and comparing the new and the old sum. If the checksum has changed, a change must also have occurred in the file itself[25][p. 70-71]. Anti-virus programs will usually use this technique to check themselves to verify that they still have their integrity intact.

Advantages and Disadvantages of Static Detection

Grégoire et al.[26] lists the advantages and disadvantages of static detection systems:

Advantages:

- Fast algorithms for matching
- Low false positive rate

Disadvantages:

- Signature creation is slow, requires much manual work, and signature creators can quickly be overwhelmed by the sheer amount of malware
- New signatures have to be distributed to the users
- Easily fooled by small modifications and/or obfuscation techniques
- The signature database can become overwhelmingly large, and older strains have to be removed, thus leaving the users unprotected against older malware

2.3.2 Dynamic Anti-Virus Detection

The counterpart to static analysis is dynamic analysis. Dynamic analysis works by actually running the code and monitoring its behavior. Aycock divides behavior monitoring into two categories: behavior monitors and behavior blockers[25][p. 72]. Behavior monitors detect that something bad is happening, while a behavior blocker will take an additional step and try to stop it from occurring. A behavior blocker will have three ways of describing normal behavior[25][p. 72]:

- Positive detection, which is permitted actions
- Negative detection, which is disallowed actions
- A combination of the two

Aycock says that a behavior blocker does not need to examine everything that is happening on the system, but only that which is interesting from a security perspective. He also mentions a very important point, namely that the code can be as statically obfuscated as possible, but the API calls performed by the application will still appear when the program is run. This is an important foundation for our work.

Grégoire et al. have summarized the existing work on behavioral detection for detecting malware[26]. According to them, the field started out in 1986 from the work of Cohen. Cohen stated that because malware also has to use the same resources as legitimate programs, the problem of behavior detection is to find out what is legitimate use of these resources and what is not. This problem can again be reduced down to the undecidability problem when looking at the system usage as input in a Turing-machine.

Grégoire et al. states that there are two types of behavior detection[26]:

- **Modeling normal behavior** - This approach seeks to model normal system usage and defines everything that deviates from this as unusual. This is the same approach that we see in intrusion detection systems where it is called anomaly detection. They state that this approach is seldom used in virology because of high false positive rates and difficulties in modeling system behavior because of large differences in program behavior.
- **Modeling suspicious behavior** - The second approach is the complete opposite. Finding suspicious behavior is about detecting behavior that can be defined as dangerous for the system. Grégoire et al. points out that this approach is usually taken by the virology community, while the other one is used by the intrusion detection community.

2.3.3 Properties

Grégoire et al.[26] give some properties that a behavioral detection system must have:

Completeness and Accuracy

Completeness describes the system's ability to detect malware, i.e. that a system which is "complete" has a high detection rate, and one which has a high false negative rate is "incomplete". Accuracy is the false positive rate. Such a system should however be adaptable, meaning that its signatures can be updated to change the false positive and false negative rates.

Resilience

The system should be resistant against techniques such as obfuscation and attempts at hiding.

Efficiency

The monitoring system should not put an unreasonable strain on the computer it runs on, nor should it take 100 years to decide if a program is malicious or not. The paper states that it is the last point that usually is the problem today, since computers now have become so fast and cheap that it is hard to bog them down with detection systems.

Timeliness

This means that the malware is detected before it can do damage to the system.

Fault-tolerance and Unobtrusiveness

Fault-tolerance means that the system can withstand directed attacks from the malware. Unobtrusiveness means that the system must not disturb the malware's execution.

Advantages and Disadvantages of Dynamic Detection

Below is a summary of the advantages and disadvantages of dynamic detection systems[26].

Advantages:

- Small database because it only needs general behavior signatures.
- Should be able to detect all malware from the same family.

Disadvantages:

- Can only capture the current execution path, and can thus not take into account randomness or conditional branches.
- The systems can have a high false-positive rate because deciding what is malicious behavior and what is not is difficult to create rules for.

2.3.4 Intrusion Detection Systems

To supplement and replace the mentioned anti-virus systems there exists intrusion detection systems that aim to detect intrusions based on either signatures or deviations from a baseline of normal behavior. Bishop [13] describes three types of intrusion detection systems:

Anomaly Detection

Anomaly detection systems works by creating a baseline of normal system behavior. This baseline includes the usage of users, processes or groups of users, and flags deviations from this baseline as being suspicious. As the behavior of the process and users changes over time, the baseline has to be updated to avoid an increase in false positives from these new but legitimate behavioral changes[13].

Misuse Detection Systems

A misuse detection system will look for attacks that matches the sets of rules that the systems has stored. This can be done by looking at both transition and state-changes to discover the attack sequence[13].

Specification-Based Detection

Specification-based Detection monitors programs deviations from a given set of rules. If the program deviates from these then it might have been subjected to a possible intrusion[13].

2.4 Technical Details

This section describes the technical background material related to monitoring running processes and capturing their API calls. We explain what API calls are, the techniques for capturing them, and round of with an explanation of some techniques that can be used to obfuscate programs to fool current static-analysis anti-virus tools.

2.4.1 User and Kernel Mode

Modern operating systems allows a program to run in either user or kernel mode. In user mode, only a subset of the CPUs instructions are available, while in kernel mode the whole instruction set is. The instructions that are disallowed in user mode are usually related to I/O and memory protection[15]. The operating system itself runs in kernel mode.

2.4.2 System Calls

A program running in user mode has to make a system call to perform operating system services, such as writing to a file or network socket. Performing a system call invokes the **trap** instruction which again invokes the operating system who takes control and perform the desired operations on behalf of the calling program. After executing the given instructions, control is returned back to the calling program [15].

Tanenbaum describes the process as being essentially like this:

1. The user mode program wants to perform an operating system task and executes a trap instruction.
2. The operating system looks at the call and parameters to determine what the the calling program wants to do.
3. The operating system performs the correct actions and returns control to the instruction after the system call.

2.4.3 API calls

System calls are usually invoked through an Application Programming Interface (API), and not directly by performing the system call itself. This is a very important distinction, as this leads to the implementation of libraries that "hides" the real system calls behind a set of APIs that the programmer has to call instead of the more low-level API calls themselves.

On UNIX systems there is almost a one-to-one mapping between the names of the system calls and the names of the library calls, while on Windows, this is a bit more convoluted, with API calls containing several system calls[15]. The distinction between an API call and a system call is that API calls are part of the Windows API, and consist of functions such as **CreateProcess** and **CreateFile**. System calls on Windows systems are undocumented to the public but consist of such calls as **NtCreateUserProcess**, which is the system call called by the API call **CreateProcess** to make a new process[23]. The Windows API is implemented as a set of DLLs that are linked dynamically when the program runs[27].

Using an API allows Microsoft to change the underlying system calls without breaking compatibility with earlier programs and at the same time exposing a consistent API to the programmers. This does however have the caveat that it is impossible to know whether a call is actually a true system call in that it traps to the kernel, or if it all is done in user mode, since the underlying code is hidden inside the API, and the API names tell little about whether or not actual system calls are performed.

The Windows API is divided into the following major categories, as copied from Russinovich et al.[28]:

- Base Services
- Component Services
- User Interface Services
- Graphics and Multimedia Services
- Messaging and Collaboration
- Networking
- Web Services

2.4.4 Hooks

Our RAT monitoring has been performed by hooking the API calls made by the RATs. Hooking is a technique for intercepting events on a Windows system[29]. Hoglund and Butler[30] explains that there are two types of hooks - userland and kernel. Userland hooks operate on a single process running in userland, and can can be used to do things like hiding a file from a file explorer. Kernel hooks exist globally and modifies the system with the same privileges as the kernel, meaning that it can subvert the whole operating system. The sections below explains these concepts more in depth. Some of these techniques are of course quite dated, and not used anymore because they are easy to detect (and sometimes complicated to implement), but are still included for completeness.

Userland Hooks

Since most processes need to perform privileged tasks such as writing files and doing networking, they rely on built-in Windows APIs for performing these tasks. The example given by Hoglund and Butler is quite instructive for explaining how userland hooks work, so we will repeat it here.

The example is a Win32 application that reads all the files in a directory. To do this it needs to import the **Kernel32.dll** to access the **FindFirstFile** function which returns a handle to the first file in the directory. To iterate through the rest of the files, **FindNextFile** is called. Both of these exist in the **Kernel32.dll**, and this DLL is loaded when the application runs, and the memory addresses for these two functions are copied into the Import Address Table (IAT), which means that when the functions **FindFirstFile** or **FindNextFile** is encountered in the program, it jumps to the IAT, which again points into **Kernel32.dll** and executes the correct function from there. If the function that was called was **FindNextFile**, **Kernel32.dll** goes into **Ntdll.dll** and calls **NtQueryDirectoryFile**, loads up the appropriate registers with parameters, and traps into the kernel.

The clue here is that if you can access the memory of the process, you can also modify the IAT or the **Kernel32.dll** itself, and thus do things like hiding a file from view, redirect to another file, hide network connections, and so forth, by either overwriting the IAT and pointing the API calls to code of your own choosing, or else rewriting the code in the DLL. This can be done because the **Kernel32.dll** is put inside the memory of the process itself and its memory location can thus be reached.

Import Address Table Hooking

By patching the IAT in memory you can replace the address of a function in the IAT with your own, and thus take control of the program flow. The disadvantage of this technique is that it is easy to detect. And if the application uses **LoadLibrary** and **GetProcAddress** it will not work either.

Inline Function Hooking

Another method is Inline Function Hooking where you overwrite the target function instead of the address to it. This means that you avoid some of the problems with IAT hooking, such as problems with address resolving, ensuring that the call always gets hooked.

This technique is done by replacing the first bytes of the target function that will be hooked, and replacing them with the address of the rootkit's detour function instead. The address of the original function is stored, so when the rootkit code is executed it can again execute the original function, get the data returned from it, and continue on in the rootkit code. The original code in the target function is saved in what is called a trampoline[31]. The detour function is placed in the source function. Then you jump to the target function from the trampoline, and from here you return to the detour function where you receive the results from the target function and can alter them. All of this happens in memory, and thus enables the hooking to preserve the original program without rewriting any of its code[31].

Injecting a DLL using Windows Hooks

By using the function for hooking window messages, a DLL can be injected into another process. By using the **SetWindowsHookEx** call, the calling application can install a hook for a specific event, e.g. a key being pressed on the keyboard. When this event happens, the process will execute the function specified by the hook, instead of the original code.

Kernel Hooks

The next step up from the userland hook is the kernel hook. The reason for having a hook in the kernel is that it gains full control of the system, and is not as easily detectable as userland hooks because it can manipulate what the operating system sees and thus evade detection software that runs in kernel mode.

A popular kernel hooking method is to hook the system service descriptor table (SSDT). **KeServiceDescriptorTable** is a table which points to the System Service Dispatch Table and the System Service Parameter Table. The first one is a list of addresses to the system calls for Win32, POSIX and OS/2, and the second is the number of bytes for the parameters for these system calls[30]. This table can be changed so that the tables point to your functions instead of the operating system's[30].

Hook Chains

When the processes are hooked they enter what is called a "hook chain". This chain is made up of all the hooks that wants to be executed when a specific event happen, and they form a queue where control is given to the next hook in the chain when the first one has done its work. Transferring control is done with the **CallNextHookEx**[32] API call. Not calling this function however will prevent the hook control from moving on, and will hide the fact that you have hooked the process, but may cause undesirable behavior as the other processes probably expected you to play nice and receive the hook after you have finished using it[32].

2.4.5 Summary

As a quick summary, we can see that there are numerous ways of hooking into a process and manipulating it. The relation to our work is that we have to observe what a running process is doing, and therefore we have to modify it so that we can capture the API calls that it makes. This is where the hooking techniques are used. The goal of this section has been to give the reader a feel for the different ways of doing so, and thereby understand how a monitoring system interacts with the system and the monitored process.

The exact hooking technique used will be revealed when we discuss the tool used for capturing API calls in Section 4.2. We have not implemented any of these hooking techniques ourselves, we simply rely on existing tools that use them, but we still feel that it is important to mention these techniques so that we understand how we can intrude into a running process and capture its behavior.

2.4.6 Obfuscation

In our experiments we have given a short demonstration of how fast we can create new RATs that are undetectable by current anti-virus solutions, yet still remaining fully functional, and thus illustrating the danger and need for new malware detection mechanisms. We will in this section describe some common obfuscation techniques that are commonly used to make programs hard to analyze.

Dang et al. describes the goal of software obfuscation to be to produce a new program that does the same as the input program it is created from, but which is harder to analyze[23]. They divide obfuscation into two camps - data-based obfuscation and control-based obfuscation. Data-based obfuscation affects data structures, while control-based affects program flow.

Data-based Obfuscation

Data-based obfuscation affects data and attempts to obfuscate what it is. Below are different techniques for it explained.

Constant Unfolding

Dang et al. describes a compiler optimization technique called "constant folding" which seeks to replace computations that can be calculated when the program is being compiled with the result of this computation. An example is the C-statement **x = 4 * 5;**, which instead can be replaced with **x = 20** by the compiler when compiling the program, because 4 and 5 are already known constants, and therefore won't change during run-time. Constant unfolding would be the inverse operation, where a constant is replaced with a computation that creates the same value as the constant it replaced.

Dead Code Insertion

Another technique is to add meaningless variables that are normally removed by the compiler. The example given by Dang et al. is as follows:

```
int f()
{
int x, y;

x = 1; // this assignment to x is dead
y = 2; // y is not used again, so it is dead
x = 3; // x above here is not live
return x; // x is live
}
```

Here, the compiler would remove the unused instances of the variables **x** and **y** because they are dead - meaning that they will not be used in the actual running program. The obfuscation technique is to add such variables to the code, and this makes a reverse engineers job harder by requiring him to keep track of all variables and check if they are actually used in a computation or not.

Control-Based Obfuscation

By abusing the expectations of how a program will behave, obfuscated code can be created. Examples of such expectations are that **call** instructions will return, and that all code branches have a possibility of being taken.

Functions In/Out-Lining

A highly illogical call graph can be created by using inline and outline functions to add and remove code to the call graph. Inline functions adds the code of a subfunction to the caller and can quickly increase the size of the code if the subfunction is called multiple times. Outline functions does the opposite and takes a piece of code and makes it into a new function call instead.

Processor-Based Control Indirection

Different assembly calls can be abused by using them for other purposes than what they are normally used for. An example from the Dang et al. is the **call** instruction which calls a subfunction. It is assumed that this function will also return, and that the calling address actually is the entry point of a function. By not returning, the **call** instruction has instead become a **jmp**, and this should help to confuse a disassembler or human analyst.

Opaque Predicates

Opaque predicates are boolean expressions that either evaluate to true or false. What is special about them is that they use a problem that is hard to compute to decide if branches should be taken or not. This can be a mathematically hard problem, or environment variables that are constant but only known when compiling or obfuscating. And to make things even more complicated, the new branch should be bogus, but look like real code.

Inserting Junk Code

Control-flow and data-based obfuscation can also be combined. One technique is inserting junk code, where the junk code can reference invalid instructions, or branch right into the middle of valid code. A simple assembly example is given by Dang et al.:

```
01: jmp label
02: <junk>
03: label:
04: <real code>
```

2.4.7 Summary

There are several types of obfuscation techniques that can be used to confuse and mislead a disassembler or human analyst. A very practical effect of this is that signature-based anti-virus solutions can be easily evaded by changing the byte sequences of a program by using these obfuscation techniques while still maintaining the same functionality in their code. This fact is an important proponent for behavior based detection, since a programs behavior will stay the same even if the code is as obfuscated as possible[25][p. 72].

2.5 Finite State Machines

We have used finite-state machines (FSM) in our work to create the behavior models for the RATs we have analyzed. Kosoresow and Hofmeyr have argued for using finite-state machines with API calls because API calls are repeating and conform to a specific structure, and are therefore well-suited to be modeled with them[33]. This sentiment has also been echoed by Gao et al.[34].

This section describes the theory behind finite-state machines. A finite state machine is a way to model a computer or other system, and is used for things like spell checking, text search, speech recognition, and specifying network protocols[35]. A finite-state machine consists of[35]:

- a finite set of states
- a dedicated starting state
- an input alphabet
- a transition function that puts the machine in a new state for every input

There are two types of finite-state machine, finite-state machines with output and finite-state machines without output. Finite-state machines with output outputs something when changing state, such as the number 42, 5 dollars, or the letter "k". Finite-state machines with no output is the opposite and provides no output when changing states. Finite-state machines with no output are often used for recognizing a language, where the a string from the language is recognized only if it reaches one of the final end states[35].

2.5.1 Finite-State Machine with Output

Rosen gives a formal definition of a finite-state machine with output[35]:
"*A finite-state machine M = (S, I, O, f, g, s_0) consists of a finite set S of states, a finite input alphabet I, a finite output alphabet O, a transition function f that assigns to each state I, a finite output alphabet O, a transition function f that assigns to each state and input pair a new state, an output function g that assigns to each state and input pair an output, and an initial state* s_0".

Rosen provides a second definition of when a FSM recognizes, meaning accepts, an input:
"*Let M = (S, I, O, f, g, s_0) be a finite-state machine, and L $\subseteq$ I*. We say that M recognizes (or accepts) L if an input string x belongs to L if and only if the last output bit produced by M when given x as input is a 1.*"

2.5.2 Finite-State Machine without Output

Rosen also gives a formal definition of finite-state machines with no output[35]:
"*A finite-state automaton M = (S, I, f, s_0, F) consists of a finite set S of states, a finite input alphabet I, a transition function f that assigns a next state to every pair of state and input (so that f : S x I $\longrightarrow$ S), an initial or start state s_0, and a subset F of S consisting of final (or accepting states).*"

2.5.3 Using Finite-State Machines to Detect Malicious Behavior

Grégoire et al. says that finite-state machines can be used for detecting malicious behavior by modeling sequences of system calls and looking for state changes that corresponds to malicious behavior[26]. They give the following principles for such a system, as copied from them:

- "*The states S of an automaton corresponds to the internal states of the malware along their lifecycle.*"
- "*The set of input symbols defined upon the collected data which are mainly system calls.*"
- "*The transition function T describes the symbol sequences known as suspicious.*"
- "*The initial state s_0 corresponds to the beginning of the analysis.*"
- "*The set of accepting states A conveying the detection of a suspicious behavior.*"

3 Related Work

This chapter describes the related work. We have tried to get a broad overview of four main categories of related work, namely why other people have used API call monitoring, how they did it, what they looked for and how they used this to look for malware.

Most of the identified papers have malware detection as an overarching goal, by e.g. using sequences of API calls as signatures for specific pieces or families of malware. Some also look at behavior, but uses other identifiers for detecting it than we have.

The sections below detail the different previous works.

3.1 API Call Monitoring

API call monitoring as a method has been argued in favor for by Forrest et al. [36] who states that API call monitoring is a powerful technique that lets you monitor programs without recompilation or instrumentation. This makes it a very portable solution for monitoring the behavior of programs. The same is stated by Garfinkel [37] who says that API call monitoring allows almost all of the program's activity to be captured, including network traffic and file system activity.

The process of creating and implementing API call monitoring has been described by Garfinkel who detailed problems and pitfalls when building Janus, a system for monitoring and controlling API calls on a Unix operating system[37]. Janus is analogous to a firewall between the application and the operating system, but it uses API calls instead of network traffic.

Because of the complexity of the Unix API, Garfinkel encountered several problems. The five categories of problems were: "*incorrectly replicating OS semantics, overlooking indirect paths to resources, race conditions, incorrectly subsetting a complex interface, and side effects of denying API calls*"[37].

An example of this was trying to determine if a **bind** call for connecting two IP sockets should be allowed based on the protocol type used. The first approach was to read the socket's protocol during creation, but this did not account for the protocol type being modified during program execution, and thus allowed a malicious program to fool Janus by changing the protocol type of the socket after it was created.

What this means for API call monitoring as a technique is that it can be difficult to "get right" due to the large amount of context related information pertaining to API calls. Fortunately for us however, we have avoided most of the context-related problems that Garfinkel mentioned by instead looking at the API calls as static units which are not subject to any context-changes of the kind that Garfinkel experienced.

There are however disadvantages with API call monitoring. It can be subjected to something called a mimicry attack where sequences of malicious API calls are inserted into sequences of legitimate ones, so as to "hide" among the legitimate ones instead of performing all of its malicious calls at once and thus show itself as malicious[34]. This is especially a problem among systems based on N-grams[38].

API call monitoring as a technique has been frequently used for intrusion detection systems where they are used to discriminate between normal and malicious behavior. Forrest et al. [36] states that this technique is effective for detecting attacks on programs because attacks usually cause the programs to run seldom used code which creates new patterns of API calls. The types of attacks that can be detected include "*buffer overflow attacks, SYN floods, configuration errors, race conditions, and Trojan horses*" since all of these would lead to new code paths being executed.

The problem with such systems is that monitoring for seldom used code is no guarantee that something malicious is happening. Certain maintenance functions could be seldom run, but would create completely diverging code paths, and thus one would suspect that this system could lead to lots of false positives.

Canali et al.[39] have tested several malware detectors based on API call monitoring against a common set of tests. The authors observed that most proposed API call based malware scanners reported unusually high detection rates and similarly low false positive rates, but that the data sets used for testing differed greatly. They therefore wanted to create a larger dataset that could be used for comparing several different solutions against each other with the same data. Their conclusion was that the most accurate API call monitoring systems were the ones who relied on few types of calls, but also took into account the calls' arguments. Forrest et al. [36] reports on the work of Tandon and Chan who added API call parameters to their detection system[40]. This increased the accuracy, but made the system run 4-10 times slower.

3.2 Signatures

Numerous research papers have written about using API call sequences and frequencies as signatures for detecting malicious software. Their reasoning is that malware will make specific sequences of API calls when running, and that these can be used to create a unique fingerprint for that piece of malware.

Sequences of API calls are used by Faruki et al.[41] who creates fingerprints of API calls under Windows, Pu et al. [42] who uses the time intervals between API calls to create a baseline for normal system behavior, and Pungila[43] who describe an intrusion detection system that monitors API calls and their timing and measures their similarity based on frequency. Ahmed et al.[44] examines various statistical features by using machine learning algorithms. They split their dataset into viruses, worms and trojans, and noted that trojans was the most difficult to detect because they want to appear as similar to benign programs as possible, while worms were the easiest to detect. Qiao et al. [45] proposes to use the frequency of parameters and API call names with parameters to distinguish malware. Liu et al. [46] examines malware that splits itself into several DLLs.

The problem with these approaches is that they create static signatures based on frequencies and/or sequences, and does require having seen the malware sample before, or at least one that comes from the same family. New malware with the same behavior will not necessarily be detected, unless it makes the same API calls. Systems based on N-grams will also have a high false positive rate because deviations from what was seen in training can be flagged as suspicious [38]. Sekar et al. says that using finite-state machines is a good alternative to signatures based on N-grams because they can represent arbitrary length sequences with small memory requirements, while also being able to capture behavior that is not exactly similar to what was seen in training[38].

3.3 Policy

Li et al.[47] proposes a system called AGIS which creates signatures based on the API calls of the malware while it runs on the system it has infected. They claim their approach is novel in that suspicious behavior is detected by looking at breaches of user-defined policies. Such a policy can be searching the computer for email addresses, exporting log files or hooking DLLs used for keyboard input. AGIS monitors the programs API calls to find out what it is doing, and gives notice if behavior that breaches the security policy is detected. This behavior is then further examined through static analysis which looks at the code paths.

The paper describes the security policies as defining general behavior that is malicious, such as keyloggers capturing keystrokes and worms collecting email addresses.

3.4 Graphs

Kolbitsch et al. suggest using data flow from API calls to detect malicious behavior. They examine a program and creates a behavior graph based on the API calls and and their arguments[48]. A malicious program is found if its behavior graph matches the one of a known malicious behavior graph. Elhadi et al. [49] has used API call graphs and applied the Longest Common Subsequence algorithm for detection in their work.

3.5 Similarity Classification

Another common approach is to use API calls to find malware that comes from the same family. The idea is that malware from the same family will make the same API calls, which then can be used as a signature for that given malware family and thus detect new strains from it based on it making the same API calls as its other family members. Salehi et al. [50] claims that malware with the same behavior has to make the same API calls, including the same arguments. Yan et al.[51] has taken a more extensive approach by also using disassembly, PE header examination and dynamic tracing, to gather API call frequency. By using this frequency they were successful in classifying malware into different families. Sami et al.[52] extracts API calls used from executables without running them, and classifies them based on similar API calls. Zhao et al. [53] uses API calls to classify malware into families by using a Radial Basic Function in a support vector machine.

Tian et al.[54] uses the API calls of programs to both separate malware from benign files, and to classify the discovered malware into different families. The classification into different malware families was based on how often a given call occurs in a given family and then comparing it to how often the call occurs in a given program sample.

Moffie et al.[55] captures API calls to look for trojan horses. Their system is called Harrier, and looks for such things as resource abuse with the **execve** call and resources accessed with **write** and **read** calls.

Part et al.[56] proposes to detect malware by looking at which kernel objects are accessed by monitoring the API calls, and then constructing a graph based on this. This graph is compared with the graphs of known families of malware to see if the new sample belongs to any of them.

Wagener et al. [57] attempts to classify malware based on how similar two pieces are by using a phylogonetic tree which is used for showing how species branches into a family. Their approach is to monitor the behavior of a program through API calls and decide if a piece of malware is similar to an already known one.

Jang et al.[58] proposes to use API calls to measure how similar two pieces of software are to e.g. detect pirated copies. Dongjin et al.[59] does also propose using API calls to discriminate between applications by fingerprinting their frequency. They also claim that because of interleaving threads in the targeted application, API call sequences are unreliable for fingerprinting, and that call frequencies should be used instead.

3.6 Identification of Dangerous API Calls

Dunham at the SANS Institute has written a paper about analyzing malware by looking for API calls by using static and dynamic analysis tools[60]. He examined the API calls in more than 600 malware samples and ended up with this list of the most common ones:

- 571 LoadLibraryA
- 133 GetUserNameA
- 119 GetComputerNameA
- 116 GetVersionExA
- 104 GetModuleFileNameA
- 101 GetStartupInfoA

- 96 IsCharAlphaA
- 92 IsBadStringPtrA
- 84 IsCharUpperA
- 78 GetWindowTextA
- 68 IsCharAlphaNumericA
- 67 IsCharLowerA
- 67 GetWindowTextLengthA
- 38 GetModuleHandleA
- 37 MessageBoxA
- 36 GetCommandLineA
- 19 LCMapStringA
- 19 GetStringTypeA
- 19 FreeEnvironmentStringsA
- 19 ChooseFontA

Some calls are also claimed to almost always be malicious, such as **URLDownloadToFile**.

What is important to note here is that many of these API calls are common in the sense that they tell little of what the program is actually doing, except usual "behind the scenes" activities such as loading DLLs and checking the case of a character.

3.7 Behavior Classification

Bailey et al. [61] describes a method for classifying malware based on its behavior, such as files written and process created, instead of sequences and frequencies of the API calls. The observed behavior is used as a fingerprint for the malware, and the authors argue that this approach is more useful because it enables easier risk analysis and damage assessment by knowing what actions the malware could perform.

Their definition of behavior is "*non-transient state changes that the malware causes on the system*". This is captured at a level "higher" than individual API calls, since they claim that API call information is to low-level to be useful. Instead, they gather information from the event logs generated when executing the malware, and use this information to determine what the malware does. The malware's behavior profile is constructed from information such as "*spawned process names, modified registry keys, modified file names, and network connection attempts*". The malware is being run inside a virtual machine, and the system events are captured using Backtracker. To classify how close two malware behaviors were, they used normalized compression distance, which sees similar behavior as being closer together. Similar behavior can be writing files with random file names to the same location. This system can also find emerging threats by looking at how unknown malware fits in with the behavior of already known samples of malware.

Other types of behavior is captured by Kirda et al. [62] who used a combination of static and dynamic analysis to find browser helper objects (BHO) that is spyware. They classified a BHO as spyware if it, in response to browser events, captured the users behavior and used a Windows API call that could transfer away information. Gao et al. [34] have used finite-state machines for modeling program behavior. They discovered malicious behavior by looking for specific API call sequences, such as the rule for detecting a backdoor which was the sequence "**(chroot, chdir, chroot, open, write, close)**. Our work differs from theirs in that we have looked at different types of behavior.

Rieck et al. [63] have used machine learning algorithms to successfully categorize malware into different families based on their exhibited behavior. The captured malware behavior using CWSandbox which catches API calls performed by the examined application by hooking them using the Detours library. Their explanation of how to use the system is as follows: "*To apply our method in practice, it suffices to collect a large number of malware samples, analyze its behavior using a sand box environment, identify typical malware families to be classified by running a standard anti-virus software and construct a malware behavior classifier by learning single-family models using a machine learning toolbox.*"

Saxe et al. [64] has mined posts from StackOverflow.com for text strings and have compared these with occurrences in StackOverflow.com code listings to identify likely capabilities of malware. They examined 1982 pieces of malware and found that 46 of them had the ability for "webcam grabbing".

Christodorescu et al. [65] shows that malware can be detected by looking at where malware and benign programs differ, i.e. which behavior is present in the malware and not in the benign programs.

Mehdi et al.[66] monitors the API calls of running processes, and creates what the authors call an "impression" of the running process, which is a measure of its maliciousness, and if it reaches a given threshold it will be terminated. They used N-grams to represent the sequence of API calls. They checked if the N-grams were present in either the benign, malicious or both processes and named them thus after which category they were exclusively present in. Each unique N-gram is given a "goodness value". The "goodness value" spans from +1 for the benign to -1 for the malicious.

3.8 Finite-State Machines

Kosoresow and Hofmeyr [33] showed how API calls can be represented by finite-state machines in a 1997 paper. They used finite-state machines because they noted that API calls were often repeated and conformed to a specific structure. Dividing the programs behavior into smaller chunks allowed them to model the behavior and deviations from them as finite-state machines states. Modeling with finite-state machines has also been done by Sekar et al. [38].

Charlier et al. [2] wrote a paper in 1995 about using finite-state machines to create generic rules for detecting infections. A detection rule created by them was used for detecting infections of COM files. An example of such a rule was to assume that a virus would run before the original program was, meaning that it would write itself to the beginning of the infected file. An example of such a finite-state machine can be seen in Figure 2.

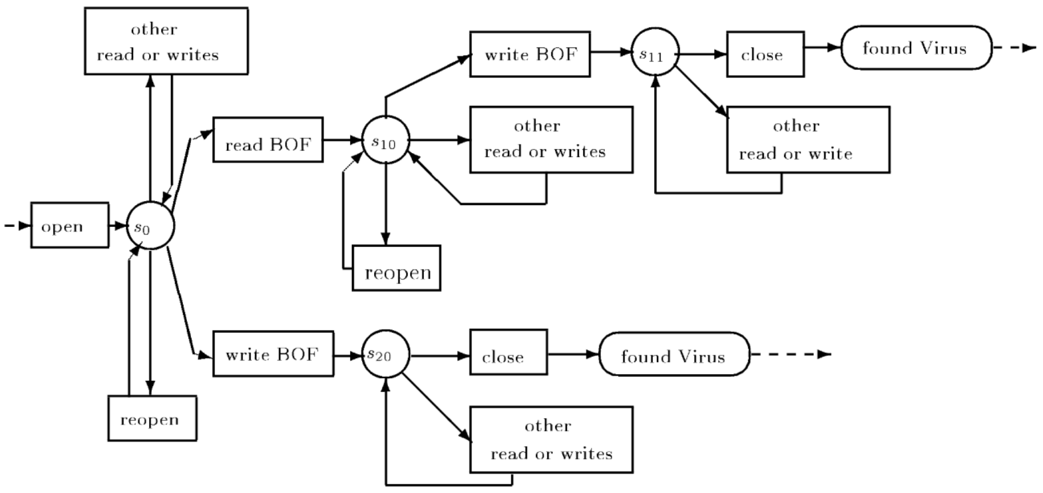


Figure 2: An example rule using a finite-state machine. Figure from Charlier et al. [2].

The difference from our work is that Charlier et al. still had as goal to detect a virus' presence based on the behavior used for infecting the system, while we look at the behavior occurring when the malicious program is running.

4 Methodology

This chapter describes our methodology for testing the RATs and discovering their behavior. The chapter is divided into two parts:

- An introduction to our testing methodology and the rationale behind it.
- A description of our test environment and the tools we have used.

4.1 Rationale for Testing

As we have seen in the previous section, there are vast amounts of existing research related to malware detection with API calls. However, little of this concentrates on modeling the behavior and actions of malware, but does rather seek to identify it based on signature extraction from the API calls and their arguments, which has been demonstrated by Gao et al. as being easily fooled by reordering the performed API calls[34].

So our approach is therefore a bit "opposite". The API calls might be easily reordered, but it is difficult is to avoid using these API calls at all [25]. If you call the API once, you can just as well call it a million times, and essentially keep the same functionality in your code. So if you forget about frequencies and sequences, and instead ask yourself "what does this API call actually *do*?", then you can create models of the examined applications that identify them based on their behavior, rather than what they are.

The next sections will explain our test setup and how we analyzed the RATs to extract their API calls and subsequently their behavior. What we have done is a qualitative study where we dissected and analyzed the RATs with the goal of learning how different types of behavior is signaled by certain API calls. So in short, the study has been an exploratory journey where we have been a bit unsure about what we would find, and therefore felt that observation would be the methodology that would give the most useful results to base future work on.

4.2 Test Setup

- **VirtualBox version:** 4.3.10
- **Windows Version:** Windows 7 x64
- **API Call Capture Tool:** Rohitab API Monitor v2 Alpha-r13

Our API calls were captured using two Windows 7 x64 virtual machines which were connected through a virtual network, with one being the RAT controller and the other the victim. The

API calls were captured using Rohitab API Monitor on the victim machine. The sections below describe the various parts of the test setup and the rationale for choosing the specific tools.

Windows 7

Windows 7 was used as a test platform because of Windows' dominant market share on desktop computers and the long-standing tradition of creating malware for it. Windows 7 has also posed itself as the inheritor to Windows XP as the stable and long-term supported Windows platform for businesses and institutions, which means that Windows 7 with all likelihood will be commonplace on important systems for years to come.

The Windows version used in our test lab was kept up-to-date with the latest security patches and had User Account Control (UAC) activated to simulate the default setup of a Windows 7 installation. None of the examined programs asked for administrative privileges using UAC (except Dark Comet during install), so they were effectively able to perform their actions without using elevated privileges.

VirtualBox

VirtualBox was chosen because it is free and open source and allows us to create virtual networks between two computers. We chose to use virtual machines in our experiments to simplify our hardware needs and system setup times, seen as we could easily revert machine states to an uninfected one.

A problem with using virtual machines however is that running programs can easily detect if they are being executed in a visualized environment, and can thus change their behavior accordingly. We have no reason to believe that this happened during our experiments, since all functionality behaved as advertised, but it is a cheap and simple trick that is often employed by malware authors to make an analyst's job slightly more difficult.

Rohitab API Monitor

Rohitab API Monitor is a Windows tool for monitoring processes and capturing their API calls[67]. It is to the best of our knowledge the API monitoring tool with the most comprehensive list of API calls that can be hooked (13 000), and was thus chosen for use in our experiments. This tool allows us to filter on individual API calls, and the APIs are categorized into logical categories that makes it very easy to find API calls for different types of behavior.

Another candidate was WinAPIOverride[68], which also monitors API calls, but our impression is that Rohitab API Monitor has a much larger collection of API calls, a better interface, and vastly better performance, in addition to being more stable. WinAPIOverride can however be scripted, a feature that is not yet available in Rohitab API Monitor.

Rohitab API Monitor does not give any documentation about what kind of hooking method it employs, and the program's authors have not responded to our request for this information. It would not be unreasonable to suspect however that the program is based upon the Microsoft Detours[31] library and the techniques described in Section 2.4.4, seen as this library is the

Microsoft-supported way of hooking processes and that the low performance impact of Rohitab API Monitor could be in-line with the performance numbers that the Detours paper reports.

ManyCam

We used ManyCam to virtualize our webcamera and microphone[69]. This allowed us to simulate having these two devices without actually having any connected physically, so instead of showing video from a capture device, a video and/or audio file was played back instead.

Obfuscation Tools

We used two different tools to obfuscate .NET-based binaries and Java RATs respectively.

Confuser

Confuser is an obfuscator for .NET programs that encrypts constants and resources, renames variables and prevents debugging and memory dumping[70]. In other words, it makes the binaries harder to detect for anti-virus programs in the same way as Confuser did.

ProGuard

ProGuard is a Java obfuscator[71]. We used it on the Java-based RATs to make them harder to detect for the anti-virus programs.

Test Network

We tested our RATs on a virtual network set up between two VirtualBox virtual machines. The setup was configured as shown in Figure 3, with one machine being the victim and the other the controller. The victim had Rohitab API Monitor installed, along with software for emulating a webcam and microphone. No firewall was present between the machines.

Our testing approach was basically to start a feature from the controller, observe the API calls on the victim, start a new feature, observe the new API calls, and so forth. Due to the large amounts of API calls being made in a short amount of time, we spent a lot of time finding out what to filter out and reading documentation from Microsoft so as to actually being able to pick out the relevant API calls that describes the features that we wanted to look for in the RATs.

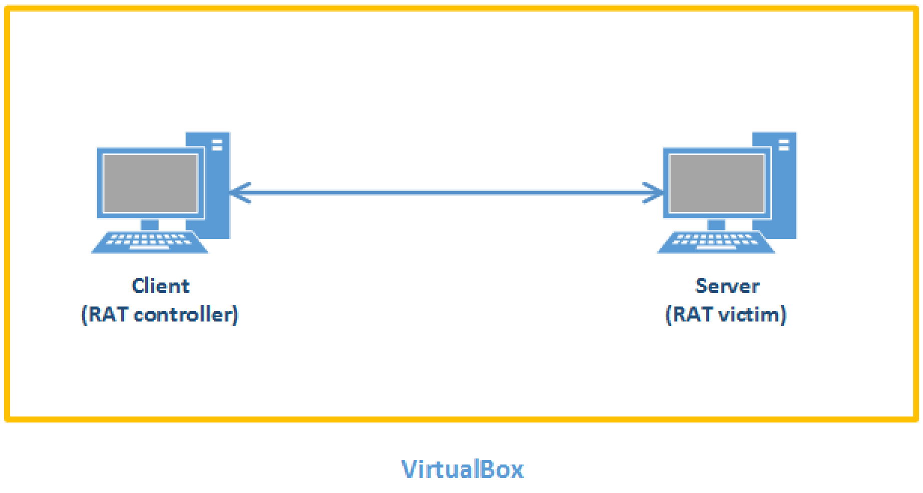


Figure 3: Our test network.

4.2.1 Experiments

We started our experiments by defining what kind of behavior we wanted to look for in the RATs. As we saw in Section 2.1, a RAT has several properties that makes them unique. Thus, we picked four different features that we felt were the most privacy invading ones, in addition to taking a more traditional IDS approach by looking at the network traffic, but instead choosing to look at it from the perspective of the API calls being made when sending and receiving the data traffic instead of looking at the data traffic itself when it is being sent over the network. Taking this approach allowed us to more accurately identify some RATs, and thus grant us a more precise detection of which RAT is present. Looking at it from the "inside" does also make us able to monitor the traffic that is sent before it is subjected to encryption of the network socket, except in cases where the encryption is applied inside the application.

The examined features were:

- Use of the webcam - We tested the webcam feature by activating the webcam viewer on the RAT controller and observing which API calls that then appear on the victim.
- Use of the microphone - We tested the microphone by activating it from the RAT and looking at the API calls that appeared on the victim.
- Logging of keystrokes - The keylogger was tested by writing text into Notepad.exe and looking at which API calls that happen in the RATs process when this is done.
- Identifiers in network traffic - The network traffic was examined when the webcam, microphone and keylogger was used.
- User interaction with the program - User interaction was tested by seeing if any API calls related to it happened when the RAT was active.

User Interaction

We also looked at which API calls that showed that the user was interacting with the analyzed program. Our theory was that the RATs would perform all the aforementioned actions, except things that involves user interactivity, since they want to stay hidden from view and will not create windows or other visible objects that the user can observe on their screen. This is not an unreasonable assumption, according to Aycock[25], who says that "visible" behavior is uncommon in malware.

5 Analysis

This section describes how we analyzed the different RATs by using the methodology described in the previous chapter. We chose to examine six different RATs that we felt offered a varied mix of both well-known RATs from published attacks, and new "under-the-radar" homebrews that have yet to show their potential. Our plan has been to pick something from the whole specter so that we can get a feeling about what kind of behavior and design that is present in the different types of RATs.

5.1 DarkComet

DarkComet is a RAT created by Jean-Pierre Lesueur under the nickname of DarkCoderSC[72]. Jean-Pierre Lesueur shut down the development of DarkComet in 2012 after several other notable RAT-creators had been arrested, stating that he did not want to be held responsible for the actions of others with regards to the use of his software[73].

DarkComet was previously available in a free and a paid version, where you could receive support and suggest features for a fee of $25 in the paid version. Currently, the only officially available version of DarkComet is the 5.4.1 Legacy version which is a free, stripped-down release, notably lacking some of the more malicious features such as the ability to compile a GUI-less server executable that can be provided to the victim[74]. Previous full-featured releases are however still being distributed on underground forums and file-sharing sites.

Among the most privacy-invading features of DarkComet is the ability for the controller to listen in on the microphone and capture video from the webcamera of its victim. It also contains standard RAT features like a keylogger and remote shell. The previous versions of DarkComet did also provide several "Fun Functions" which included such "features" as opening the CD-tray, hiding the icons on the desktop, hiding the start-button, and other useful functionality. The RAT controller could also open a chat window and chat with the victim[74].

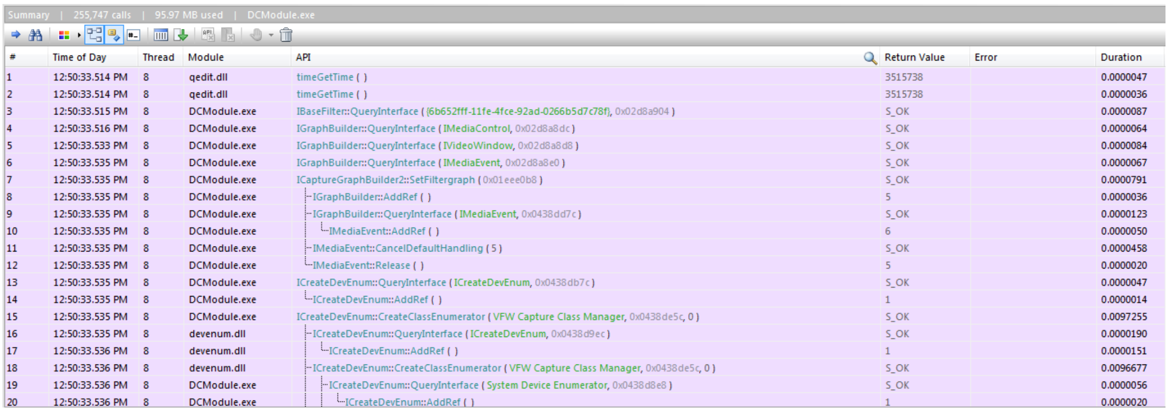
DarkComet has been used in several reported attacks, perhaps most notably in a social-engineering attack against Syrian users in 2012. DarkComet was disguised as an encryption tool for Skype, but did in reality install DarkComet and attempted to connect back to a controller in the Syrian IP range[75].

The sections below details the functionality of DarkComet and how we can detect it by looking at the API calls that is made when this functionality is performed. In addition, we look at how the network traffic can be observed by looking at the API calls used for sending and receiving data over the network. We tested the 5.4.1 Legacy version which is provided on the official Dark Comet website[72].

5.1.1 Webcam

Figure 4 shows the API calls that appear when the controller is watching the webcam of the victim. Let us take a closer look at them. There are quite a number of API calls here, but most of them can be discarded because they do not tell if the webcamera is recording or not. By trimming down the amount of calls we can get to the meat, and we end up with simply having to monitor the **IMediaSample::GetPointer**[76] API call to know if the webcamera is recording or not. This can be seen in Figure 5.

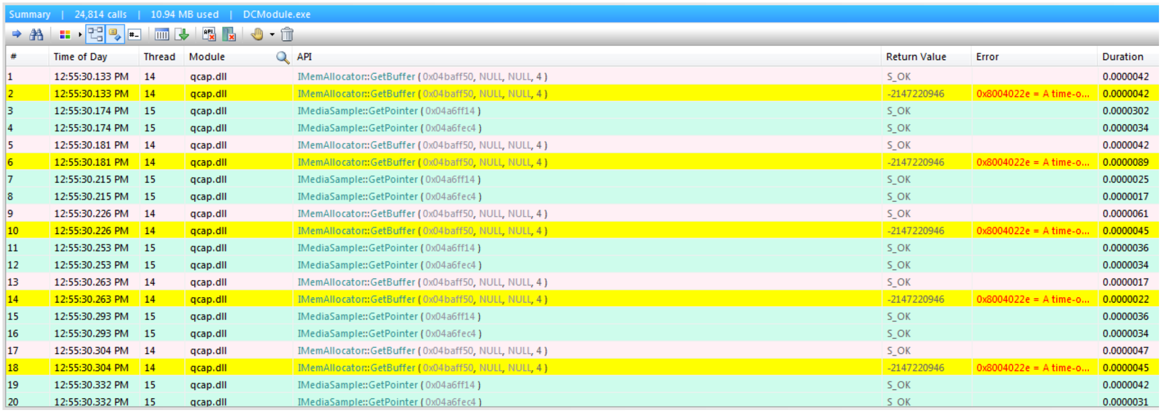
Thus, by looking at only *one* API call, we can determine whether the monitored process is currently watching the webcam.



Summary | 255,747 calls | 95.97 MB used | DCModule.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	12:50:33.514 PM	8	qedit.dll	timeGetTime ( )	3515738		0.0000047
2	12:50:33.514 PM	8	qedit.dll	timeGetTime ( )	3515738		0.0000036
3	12:50:33.515 PM	8	DCModule.exe	IBaseFilter::QueryInterface ({6b652fff-11fe-4fce-92ad-0266b5d7c78f}, 0x02d8a904)	S_OK		0.0000087
4	12:50:33.516 PM	8	DCModule.exe	IGraphBuilder::QueryInterface (IMediaControl, 0x02d8a8dc)	S_OK		0.0000064
5	12:50:33.533 PM	8	DCModule.exe	IGraphBuilder::QueryInterface (IVideoWindow, 0x02d8a8d8)	S_OK		0.0000084
6	12:50:33.535 PM	8	DCModule.exe	IGraphBuilder::QueryInterface (IMediaEvent, 0x02d8a8e0)	S_OK		0.0000067
7	12:50:33.535 PM	8	DCModule.exe	ICaptureGraphBuilder2::SetFiltergraph (0x01eee0b8)	S_OK		0.0000791
8	12:50:33.535 PM	8	DCModule.exe	├IGraphBuilder::AddRef ( )	5		0.0000036
9	12:50:33.535 PM	8	DCModule.exe	├IGraphBuilder::QueryInterface (IMediaEvent, 0x0438dd7c)	S_OK		0.0000123
10	12:50:33.535 PM	8	DCModule.exe	│└IMediaEvent::AddRef ( )	6		0.0000050
11	12:50:33.535 PM	8	DCModule.exe	├IMediaEvent::CancelDefaultHandling (5)	S_OK		0.0000458
12	12:50:33.535 PM	8	DCModule.exe	└IMediaEvent::Release ( )	5		0.0000020
13	12:50:33.535 PM	8	DCModule.exe	ICreateDevEnum::QueryInterface (ICreateDevEnum, 0x0438db7c)	S_OK		0.0000047
14	12:50:33.535 PM	8	DCModule.exe	└ICreateDevEnum::AddRef ( )	1		0.0000014
15	12:50:33.535 PM	8	DCModule.exe	ICreateDevEnum::CreateClassEnumerator (VFW Capture Class Manager, 0x0438de5c, 0)	S_OK		0.0097255
16	12:50:33.535 PM	8	devenum.dll	├ICreateDevEnum::QueryInterface (ICreateDevEnum, 0x0438d9ec)	S_OK		0.0000190
17	12:50:33.536 PM	8	devenum.dll	│└ICreateDevEnum::AddRef ( )	1		0.0000151
18	12:50:33.536 PM	8	devenum.dll	├ICreateDevEnum::CreateClassEnumerator (VFW Capture Class Manager, 0x0438de5c, 0)	S_OK		0.0096677
19	12:50:33.536 PM	8	DCModule.exe	│├ICreateDevEnum::QueryInterface (System Device Enumerator, 0x0438d8e8)	S_OK		0.0000056
20	12:50:33.536 PM	8	DCModule.exe	││└ICreateDevEnum::AddRef ( )	1		0.0000020

Figure 4: The API calls that appear on the victim when the webcam is recording.



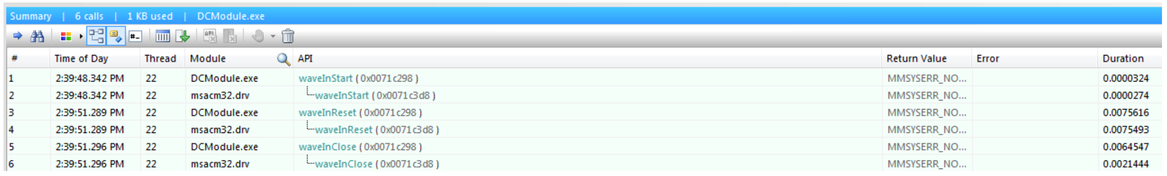
Summary | 24,814 calls | 10.94 MB used | DCModule.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	12:55:30.133 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	S_OK		0.0000042
2	12:55:30.133 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000042
3	12:55:30.174 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6ff14)	S_OK		0.0000302
4	12:55:30.174 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6fec4)	S_OK		0.0000034
5	12:55:30.181 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	S_OK		0.0000042
6	12:55:30.181 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000089
7	12:55:30.215 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6ff14)	S_OK		0.0000025
8	12:55:30.215 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6fec4)	S_OK		0.0000017
9	12:55:30.226 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	S_OK		0.0000061
10	12:55:30.226 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000045
11	12:55:30.253 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6ff14)	S_OK		0.0000036
12	12:55:30.253 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6fec4)	S_OK		0.0000034
13	12:55:30.263 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	S_OK		0.0000017
14	12:55:30.263 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000022
15	12:55:30.293 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6ff14)	S_OK		0.0000036
16	12:55:30.293 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6fec4)	S_OK		0.0000034
17	12:55:30.304 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	S_OK		0.0000047
18	12:55:30.304 PM	14	qcap.dll	IMemAllocator::GetBuffer (0x04baff50, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000045
19	12:55:30.332 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6ff14)	S_OK		0.0000042
20	12:55:30.332 PM	15	qcap.dll	IMediaSample::GetPointer (0x04a6fec4)	S_OK		0.0000031

Figure 5: The essential API calls for knowing when the webcam is recording.

5.1.2 Audio Capture

Discovering when the microphone is recording is a similar process. By removing the unnecessary API calls we are left with two that tells us when the microphone has started and stopped recording. When the microphone starts recording we see the **waveInStart**[77] API call which, as its name suggests, is called when a program starts to record audio from a given audio input device. The **waveInClose**[78] call is performed when the recording is finished. These two calls are displayed in Figure 6.

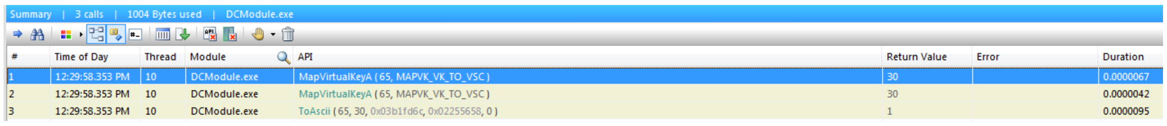

Summary | 6 calls | 1 KB used | DCModule.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	2:39:48.342 PM	22	DCModule.exe	waveInStart (0x0071c298)	MMSYSERR_NO...		0.0000324
2	2:39:48.342 PM	22	msacm32.drv	waveInStart (0x0071c3d8)	MMSYSERR_NO...		0.0000274
3	2:39:51.289 PM	22	DCModule.exe	waveInReset (0x0071c298)	MMSYSERR_NO...		0.0075616
4	2:39:51.289 PM	22	msacm32.drv	waveInReset (0x0071c3d8)	MMSYSERR_NO...		0.0075493
5	2:39:51.296 PM	22	DCModule.exe	waveInClose (0x0071c298)	MMSYSERR_NO...		0.0064547
6	2:39:51.296 PM	22	msacm32.drv	waveInClose (0x0071c3d8)	MMSYSERR_NO...		0.0021444

Figure 6: Audio recording API calls.

5.1.3 Keylogging

Detecting a keylogger can be a bit more tricky. Figure 7 shows the performed API calls when the user presses the key "**a**" in Notepad.exe. As shown in the figure, the argument value in the call **MapVirtualKey**[79] is **65**, which corresponds to the key "**a**" on the keyboard according to the ASCII-table. We can also see the call **ToAscii**[80] appear, which also contains decimal **65**.


Summary | 3 calls | 1004 Bytes used | DCModule.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	12:29:58.353 PM	10	DCModule.exe	MapVirtualKeyA (65, MAPVK_VK_TO_VSC)	30		0.0000067
2	12:29:58.353 PM	10	DCModule.exe	MapVirtualKeyA (65, MAPVK_VK_TO_VSC)	30		0.0000042
3	12:29:58.353 PM	10	DCModule.exe	ToAscii (65, 30, 0x03b1fd6c, 0x02255658, 0)	1		0.0000095

Figure 7: The API calls for the key "a".

5.1.4 Network Traffic

In addition to the indicators above, we have also looked at the network traffic as it is being sent and received through the RAT's API calls. This has allowed us to get a different angle with regards to what is going on with the RAT.

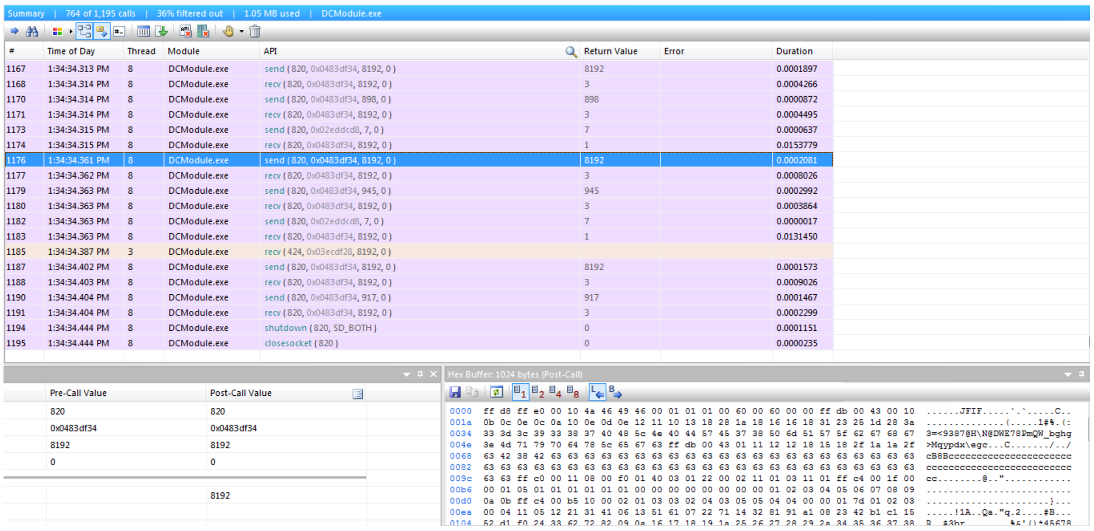
Webcam Traffic

Figure 8 shows the network traffic when the webcam is recording. As we can see, the data being sent is prepended by the word **JFIF**, which in all likelihood stands for "Jpeg File Interchange Format", and is therefore appended by binary data corresponding to frames for the video captured in the webcam. Thus, by looking at the sent network traffic, we can see when the webcam is sending away images.

Microphone Traffic

When doing a microphone capture we can see a Remote Procedure Call (RPC) being made, which contains the string "Audiosrv", as shown in Figure 9.

In Figure 10 we can see that the **send** call alternates between receiving the words "A.C", sending several "." and sometimes sending the string "EndReceive". The last **send** from the client contains an "EndReceive" before the socket is closed. Again, these are keywords that can be used to detect when the microphone is being used by the RAT.



Summary | 764 of 1,195 calls | 36% filtered out | 1.05 MB used | DCModule.exe

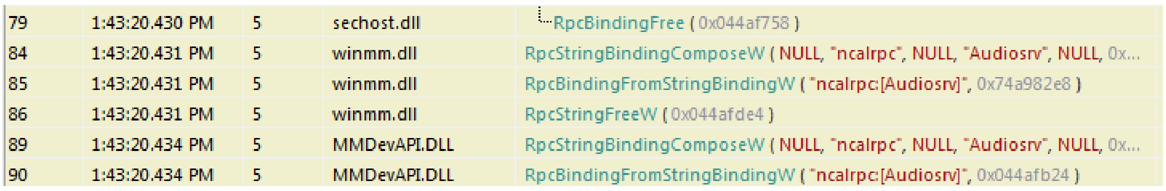
#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1167	1:34:34.313 PM	8	DCModule.exe	send (820, 0x0483df34, 8192, 0)	8192		0.0001897
1168	1:34:34.314 PM	8	DCModule.exe	recv (820, 0x0483df34, 8192, 0)	3		0.0004266
1170	1:34:34.314 PM	8	DCModule.exe	send (820, 0x0483df34, 898, 0)	898		0.0000872
1171	1:34:34.314 PM	8	DCModule.exe	recv (820, 0x0483df34, 8192, 0)	3		0.0004495
1173	1:34:34.315 PM	8	DCModule.exe	send (820, 0x02eddcd8, 7, 0)	7		0.0000637
1174	1:34:34.315 PM	8	DCModule.exe	recv (820, 0x0483df34, 8192, 0)	1		0.0153779
1176	1:34:34.361 PM	8	DCModule.exe	send (820, 0x0483df34, 8192, 0)	8192		0.0002081
1177	1:34:34.362 PM	8	DCModule.exe	recv (820, 0x0483df34, 8192, 0)	3		0.0008026
1179	1:34:34.363 PM	8	DCModule.exe	send (820, 0x0483df34, 945, 0)	945		0.0002992
1180	1:34:34.363 PM	8	DCModule.exe	recv (820, 0x0483df34, 8192, 0)	3		0.0003864
1182	1:34:34.363 PM	8	DCModule.exe	send (820, 0x02eddcd8, 7, 0)	7		0.0000017
1183	1:34:34.363 PM	8	DCModule.exe	recv (820, 0x0483df34, 8192, 0)	1		0.0131450
1185	1:34:34.387 PM	3	DCModule.exe	recv (424, 0x03ecdf28, 8192, 0)			
1187	1:34:34.402 PM	8	DCModule.exe	send (820, 0x0483df34, 8192, 0)	8192		0.0001573
1188	1:34:34.403 PM	8	DCModule.exe	recv (820, 0x0483df34, 8192, 0)	3		0.0009026
1190	1:34:34.404 PM	8	DCModule.exe	send (820, 0x0483df34, 917, 0)	917		0.0001467
1191	1:34:34.404 PM	8	DCModule.exe	recv (820, 0x0483df34, 8192, 0)	3		0.0002299
1194	1:34:34.444 PM	8	DCModule.exe	shutdown (820, SD_BOTH)	0		0.0001151
1195	1:34:34.444 PM	8	DCModule.exe	closesocket (820)	0		0.0000235

Pre-Call Value	Post-Call Value
820	820
0x0483df34	0x0483df34
8192	8192
0	0
	8192

Hex Buffer: 1024 bytes (Post-Call)

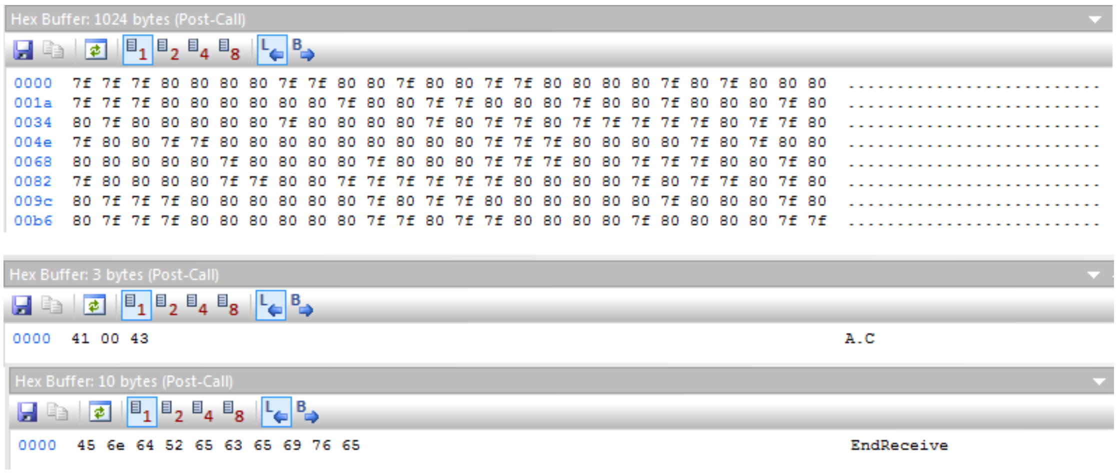
```
0000  ff d8 ff e0 00 10 4a 46 49 46 00 01 01 01 00 60 00 60 00 00 ff db 00 43 00 10  ......JFIF.....`.`.....C..
001a  0b 0c 0e 0c 0a 10 0e 0d 0e 12 11 10 13 18 28 1a 18 16 16 18 31 23 25 1d 28 3a  ..............(.....1#%.(:
0034  33 3d 3c 39 33 38 37 40 48 5c 4e 40 44 57 45 37 38 50 6d 51 57 5f 62 67 68 67  3=<9387@H\N@DWE78PmQW_bghg
004e  3e 4d 71 79 70 64 78 5c 65 67 63 ff db 00 43 01 11 12 12 18 15 18 2f 1a 1a 2f  >Mqypdx\egc...C......./../
0068  63 42 38 42 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63  cB8Bcccccccccccccccccccccc
0082  63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63  cccccccccccccccccccccccccc
009c  63 63 ff c0 00 11 08 00 f0 01 40 03 01 22 00 02 11 01 03 11 01 ff c4 00 1f 00  cc........@..".............
00b6  00 01 05 01 01 01 01 01 01 00 00 00 00 00 00 00 00 01 02 03 04 05 06 07 08 09  ..........................
00d0  0a 0b ff c4 00 b5 10 00 02 01 03 03 02 04 03 05 05 04 04 00 00 01 7d 01 02 03  .......................}...
00ea  00 04 11 05 12 21 31 41 06 13 51 61 07 22 71 14 32 81 91 a1 08 23 42 b1 c1 15  .....!1A..Qa."q.2....#B...
0104  52 d1 f0 24 33 62 72 82 09 0a 16 17 18 19 1a 25 26 27 28 29 2a 34 35 36 37 38  R..$3br.........%&'()*45678
```

Figure 8: Webcam traffic.



79	1:43:20.430 PM	5	sechost.dll	└RpcBindingFree (0x044af758)
84	1:43:20.431 PM	5	winmm.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "Audiosrv", NULL, 0x...
85	1:43:20.431 PM	5	winmm.dll	RpcBindingFromStringBindingW ("ncalrpc:[Audiosrv]", 0x74a982e8)
86	1:43:20.431 PM	5	winmm.dll	RpcStringFreeW (0x044afde4)
89	1:43:20.434 PM	5	MMDevAPI.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "Audiosrv", NULL, 0x...
90	1:43:20.434 PM	5	MMDevAPI.DLL	RpcBindingFromStringBindingW ("ncalrpc:[Audiosrv]", 0x044afb24)

Figure 9: Microphone traffic contains the word "Audiosrv".



Hex Buffer: 1024 bytes (Post-Call)

```
0000  7f 7f 7f 80 80 80 80 7f 7f 80 80 7f 80 80 7f 7f 80 80 80 80 7f 80 7f 80 80 80  ..........................
001a  7f 7f 7f 80 80 80 80 80 80 7f 80 80 7f 7f 80 80 80 7f 80 80 7f 80 80 80 7f 80  ..........................
0034  80 7f 80 80 80 80 80 7f 80 80 80 80 7f 80 7f 7f 80 7f 7f 7f 7f 7f 80 7f 7f 80  ..........................
004e  7f 80 80 7f 7f 80 80 80 80 80 80 80 80 80 7f 7f 7f 80 80 80 80 7f 80 7f 80 80  ..........................
0068  80 80 80 80 80 7f 80 80 80 80 7f 80 80 80 7f 7f 7f 80 80 7f 7f 7f 80 80 7f 80  ..........................
0082  7f 80 80 80 80 7f 7f 80 80 7f 7f 7f 7f 7f 7f 80 80 80 80 7f 80 7f 7f 80 7f 80  ..........................
009c  80 7f 7f 7f 80 80 80 80 80 80 7f 80 7f 7f 80 80 80 80 80 80 7f 80 80 80 7f 80  ..........................
00b6  80 7f 7f 7f 80 80 80 80 80 80 7f 7f 80 7f 7f 80 80 80 80 7f 80 80 80 80 7f 7f  ..........................
```

Hex Buffer: 3 bytes (Post-Call)

```
0000  41 00 43                                                                      A.C
```

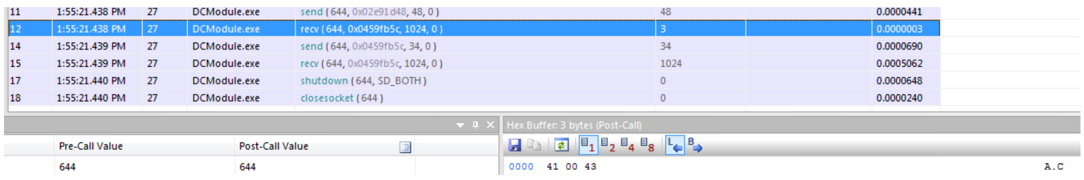
Hex Buffer: 10 bytes (Post-Call)

```
0000  45 6e 64 52 65 63 65 69 76 65                                                 EndReceive
```

Figure 10: The content of the **send** calls and **receive** calls. The first and last calls are sent, the middle received.

Keylogger Traffic

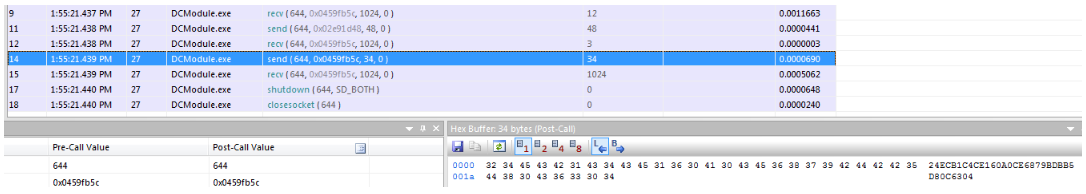
We only get scrambled data when downloading using the keylogger feature, with no traffic discerning what has been typed. We also receive the "A.C" string as we saw in the microphone traffic. This is shown in Figure 11. Figure 12 shows the data.



```
11   1:55:21.438 PM   27   DCModule.exe   send ( 644, 0x02e91d48, 48, 0 )          48     0.0000441
12   1:55:21.438 PM   27   DCModule.exe   recv ( 644, 0x0459fb5c, 1024, 0 )        3      0.0000003
14   1:55:21.439 PM   27   DCModule.exe   send ( 644, 0x0459fb5c, 34, 0 )          34     0.0000690
15   1:55:21.439 PM   27   DCModule.exe   recv ( 644, 0x0459fb5c, 1024, 0 )        1024   0.0005062
17   1:55:21.440 PM   27   DCModule.exe   shutdown ( 644, SD_BOTH )                0      0.0000648
18   1:55:21.440 PM   27   DCModule.exe   closesocket ( 644 )                      0      0.0000240

Pre-Call Value   Post-Call Value        Hex Buffer: 3 bytes (Post-Call)
644              644                    0000  41 00 43                A.C
```

Figure 11: Keylogger sends the word "A.C".



```
9    1:55:21.437 PM   27   DCModule.exe   recv ( 644, 0x0459fb5c, 1024, 0 )        12     0.0011663
11   1:55:21.438 PM   27   DCModule.exe   send ( 644, 0x02e91d48, 48, 0 )          48     0.0000441
12   1:55:21.438 PM   27   DCModule.exe   recv ( 644, 0x0459fb5c, 1024, 0 )        3      0.0000003
14   1:55:21.439 PM   27   DCModule.exe   send ( 644, 0x0459fb5c, 34, 0 )          34     0.0000690
15   1:55:21.439 PM   27   DCModule.exe   recv ( 644, 0x0459fb5c, 1024, 0 )        1024   0.0005062
17   1:55:21.440 PM   27   DCModule.exe   shutdown ( 644, SD_BOTH )                0      0.0000648
18   1:55:21.440 PM   27   DCModule.exe   closesocket ( 644 )                      0      0.0000240

Pre-Call Value   Post-Call Value        Hex Buffer: 34 bytes (Post-Call)
644              644                    0000  32 34 45 43 42 31 43 34 43 45 31 36 30 41 30 43 45 36 38 37 39 42 44 42 42 35  24ECB1C4CE160A0CE6879BDBB5
0x0459fb5c       0x0459fb5c             001a  44 38 30 43 36 33 30 34                                                        D80C6304
```

Figure 12: Keylogger data sent over the network.

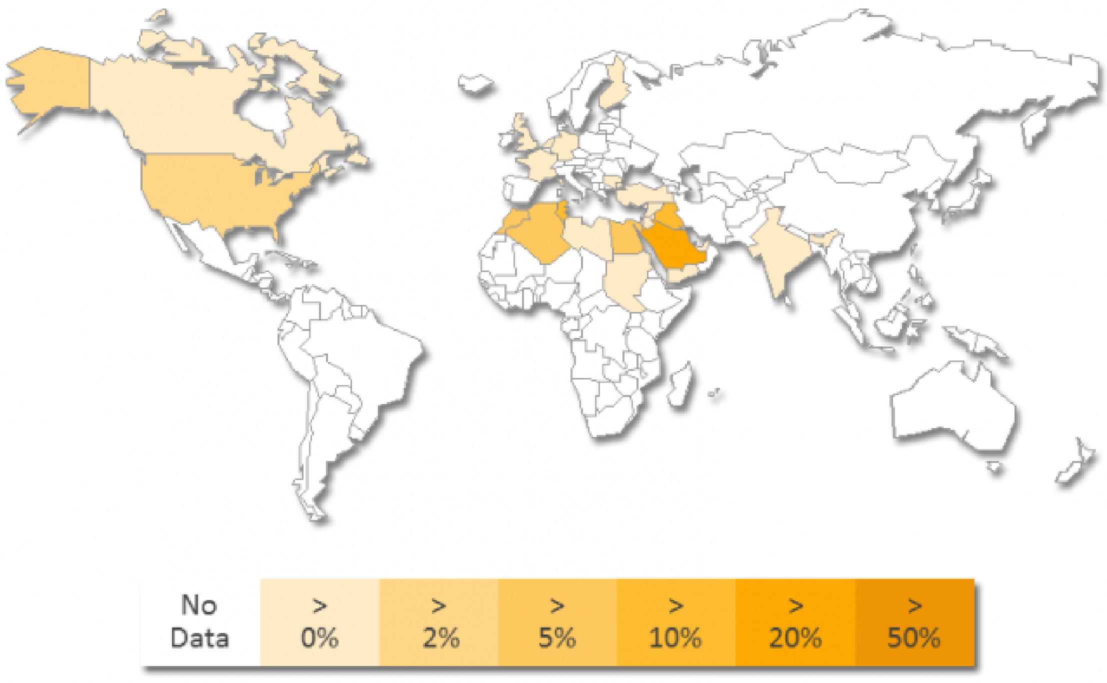


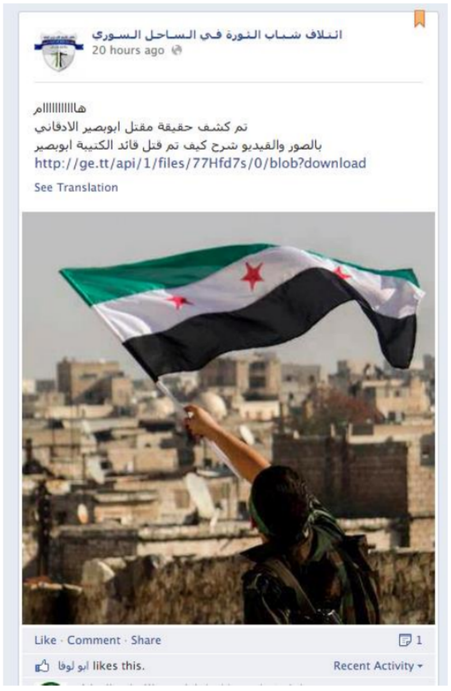
Figure 13: Spread map of njRAT from March 31st 2014. Figure by Symantec[3].

5.2 njRAT

njRAT is a RAT written in Visual Basic .NET by a Kuwaiti individual going by the nickname njq8 [81, 5]. Symantec believes that njRAT has been live since at least November 20th 2012, based on an infected screen saver containing the RAT that has been distributed since that date[3]. The public release was not before June 2013 however. Symantec have traced the command and control (C&C) servers back to IP addresses of ADSL lines, suggesting to them that most of the RAT controllers actually are home users.

The blog post from Symantec was published on the 31st of March, and said that the RAT had recently seen an upsurge in infections in the Middle East region[3]. They speculated that because the coder is an Arabic speaker, njRAT is popular in the Arabic speaking world because of the availability of tutorials and support in Arabic.

Symantec's analysis uncovered 542 command and control server domain names and 24 000 infected computers around the world, with 80% of them being in the Middle East. Figure 13 shows a map of the spread around the world. Symantec suspects that the RAT will continue to be used in the future, and obfuscated to avoid detection by anti-virus companies.
njRAT has also been used in malware campaigns in Syria, according to a joint report by EFF and Citizen Lab[4], and some of the C&C servers have been identified as belonging to the Assad regime. Figure 14 shows a Facebook post on the Facebook page of the "Revolution Youth Coalition


ائتلاف شباب الثورة في الساحل السوري
20 hours ago

هاااااااام
تم كشف حقيقة مقتل ابوبصير الادقاني
بالصور والفيديو شرح كيف تم قتل قائد الكتيبة ابوبصير
http://ge.tt/api/1/files/77Hfd7s/0/blob?download

Figure 14: Facebook message attempting to spread njRAT. Figure by EFF and Citizen Lab[4].

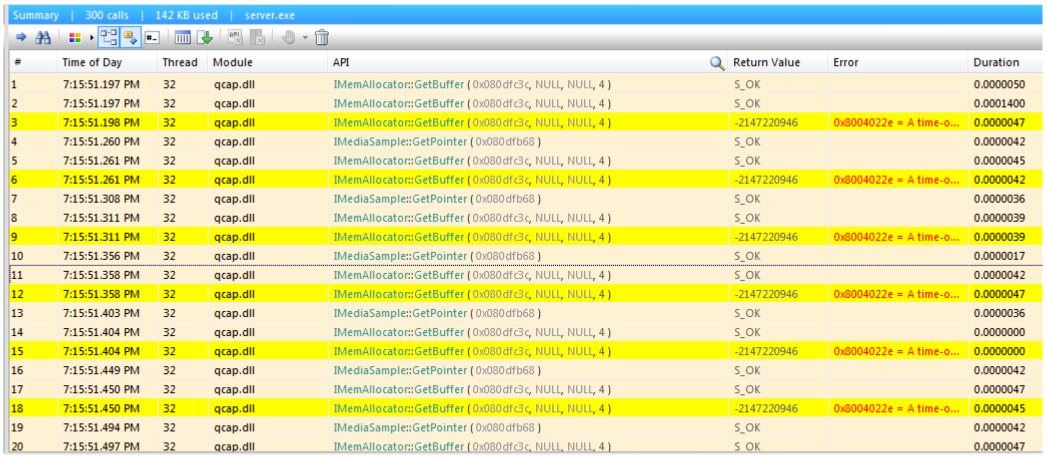
on the Syrian Cost", which is a pro-opposition group. The posted link contained a download with njRAT. The translation of the post is according to the report as follows[4]:
"*Important*
The truth about killing Abu Basir al-Adkani has been revealed.
Using photos and videos, an explanation as how Abu Basir, the battalion leader was killed."
The Facebook page appeared to be hijacked, as messages warning of the link's content were deleted.
We used njRAT 0.7, obtained from Hackforums.net in our tests[82]. Running API Monitor at njRAT's startup made the program less than cooperative and it refused to launch, but monitoring it right after it had started caused no problems.

5.2.1 Webcam

If we look at the API calls that appear when the webcam is recording, we can see that they are the same that we observed when DarkComet was using the webcam, i.e. **IMediaSample::GetPointer**. Figure 15 shows the observed API calls.

5.2.2 Microphone

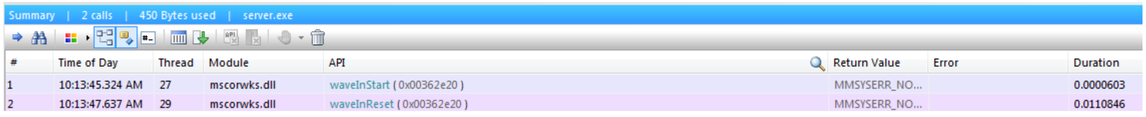
By looking at the API calls that appear when we use the microphone we can see that **waveInStart** and **waveInReset** appears. These two API calls are shown in Figure 16. In contrast to Dark Comet and the other RATs that we look at later, no **waveInStop** call is performed.



Summary | 300 calls | 142 KB used | server.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	7:15:51.197 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	S_OK		0.0000050
2	7:15:51.197 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	S_OK		0.0001400
3	7:15:51.198 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000047
4	7:15:51.260 PM	32	qcap.dll	IMediaSample::GetPointer (0x080dfb68)	S_OK		0.0000042
5	7:15:51.261 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	S_OK		0.0000045
6	7:15:51.261 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000042
7	7:15:51.308 PM	32	qcap.dll	IMediaSample::GetPointer (0x080dfb68)	S_OK		0.0000036
8	7:15:51.311 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	S_OK		0.0000039
9	7:15:51.311 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000039
10	7:15:51.356 PM	32	qcap.dll	IMediaSample::GetPointer (0x080dfb68)	S_OK		0.0000017
11	7:15:51.358 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	S_OK		0.0000042
12	7:15:51.358 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000047
13	7:15:51.403 PM	32	qcap.dll	IMediaSample::GetPointer (0x080dfb68)	S_OK		0.0000036
14	7:15:51.404 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	S_OK		0.0000000
15	7:15:51.404 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000000
16	7:15:51.449 PM	32	qcap.dll	IMediaSample::GetPointer (0x080dfb68)	S_OK		0.0000042
17	7:15:51.450 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	S_OK		0.0000047
18	7:15:51.450 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000045
19	7:15:51.494 PM	32	qcap.dll	IMediaSample::GetPointer (0x080dfb68)	S_OK		0.0000042
20	7:15:51.497 PM	32	qcap.dll	IMemAllocator::GetBuffer (0x080dfc3c, NULL, NULL, 4)	S_OK		0.0000047

Figure 15: API calls when the webcam records.



Summary | 2 calls | 450 Bytes used | server.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	10:13:45.324 AM	27	mscorwks.dll	waveInStart (0x00362e20)	MMSYSERR_NO...		0.0000603
2	10:13:47.637 AM	29	mscorwks.dll	waveInReset (0x00362e20)	MMSYSERR_NO...		0.0110846

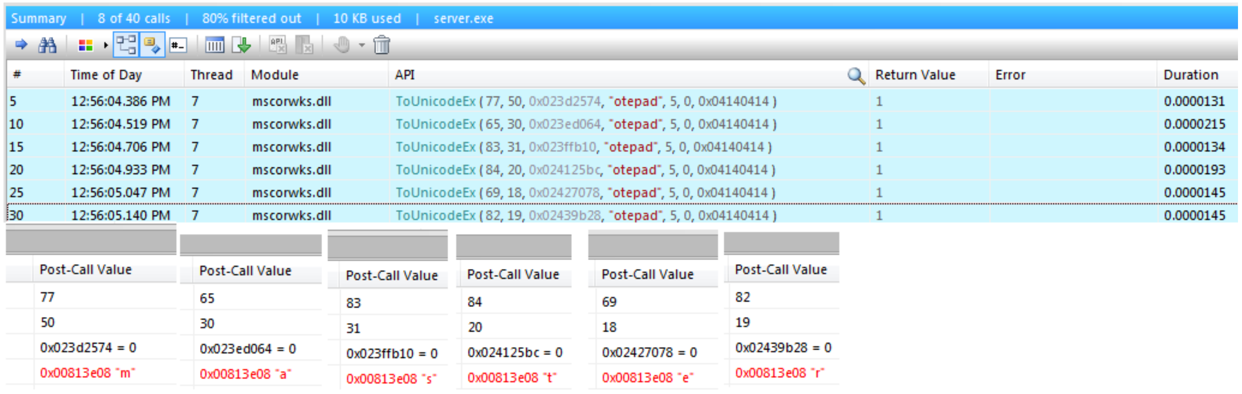
Figure 16: API calls when the microphone starts and stops recording.

5.2.3 Keylogger

Figure 17 shows the API calls that are made when we type the word **master** into Notepad.exe. As we can see, the **ToUnicode** call appears, and it contains the respective characters in each call, as shown in the bottom part of the figure. In the calls we can also see the word "otepad" appear, which probably is a reference to "Notepad" with the first character missing.

5.2.4 Network Traffic

The sections below details our findings when we examined the network traffic.

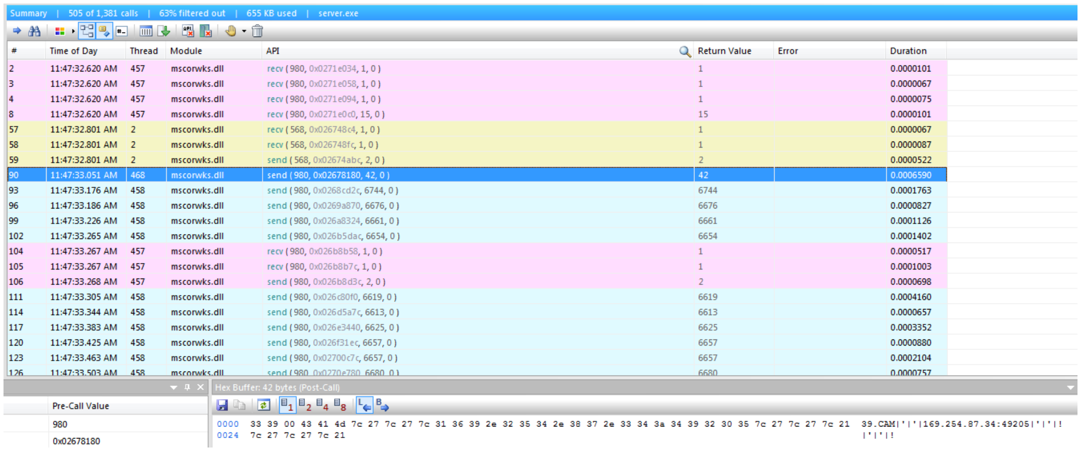


Summary | 8 of 40 calls | 80% filtered out | 10 KB used | server.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
5	12:56:04.386 PM	7	mscorwks.dll	ToUnicodeEx (77, 50, 0x023d2574, "otepad", 5, 0, 0x04140414)	1		0.0000131
10	12:56:04.519 PM	7	mscorwks.dll	ToUnicodeEx (65, 30, 0x023ed064, "otepad", 5, 0, 0x04140414)	1		0.0000215
15	12:56:04.706 PM	7	mscorwks.dll	ToUnicodeEx (83, 31, 0x023ffb10, "otepad", 5, 0, 0x04140414)	1		0.0000134
20	12:56:04.933 PM	7	mscorwks.dll	ToUnicodeEx (84, 20, 0x024125bc, "otepad", 5, 0, 0x04140414)	1		0.0000193
25	12:56:05.047 PM	7	mscorwks.dll	ToUnicodeEx (69, 18, 0x02427078, "otepad", 5, 0, 0x04140414)	1		0.0000145
30	12:56:05.140 PM	7	mscorwks.dll	ToUnicodeEx (82, 19, 0x02439b28, "otepad", 5, 0, 0x04140414)	1		0.0000145

Post-Call Value	Post-Call Value	Post-Call Value	Post-Call Value	Post-Call Value	Post-Call Value
77	65	83	84	69	82
50	30	31	20	18	19
0x023d2574 = 0	0x023ed064 = 0	0x023ffb10 = 0	0x024125bc = 0	0x02427078 = 0	0x02439b28 = 0
0x00813e08 "m"	0x00813e08 "a"	0x00813e08 "s"	0x00813e08 "t"	0x00813e08 "e"	0x00813e08 "r"

Figure 17: njRAT keylogger API calls.


Summary | 505 of 1,381 calls | 63% filtered out | 655 KB used | server.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
2	11:47:32.620 AM	457	mscorwks.dll	recv (980, 0x0271e034, 1, 0)	1		0.0000101
3	11:47:32.620 AM	457	mscorwks.dll	recv (980, 0x0271e058, 1, 0)	1		0.0000067
4	11:47:32.620 AM	457	mscorwks.dll	recv (980, 0x0271e094, 1, 0)	1		0.0000075
8	11:47:32.620 AM	457	mscorwks.dll	recv (980, 0x0271e0c0, 15, 0)	15		0.0000101
57	11:47:32.801 AM	2	mscorwks.dll	recv (568, 0x026748c4, 1, 0)	1		0.0000067
58	11:47:32.801 AM	2	mscorwks.dll	recv (568, 0x026748fc, 1, 0)	1		0.0000087
59	11:47:32.801 AM	2	mscorwks.dll	send (568, 0x02674abc, 2, 0)	2		0.0000522
90	11:47:33.051 AM	468	mscorwks.dll	send (980, 0x02678180, 42, 0)	42		0.0006590
93	11:47:33.176 AM	458	mscorwks.dll	send (980, 0x0268cd2c, 6744, 0)	6744		0.0001763
96	11:47:33.186 AM	458	mscorwks.dll	send (980, 0x0269a870, 6676, 0)	6676		0.0000827
99	11:47:33.226 AM	458	mscorwks.dll	send (980, 0x026a8324, 6661, 0)	6661		0.0001126
102	11:47:33.265 AM	458	mscorwks.dll	send (980, 0x026b5dac, 6654, 0)	6654		0.0001402
104	11:47:33.267 AM	457	mscorwks.dll	recv (980, 0x026b8b58, 1, 0)	1		0.0000517
105	11:47:33.267 AM	457	mscorwks.dll	recv (980, 0x026b8b7c, 1, 0)	1		0.0001003
106	11:47:33.268 AM	457	mscorwks.dll	send (980, 0x026b8d3c, 2, 0)	2		0.0000698
111	11:47:33.305 AM	458	mscorwks.dll	send (980, 0x026c80f0, 6619, 0)	6619		0.0004160
114	11:47:33.344 AM	458	mscorwks.dll	send (980, 0x026d5a7c, 6613, 0)	6613		0.0000657
117	11:47:33.383 AM	458	mscorwks.dll	send (980, 0x026e3440, 6625, 0)	6625		0.0003352
120	11:47:33.425 AM	458	mscorwks.dll	send (980, 0x026f31ec, 6657, 0)	6657		0.0000880
123	11:47:33.463 AM	458	mscorwks.dll	send (980, 0x02700c7c, 6657, 0)	6657		0.0002104
126	11:47:33.503 AM	458	mscorwks.dll	send (980, 0x0270e780, 6680, 0)	6680		0.0000757

	Pre-Call Value
	980
	0x02678180

Hex Buffer: 42 bytes (Post-Call)

```
0000  33 39 00 43 41 4d 7c 27 7c 27 7c 31 36 39 2e 32 35 34 2e 38 37 2e 33 34 3a 34 39 32 30 35 7c 27 7c 27 7c 21  39.CAM|'|'|169.254.87.34:49205|'|'|!
0024  7c 27 7c 27 7c 21                                                                                              |'|'|!
```

Figure 18: Network traffic with the word **CAM** in njRAT.

Webcam Traffic

After activating the camera, we can observe the network traffic in Figure 18. We also see some delimiter It is interesting to note that all the calls relating to the webcam appears to start with the keyword CAM, and this can thus be used for identifying when webcam related activities start. We can also see the synbos |'|'| appear as a delimiter sequence for the different bits of information that is being sent. This sequence can also be used to fingerprint njRAT. These symbols are also shown as appearing in Fidelis Cybersecurity Solutions report on njRAT[81].

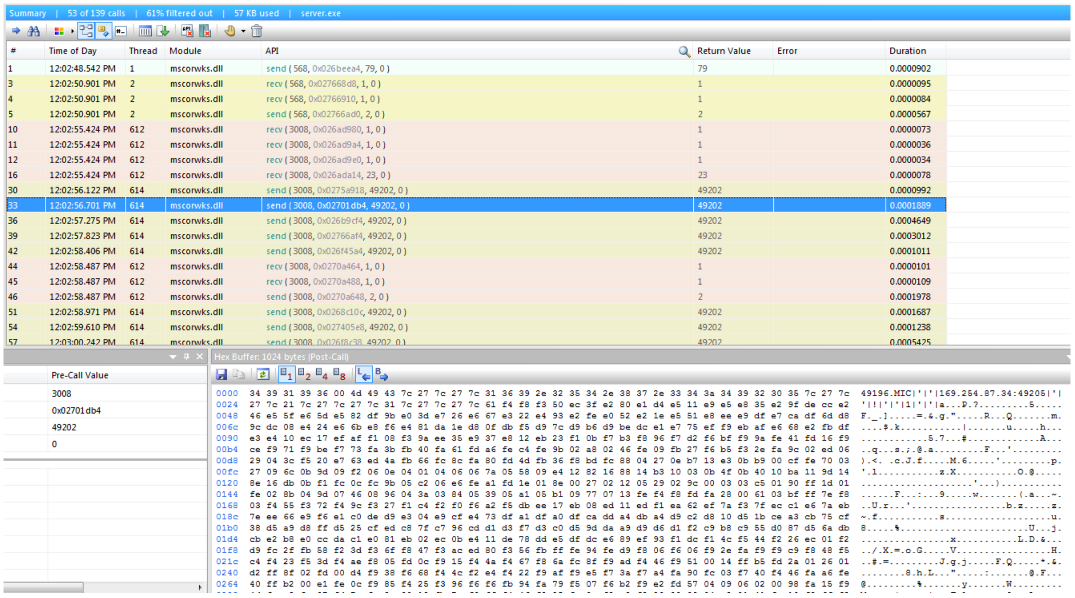
Microphone Traffic

Using the microphone, we get network traffic as shown in Figure 19. It starts with the word "MIC" and the same delimiters as with the webcam.

Keylogger Traffic

The keylogger traffic starts with the word "kl" as shown in Figure 20 and the same delimiter symbols as before.
What we also can see in the network traffic is the string below.

Y21kW0VOVEVSXQ0KDQoBMTQuM
DQuMDYgY21kIEM6XFdpbmRvd3Ncc3lzdGVtM
zJcY21kLmV4ZQENCmlwY29uZmlnDQoBMTQuM
DQuMDYgY21kIEM6XFdpbmRvd3Ncc3lzdGVtM
zJcY21kLmV4ZSAtIGlwY29uZmlnAQ0KW0VOV
EVSXQ0KYWN0aXZhdW0VOVEVSXQ0KDQoBMTQuM
DQuMDcgY21kIEM6XFdpbmRvd3Ncc3lzdGVtM
zJcY21kLmV4ZQENCmlwY29uZmlnDQoBMTQuM
DQuMDcgY21kIEM6XFdpbmRvd3Ncc3lzdGVtM
zJcY21kLmV4ZSAtIGlwY29uZmlnAQ0KW0VOV
EVSXQ0K



Summary | 53 of 139 calls | 61% filtered out | 57 KB used | server.exe

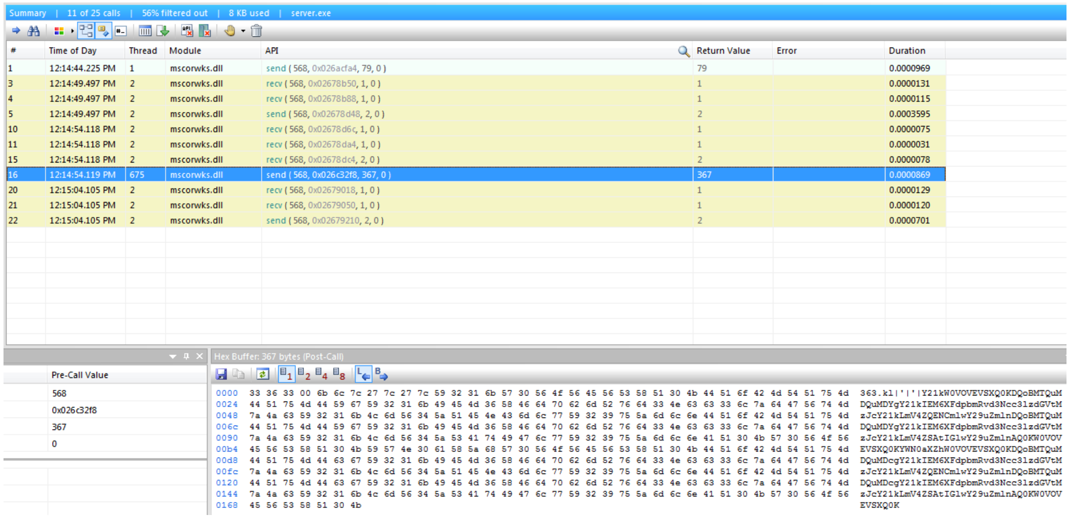
#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	12:02:48.542 PM	1	mscorwks.dll	send (568, 0x026beea4, 79, 0)	79		0.0000902
3	12:02:50.901 PM	2	mscorwks.dll	recv (568, 0x027668d8, 1, 0)	1		0.0000095
4	12:02:50.901 PM	2	mscorwks.dll	recv (568, 0x02766910, 1, 0)	1		0.0000084
5	12:02:50.901 PM	2	mscorwks.dll	send (568, 0x02766ad0, 2, 0)	2		0.0000567
10	12:02:55.424 PM	612	mscorwks.dll	recv (3008, 0x026ad980, 1, 0)	1		0.0000073
11	12:02:55.424 PM	612	mscorwks.dll	recv (3008, 0x026ad9a4, 1, 0)	1		0.0000036
12	12:02:55.424 PM	612	mscorwks.dll	recv (3008, 0x026ad9e0, 1, 0)	1		0.0000034
16	12:02:55.424 PM	612	mscorwks.dll	recv (3008, 0x026ada14, 23, 0)	23		0.0000078
30	12:02:56.122 PM	614	mscorwks.dll	send (3008, 0x0275a918, 49202, 0)	49202		0.0000992
33	12:02:56.701 PM	614	mscorwks.dll	send (3008, 0x02701db4, 49202, 0)	49202		0.0001889
36	12:02:57.275 PM	614	mscorwks.dll	send (3008, 0x026b9cf4, 49202, 0)	49202		0.0004649
39	12:02:57.823 PM	614	mscorwks.dll	send (3008, 0x02766af4, 49202, 0)	49202		0.0003012
42	12:02:58.406 PM	614	mscorwks.dll	send (3008, 0x026f45a4, 49202, 0)	49202		0.0001011
44	12:02:58.487 PM	612	mscorwks.dll	recv (3008, 0x0270a464, 1, 0)	1		0.0000101
45	12:02:58.487 PM	612	mscorwks.dll	recv (3008, 0x0270a488, 1, 0)	1		0.0000109
46	12:02:58.487 PM	612	mscorwks.dll	send (3008, 0x0270a648, 2, 0)	2		0.0001978
51	12:02:58.971 PM	614	mscorwks.dll	send (3008, 0x0268c10c, 49202, 0)	49202		0.0001687
54	12:02:59.610 PM	614	mscorwks.dll	send (3008, 0x027405e8, 49202, 0)	49202		0.0001238
57	12:03:00.242 PM	614	mscorwks.dll	send (3008, 0x026f8c38, 49202, 0)	49202		0.0005425

	Pre-Call Value
	3008
	0x02701db4
	49202
	0

Hex Buffer: 1024 bytes (Post-Call)

```
0000  34 39 31 39 36 00 4d 49 43 7c 27 7c 27 7c 31 36 39 2e 32 35 34 2e 38 37 2e 33 34 3a 34 39 32 30 35 7c 27 7c  49196.MIC|'|'|169.254.87.34:49205|'|
0024  27 7c 21 7c 27 7c 27 7c 31 7c 27 7c 27 7c 61 f4 f8 f3 50 ec 3f e2 80 e1 d4 e5 11 e9 e5 e8 35 e2 9f de cc e2  '|!|'|'|1|'|'|a...P.?.........5.....
0048  46 e5 5f e6 5d e5 82 df 9b e0 3d e7 26 e6 67 e3 22 e4 93 e2 fe e0 52 e2 1e e5 51 e8 ee e9 df e7 ca df 6d d8  F._.].....=.&.g.".....R...Q.......m.
006c  9c dc 08 e4 24 e6 6b e8 f6 e4 81 da 1e d8 0f db f5 d9 7c d9 b6 d9 be dc e1 e7 75 ef f9 eb af e6 68 e2 fb df  ....$.k...........|.......u.....h...
0090  e3 e4 10 ec 17 ef af f1 08 f3 9a ee 35 e9 37 e8 12 eb 23 f1 0b f7 b3 f8 96 f7 d2 f6 bf f9 9a fe 41 fd 16 f9  ............5.7...#.............A...
00b4  ce f9 71 f9 be f7 73 fa 3b fb 40 fa 61 fd a6 fe c4 fe 9b 02 a8 02 46 fe 09 fb 27 f6 b5 f3 2e fa 9c 02 ed 06  ..q...s.;.@.a.........F...'.........
00d8  29 04 3c f5 20 e7 63 ed 4a fb 66 fc 8c fa 80 fd 4d fb 36 f8 bd fc 88 04 27 0e b7 13 e3 0b b9 00 cf fe 70 03  ).<. .c.J.f.....M.6.....'.........p.
00fc  27 09 6c 0b 9d 09 f2 06 0e 04 01 04 06 06 7a 05 58 09 e4 12 82 16 88 14 b3 10 03 0b 4f 0b 40 10 ba 11 9d 14  '.l...........z.X...........O.@.....
0120  8e 16 db 0b f1 fc 0c fc 9b 05 c2 06 e6 fe a1 fd 1e 01 8e 00 27 02 12 05 29 02 9c 00 03 03 c5 01 90 ff 1d 01  .....................'...)..........
0144  fe 02 8b 04 9d 07 46 08 96 04 3a 03 84 05 39 05 a1 05 b1 09 77 07 13 fe f4 f8 fd fa 28 00 61 03 bf ff 7e f8  ......F...:...9.....w.......(.a...~.
0168  03 f4 55 f3 72 f4 9c f3 27 f1 c4 f2 f0 f6 a2 f5 db ee 17 eb 08 ed 11 ed f1 ea 62 ef 7a f3 7f ec c1 e6 7a eb  ..U.r...'................b.z.....z.
018c  7e ee 66 e9 f6 e1 c0 de d9 e3 04 e9 cf e4 73 df a1 df a0 df ca dd a4 db a4 d9 c2 d8 10 d5 1b ce a3 cb 75 cf  ~.f...........s....................u.
01b0  38 d5 a9 d8 ff d5 25 cf ed c8 7f c7 96 cd d1 d3 f7 d3 c0 d5 9d da a9 d9 d6 d1 f2 c9 b8 c9 55 d0 87 d5 6a db  8.....%.........................U...j.
01d4  cb e2 b8 e0 cc da c1 e0 81 eb 02 ec 0b e4 11 de 78 dd e5 df dc e6 89 ef 93 f1 dc f1 4c f5 44 f2 26 ec 01 f2  .................x..........L.D.&...
01f8  d9 fc 2f fb 58 f2 3d f3 6f f8 47 f3 ac ed 80 f3 56 fb ff fe 94 fe d9 f8 06 f6 06 f9 2e fa f9 f9 c9 f8 48 f5  ../.X.=.o.G.....V.................H.
021c  c4 f4 23 f5 3d f4 ae f8 05 fd 0c f9 15 f4 4a f4 67 f8 6a fc 8f f9 ad f4 46 f9 51 00 14 ff b5 fd 2a 01 26 01  ..#.=.........J.g.j.....F.Q.....*.&.
0240  d2 ff 8f 02 fd 00 d4 f9 38 f6 68 f4 4c f2 e4 f4 22 f9 af f9 e5 f7 3a f7 a4 fa 90 fc 03 f7 40 f4 46 fa a6 fe  ........8.h.L..."......:.......@.F...
0264  40 ff b2 00 e1 fe 0c f9 85 f4 25 f3 96 f6 f6 fb 94 fa 79 f5 07 f6 b2 f9 e2 fd 57 04 09 06 02 00 98 fa 15 f9  @.........%.......y.......W.........
```

Figure 19: Network traffic with mic in njRAT.



Summary | 11 of 25 calls | 56% filtered out | 8 KB used | server.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	12:14:44.225 PM	1	mscorwks.dll	send (568, 0x026acfa4, 79, 0)	79		0.0000969
3	12:14:49.497 PM	2	mscorwks.dll	recv (568, 0x02678b50, 1, 0)	1		0.0000131
4	12:14:49.497 PM	2	mscorwks.dll	recv (568, 0x02678b88, 1, 0)	1		0.0000115
5	12:14:49.497 PM	2	mscorwks.dll	send (568, 0x02678d48, 2, 0)	2		0.0003595
10	12:14:54.118 PM	2	mscorwks.dll	recv (568, 0x02678d6c, 1, 0)	1		0.0000075
11	12:14:54.118 PM	2	mscorwks.dll	recv (568, 0x02678da4, 1, 0)	1		0.0000031
15	12:14:54.118 PM	2	mscorwks.dll	recv (568, 0x02678dc4, 2, 0)	2		0.0000078
16	12:14:54.119 PM	675	mscorwks.dll	send (568, 0x026c32f8, 367, 0)	367		0.0000869
20	12:15:04.105 PM	2	mscorwks.dll	recv (568, 0x02679018, 1, 0)	1		0.0000129
21	12:15:04.105 PM	2	mscorwks.dll	recv (568, 0x02679050, 1, 0)	1		0.0000120
22	12:15:04.105 PM	2	mscorwks.dll	send (568, 0x02679210, 2, 0)	2		0.0000701

	Pre-Call Value
	568
	0x026c32f8
	367
	0

Hex Buffer: 367 bytes (Post-Call)

```
0000  33 36 33 00 6b 6c 7c 27 7c 27 7c 59 32 31 6b 57 30 56 4f 56 45 56 53 58 51 30 4b 44 51 6f 42 4d 54 51 75 4d  363.kl|'|'|Y21kW0VOVEVSXQ0KDQoBMTQuM
0024  44 51 75 4d 44 59 67 59 32 31 6b 49 45 4d 36 58 46 64 70 62 6d 52 76 64 33 4e 63 63 33 6c 7a 64 47 56 74 4d  DQuMDYgY21kIEM6XFdpbmRvd3Ncc3lzdGVtM
0048  7a 4a 63 59 32 31 6b 4c 6d 56 34 5a 51 45 4e 43 6d 6c 77 59 32 39 75 5a 6d 6c 6e 44 51 6f 42 4d 54 51 75 4d  zJcY21kLmV4ZQENCmlwY29uZmlnDQoBMTQuM
006c  44 51 75 4d 44 59 67 59 32 31 6b 49 45 4d 36 58 46 64 70 62 6d 52 76 64 33 4e 63 63 33 6c 7a 64 47 56 74 4d  DQuMDYgY21kIEM6XFdpbmRvd3Ncc3lzdGVtM
0090  7a 4a 63 59 32 31 6b 4c 6d 56 34 5a 53 41 74 49 47 6c 77 59 32 39 75 5a 6d 6c 6e 41 51 30 4b 57 30 56 4f 56  zJcY21kLmV4ZSAtIGlwY29uZmlnAQ0KW0VOV
00b4  45 56 53 58 51 30 4b 59 57 4e 30 61 58 5a 68 57 30 56 4f 56 45 56 53 58 51 30 4b 44 51 6f 42 4d 54 51 75 4d  EVSXQ0KYWN0aXZhW0VOVEVSXQ0KDQoBMTQuM
00d8  44 51 75 4d 44 63 67 59 32 31 6b 49 45 4d 36 58 46 64 70 62 6d 52 76 64 33 4e 63 63 33 6c 7a 64 47 56 74 4d  DQuMDcgY21kIEM6XFdpbmRvd3Ncc3lzdGVtM
00fc  7a 4a 63 59 32 31 6b 4c 6d 56 34 5a 51 45 4e 43 6d 6c 77 59 32 39 75 5a 6d 6c 6e 44 51 6f 42 4d 54 51 75 4d  zJcY21kLmV4ZQENCmlwY29uZmlnDQoBMTQuM
0120  44 51 75 4d 44 63 67 59 32 31 6b 49 45 4d 36 58 46 64 70 62 6d 52 76 64 33 4e 63 63 33 6c 7a 64 47 56 74 4d  DQuMDcgY21kIEM6XFdpbmRvd3Ncc3lzdGVtM
0144  7a 4a 63 59 32 31 6b 4c 6d 56 34 5a 53 41 74 49 47 6c 77 59 32 39 75 5a 6d 6c 6e 41 51 30 4b 57 30 56 4f 56  zJcY21kLmV4ZSAtIGlwY29uZmlnAQ0KW0VOV
0168  45 56 53 58 51 30 4b                                                                                          EVSXQ0K
```

Figure 20: The keylogger traffic contains the word "kl".

This text is base64 encoded, and decodes into the text shown below.

```
cmd[ENTER]

14.04.06 cmd C:\Windows\system32\cmd.exe
ipconfig
14.04.06 cmd C:\Windows\system32\cmd.exe – ipconfig
[ENTER]
activa[ENTER]

14.04.07 cmd C:\Windows\system32\cmd.exe
ipconfig
14.04.07 cmd C:\Windows\system32\cmd.exe – ipconfig
[ENTER]
```

This information corresponds to what we typed and which program we did it in. In this case we wrote **ipconfig** in **cmd.exe** several times.

5.2.5 The Future

In August 2013, FireEye Labs wrote about Njw0rm, a worm forked from njRAT, which added the capability to spread through USB memory sticks and similar removable devices[5]. FireEye Labs claims that njq8 is the author of this RAT as well, based on the observation that all communications with the RAT starts with "lv" and that "0njxq80" is used as a delimiter symbol. Based on FireEye Lab's dissection, njw0rm appears to have had its webcam and microphone logging ability removed, as shown in Figure 21, which displays the functionality available from the control panel in Njw0rm.

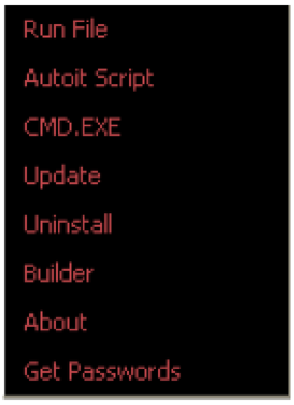


Figure 21: The functionality in Njw0rm. Picture from FireEye Labs[5].

5.3 jRAT

jRAT is a RAT written in Java, which means that it uses the Java Runtime Environment (JRE), and can thus execute on any computer that has Java installed[83]. It even has experimental support for FreeBSD.

Since jRAT is relatively new it has not been used in many well-published attacks. It has however recently been used in a spam campaign against United Kingdom and United Arab Emirates users, according to Symantec[84].

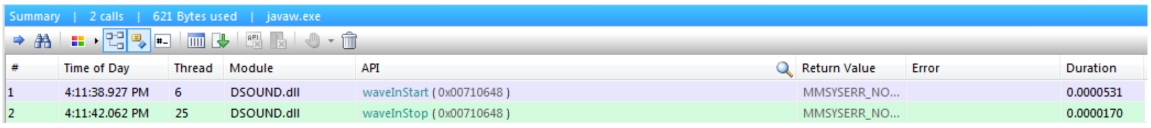
jRAT does not support webcam capture or keylogging in the free version available on their website, but a commercial version is available for $50, and the changelog mentions webcam functionality[85].

Like DarkComet, jRAT does also have a user interface, although it is limited to a small tray icon and not a full-featured GUI with windows.

5.3.1 Microphone

Despite being written in Java, the JRE still performs API calls using the Windows libraries, and we can thus capture them with our monitoring tool.

When we start the audio capturing functionality, the API calls in Figure 22 can be observed on the victim's computer. As we have seen previously, the call **waveInStart**[77] is called when the microphone starts recording, and it appears again here as well. The call **waveInClose**[78] is called when the microphone stops recording, so by monitoring just these two calls we can easily detect if the microphone is recording or not.

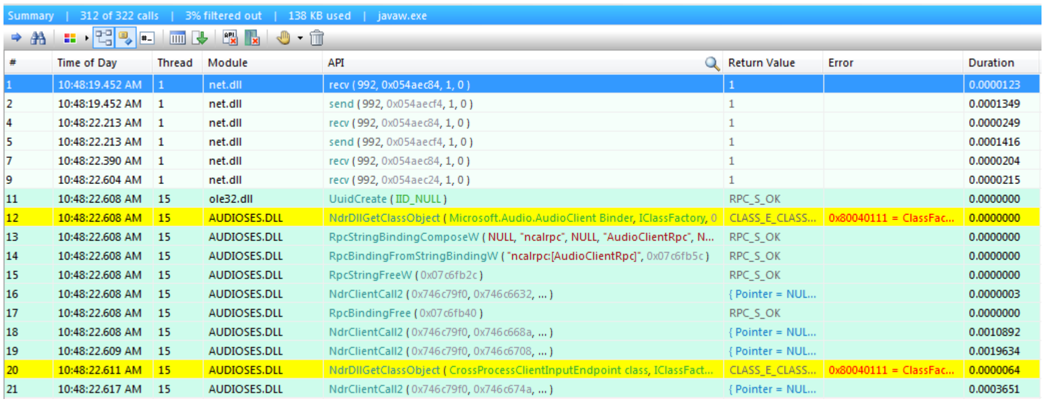

Summary | 2 calls | 621 Bytes used | javaw.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	4:11:38.927 PM	6	DSOUND.dll	waveInStart (0x00710648)	MMSYSERR_NO...		0.0000531
2	4:11:42.062 PM	25	DSOUND.dll	waveInStop (0x00710648)	MMSYSERR_NO...		0.0000170

Figure 22: API calls done when starting and stopping the microphone recording.

5.3.2 Network Traffic

If we look at the network traffic API calls that appear when we use the microphone, we can see the call **RpcStringBindingCompose** appear and that it contains the parameter "AudioClientRpc". This is shown in Figure 23. The **send** and **receive** API calls contain no information of value.



Summary | 312 of 322 calls | 3% filtered out | 138 KB used | javaw.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	10:48:19.452 AM	1	net.dll	recv (992, 0x054aec84, 1, 0)	1		0.0000123
2	10:48:19.452 AM	1	net.dll	send (992, 0x054aecf4, 1, 0)	1		0.0001349
4	10:48:22.213 AM	1	net.dll	recv (992, 0x054aec84, 1, 0)	1		0.0000249
5	10:48:22.213 AM	1	net.dll	send (992, 0x054aecf4, 1, 0)	1		0.0001416
7	10:48:22.390 AM	1	net.dll	recv (992, 0x054aec84, 1, 0)	1		0.0000204
9	10:48:22.604 AM	1	net.dll	recv (992, 0x054aec24, 1, 0)	1		0.0000215
11	10:48:22.608 AM	15	ole32.dll	UuidCreate (IID_NULL)	RPC_S_OK		0.0000000
12	10:48:22.608 AM	15	AUDIOSES.DLL	NdrDllGetClassObject (Microsoft.Audio.AudioClient Binder, IClassFactory, 0	CLASS_E_CLASS...	0x80040111 = ClassFac...	0.0000000
13	10:48:22.608 AM	15	AUDIOSES.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "AudioClientRpc", N...	RPC_S_OK		0.0000000
14	10:48:22.608 AM	15	AUDIOSES.DLL	RpcBindingFromStringBindingW ("ncalrpc:[AudioClientRpc]", 0x07c6fb5c)	RPC_S_OK		0.0000000
15	10:48:22.608 AM	15	AUDIOSES.DLL	RpcStringFreeW (0x07c6fb2c)	RPC_S_OK		0.0000000
16	10:48:22.608 AM	15	AUDIOSES.DLL	NdrClientCall2 (0x746c79f0, 0x746c6632, ...)	{ Pointer = NUL...		0.0000003
17	10:48:22.608 AM	15	AUDIOSES.DLL	RpcBindingFree (0x07c6fb40)	RPC_S_OK		0.0000000
18	10:48:22.608 AM	15	AUDIOSES.DLL	NdrClientCall2 (0x746c79f0, 0x746c668a, ...)	{ Pointer = NUL...		0.0010892
19	10:48:22.609 AM	15	AUDIOSES.DLL	NdrClientCall2 (0x746c79f0, 0x746c6708, ...)	{ Pointer = NUL...		0.0019634
20	10:48:22.611 AM	15	AUDIOSES.DLL	NdrDllGetClassObject (CrossProcessClientInputEndpoint class, IClassFact...	CLASS_E_CLASS...	0x80040111 = ClassFac...	0.0000064
21	10:48:22.617 AM	15	AUDIOSES.DLL	NdrClientCall2 (0x746c79f0, 0x746c674a, ...)	{ Pointer = NUL...		0.0003651

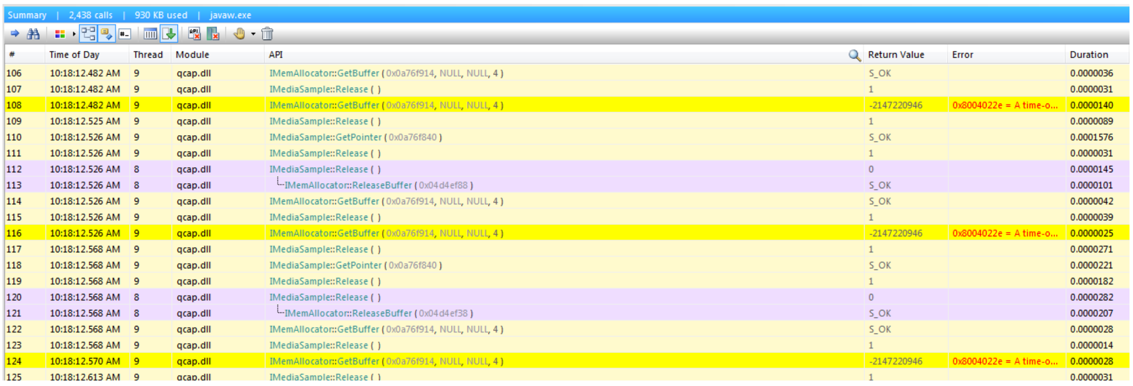
Figure 23: The microphone traffic API calls.

5.4 jSpy

jSPY is a Java-based RAT created by a 17 year old programmer from London[86]. To the best of our knowledge jSpy has not been used in any reported attacks, so it is probably most commonly used by "home users". We examined version 008 in our experiments.

5.4.1 Webcam

When starting the webcam functionality in jSpy we can again observe the same API calls as in the previous RATs. In Figure 24 we see that **IMediaSample::GetPointer** appears and indicates that the webcam is recording.



Summary | 2,438 calls | 930 KB used | javaw.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
106	10:18:12.482 AM	9	qcap.dll	IMemAllocator::GetBuffer (0x0a76f914, NULL, NULL, 4)	S_OK		0.0000036
107	10:18:12.482 AM	9	qcap.dll	IMediaSample::Release ( )	1		0.0000031
108	10:18:12.482 AM	9	qcap.dll	IMemAllocator::GetBuffer (0x0a76f914, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000140
109	10:18:12.525 AM	9	qcap.dll	IMediaSample::Release ( )	1		0.0000089
110	10:18:12.526 AM	9	qcap.dll	IMediaSample::GetPointer (0x0a76f840)	S_OK		0.0001576
111	10:18:12.526 AM	9	qcap.dll	IMediaSample::Release ( )	1		0.0000031
112	10:18:12.526 AM	8	qcap.dll	IMediaSample::Release ( )	0		0.0000145
113	10:18:12.526 AM	8	qcap.dll	└IMemAllocator::ReleaseBuffer (0x04d4ef88)	S_OK		0.0000101
114	10:18:12.526 AM	9	qcap.dll	IMemAllocator::GetBuffer (0x0a76f914, NULL, NULL, 4)	S_OK		0.0000042
115	10:18:12.526 AM	9	qcap.dll	IMediaSample::Release ( )	1		0.0000039
116	10:18:12.526 AM	9	qcap.dll	IMemAllocator::GetBuffer (0x0a76f914, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000025
117	10:18:12.568 AM	9	qcap.dll	IMediaSample::Release ( )	1		0.0000271
118	10:18:12.568 AM	9	qcap.dll	IMediaSample::GetPointer (0x0a76f840)	S_OK		0.0000221
119	10:18:12.568 AM	9	qcap.dll	IMediaSample::Release ( )	1		0.0000182
120	10:18:12.568 AM	8	qcap.dll	IMediaSample::Release ( )	0		0.0000282
121	10:18:12.568 AM	8	qcap.dll	└IMemAllocator::ReleaseBuffer (0x04d4ef38)	S_OK		0.0000207
122	10:18:12.568 AM	9	qcap.dll	IMemAllocator::GetBuffer (0x0a76f914, NULL, NULL, 4)	S_OK		0.0000028
123	10:18:12.568 AM	9	qcap.dll	IMediaSample::Release ( )	1		0.0000014
124	10:18:12.570 AM	9	qcap.dll	IMemAllocator::GetBuffer (0x0a76f914, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000028
125	10:18:12.613 AM	9	qcap.dll	IMediaSample::Release ( )	1		0.0000031

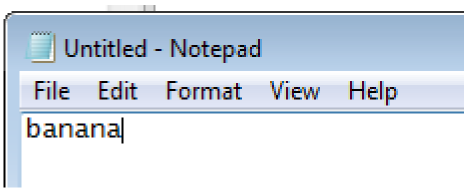
Figure 24: API calls when monitoring the webcam.

5.4.2 Microphone

jSpy does not support microphone recording.

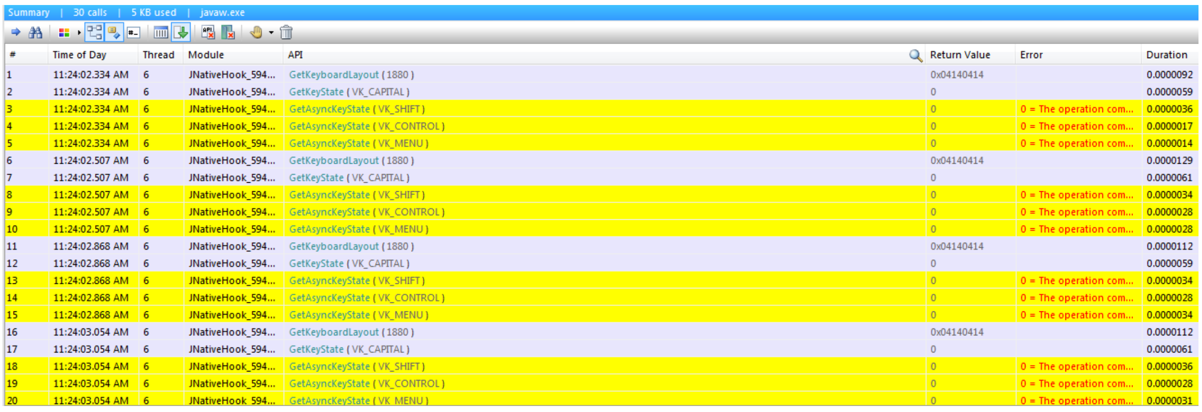
5.4.3 Keylogger

jSpy does include keylogging functionality, and in Figure 25 we can see the word "banana" being written in Notepad.exe. In Figure 26 we see the calls that happens on the victim machine as the RAT records the keystrokes when the user types "banana" in Notepad.exe. However, none of them tells exactly which key has been pressed, as we saw with the other RATs. Thus, we are unable to see exactly which keys are being pressed and can not discern that jSpy logs key presses.



banana

Figure 25: The word "banana" written in Notepad.



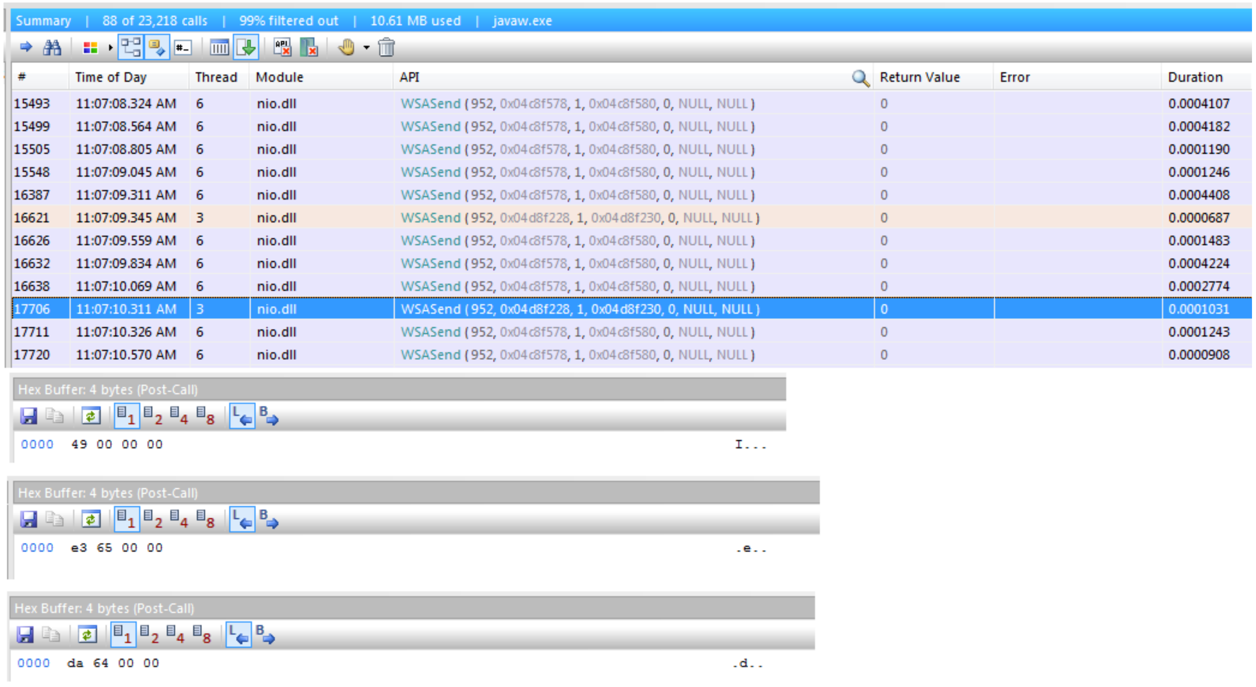
Summary | 30 calls | 5 KB used | javaw.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	11:24:02.334 AM	6	JNativeHook_594...	GetKeyboardLayout (1880)	0x04140414		0.0000092
2	11:24:02.334 AM	6	JNativeHook_594...	GetKeyState (VK_CAPITAL)	0		0.0000059
3	11:24:02.334 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_SHIFT)	0	0 = The operation com...	0.0000036
4	11:24:02.334 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_CONTROL)	0	0 = The operation com...	0.0000017
5	11:24:02.334 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_MENU)	0	0 = The operation com...	0.0000014
6	11:24:02.507 AM	6	JNativeHook_594...	GetKeyboardLayout (1880)	0x04140414		0.0000129
7	11:24:02.507 AM	6	JNativeHook_594...	GetKeyState (VK_CAPITAL)	0		0.0000061
8	11:24:02.507 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_SHIFT)	0	0 = The operation com...	0.0000034
9	11:24:02.507 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_CONTROL)	0	0 = The operation com...	0.0000028
10	11:24:02.507 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_MENU)	0	0 = The operation com...	0.0000028
11	11:24:02.868 AM	6	JNativeHook_594...	GetKeyboardLayout (1880)	0x04140414		0.0000112
12	11:24:02.868 AM	6	JNativeHook_594...	GetKeyState (VK_CAPITAL)	0		0.0000059
13	11:24:02.868 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_SHIFT)	0	0 = The operation com...	0.0000034
14	11:24:02.868 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_CONTROL)	0	0 = The operation com...	0.0000028
15	11:24:02.868 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_MENU)	0	0 = The operation com...	0.0000034
16	11:24:03.054 AM	6	JNativeHook_594...	GetKeyboardLayout (1880)	0x04140414		0.0000112
17	11:24:03.054 AM	6	JNativeHook_594...	GetKeyState (VK_CAPITAL)	0		0.0000061
18	11:24:03.054 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_SHIFT)	0	0 = The operation com...	0.0000036
19	11:24:03.054 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_CONTROL)	0	0 = The operation com...	0.0000028
20	11:24:03.054 AM	6	JNativeHook_594...	GetAsyncKeyState (VK_MENU)	0	0 = The operation com...	0.0000031

Figure 26: API calls appearing when keystrokes are entered on the victim.

5.4.4 Network Traffic

As we can see in Figure 27, no interesting information can be observed when we look at the network traffic. The content of the calls appears to be random characters, such as shown in the figure when using the webcam. The character "I" is sent a lot.



Summary | 88 of 23,218 calls | 99% filtered out | 10.61 MB used | javaw.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
15493	11:07:08.324 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0004107
15499	11:07:08.564 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0004182
15505	11:07:08.805 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0001190
15548	11:07:09.045 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0001246
16387	11:07:09.311 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0004408
16621	11:07:09.345 AM	3	nio.dll	WSASend (952, 0x04d8f228, 1, 0x04d8f230, 0, NULL, NULL)	0		0.0000687
16626	11:07:09.559 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0001483
16632	11:07:09.834 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0004224
16638	11:07:10.069 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0002774
17706	11:07:10.311 AM	3	nio.dll	WSASend (952, 0x04d8f228, 1, 0x04d8f230, 0, NULL, NULL)	0		0.0001031
17711	11:07:10.326 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0001243
17720	11:07:10.570 AM	6	nio.dll	WSASend (952, 0x04c8f578, 1, 0x04c8f580, 0, NULL, NULL)	0		0.0000908

Hex Buffer: 4 bytes (Post-Call)

```
0000  49 00 00 00                                    I...
```

Hex Buffer: 4 bytes (Post-Call)

```
0000  e3 65 00 00                                    .e..
```

Hex Buffer: 4 bytes (Post-Call)

```
0000  da 64 00 00                                    .d..
```

Figure 27: Network traffic from jSpy.

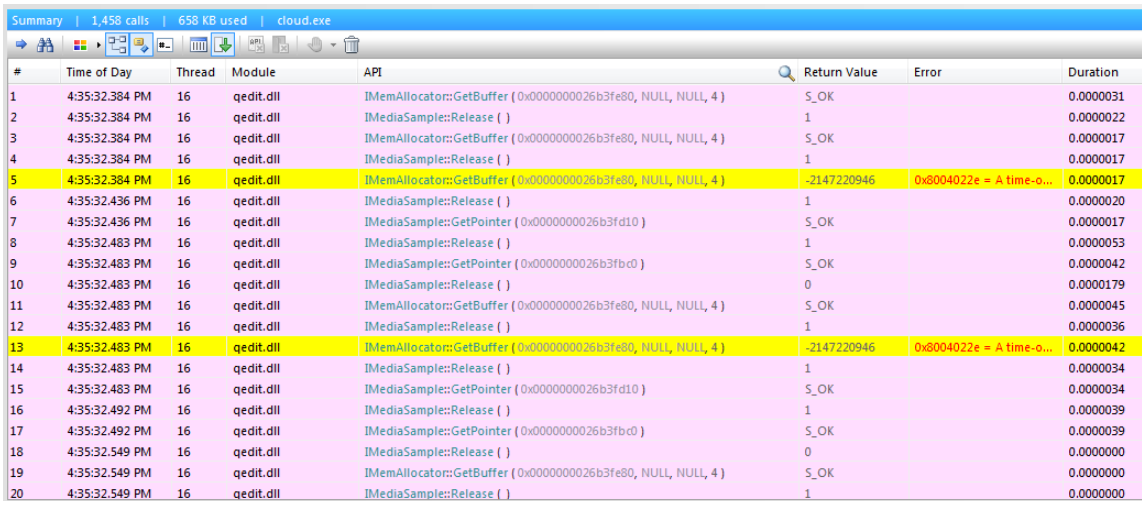
5.5 Cloud Net

Cloud Net is a freely available RAT written in C#. It is still in beta and is under constant development[87]. The last available update has been published on January 24. 2014. We tested Beta v1,5.

Cloud Net appears to implement protections against hooking by not allowing us to hook the process while it starts. But hooking it after it has started poses no problems and lets us monitor it.

5.5.1 Webcam

As shown in Figure 28, we can see the **IMediaSample::GetPointer** API calls being made when the video is recorded from the webcam.


Summary | 1,458 calls | 658 KB used | cloud.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	4:35:32.384 PM	16	qedit.dll	IMemAllocator::GetBuffer (0x000000026b3fe80, NULL, NULL, 4)	S_OK		0.0000031
2	4:35:32.384 PM	16	qedit.dll	IMediaSample::Release ( )	1		0.0000022
3	4:35:32.384 PM	16	qedit.dll	IMemAllocator::GetBuffer (0x000000026b3fe80, NULL, NULL, 4)	S_OK		0.0000017
4	4:35:32.384 PM	16	qedit.dll	IMediaSample::Release ( )	1		0.0000017
5	4:35:32.384 PM	16	qedit.dll	IMemAllocator::GetBuffer (0x000000026b3fe80, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000017
6	4:35:32.436 PM	16	qedit.dll	IMediaSample::Release ( )	1		0.0000020
7	4:35:32.436 PM	16	qedit.dll	IMediaSample::GetPointer (0x000000026b3fd10)	S_OK		0.0000017
8	4:35:32.483 PM	16	qedit.dll	IMediaSample::Release ( )	1		0.0000053
9	4:35:32.483 PM	16	qedit.dll	IMediaSample::GetPointer (0x000000026b3fbc0)	S_OK		0.0000042
10	4:35:32.483 PM	16	qedit.dll	IMediaSample::Release ( )	0		0.0000179
11	4:35:32.483 PM	16	qedit.dll	IMemAllocator::GetBuffer (0x000000026b3fe80, NULL, NULL, 4)	S_OK		0.0000045
12	4:35:32.483 PM	16	qedit.dll	IMediaSample::Release ( )	1		0.0000036
13	4:35:32.483 PM	16	qedit.dll	IMemAllocator::GetBuffer (0x000000026b3fe80, NULL, NULL, 4)	-2147220946	0x8004022e = A time-o...	0.0000042
14	4:35:32.483 PM	16	qedit.dll	IMediaSample::Release ( )	1		0.0000034
15	4:35:32.483 PM	16	qedit.dll	IMediaSample::GetPointer (0x000000026b3fd10)	S_OK		0.0000034
16	4:35:32.492 PM	16	qedit.dll	IMediaSample::Release ( )	1		0.0000039
17	4:35:32.492 PM	16	qedit.dll	IMediaSample::GetPointer (0x000000026b3fbc0)	S_OK		0.0000039
18	4:35:32.549 PM	16	qedit.dll	IMediaSample::Release ( )	0		0.0000000
19	4:35:32.549 PM	16	qedit.dll	IMemAllocator::GetBuffer (0x000000026b3fe80, NULL, NULL, 4)	S_OK		0.0000000
20	4:35:32.549 PM	16	qedit.dll	IMediaSample::Release ( )	1		0.0000000

Figure 28: API calls when the webcam starts.

5.5.2 Microphone

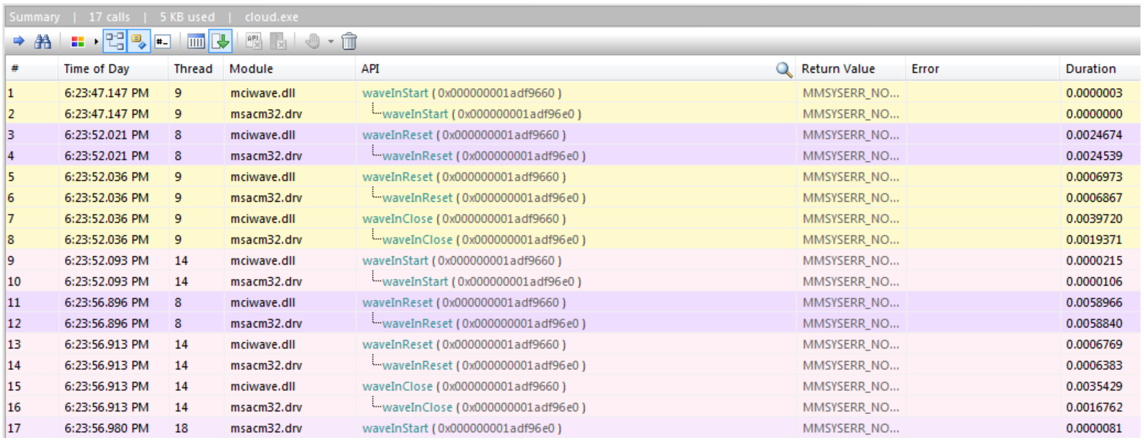
Cloud Net can also record audio through the microphone. It records in bursts of 5 seconds and saves the corresponding audio clip. Figure 29 shows the relevant API calls. Here we can see that **waveInStart** and **waveInClose** appears when the audio recording starts and stops. The reason for the multiple start and close calls is that Cloud Net records audio in bursts.

5.5.3 Keylogger

If we type the word "banana" we can see that the calls **ToUnicodeEx** appear. These calls contain our typed keystrokes, as shown in Figure 30 for the key "b". The rest of the word contains the same calls and is thus omitted.

5.5.4 Network Traffic

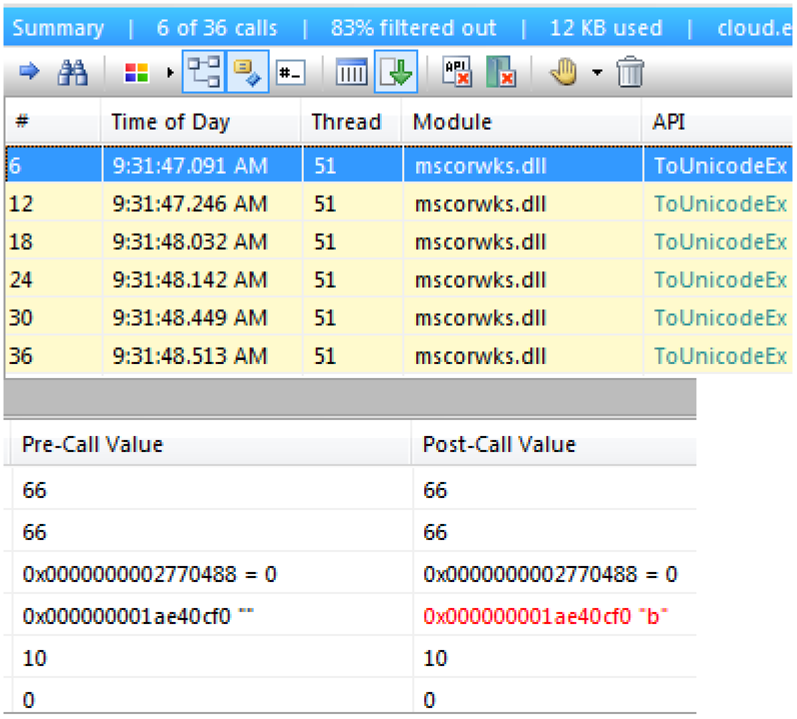
Cloud Net operates through Remote Procedure Calls (RPC), which means that we do not see any **send** or **receive** API calls.



Summary | 17 calls | 5 KB used | cloud.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	6:23:47.147 PM	9	mciwave.dll	waveInStart (0x000000001adf9660)	MMSYSERR_NO...		0.0000003
2	6:23:47.147 PM	9	msacm32.drv	└waveInStart (0x000000001adf96e0)	MMSYSERR_NO...		0.0000000
3	6:23:52.021 PM	8	mciwave.dll	waveInReset (0x000000001adf9660)	MMSYSERR_NO...		0.0024674
4	6:23:52.021 PM	8	msacm32.drv	└waveInReset (0x000000001adf96e0)	MMSYSERR_NO...		0.0024539
5	6:23:52.036 PM	9	mciwave.dll	waveInReset (0x000000001adf9660)	MMSYSERR_NO...		0.0006973
6	6:23:52.036 PM	9	msacm32.drv	└waveInReset (0x000000001adf96e0)	MMSYSERR_NO...		0.0006867
7	6:23:52.036 PM	9	mciwave.dll	waveInClose (0x000000001adf9660)	MMSYSERR_NO...		0.0039720
8	6:23:52.036 PM	9	msacm32.drv	└waveInClose (0x000000001adf96e0)	MMSYSERR_NO...		0.0019371
9	6:23:52.093 PM	14	mciwave.dll	waveInStart (0x000000001adf9660)	MMSYSERR_NO...		0.0000215
10	6:23:52.093 PM	14	msacm32.drv	└waveInStart (0x000000001adf96e0)	MMSYSERR_NO...		0.0000106
11	6:23:56.896 PM	8	mciwave.dll	waveInReset (0x000000001adf9660)	MMSYSERR_NO...		0.0058966
12	6:23:56.896 PM	8	msacm32.drv	└waveInReset (0x000000001adf96e0)	MMSYSERR_NO...		0.0058840
13	6:23:56.913 PM	14	mciwave.dll	waveInReset (0x000000001adf9660)	MMSYSERR_NO...		0.0006769
14	6:23:56.913 PM	14	msacm32.drv	└waveInReset (0x000000001adf96e0)	MMSYSERR_NO...		0.0006383
15	6:23:56.913 PM	14	mciwave.dll	waveInClose (0x000000001adf9660)	MMSYSERR_NO...		0.0035429
16	6:23:56.913 PM	14	msacm32.drv	└waveInClose (0x000000001adf96e0)	MMSYSERR_NO...		0.0016762
17	6:23:56.980 PM	18	msacm32.drv	waveInStart (0x000000001adf96e0)	MMSYSERR_NO...		0.0000081

Figure 29: The microphone capture API calls.



Summary | 6 of 36 calls | 83% filtered out | 12 KB used | cloud.e

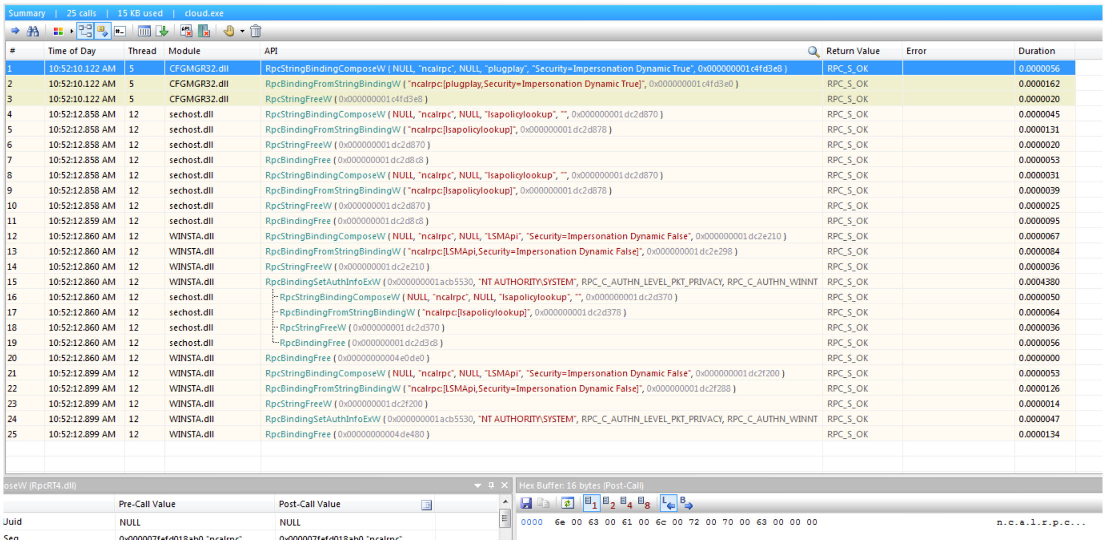
#	Time of Day	Thread	Module	API
6	9:31:47.091 AM	51	mscorwks.dll	ToUnicodeEx
12	9:31:47.246 AM	51	mscorwks.dll	ToUnicodeEx
18	9:31:48.032 AM	51	mscorwks.dll	ToUnicodeEx
24	9:31:48.142 AM	51	mscorwks.dll	ToUnicodeEx
30	9:31:48.449 AM	51	mscorwks.dll	ToUnicodeEx
36	9:31:48.513 AM	51	mscorwks.dll	ToUnicodeEx

Pre-Call Value	Post-Call Value
66	66
66	66
0x0000000002770488 = 0	0x0000000002770488 = 0
0x000000001ae40cf0 ""	0x000000001ae40cf0 "b"
10	10
0	0

Figure 30: API call for the key "b".

Webcam Network Traffic

Figure 31 shows the networking related API calls when viewing the webcam. Unfortunately, none of them appear to give any indication about webcam network traffic actually occurring.



Summary | 25 calls | 15 KB used | cloud.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	10:52:10.122 AM	5	CFGMGR32.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "plugplay", "Security=Impersonation Dynamic True", 0x000000001c4fd3e8)	RPC_S_OK		0.0000056
2	10:52:10.122 AM	5	CFGMGR32.dll	RpcBindingFromStringBindingW ("ncalrpc:[plugplay,Security=Impersonation Dynamic True]", 0x000000001c4fd3e0)	RPC_S_OK		0.0000162
3	10:52:10.122 AM	5	CFGMGR32.dll	RpcStringFreeW (0x000000001c4fd3e8)	RPC_S_OK		0.0000020
4	10:52:12.858 AM	12	sechost.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "lsapolicylookup", "", 0x000000001dc2d870)	RPC_S_OK		0.0000045
5	10:52:12.858 AM	12	sechost.dll	RpcBindingFromStringBindingW ("ncalrpc:[lsapolicylookup]", 0x000000001dc2d878)	RPC_S_OK		0.0000131
6	10:52:12.858 AM	12	sechost.dll	RpcStringFreeW (0x000000001dc2d870)	RPC_S_OK		0.0000020
7	10:52:12.858 AM	12	sechost.dll	RpcBindingFree (0x000000001dc2d8c8)	RPC_S_OK		0.0000053
8	10:52:12.858 AM	12	sechost.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "lsapolicylookup", "", 0x000000001dc2d870)	RPC_S_OK		0.0000031
9	10:52:12.858 AM	12	sechost.dll	RpcBindingFromStringBindingW ("ncalrpc:[lsapolicylookup]", 0x000000001dc2d878)	RPC_S_OK		0.0000039
10	10:52:12.858 AM	12	sechost.dll	RpcStringFreeW (0x000000001dc2d870)	RPC_S_OK		0.0000025
11	10:52:12.859 AM	12	sechost.dll	RpcBindingFree (0x000000001dc2d8c8)	RPC_S_OK		0.0000095
12	10:52:12.860 AM	12	WINSTA.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "LSMApi", "Security=Impersonation Dynamic False", 0x000000001dc2e210)	RPC_S_OK		0.0000067
13	10:52:12.860 AM	12	WINSTA.dll	RpcBindingFromStringBindingW ("ncalrpc:[LSMApi,Security=Impersonation Dynamic False]", 0x000000001dc2e298)	RPC_S_OK		0.0000084
14	10:52:12.860 AM	12	WINSTA.dll	RpcStringFreeW (0x000000001dc2e210)	RPC_S_OK		0.0000036
15	10:52:12.860 AM	12	WINSTA.dll	RpcBindingSetAuthInfoExW (0x000000001acb5530, "NT AUTHORITY\SYSTEM", RPC_C_AUTHN_LEVEL_PKT_PRIVACY, RPC_C_AUTHN_WINNT	RPC_S_OK		0.0004380
16	10:52:12.860 AM	12	sechost.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "lsapolicylookup", "", 0x000000001dc2d370)	RPC_S_OK		0.0000050
17	10:52:12.860 AM	12	sechost.dll	RpcBindingFromStringBindingW ("ncalrpc:[lsapolicylookup]", 0x000000001dc2d378)	RPC_S_OK		0.0000064
18	10:52:12.860 AM	12	sechost.dll	RpcStringFreeW (0x000000001dc2d370)	RPC_S_OK		0.0000036
19	10:52:12.860 AM	12	sechost.dll	RpcBindingFree (0x000000001dc2d3c8)	RPC_S_OK		0.0000056
20	10:52:12.860 AM	12	WINSTA.dll	RpcBindingFree (0x00000000004e0de0)	RPC_S_OK		0.0000000
21	10:52:12.899 AM	12	WINSTA.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "LSMApi", "Security=Impersonation Dynamic False", 0x000000001dc2f200)	RPC_S_OK		0.0000053
22	10:52:12.899 AM	12	WINSTA.dll	RpcBindingFromStringBindingW ("ncalrpc:[LSMApi,Security=Impersonation Dynamic False]", 0x000000001dc2f288)	RPC_S_OK		0.0000126
23	10:52:12.899 AM	12	WINSTA.dll	RpcStringFreeW (0x000000001dc2f200)	RPC_S_OK		0.0000014
24	10:52:12.899 AM	12	WINSTA.dll	RpcBindingSetAuthInfoExW (0x000000001acb5530, "NT AUTHORITY\SYSTEM", RPC_C_AUTHN_LEVEL_PKT_PRIVACY, RPC_C_AUTHN_WINNT	RPC_S_OK		0.0000047
25	10:52:12.899 AM	12	WINSTA.dll	RpcBindingFree (0x00000000004de480)	RPC_S_OK		0.0000134

oseW (RpcRT4.dll)

	Pre-Call Value	Post-Call Value
Uuid	NULL	NULL
Seq	0x000007fefd018ab0 "ncalrpc"	0x000007fefd018ab0 "ncalrpc"

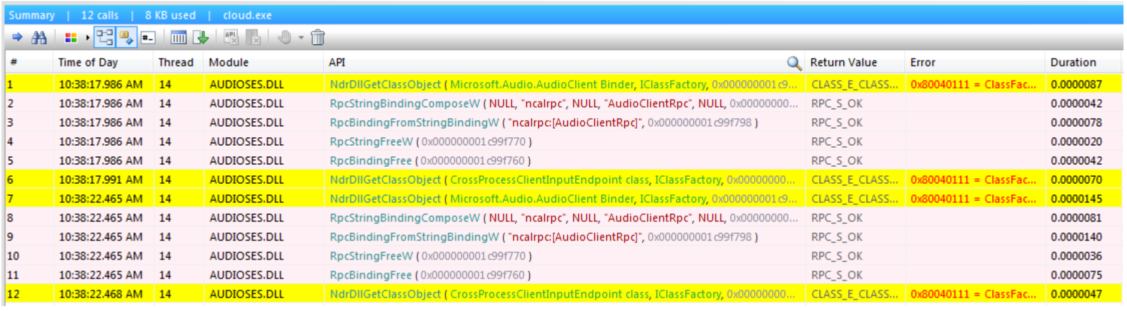
Hex Buffer: 16 bytes (Post-Call)

```
0000  6e 00 63 00 61 00 6c 00 72 00 70 00 63 00 00 00            n.c.a.l.r.p.c...
```

Figure 31: Network traffic when viewing the webcam.

Audio Network Traffic

Figure 32 shows the API calls when recording the audio from the microphone. As we can see, there are references to "AudioClientRpc" in the calls **RpcStringBindingCompose** and **RpcBindingFromStringBinding**.



Summary | 12 calls | 8 KB used | cloud.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	10:38:17.986 AM	14	AUDIOSES.DLL	NdrDllGetClassObject (Microsoft.Audio.AudioClient Binder, IClassFactory, 0x000000001c9...	CLASS_E_CLASS...	0x80040111 = ClassFac...	0.0000087
2	10:38:17.986 AM	14	AUDIOSES.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "AudioClientRpc", NULL, 0x00000000...	RPC_S_OK		0.0000042
3	10:38:17.986 AM	14	AUDIOSES.DLL	RpcBindingFromStringBindingW ("ncalrpc:[AudioClientRpc]", 0x000000001c99f798)	RPC_S_OK		0.0000078
4	10:38:17.986 AM	14	AUDIOSES.DLL	RpcStringFreeW (0x000000001c99f770)	RPC_S_OK		0.0000020
5	10:38:17.986 AM	14	AUDIOSES.DLL	RpcBindingFree (0x000000001c99f760)	RPC_S_OK		0.0000042
6	10:38:17.991 AM	14	AUDIOSES.DLL	NdrDllGetClassObject (CrossProcessClientInputEndpoint class, IClassFactory, 0x00000000...	CLASS_E_CLASS...	0x80040111 = ClassFac...	0.0000070
7	10:38:22.465 AM	14	AUDIOSES.DLL	NdrDllGetClassObject (Microsoft.Audio.AudioClient Binder, IClassFactory, 0x000000001c9...	CLASS_E_CLASS...	0x80040111 = ClassFac...	0.0000145
8	10:38:22.465 AM	14	AUDIOSES.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "AudioClientRpc", NULL, 0x00000000...	RPC_S_OK		0.0000081
9	10:38:22.465 AM	14	AUDIOSES.DLL	RpcBindingFromStringBindingW ("ncalrpc:[AudioClientRpc]", 0x000000001c99f798)	RPC_S_OK		0.0000140
10	10:38:22.465 AM	14	AUDIOSES.DLL	RpcStringFreeW (0x000000001c99f770)	RPC_S_OK		0.0000036
11	10:38:22.465 AM	14	AUDIOSES.DLL	RpcBindingFree (0x000000001c99f760)	RPC_S_OK		0.0000075
12	10:38:22.468 AM	14	AUDIOSES.DLL	NdrDllGetClassObject (CrossProcessClientInputEndpoint class, IClassFactory, 0x00000000...	CLASS_E_CLASS...	0x80040111 = ClassFac...	0.0000047

Figure 32: Network traffic when recording through the microphone.

Keylogger Network Traffic

The RAT makes no observable network traffic API calls when using the keylogger.

5.6 LuxNET

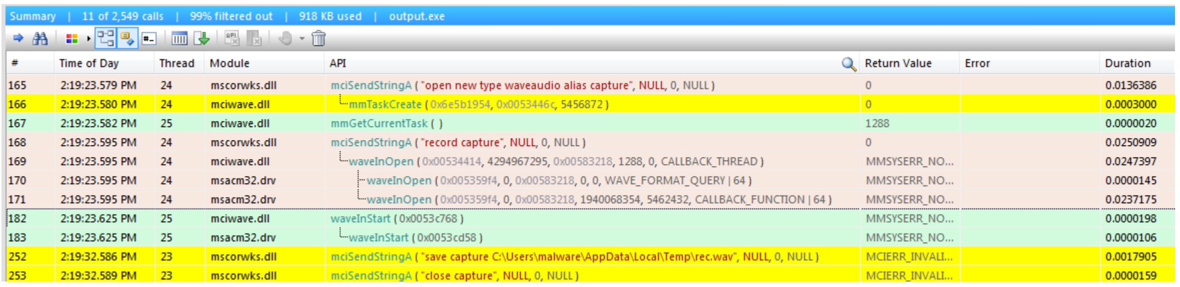
LuxNET is a Visual Basic RAT created by someone going by the handle Xillux[88]. We have not found this RAT used in any published attacks, but it is still a feature rich piece of software that equals most other RATs in terms of functionality. The fact that it is little known does make it all the more stealthy, and less likely to be detected by anti-virus software. We tested version Alpha 1 obtained from Hackforums.net in our experiments[88].

5.6.1 Webcam

LuxNET does not support webcam capturing, but it does provide a menu option for it, so it is probably a planned feature.

5.6.2 Microphone

Figure 33 shows the API calls that occur when we record from the microphone on the victim. Here we actually see some difference from the other RATs when recording. The **waveInStart** call appears, but in addition we see **mciSendString** which contains the strings "*open new type waveaudio alias capture*", "*record capture*", "*save capture C:\Users\malware\AppData\Local\Temp\rec.wav*", and "*close capture*".



Summary | 11 of 2,549 calls | 99% filtered out | 918 KB used | output.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
165	2:19:23.579 PM	24	mscorwks.dll	mciSendStringA ("open new type waveaudio alias capture", NULL, 0, NULL)	0		0.0136386
166	2:19:23.580 PM	24	mciwave.dll	└mmTaskCreate (0x6e5b1954, 0x0053446c, 5456872)	0		0.0003000
167	2:19:23.582 PM	25	mciwave.dll	mmGetCurrentTask ( )	1288		0.0000020
168	2:19:23.595 PM	24	mscorwks.dll	mciSendStringA ("record capture", NULL, 0, NULL)	0		0.0250909
169	2:19:23.595 PM	24	mciwave.dll	└waveInOpen (0x00534414, 4294967295, 0x00583218, 1288, 0, CALLBACK_THREAD)	MMSYSERR_NO...		0.0247397
170	2:19:23.595 PM	24	msacm32.drv	├waveInOpen (0x005359f4, 0, 0x00583218, 0, 0, WAVE_FORMAT_QUERY | 64)	MMSYSERR_NO...		0.0000145
171	2:19:23.595 PM	24	msacm32.drv	└waveInOpen (0x005359f4, 0, 0x00583218, 1940068354, 5462432, CALLBACK_FUNCTION | 64)	MMSYSERR_NO...		0.0237175
182	2:19:23.625 PM	25	mciwave.dll	waveInStart (0x0053c768)	MMSYSERR_NO...		0.0000198
183	2:19:23.625 PM	25	msacm32.drv	└waveInStart (0x0053cd58)	MMSYSERR_NO...		0.0000106
252	2:19:32.586 PM	23	mscorwks.dll	mciSendStringA ("save capture C:\Users\malware\AppData\Local\Temp\rec.wav", NULL, 0, NULL)	MCIERR_INVALI...		0.0017905
253	2:19:32.589 PM	23	mscorwks.dll	mciSendStringA ("close capture", NULL, 0, NULL)	MCIERR_INVALI...		0.0000159

Figure 33: API calls when recording from the microphone.

5.6.3 Keylogger

In Figure 34 we can see the API calls that happen when we type the word "master" on the victim machine. As we have seen on the other RATs, **ToUnicodeEx** appears and contains the letter "m" as shown. The other keys are omitted to save space.

5.6.4 Network Traffic

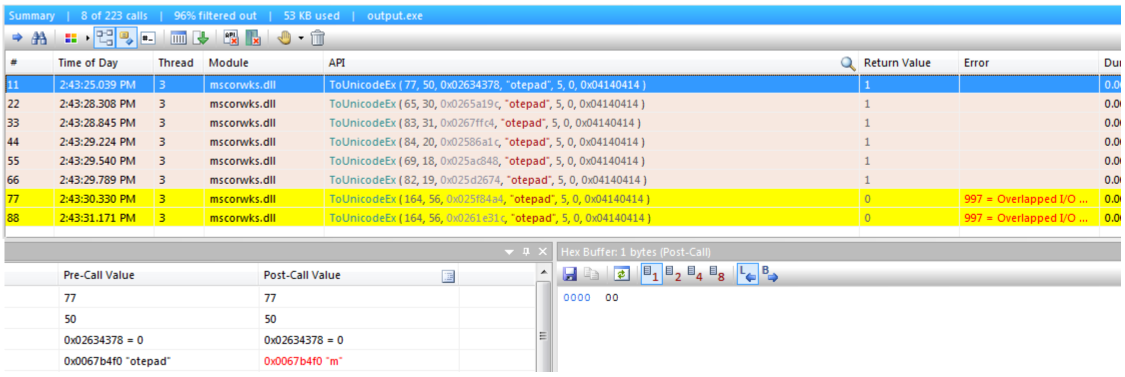
LuxNET communicates through RPC calls and they can be used to derive information about what is happening in the RAT.

Microphone

In Figure 35 we see the API calls appearing from the network traffic when we use the microphone. **RpcStringBindingCompose** and **RpicBindingFromStringBinding** contains an argument called "AudioClientRpc".

Keylogger

LuxNET gave us no discernible network traffic when using the keylogger.



Summary | 8 of 223 calls | 96% filtered out | 53 KB used | output.exe

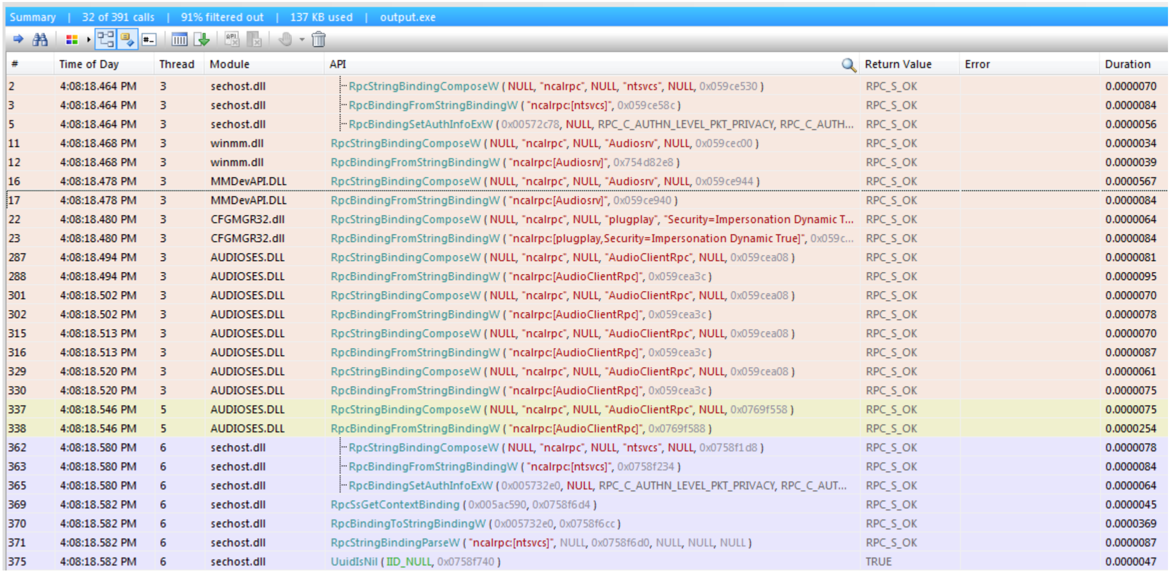
#	Time of Day	Thread	Module	API	Return Value	Error	Du
11	2:43:25.039 PM	3	mscorwks.dll	ToUnicodeEx (77, 50, 0x02634378, "otepad", 5, 0, 0x04140414)	1		0.0
22	2:43:28.308 PM	3	mscorwks.dll	ToUnicodeEx (65, 30, 0x0265a19c, "otepad", 5, 0, 0x04140414)	1		0.0
33	2:43:28.845 PM	3	mscorwks.dll	ToUnicodeEx (83, 31, 0x0267ffc4, "otepad", 5, 0, 0x04140414)	1		0.0
44	2:43:29.224 PM	3	mscorwks.dll	ToUnicodeEx (84, 20, 0x02586a1c, "otepad", 5, 0, 0x04140414)	1		0.0
55	2:43:29.540 PM	3	mscorwks.dll	ToUnicodeEx (69, 18, 0x025ac848, "otepad", 5, 0, 0x04140414)	1		0.0
66	2:43:29.789 PM	3	mscorwks.dll	ToUnicodeEx (82, 19, 0x025d2674, "otepad", 5, 0, 0x04140414)	1		0.0
77	2:43:30.330 PM	3	mscorwks.dll	ToUnicodeEx (164, 56, 0x025f84a4, "otepad", 5, 0, 0x04140414)	0	997 = Overlapped I/O ...	0.0
88	2:43:31.171 PM	3	mscorwks.dll	ToUnicodeEx (164, 56, 0x0261e31c, "otepad", 5, 0, 0x04140414)	0	997 = Overlapped I/O ...	0.0

Pre-Call Value	Post-Call Value
77	77
50	50
0x02634378 = 0	0x02634378 = 0
0x0067b4f0 "otepad"	0x0067b4f0 "m"

Hex Buffer: 1 bytes (Post-Call)

```
0000   00
```

Figure 34: API calls when typing on the keyboard.



Summary | 32 of 391 calls | 91% filtered out | 137 KB used | output.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
2	4:08:18.464 PM	3	sechost.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "ntsvcs", NULL, 0x059ce530)	RPC_S_OK		0.0000070
3	4:08:18.464 PM	3	sechost.dll	RpcBindingFromStringBindingW ("ncalrpc:[ntsvcs]", 0x059ce58c)	RPC_S_OK		0.0000084
5	4:08:18.464 PM	3	sechost.dll	RpcBindingSetAuthInfoExW (0x00572c78, NULL, RPC_C_AUTHN_LEVEL_PKT_PRIVACY, RPC_C_AUTH...	RPC_S_OK		0.0000056
11	4:08:18.468 PM	3	winmm.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "Audiosrv", NULL, 0x059cec00)	RPC_S_OK		0.0000034
12	4:08:18.468 PM	3	winmm.dll	RpcBindingFromStringBindingW ("ncalrpc:[Audiosrv]", 0x754d82e8)	RPC_S_OK		0.0000039
16	4:08:18.478 PM	3	MMDevAPI.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "Audiosrv", NULL, 0x059ce944)	RPC_S_OK		0.0000567
17	4:08:18.478 PM	3	MMDevAPI.DLL	RpcBindingFromStringBindingW ("ncalrpc:[Audiosrv]", 0x059ce940)	RPC_S_OK		0.0000084
22	4:08:18.480 PM	3	CFGMGR32.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "plugplay", "Security=Impersonation Dynamic T...	RPC_S_OK		0.0000064
23	4:08:18.480 PM	3	CFGMGR32.dll	RpcBindingFromStringBindingW ("ncalrpc:[plugplay,Security=Impersonation Dynamic True]", 0x059c...	RPC_S_OK		0.0000084
287	4:08:18.494 PM	3	AUDIOSES.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "AudioClientRpc", NULL, 0x059cea08)	RPC_S_OK		0.0000081
288	4:08:18.494 PM	3	AUDIOSES.DLL	RpcBindingFromStringBindingW ("ncalrpc:[AudioClientRpc]", 0x059cea3c)	RPC_S_OK		0.0000095
301	4:08:18.502 PM	3	AUDIOSES.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "AudioClientRpc", NULL, 0x059cea08)	RPC_S_OK		0.0000070
302	4:08:18.502 PM	3	AUDIOSES.DLL	RpcBindingFromStringBindingW ("ncalrpc:[AudioClientRpc]", 0x059cea3c)	RPC_S_OK		0.0000078
315	4:08:18.513 PM	3	AUDIOSES.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "AudioClientRpc", NULL, 0x059cea08)	RPC_S_OK		0.0000070
316	4:08:18.513 PM	3	AUDIOSES.DLL	RpcBindingFromStringBindingW ("ncalrpc:[AudioClientRpc]", 0x059cea3c)	RPC_S_OK		0.0000087
329	4:08:18.520 PM	3	AUDIOSES.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "AudioClientRpc", NULL, 0x059cea08)	RPC_S_OK		0.0000061
330	4:08:18.520 PM	3	AUDIOSES.DLL	RpcBindingFromStringBindingW ("ncalrpc:[AudioClientRpc]", 0x059cea3c)	RPC_S_OK		0.0000075
337	4:08:18.546 PM	5	AUDIOSES.DLL	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "AudioClientRpc", NULL, 0x0769f558)	RPC_S_OK		0.0000075
338	4:08:18.546 PM	5	AUDIOSES.DLL	RpcBindingFromStringBindingW ("ncalrpc:[AudioClientRpc]", 0x0769f588)	RPC_S_OK		0.0000254
362	4:08:18.580 PM	6	sechost.dll	RpcStringBindingComposeW (NULL, "ncalrpc", NULL, "ntsvcs", NULL, 0x0758f1d8)	RPC_S_OK		0.0000078
363	4:08:18.580 PM	6	sechost.dll	RpcBindingFromStringBindingW ("ncalrpc:[ntsvcs]", 0x0758f234)	RPC_S_OK		0.0000084
365	4:08:18.580 PM	6	sechost.dll	RpcBindingSetAuthInfoExW (0x005732e0, NULL, RPC_C_AUTHN_LEVEL_PKT_PRIVACY, RPC_C_AUT...	RPC_S_OK		0.0000064
369	4:08:18.582 PM	6	sechost.dll	RpcSsGetContextBinding (0x005ac590, 0x0758f6d4)	RPC_S_OK		0.0000045
370	4:08:18.582 PM	6	sechost.dll	RpcBindingToStringBindingW (0x005732e0, 0x0758f6cc)	RPC_S_OK		0.0000369
371	4:08:18.582 PM	6	sechost.dll	RpcStringBindingParseW ("ncalrpc:[ntsvcs]", NULL, 0x0758f6d0, NULL, NULL, NULL)	RPC_S_OK		0.0000087
375	4:08:18.582 PM	6	sechost.dll	UuidIsNil (IID_NULL, 0x0758f740)	TRUE		0.0000047

Figure 35: API calls for network traffic when using the microphone.

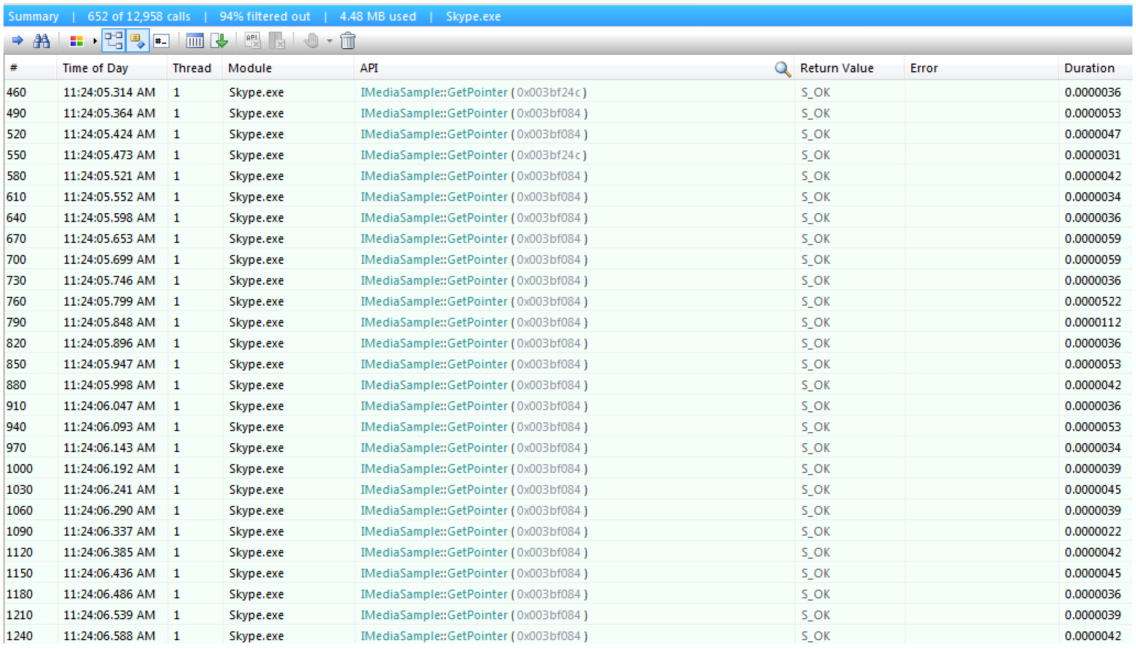
5.7 Skype

This section describes the video chat program Skype[89]. We chose to examine Skype because it shares a lot of common features with a RAT, such as recording video and audio, and capturing keystrokes. The difference however is that this behavior is both expected and desired in a program such as Skype, and we should therefore see a different way of using such a program compared to how a RAT would "interact" with its victim. The reason for studying Skype is thus to find out what kind of behavior (i.e. API calls) that separates a benign program like Skype from a RAT.

To perform our testing we started a video chat between two Skype clients over the Internet and recorded the API calls on one of the machines when the different types of behavior was performed.

5.7.1 Webcam

When we start a webcam chat we can see the calls in Figure 36 appearing. Once again, we can notice that the call **IMediaSample::GetPointer** appears when the webcam is used, and this does provide an indicator for this behavior. The webcam session lasted for 14 seconds, and the API calls were consistently being made throughout that time period.



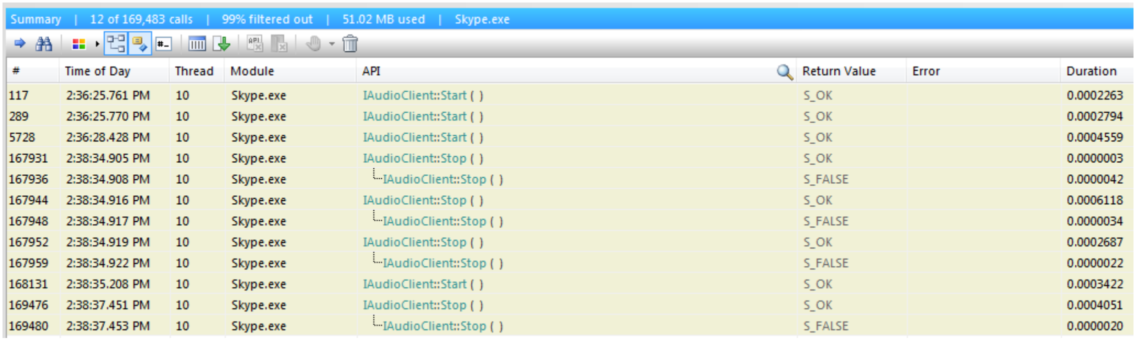
Summary | 652 of 12,958 calls | 94% filtered out | 4.48 MB used | Skype.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
460	11:24:05.314 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf24c)	S_OK		0.0000036
490	11:24:05.364 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000053
520	11:24:05.424 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000047
550	11:24:05.473 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf24c)	S_OK		0.0000031
580	11:24:05.521 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000042
610	11:24:05.552 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000034
640	11:24:05.598 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000036
670	11:24:05.653 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000059
700	11:24:05.699 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000059
730	11:24:05.746 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000036
760	11:24:05.799 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000522
790	11:24:05.848 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000112
820	11:24:05.896 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000036
850	11:24:05.947 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000053
880	11:24:05.998 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000042
910	11:24:06.047 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000036
940	11:24:06.093 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000053
970	11:24:06.143 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000034
1000	11:24:06.192 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000039
1030	11:24:06.241 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000045
1060	11:24:06.290 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000039
1090	11:24:06.337 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000022
1120	11:24:06.385 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000042
1150	11:24:06.436 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000045
1180	11:24:06.486 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000036
1210	11:24:06.539 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000039
1240	11:24:06.588 AM	1	Skype.exe	IMediaSample::GetPointer (0x003bf084)	S_OK		0.0000042

Figure 36: The API calls that appear when we use the webcam.

5.7.2 Microphone

By making a voice-only call through Skype we get the calls shown in Figure 37. The call lasted for 2 minutes, and we can see the **IAudioClient::Start** and **IAudioClient::Stop** calls when the recording starts and stops. Note that this is different from the **waveInStart** and **waveInStop** calls that we have seen previously in the RATs. This is probably caused by different libraries being used, seen as the **waveIn** functions belong to the **Winmm.dll**[77] library, while **IAudioClient** resides in the Windows Audio Session API (WASAPI)[90].


Summary | 12 of 169,483 calls | 99% filtered out | 51.02 MB used | Skype.exe

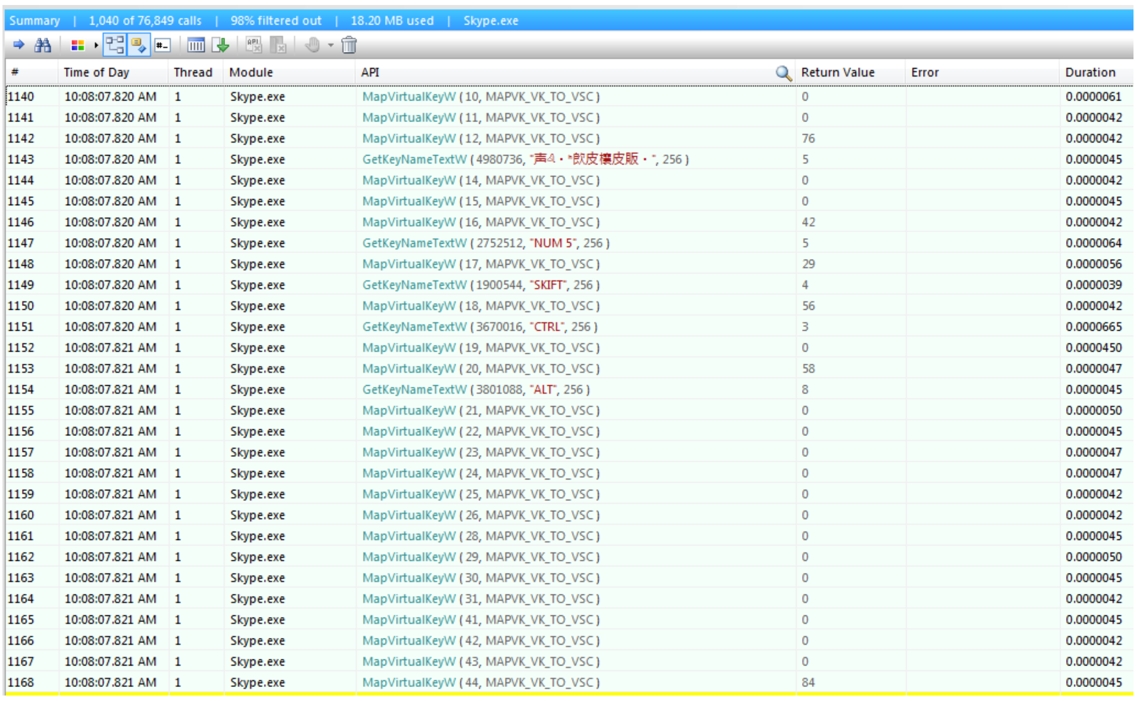
#	Time of Day	Thread	Module	API	Return Value	Error	Duration
117	2:36:25.761 PM	10	Skype.exe	IAudioClient::Start ( )	S_OK		0.0002263
289	2:36:25.770 PM	10	Skype.exe	IAudioClient::Start ( )	S_OK		0.0002794
5728	2:36:28.428 PM	10	Skype.exe	IAudioClient::Start ( )	S_OK		0.0004559
167931	2:38:34.905 PM	10	Skype.exe	IAudioClient::Stop ( )	S_OK		0.0000003
167936	2:38:34.908 PM	10	Skype.exe	└IAudioClient::Stop ( )	S_FALSE		0.0000042
167944	2:38:34.916 PM	10	Skype.exe	IAudioClient::Stop ( )	S_OK		0.0006118
167948	2:38:34.917 PM	10	Skype.exe	└IAudioClient::Stop ( )	S_FALSE		0.0000034
167952	2:38:34.919 PM	10	Skype.exe	IAudioClient::Stop ( )	S_OK		0.0002687
167959	2:38:34.922 PM	10	Skype.exe	└IAudioClient::Stop ( )	S_FALSE		0.0000022
168131	2:38:35.208 PM	10	Skype.exe	IAudioClient::Start ( )	S_OK		0.0003422
169476	2:38:37.451 PM	10	Skype.exe	IAudioClient::Stop ( )	S_OK		0.0004051
169480	2:38:37.453 PM	10	Skype.exe	└IAudioClient::Stop ( )	S_FALSE		0.0000020

Figure 37: The microphone API calls.

5.7.3 Keyboard Input

Skype (obviously) does not contain a keylogger function, but it does provide the ability to send chat messages to contacts, and can thus capture keyboard input. Therefore we should expect that Skype will make API calls that tells it which keys have been pressed, and that it therefore will make the same ones as a keylogger does.

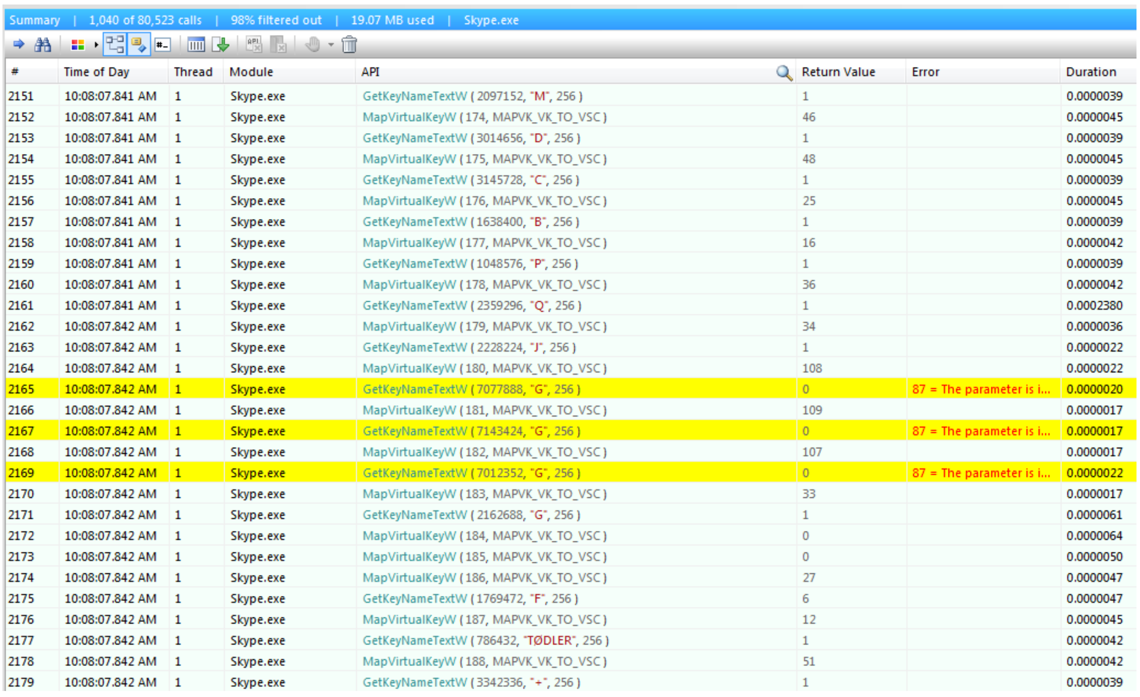
By observation we can see that the functionality of Skype is a bit different than the RATs, for where they would capture one key at a time, Skype enumerates all the keys using the **MapVirtualKeyW** and **GetKeyNameTextW** API calls at the moment you select a contact to chat with in Skype. It does not make any individual API calls when individual keys are pressed however, so it is impossible to distinguish individual keystrokes. Figure 38 and 39 shows the **MapVirtualKeyW** and **GetKeyNameTextW** API calls that enumerates all the keys on the keyboard from alphabet start to alphabet end.
A consequence of this is that by hooking the API calls performed by Skype we can *not* see which keys that have been pressed, because Skype does not make any API calls related to individual key presses, and we are thus prevented from observing what is written to Skype. This might be an intentional counter-measure to prevent snooping of Skype data by other programs.



Summary | 1,040 of 76,849 calls | 98% filtered out | 18.20 MB used | Skype.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1140	10:08:07.820 AM	1	Skype.exe	MapVirtualKeyW (10, MAPVK_VK_TO_VSC)	0		0.0000061
1141	10:08:07.820 AM	1	Skype.exe	MapVirtualKeyW (11, MAPVK_VK_TO_VSC)	0		0.0000042
1142	10:08:07.820 AM	1	Skype.exe	MapVirtualKeyW (12, MAPVK_VK_TO_VSC)	76		0.0000042
1143	10:08:07.820 AM	1	Skype.exe	GetKeyNameTextW (4980736, "声4・*飲皮欔皮販・", 256)	5		0.0000045
1144	10:08:07.820 AM	1	Skype.exe	MapVirtualKeyW (14, MAPVK_VK_TO_VSC)	0		0.0000042
1145	10:08:07.820 AM	1	Skype.exe	MapVirtualKeyW (15, MAPVK_VK_TO_VSC)	0		0.0000045
1146	10:08:07.820 AM	1	Skype.exe	MapVirtualKeyW (16, MAPVK_VK_TO_VSC)	42		0.0000042
1147	10:08:07.820 AM	1	Skype.exe	GetKeyNameTextW (2752512, "NUM 5", 256)	5		0.0000064
1148	10:08:07.820 AM	1	Skype.exe	MapVirtualKeyW (17, MAPVK_VK_TO_VSC)	29		0.0000056
1149	10:08:07.820 AM	1	Skype.exe	GetKeyNameTextW (1900544, "SKIFT", 256)	4		0.0000039
1150	10:08:07.820 AM	1	Skype.exe	MapVirtualKeyW (18, MAPVK_VK_TO_VSC)	56		0.0000042
1151	10:08:07.820 AM	1	Skype.exe	GetKeyNameTextW (3670016, "CTRL", 256)	3		0.0000665
1152	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (19, MAPVK_VK_TO_VSC)	0		0.0000450
1153	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (20, MAPVK_VK_TO_VSC)	58		0.0000047
1154	10:08:07.821 AM	1	Skype.exe	GetKeyNameTextW (3801088, "ALT", 256)	8		0.0000045
1155	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (21, MAPVK_VK_TO_VSC)	0		0.0000050
1156	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (22, MAPVK_VK_TO_VSC)	0		0.0000045
1157	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (23, MAPVK_VK_TO_VSC)	0		0.0000047
1158	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (24, MAPVK_VK_TO_VSC)	0		0.0000047
1159	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (25, MAPVK_VK_TO_VSC)	0		0.0000042
1160	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (26, MAPVK_VK_TO_VSC)	0		0.0000042
1161	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (28, MAPVK_VK_TO_VSC)	0		0.0000045
1162	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (29, MAPVK_VK_TO_VSC)	0		0.0000050
1163	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (30, MAPVK_VK_TO_VSC)	0		0.0000045
1164	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (31, MAPVK_VK_TO_VSC)	0		0.0000042
1165	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (41, MAPVK_VK_TO_VSC)	0		0.0000045
1166	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (42, MAPVK_VK_TO_VSC)	0		0.0000042
1167	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (43, MAPVK_VK_TO_VSC)	0		0.0000042
1168	10:08:07.821 AM	1	Skype.exe	MapVirtualKeyW (44, MAPVK_VK_TO_VSC)	84		0.0000045

Figure 38: The start of the keyboard API calls.



Summary | 1,040 of 80,523 calls | 98% filtered out | 19.07 MB used | Skype.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
2151	10:08:07.841 AM	1	Skype.exe	GetKeyNameTextW (2097152, "M", 256)	1		0.0000039
2152	10:08:07.841 AM	1	Skype.exe	MapVirtualKeyW (174, MAPVK_VK_TO_VSC)	46		0.0000045
2153	10:08:07.841 AM	1	Skype.exe	GetKeyNameTextW (3014656, "D", 256)	1		0.0000039
2154	10:08:07.841 AM	1	Skype.exe	MapVirtualKeyW (175, MAPVK_VK_TO_VSC)	48		0.0000045
2155	10:08:07.841 AM	1	Skype.exe	GetKeyNameTextW (3145728, "C", 256)	1		0.0000039
2156	10:08:07.841 AM	1	Skype.exe	MapVirtualKeyW (176, MAPVK_VK_TO_VSC)	25		0.0000045
2157	10:08:07.841 AM	1	Skype.exe	GetKeyNameTextW (1638400, "B", 256)	1		0.0000039
2158	10:08:07.841 AM	1	Skype.exe	MapVirtualKeyW (177, MAPVK_VK_TO_VSC)	16		0.0000042
2159	10:08:07.841 AM	1	Skype.exe	GetKeyNameTextW (1048576, "P", 256)	1		0.0000039
2160	10:08:07.841 AM	1	Skype.exe	MapVirtualKeyW (178, MAPVK_VK_TO_VSC)	36		0.0000042
2161	10:08:07.841 AM	1	Skype.exe	GetKeyNameTextW (2359296, "Q", 256)	1		0.0002380
2162	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (179, MAPVK_VK_TO_VSC)	34		0.0000036
2163	10:08:07.842 AM	1	Skype.exe	GetKeyNameTextW (2228224, "J", 256)	1		0.0000022
2164	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (180, MAPVK_VK_TO_VSC)	108		0.0000022
2165	10:08:07.842 AM	1	Skype.exe	GetKeyNameTextW (7077888, "G", 256)	0	87 = The parameter is i...	0.0000020
2166	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (181, MAPVK_VK_TO_VSC)	109		0.0000017
2167	10:08:07.842 AM	1	Skype.exe	GetKeyNameTextW (7143424, "G", 256)	0	87 = The parameter is i...	0.0000017
2168	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (182, MAPVK_VK_TO_VSC)	107		0.0000017
2169	10:08:07.842 AM	1	Skype.exe	GetKeyNameTextW (7012352, "G", 256)	0	87 = The parameter is i...	0.0000022
2170	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (183, MAPVK_VK_TO_VSC)	33		0.0000017
2171	10:08:07.842 AM	1	Skype.exe	GetKeyNameTextW (2162688, "G", 256)	1		0.0000061
2172	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (184, MAPVK_VK_TO_VSC)	0		0.0000064
2173	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (185, MAPVK_VK_TO_VSC)	0		0.0000050
2174	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (186, MAPVK_VK_TO_VSC)	27		0.0000047
2175	10:08:07.842 AM	1	Skype.exe	GetKeyNameTextW (1769472, "F", 256)	6		0.0000047
2176	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (187, MAPVK_VK_TO_VSC)	12		0.0000045
2177	10:08:07.842 AM	1	Skype.exe	GetKeyNameTextW (786432, "TØDLER", 256)	1		0.0000042
2178	10:08:07.842 AM	1	Skype.exe	MapVirtualKeyW (188, MAPVK_VK_TO_VSC)	51		0.0000042
2179	10:08:07.842 AM	1	Skype.exe	GetKeyNameTextW (3342336, "+", 256)	1		0.0000039

Figure 39: The end of the keyboard API calls.

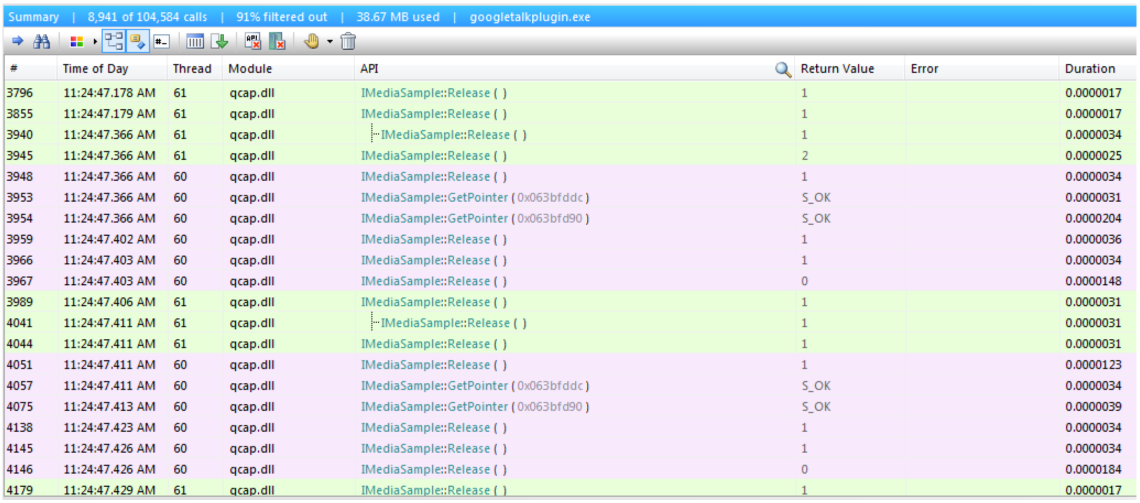
5.8 Google Hangouts

Google Hangouts is Google's interpretation of a video chat service[91]. Just like Skype, Google Hangouts produces behavior that can be said to be similar to what a RAT would do, and therefore we wanted to look at its API calls to see how they compare to those of the RATs.

This experiment was performed with Google Chrome where we used Google Hangouts with the Google Talk plugin. Using this plugin allows users to video chat through Google Hangouts. As with Skype, we set up a connection between two Google Hangouts clients over the Internet and recorded the API calls on one of the machines.

5.8.1 Webcam

In Figure 40 we can again observe the **IMediaSample::GetPointer** call appear when the webcam is recording.



Summary | 8,941 of 104,584 calls | 91% filtered out | 38.67 MB used | googletalkplugin.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
3796	11:24:47.178 AM	61	qcap.dll	IMediaSample::Release ( )	1		0.0000017
3855	11:24:47.179 AM	61	qcap.dll	IMediaSample::Release ( )	1		0.0000017
3940	11:24:47.366 AM	61	qcap.dll	IMediaSample::Release ( )	1		0.0000034
3945	11:24:47.366 AM	61	qcap.dll	IMediaSample::Release ( )	2		0.0000025
3948	11:24:47.366 AM	60	qcap.dll	IMediaSample::Release ( )	1		0.0000034
3953	11:24:47.366 AM	60	qcap.dll	IMediaSample::GetPointer (0x063bfddc)	S_OK		0.0000031
3954	11:24:47.366 AM	60	qcap.dll	IMediaSample::GetPointer (0x063bfd90)	S_OK		0.0000204
3959	11:24:47.402 AM	60	qcap.dll	IMediaSample::Release ( )	1		0.0000036
3966	11:24:47.403 AM	60	qcap.dll	IMediaSample::Release ( )	1		0.0000034
3967	11:24:47.403 AM	60	qcap.dll	IMediaSample::Release ( )	0		0.0000148
3989	11:24:47.406 AM	61	qcap.dll	IMediaSample::Release ( )	1		0.0000031
4041	11:24:47.411 AM	61	qcap.dll	IMediaSample::Release ( )	1		0.0000031
4044	11:24:47.411 AM	61	qcap.dll	IMediaSample::Release ( )	1		0.0000031
4051	11:24:47.411 AM	60	qcap.dll	IMediaSample::Release ( )	1		0.0000123
4057	11:24:47.411 AM	60	qcap.dll	IMediaSample::GetPointer (0x063bfddc)	S_OK		0.0000034
4075	11:24:47.413 AM	60	qcap.dll	IMediaSample::GetPointer (0x063bfd90)	S_OK		0.0000039
4138	11:24:47.423 AM	60	qcap.dll	IMediaSample::Release ( )	1		0.0000034
4145	11:24:47.426 AM	60	qcap.dll	IMediaSample::Release ( )	1		0.0000034
4146	11:24:47.426 AM	60	qcap.dll	IMediaSample::Release ( )	0		0.0000184
4179	11:24:47.429 AM	61	qcap.dll	IMediaSample::Release ( )	1		0.0000017

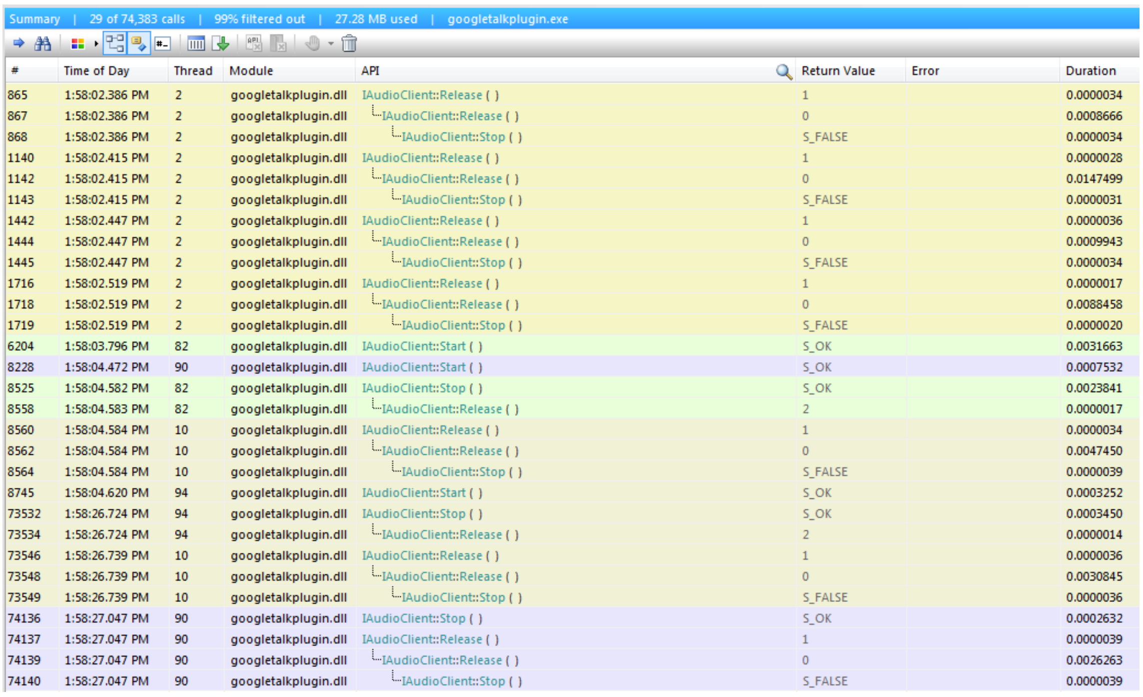
Figure 40: The API calls when the webcam records.

5.8.2 Microphone Capture

Figure 41 shows the API calls when a voice call is made. The voice call lasted for about 25 seconds. Just like in Skype, **IAudioClient::Start** and **IAudioClient::Stop** appears here as well.

5.8.3 Keyboard Input

Google Hangouts makes no API calls that allows us to discern which keys that have been pressed when we use the chat functionality.



Summary | 29 of 74,383 calls | 99% filtered out | 27.28 MB used | googletalkplugin.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
865	1:58:02.386 PM	2	googletalkplugin.dll	IAudioClient::Release ( )	1		0.0000034
867	1:58:02.386 PM	2	googletalkplugin.dll	└IAudioClient::Release ( )	0		0.0008666
868	1:58:02.386 PM	2	googletalkplugin.dll	└IAudioClient::Stop ( )	S_FALSE		0.0000034
1140	1:58:02.415 PM	2	googletalkplugin.dll	IAudioClient::Release ( )	1		0.0000028
1142	1:58:02.415 PM	2	googletalkplugin.dll	└IAudioClient::Release ( )	0		0.0147499
1143	1:58:02.415 PM	2	googletalkplugin.dll	└IAudioClient::Stop ( )	S_FALSE		0.0000031
1442	1:58:02.447 PM	2	googletalkplugin.dll	IAudioClient::Release ( )	1		0.0000036
1444	1:58:02.447 PM	2	googletalkplugin.dll	└IAudioClient::Release ( )	0		0.0009943
1445	1:58:02.447 PM	2	googletalkplugin.dll	└IAudioClient::Stop ( )	S_FALSE		0.0000034
1716	1:58:02.519 PM	2	googletalkplugin.dll	IAudioClient::Release ( )	1		0.0000017
1718	1:58:02.519 PM	2	googletalkplugin.dll	└IAudioClient::Release ( )	0		0.0088458
1719	1:58:02.519 PM	2	googletalkplugin.dll	└IAudioClient::Stop ( )	S_FALSE		0.0000020
6204	1:58:03.796 PM	82	googletalkplugin.dll	IAudioClient::Start ( )	S_OK		0.0031663
8228	1:58:04.472 PM	90	googletalkplugin.dll	IAudioClient::Start ( )	S_OK		0.0007532
8525	1:58:04.582 PM	82	googletalkplugin.dll	IAudioClient::Stop ( )	S_OK		0.0023841
8558	1:58:04.583 PM	82	googletalkplugin.dll	└IAudioClient::Release ( )	2		0.0000017
8560	1:58:04.584 PM	10	googletalkplugin.dll	IAudioClient::Release ( )	1		0.0000034
8562	1:58:04.584 PM	10	googletalkplugin.dll	└IAudioClient::Release ( )	0		0.0047450
8564	1:58:04.584 PM	10	googletalkplugin.dll	└IAudioClient::Stop ( )	S_FALSE		0.0000039
8745	1:58:04.620 PM	94	googletalkplugin.dll	IAudioClient::Start ( )	S_OK		0.0003252
73532	1:58:26.724 PM	94	googletalkplugin.dll	IAudioClient::Stop ( )	S_OK		0.0003450
73534	1:58:26.724 PM	94	googletalkplugin.dll	└IAudioClient::Release ( )	2		0.0000014
73546	1:58:26.739 PM	10	googletalkplugin.dll	IAudioClient::Release ( )	1		0.0000036
73548	1:58:26.739 PM	10	googletalkplugin.dll	└IAudioClient::Release ( )	0		0.0030845
73549	1:58:26.739 PM	10	googletalkplugin.dll	└IAudioClient::Stop ( )	S_FALSE		0.0000036
74136	1:58:27.047 PM	90	googletalkplugin.dll	IAudioClient::Stop ( )	S_OK		0.0002632
74137	1:58:27.047 PM	90	googletalkplugin.dll	IAudioClient::Release ( )	1		0.0000039
74139	1:58:27.047 PM	90	googletalkplugin.dll	└IAudioClient::Release ( )	0		0.0026263
74140	1:58:27.047 PM	90	googletalkplugin.dll	└IAudioClient::Stop ( )	S_FALSE		0.0000039

Figure 41: The voice API calls.

Application	Has a GUI	User Interactivity API Calls Appearing
Dark Comet	Yes	Yes
njRAT	No	No
jRAT	Yes, tray icon (free version)	No
jSpy	No	No
CloudNet	No	No
LuxNET	No	No
Skype	Yes	GetFocus & SetFocus
Google Hangouts	Yes	GetFocus & SetFocus

Table 1: The API calls related to user interactivity for each program.

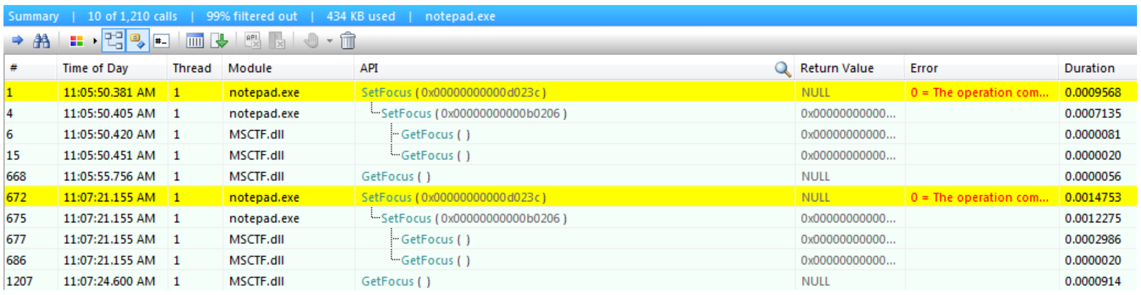
5.9 User Interaction

Now that we have seen which API calls that appear when the various kinds of behavior occur, it is time to shift our focus to those API calls that we have used to distinguish benign programs from the RATs. As we mentioned earlier, our theory has been that we can separate a malicious program from a benign one based on the API calls related to the programs interactivity with the user, meaning that a legitimate program would interact with and show itself to the user by displaying windows, buttons, dialog boxes and so on, while a RAT wants to stay as hidden as possible and not do anything that alerts or notifies the "user" to its presence. This has been explained as Sami et al. as a common way of behaving for malware[52].

To separate benign and malicious programs we have focused on two user interactivity API calls, namely **SetFocus** and **GetFocus**. These two API calls sets the keyboard's attention to the current window, or checks if it already has it, respectively, and would only occur if a window is present for the given program. We have not observed that any of these calls appear in any GUI-less programs, and do therefore appear to be a sufficient indicator of whether or not the user is interacting with the given program. Table 1 shows a summary of the API calls that appeared when we checked for user interactivity in the RATs and benign programs.

5.9.1 Example

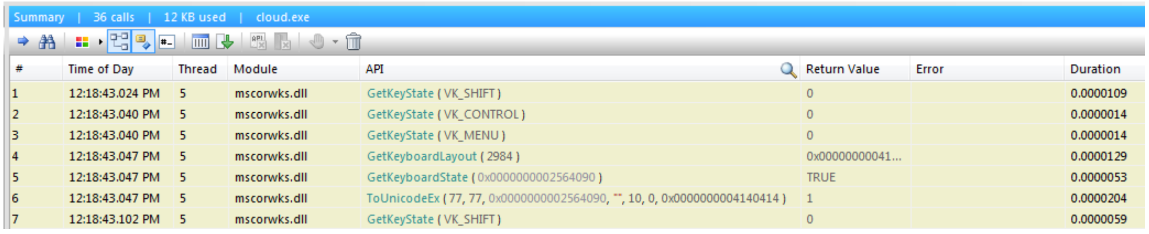
Let us look at an example to illustrate this. If we type the word "master" into Notepad.exe we can see that the API calls **GetFocus** and **SetFocus** appear as in Figure 42. This means that the Notepad.exe window has captured the keyboard input, and that it checks to see if it still has it.
If we look at the RAT CloudNet however, we can see that when we type on the keyboard, the API calls shown in Figure 43 appear, but that they do not contain the **GetFocus** or **SetFocus** calls that we saw appear in Notepad.exe when we did the same thing.
Another example can be seen in Figure 44. This is the calls that appear when we put the Notepad.exe window in the foreground, and as we can see, the API call **GetForegroundWindow**[92] is performed, and a handle is returned to the window in the foreground, which means that the monitored program, Notepad.exe, is in the foreground of the screen, and thus has the



Summary | 10 of 1,210 calls | 99% filtered out | 434 KB used | notepad.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	11:05:50.381 AM	1	notepad.exe	SetFocus (0x00000000000d023c)	NULL	0 = The operation com...	0.0009568
4	11:05:50.405 AM	1	notepad.exe	SetFocus (0x00000000000b0206)	0x00000000000...		0.0007135
6	11:05:50.420 AM	1	MSCTF.dll	GetFocus ( )	0x00000000000...		0.0000081
15	11:05:50.451 AM	1	MSCTF.dll	GetFocus ( )	0x00000000000...		0.0000020
668	11:05:55.756 AM	1	MSCTF.dll	GetFocus ( )	NULL		0.0000056
672	11:07:21.155 AM	1	notepad.exe	SetFocus (0x00000000000d023c)	NULL	0 = The operation com...	0.0014753
675	11:07:21.155 AM	1	notepad.exe	SetFocus (0x00000000000b0206)	0x00000000000...		0.0012275
677	11:07:21.155 AM	1	MSCTF.dll	GetFocus ( )	0x00000000000...		0.0002986
686	11:07:21.155 AM	1	MSCTF.dll	GetFocus ( )	0x00000000000...		0.0000020
1207	11:07:24.600 AM	1	MSCTF.dll	GetFocus ( )	NULL		0.0000914

Figure 42: The Notepad.exe API calls for user interactivity.

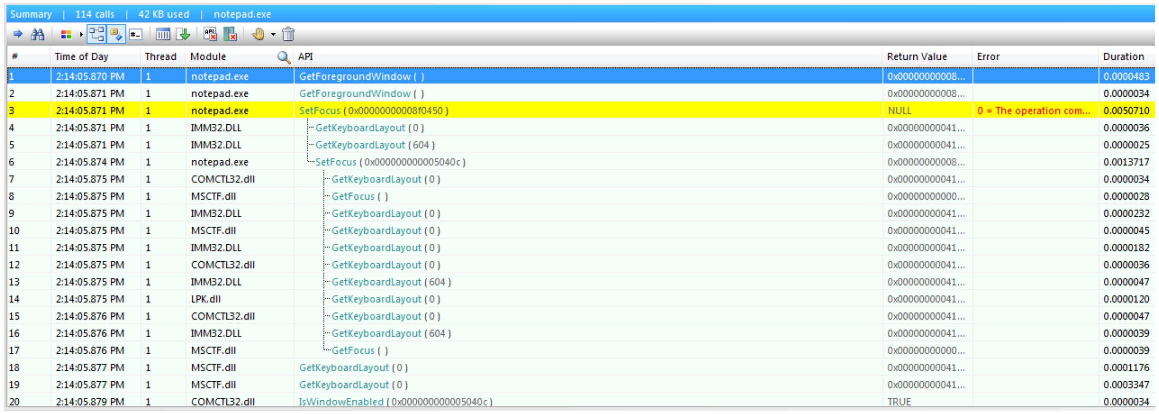


Summary | 36 calls | 12 KB used | cloud.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	12:18:43.024 PM	5	mscorwks.dll	GetKeyState (VK_SHIFT)	0		0.0000109
2	12:18:43.040 PM	5	mscorwks.dll	GetKeyState (VK_CONTROL)	0		0.0000014
3	12:18:43.040 PM	5	mscorwks.dll	GetKeyState (VK_MENU)	0		0.0000014
4	12:18:43.047 PM	5	mscorwks.dll	GetKeyboardLayout (2984)	0x00000000041...		0.0000129
5	12:18:43.047 PM	5	mscorwks.dll	GetKeyboardState (0x0000000002564090)	TRUE		0.0000053
6	12:18:43.047 PM	5	mscorwks.dll	ToUnicodeEx (77, 77, 0x0000000002564090, "", 10, 0, 0x0000000004140414)	1		0.0000204
7	12:18:43.102 PM	5	mscorwks.dll	GetKeyState (VK_SHIFT)	0		0.0000059

Figure 43: No **GetFocus** or **SetFocus** calls appear in CloudNet.

user's attention. We have not used these API calls in our testing, but we mention them here as an illustration of other API calls that also can be used.



Summary | 114 calls | 42 KB used | notepad.exe

#	Time of Day	Thread	Module	API	Return Value	Error	Duration
1	2:14:05.870 PM	1	notepad.exe	GetForegroundWindow ( )	0x00000000008...		0.0000483
2	2:14:05.871 PM	1	notepad.exe	GetForegroundWindow ( )	0x00000000008...		0.0000034
3	2:14:05.871 PM	1	notepad.exe	SetFocus (0x00000000008f0450)	NULL	0 = The operation com...	0.0050710
4	2:14:05.871 PM	1	IMM32.DLL	GetKeyboardLayout (0)	0x00000000041...		0.0000036
5	2:14:05.871 PM	1	IMM32.DLL	GetKeyboardLayout (604)	0x00000000041...		0.0000025
6	2:14:05.874 PM	1	notepad.exe	SetFocus (0x000000000005040c)	0x00000000008...		0.0013717
7	2:14:05.875 PM	1	COMCTL32.dll	GetKeyboardLayout (0)	0x00000000041...		0.0000034
8	2:14:05.875 PM	1	MSCTF.dll	GetFocus ( )	0x00000000000...		0.0000028
9	2:14:05.875 PM	1	IMM32.DLL	GetKeyboardLayout (0)	0x00000000041...		0.0000232
10	2:14:05.875 PM	1	MSCTF.dll	GetKeyboardLayout (0)	0x00000000041...		0.0000045
11	2:14:05.875 PM	1	IMM32.DLL	GetKeyboardLayout (0)	0x00000000041...		0.0000182
12	2:14:05.875 PM	1	COMCTL32.dll	GetKeyboardLayout (0)	0x00000000041...		0.0000036
13	2:14:05.875 PM	1	IMM32.DLL	GetKeyboardLayout (604)	0x00000000041...		0.0000047
14	2:14:05.875 PM	1	LPK.dll	GetKeyboardLayout (0)	0x00000000041...		0.0000120
15	2:14:05.876 PM	1	COMCTL32.dll	GetKeyboardLayout (0)	0x00000000041...		0.0000047
16	2:14:05.876 PM	1	IMM32.DLL	GetKeyboardLayout (604)	0x00000000041...		0.0000039
17	2:14:05.876 PM	1	MSCTF.dll	GetFocus ( )	0x00000000000...		0.0000039
18	2:14:05.877 PM	1	MSCTF.dll	GetKeyboardLayout (0)	0x00000000041...		0.0001176
19	2:14:05.877 PM	1	MSCTF.dll	GetKeyboardLayout (0)	0x00000000041...		0.0003347
20	2:14:05.879 PM	1	COMCTL32.dll	IsWindowEnabled (0x000000000005040c)	TRUE		0.0000034

Figure 44: API calls for checking what is the foreground window.

5.10 Obfuscation

As a quick experiment we have also tested out some various obfuscation tools to see if we could make the RATs undetectable for signature-based anti-virus software. A RAT that is undetectable by anti-virus software is usually referred to as "Fully Undetectable" or just FUD for short, and can be considered "the holy grail" of RATs[93]. In addition, the detection rate of the RATs could potentially be considered a good indicator of their popularity and spread since a more widely

Application	Before Obfuscation	After Obfuscation
Dark Comet	38/51	Unable to Obfuscate
njRAT	38/51	6/51
jRAT	12/50	3/50
jSpy	0/46	Unable to Obfuscate
CloudNet	12/50	3/50
LuxNet	19/52	2/52

Table 2: VirusTotal detection rate.

spread RAT is more likely to be detected by anti-virus software.

We used the website VirusTotal.com[94] which scans a submitted file with 52 different anti-virus programs and reports back which ones who detect the submitted file. Table 2 lists our findings before and after using the obfuscation tools. For .NET based programs we used Confuser[70], and for Java-based ones we used ProGuard[71]. We attempted to run the RAT again after performing the obfuscation, and we found no differences in the functionality between the obfuscated and unobfuscated version. This is a good argument for why we need new detection mechanisms for malware in general and RATs in particular.
In short, what these results show is that we are able to make the RATs fully undetectable for signature-based anti-virus with a minimal amount of work, but it will still make the same API calls as before it was obfuscated.

5.11 API Call Descriptions

The reader has probably noticed that we have not yet described what the API calls we have studied actually do, so we should remedy this by taking a look at how the API calls are described in the official documentation. The curious reader should of course visit Microsoft's documentation themselves for a more in-depth look, but we feel that it is in its place to repeat some of that information here. Table 3 shows the API calls we examined, where they are used, and a short description of what they do.

API Call	Behavior	Description
IMediaSample::GetPointer[95]	Webcam	Retrieves a pointer to the buffer of a media sample. In other words, it is called when video is being recorded.
waveInStart[77]	Microphone	Starts receiving input from the given audio input device.
waveInStop[96]	Microphone	Stops receiving audio input.
waveInReset[97]	Microphone	Stops receiving audio input and sends the data back to the application.
IAudioClient::Start[98]	Microphone	Starts the audio stream. I.e. starts to record audio from the microphone.
IAudioClient::Stop[99]	Microphone	Stops the audio stream. I.e. stops recording audio from the microphone.
ToUnicodeEx[100]	Keylogger	Translates a keypress to a unicode character.
ToAscii[80]	Keylogger	Translates a keypress to a character in the local alphabet.
SetFocus[101]	User Interaction	Keyboard focus is set to a specific window.
GetFocus[102]	User Interaction	Gets the handle of the window with keyboard focus.
send[103]	Network	Sends data to a socket.
RpcStringBindingCompose[104]	Network	Creates a handle to a string binding.
RpcBindingFromStringBinding[105]	Network	Creates a binding handle from a string.

Table 3: Descriptions of the API calls.

6 Results

Now it is time to look at how we can use all this data to model the behavior of the RATs. As explained before, we have theorized that the different modes of behavior in the RATs can be represented by using finite-state machines that represents the behavior as states and API calls as the input alphabet that changes these states. Very simply, this means that we look at which API calls that initiates a certain behavior.

Much of the observed RAT behavior does however, as we showed in the previous chapter, correspond to the same behavior in several legitimate programs too, such as video chat or word processing. So to allow us to get a more precise recognition we also have to model other characteristic traits of the RATs, such as the lack of API calls relating to user interaction and individual characteristics in network traffic. The clue, however, is that several of the finite-state machines can be used to describe both legitimate and malicious behavior, but combinations together with other kinds of behavior, such as hiding its presence, gives a more accurate picture that can be used to detect if the program is a RAT or not.

6.1 API Call Summary

Before we start to create the finite-state machines we should again take a look at the API calls that the different RATs used for their behavior. This is summarized in Table 4, and this is the data we will use in our finite-state machines. Table 5 shows the API calls and arguments that represent the network traffic, which will be used to model the network traffic.

Application	**Webcam**	**Microphone**	**Keylogger**
Dark Comet	IMediaSample::GetPointer	waveInStart & waveInClose	ToAscii
njRAT	IMediaSample::GetPointer	waveInStart & waveInReset	ToUnicodeEx
jRAT	Not Supported	waveInStart & waveInClose	Not Supported
jSpy	IMediaSample::GetPointer	Not Supported	No calls identify individual key strokes
CloudNet	IMediaSample::GetPointer	waveInStart & waveInClose	ToUnicodeEx
Lux Net	Not Supported	waveInStart & mciSendString("save capture `C:\Users\Username\AppData\Local\Temp\rec.wav`")	ToUnicodeEx
Skype	IMediaSample::GetPointer	IAudioClient::Start & IAudioClient::Stop	None (Enumerates all keys when entering chat, but no discernible individual keystrokes)
Google Hangouts	IMediaSample::GetPointer	IAudioClient::Start & IAudioClient::Stop	None

Table 4: The API calls denoting the different behavior.

Application	**Webcam**	**Microphone**	**Keylogger**
Dark Comet	**send** contains data prepended by the word JFIF	**RpcStringBindingCompose** & **RpcBindingFromStringBinding** containing "Audiosrc". "EndReceive" sent before socket closes.	Nothing/Scrambled Data
njRAT	**send** starts with the word "CAM"	**send** starts with the word "MIC"	**send** starts with the word "kl"
jRAT	Not Supported	**RpcStringBindingCompose** contains the parameter "AudioClientRpc"	Not Supported
jSpy	None	Not Supported	None
CloudNet	None	**RpcStringBindingCompose** & **RpcBindingFromStringBinding** contains "AudioClientRpc".	None
Lux Net	Not Supported	**RpcStringBindingCompose** & **RpcBindingFromStringBinding** contains "AudioClientRpc".	None

Table 5: The API calls and arguments for the network traffic.

6.2 Finite-State Machine for Webcam Activity

The finite-state machine in Figure 45 models the states for deciding if the webcam is recording or not. As we can see, **IMediaSample::GetPointer** calls are done when the webcam is recording, and a timeout period followed by no new **IMediaSample::GetPointer** calls means that the webcam has stopped recording, in accordance with what we observed in Section 5.

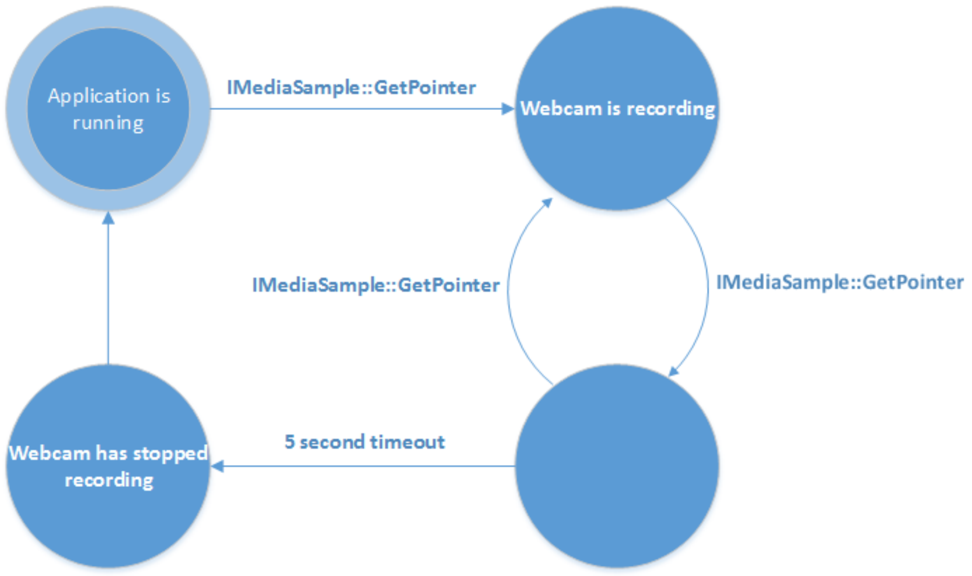


Figure 45: The finite-state machine for detecting if the webcam is recording.

6.3 Microphone Activity

Figure 46 shows the finite-state machine for deciding if the microphone is recording. **waveIn-Start** and **waveInStop** are performed as the microphone starts and stops recording, respectively, and can therefore be used to model this behavior.

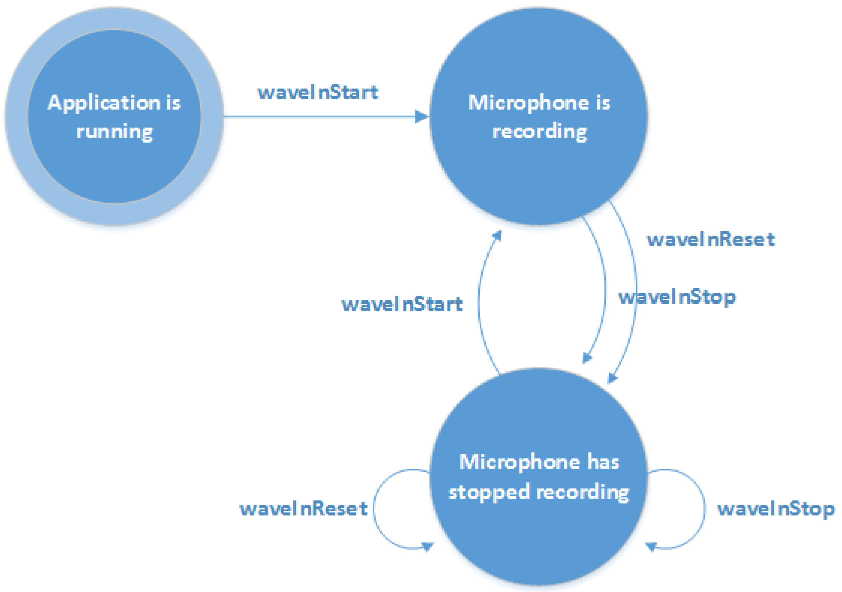


Figure 46: The finite-state machine for detecting if the microphone is recording.

6.4 Keylogger Activity

Figure 47 shows the finite-state machine for detecting if the keylogger is active or not. By looking at the API calls **ToAscii** or **ToUnicodeEx**, we can observe which keypresses that are observed by the given program.

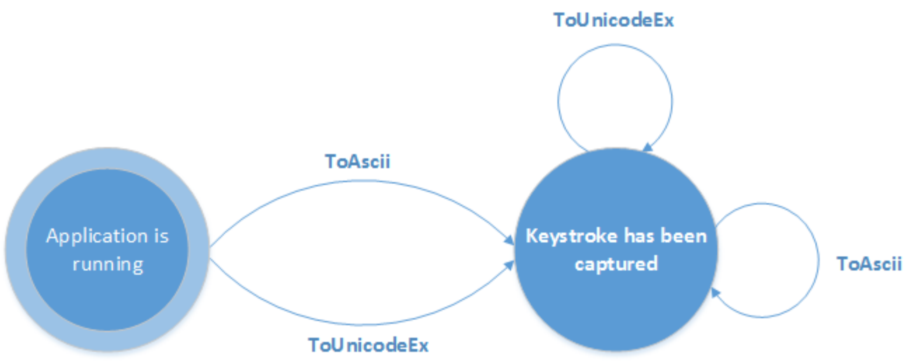


Figure 47: The FSM for detecting keylogging.

6.5 User Interactivity

The behavior models by themselves are sadly insufficient for detecting the presence of a RAT because, as we have seen in the previous chapters, other applications also perform the same API calls for the same behavior. The missing piece here is the combination of behavior API calls and the absence of user interactivity, and we can therefore use this information to create a model for distinguishing them.

Our hypothesis has been, as stated previously, that a RAT will try to stay hidden and therefore not show itself or communicate with the user. Although some of the RATs do provide the ability to send messages to the user, this is something that would be a big clue for the user that they are infected. This lack of visible interaction would be a indicator that something is wrong when combined with behavior that usually requires human interaction.

We therefore choose to regard this behavior as strange, seen as webcam, microphone and keyboard usage inherently are interactive tasks that prompts user interaction. It is thus these suspicious states where information is being recorded but no interactivity with the user is detected that we are interested in observing and modeling. If we then add the user interactivity API calls we get a finite-state machine as shown in Figure 48. This the complete model containing all the behavior seen previously, and should therefore be a suitable model for detecting RAT behavior.

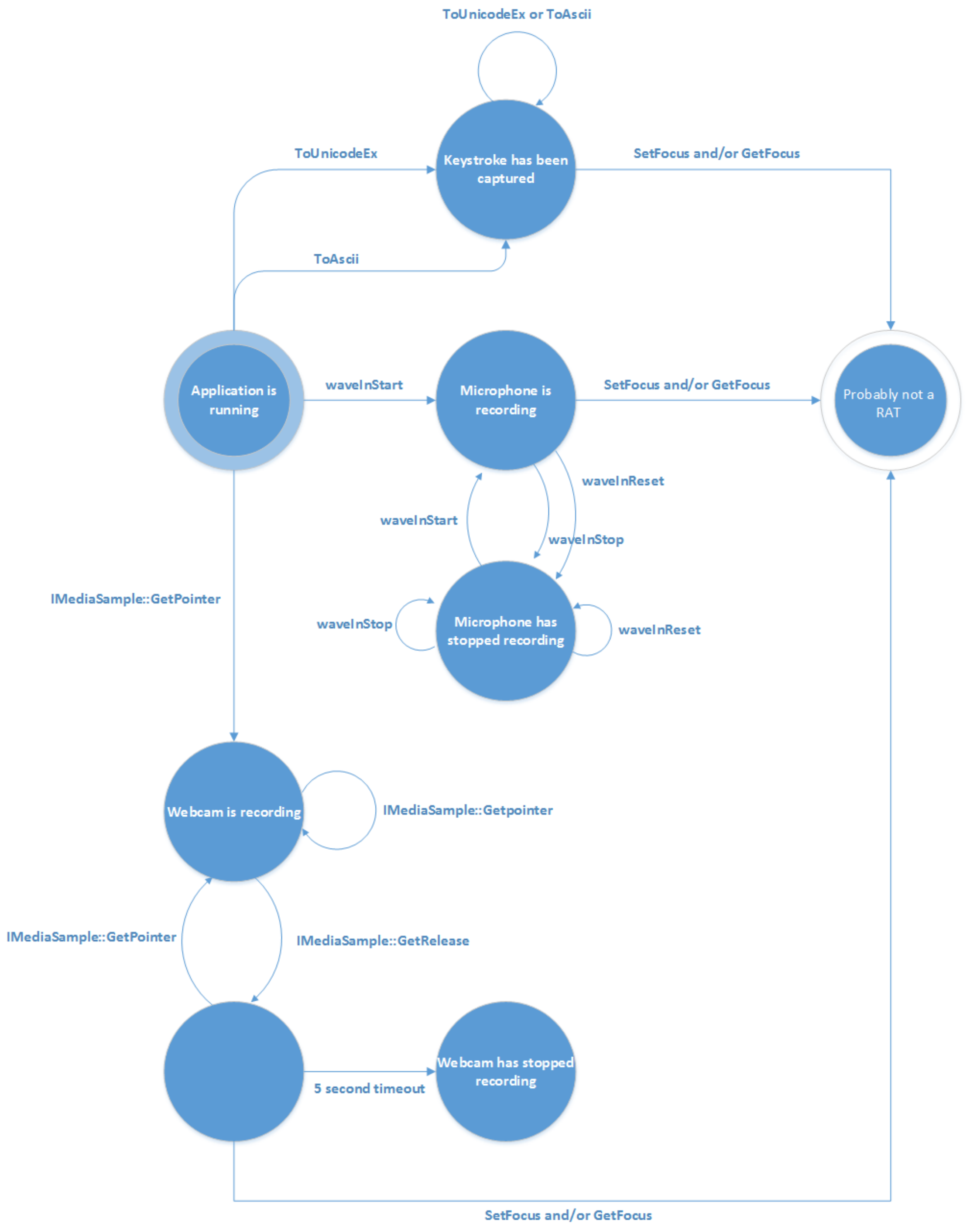


Figure 48: The complete behavior model.

6.6 Network Activity

As we also have seen previously, the RATs have different unique identifiers in the network traffic that can also be used to pinpoint exactly which RAT is present. Figure 49 shows how we can use some of the network traffic to detect what is going on in which RAT.

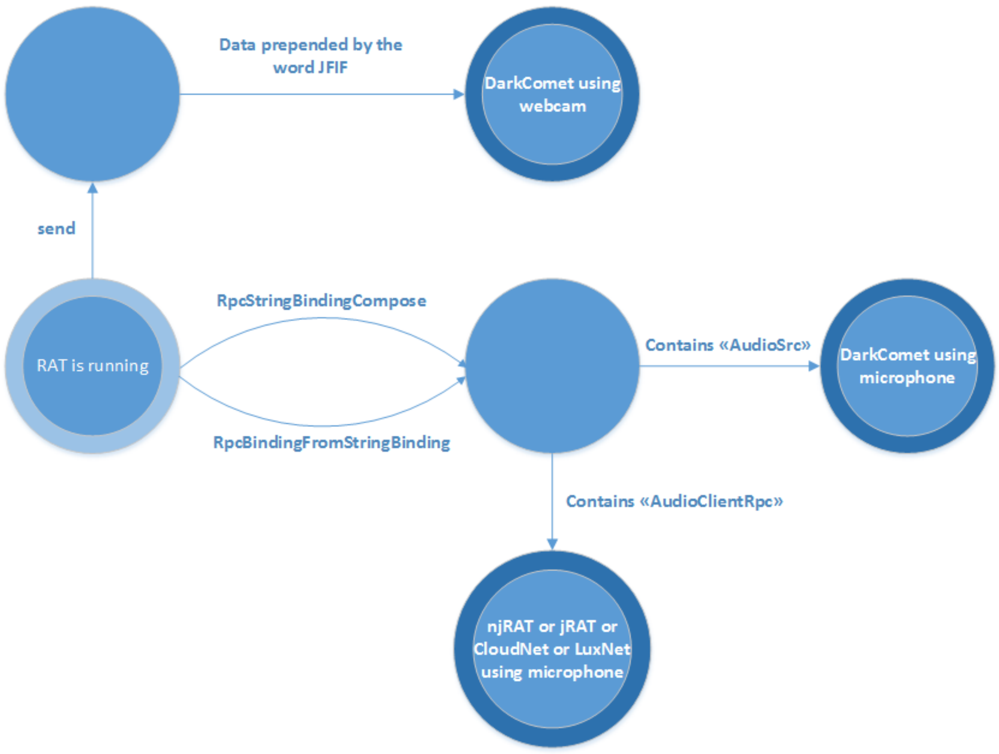

Data prepended by the word JFIF
DarkComet using webcam
send
RpcStringBindingCompose
RAT is running
RpcBindingFromStringBinding
Contains «AudioSrc»
DarkComet using microphone
Contains «AudioClientRpc»
njRAT or jRAT or CloudNet or LuxNet using microphone

Figure 49: Network traffic finite-state machine.

6.7 Summary and Research Questions

We should round of this section by doing a summary of our contributions and how they relate to the research questions we identified at the start of this thesis. Each question is listed below with a discussion of how we have fulfilled them.

6.7.1 What kind of behavior is unique for a RAT?

This research question was answered in Section 4.1 where we theorized which behavior that we could expect to see in the RATs. We have shown that this behavior did indeed occur in our analysis in Section 5, and that based on our analysis, this information could be used to discriminate a RAT from a program with similar behavior. What we also have seen is that most of the RATs made the same API calls for the same behavior, meaning that it was a straight-forward process to create general models for it.

6.7.2 Which API calls can be used to discover this behavior?

This question was attempted answered in Section 5. As we saw there, we defined three behavior features that we wanted to look at, namely webcam, microphone and keylogging. By studying the six RATs we were able to identify a set of API calls that can be used to indicate when they perform the three types of behavior, and we feel that we have answered this question in our analysis. The unique part of our work is which behavior we looked at, and we have not seen any other work that looks at the same indicators or the same behavior as we have, or who has concentrated on detecting RATs. Some similar work has been done by Gao et al. [34], Charlier et al. [2] and Kirda et al. [62] who also looked at how to detect different behavior with API calls, but they have differed in the kinds of behavior that they looked at.

6.7.3 How can we use this information to discriminate a RAT from a benign program?

This question has been answered in Section 5.9 where we looked at which API calls that appeared when user interaction with a program could be registered. We used this information to separate a RAT from a benign program that looks like a RAT by theorizing that the user would interact with the benign program, but not the RAT. This was shown to be mostly true in Section 5.9 where we saw that programs without a GUI made no API calls for user interaction, and thus could be considered as having behavior that indicated that it wanted to hide.

6.7.4 Is it possible to model the RATs behavior with a finite-state machine and detect them based on that?

We have shown how we can translate the observed behavior from Section 5 into finite-state machines that describes how the API calls puts the system into different states, i.e. performs specific behavior. By modeling this we have shown that the API calls can be simplified into models that detect the different types of behavior, based on very little information.

6.8 Discussion

The answers to these research questions also illustrates our contributions. Our main contribution has been our proposed model for detecting RATs based on behavior and user interaction. The next logical addition to this work would be to get a larger sample size of RATs and benign programs and test our proposed models better to see how it holds up against other RATs and programs, as the limited sample size is a weakness in our current work. We have not looked exhaustively at all the API calls available for detecting various types of behavior either, nor have we tried to create scenarios that may fool our proposed model. But based on the results we have obtained so far, we believe that API call monitoring is a viable technique for detecting RATs and that it might be a suitable solution for the future.

Below is a summary of the strengths and weaknesses of our proposed solution and a discussion of each of these points.

6.8.1 Advantages of Our System

- Can detect new samples based on behavior - By using the observed behavior and the created models, we believe that we can detect new RATs based on their API calls. As we have seen, the RATs make the same API calls for the same behavior and if we have observed it once we can see who matches it.

- Fast - The hooking mechanism employed by Rohitab API Monitor has in our experience been fast and provided little performance impact. We have not performed any performance analysis on the tool or other hooking techniques, but Microsoft did in their Detours paper[31] state that they had a 3% overhead when using the proposed hooking technique.
- Scalable - As we have shown, our behavior models can easily be expanded by adding new behavior. The webcam, microphone and keylogger behavior is just a start, we could also add functionality for detecting remote controlling of the screen or transferring personal files.
- Accurate - By observation we have shown that our model can be used for accurately predicting which RAT is running by modeling the specific API calls of that RAT. With our combination of API calls and parameters we are able to gather unique identifiers for each RAT, and can thus create a model that accurately pin-points the exact RAT that is present and what it is doing. By using more unique parameters and API calls we should be able to become even more accurate.
- Robust - Our model should have a greater reliability when considering offshoots with minor variations from the same malware family by considering features that usually remains constant, such as network traffic and libraries used for implementing functionality. Thereby we can remain unaffected by obfuscation techniques since we care about the performed API calls, and not about signatures based on byte sequences.

6.8.2 Disadvantages of Our System

- Requires manual updating - Currently, the model has to be updated with new RAT identifiers manually. Our system will be able to predict that a RAT might be present based on general identifiers such as the presence of webcam recording and no user interaction, but will not be able to pinpoint the exact RAT that is present since it has not been updated to know about the newest families based on network traffic. Updating with new families does thus have to be done manually, and does require some effort by manually examining the API calls and looking for relevant identifiers.
- May be easy to spoof - We have not tested how our system can be fooled. It is possible that a malicious program can perform API calls that are related user interaction, but not visible for the user, by e.g. creating a 1x1 pixel window or else calling the appropriate APIs without doing anything. We have not examined the possibility of doing this, but it could be considered further work.

7 Conclusion

In this master's thesis we have shown how Application Programming Interface (API) calls can be used to detect and model the behavior of Remote Administration Trojans (RAT). We have looked at the following features of six different RATs and used the API calls related to performing this behavior to detect when it is occurring:

- Use of the webcam
- Use of the microphone
- Logging of keystrokes
- User interactivity with the program
- Identifiers in network traffic

Our main findings are that the majority of the examined RATs appear to use the same API calls when exhibiting the same behavior, and by looking for specific API calls we can determine when a specific action, such as activating the webcam, is happening.

This information is not too useful on its own however, since we also have shown that legitimate programs such as Skype and Google Hangouts make many of the same API calls as the RATs when performing the same behavior, so we have also looked at API calls related to the user interacting with the program exhibiting the examined behavior.

The theory behind this has been that the RATs want to stay hidden, and will therefore not make any API calls that are related to user interaction, such as showing windows or dialog boxes. We have shown that this is true by comparing user interactivity API calls from the RATs with those from benign programs, and we show that the RATs do not make API calls related to user interaction, with one exception. An example of this is that both the RAT and Notepad.exe will register the user's keystrokes, but only Notepad.exe will make such API calls as **GetFocus** and **SetFocus** which relates to capturing the user's keyboard's attention into a given window. This means that the RAT will make API calls for capturing keystrokes, just like Notepad.exe, but it will do it silently, and not have a window and reveal itself to the user.

In addition, we have looked at the API calls that are used for network traffic when the RATs communicate with their master, and we show that several of the RATs give away indicators about what kind of actions it is about to perform and that this therefore can be used to predict what the RAT is up to.

We use all of this information to create finite-state machines that can be used to detect new RATs based on their exhibited behavior and their interaction with the users. The goal of this is to detect the RATs based on their behavior, rather than on classical byte-pattern signatures.

Our main conclusion, based on the data we have collected, is that it is possible to identify what kind of actions a program is performing, e.g. webcam or microphone recording, and if this is performed by a RAT or a benign program based on if the user interacts with the program or not. By modeling the API calls used for this behavior, we believe that we can detect new RATs based on the actions they perform.

8 Future Work

An obvious addition to our work would be to include a larger sample size of both malicious and benign programs to identify new API calls that might be used by other programs and to see if our proposed models still hold true as the sample size increases.

Our models have been created using manual analysis of the RATs, which has been time-consuming, and automating parts of this process could be a huge time-saver.

Another improvement would be the ability to automatically run and extract relevant information from RATs to build more accurate detection rules. The nature of the RATs does make them quite dependent on manual control for input, and it might therefore be difficult to automate. Sekar et al. has however proposed a system for automatically creating finite-state machines from program behavior [38].

Another obvious enhancement would be to look at more RATs and benign programs to better determine if there are cases where it is more difficult to separate a RAT from a benign program, i.e. situations where RATs make API calls related to user activity, or where benign programs that uses webcam, microphone, etc. does not. Such cases would require more fine-tuning of the rules and could potentially invalidate our proposed model by showing that the false-positive rate would make it infeasible of the data set becomes large enough.

Performance could also become an issue, but we have not tested it yet, but our subjective experience from our testing so far has been that as long as we do not hook several thousand different API calls at once, performance is negligible and not an issue. Microsoft quoted figures of 3% performance decrease when proposing the Detours hooking system[31]. Quantifying the performance impact and comparing it with similar systems would be an interesting indicator of its viability however.

Another interesting observation is that some RATs refuse to run if they detect that they are being hooked while launching. Creating a system for hooking all process at startup could actually prevent some malicious programs from running by using their anti-analysis tricks against them.

Bibliography

[1] Krebs, B. 2012. The Scrap Value of a Hacked PC, Revisited. http://krebsonsecurity.com/2012/10/the-scrap-value-of-a-hacked-pc-revisited/. [Online; accessed 20-May-2014].

[2] Charlier, B. L., Mounji, A., & Swimmer, M. 1995. Dynamic detection and classification of computer viruses using general behaviour patterns. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.1515.

[3] Symantec Security Response. 2014. Simple njRAT Fuels Nascent Middle East Cybercrime Scene. http://www.symantec.com/connect/blogs/simple-njrat-fuels-nascent-middle-east-cybercrime-scene. [Online; accessed 1-April-2014].

[4] Eva Galperin, M. M.-B. & Scott-Railton, J. 2013. Quantum of Surveillance: Familiar Actors and Possible False Flags in Syrian Malware Campaigns A joint Citizen Lab - EFF Report. https://www.eff.org/files/2013/12/28/quantum_of_surveillance4d.pdf. [Online; accessed 1-April-2014].

[5] Dawda, U. & Villeneuve, N. 2013. Njw0rm – Brother From the Same Mother. http://www.fireeye.com/blog/technical/malware-research/2013/08/njw0rm-brother-from-the-same-mother.html. [Online; accessed 29-March-2014].

[6] Norton. 2011. Norton Cybercrime Report 2011. http://us.norton.com/content/en/us/home_homeoffice/html/cybercrimereport/. [Online; accessed 2-May-2014].

[7] Anderson, N. 2013. Meet the men who spy on women through their webcams. http://arstechnica.com/tech-policy/2013/03/rat-breeders-meet-the-men-who-spy-on-women-through-their-webcams/. [Online; accessed 3-March-2014].

[8] Farivar, C. 2014. Sextortionist who hacked Miss Teen USA's computer sentenced to 18 months. http://arstechnica.com/tech-policy/2014/03/sextortionist-who-hacked-miss-teen-usas-computer-sentenced-to-18-months/. [Online; accessed 14-March-2014].

[9] Krebs, B. 2014. 'Blackshades' Trojan Users Had It Coming. http://krebsonsecurity.com/2014/05/blackshades-trojan-users-had-it-coming/. [Online; accessed 20-May-2014].

[10] Gallagher, R. & Greenwald, G. 2014. How the NSA Plans to Infect 'Millions' of Computers with Malware. `https://firstlook.org/theintercept/article/2014/03/12/nsa-plans-infect-millions-computers-malware/`. [Online; accessed 29-March-2014].

[11] Jyothsna, V., Prasad, V. V. R., & Prasad, K. M. August 2011. Article: A review of anomaly based intrusion detection systems. *International Journal of Computer Applications*, 28(7), 26–35. Published by Foundation of Computer Science, New York, USA.

[12] Villeneuve, N. & Bennett, J. 2012. Detecting APT Activity with Network Traffic Analysis. `http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-detecting-apt-activity-with-network-traffic-analysis.pdf`. [Online; accessed 6-April-2014].

[13] Bishop, M. 2003. *Computer Security: Art and Science*. Addison-Wesley.

[14] Rouse, M. 2009. RAT (remote access Trojan). `http://searchsecurity.techtarget.com/definition/RAT-remote-access-Trojan`. [Online; accessed 2-April-2014].

[15] Tanenbaum, A. S. 2007. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

[16] Cult of the Dead Cow. 1998. Back Orifice Windows Remote Administration Tool. `http://www.cultdeadcow.com/tools/bo.html`. [Online; accessed 3-March-2014].

[17] Cult of the Dead Cow. 1998. RUNNING A MICROSOFT OPERATING SYSTEM ON A NETWORK? OUR CONDOLENCES. `http://www.cultdeadcow.com/news/back_orifice.txt`. [Online; accessed 3-March-2014].

[18] Microsoft. 1998. Microsoft Security Bulletin MS98-010). `http://technet.microsoft.com/en-us/security/bulletin/ms98-010`. [Online; accessed 2-April-2014].

[19] Poole, O. 2007. *Network Security*. Taylor & Francis.

[20] diggerx, fluffy-stuff and novice222. 2013. BO2K. `http://sourceforge.net/projects/bo2k/`. [Online; accessed 3-March-2014].

[21] Codius. 2008. Poison Ivy - Remote Administration Tool. `http://www.poisonivy-rat.com`. [Online; accessed 3-March-2014].

[22] Kindlund, D. 2013. Poison Ivy: Assessing Damage and Extracting Intelligence. `http://www.fireeye.com/blog/technical/targeted-attack/2013/08/pivy-assessing-damage-and-extracting-intel.html`. [Online; accessed 3-March-2014].

[23] Dang, B., Gazet, A., Bachaalany, E., & Josse, S. 2014. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Wiley, 111 River Street, Hoboken, NJ, USA, 1st edition.

[24] Bridgwater, A. 2012. What is Signature Based Detection? http://blogs.avg.com/business/signature-based-detection/. [Online; accessed 20-February-2014].

[25] Aycock, J. 2006. *Computer Viruses and Malware*. Advances in Information Security. Springer.

[26] Jacob, G., Debar, H., & Filiol, E. 2008. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3), 251–266.

[27] Microsoft. 2013. Dynamic-Link Libraries. http://msdn.microsoft.com/en-us/library/ms682589%28v=vs.85%29.aspx. [Online; accessed 18-February-2014].

[28] Russinovich, M. E., Solomon, D. A., & Ionescu, A. 2012. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. Microsoft Press, 6th edition.

[29] Microsoft. 2014. Hooks Overview. http://msdn.microsoft.com/en-us/library/windows/desktop/ms644959%28v=vs.85%29.aspx. [Online; accessed 19-February-2014].

[30] Hoglund, G. & Butler, J. 2006. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Software Security Series. Addison-Wesley.

[31] Hunt, G. & Brubacher, D. 1999. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, WINSYM'99, 14–14, Berkeley, CA, USA. USENIX Association.

[32] Microsoft. 2014. CallNextHookEx function. http://msdn.microsoft.com/en-us/library/windows/desktop/ms644974%28v=vs.85%29.aspx. [Online; accessed 5-April-2014].

[33] Kosoresow, A. P. & Hofmeyr, S. A. September 1997. Intrusion detection via system call traces. *IEEE Softw.*, 14(5), 35–42.

[34] Gao, D., Reiter, M. K., & Song, D. X. 2004. On gray-box program tracking for anomaly detection. In *USENIX security symposium*, 103–118.

[35] Rosen, K. 2007. *Discrete Mathematics and Its Applications*. McGraw-Hill international edition. McGraw-Hill Higher Education.

[36] Forrest, S., Hofmeyr, S., & Somayaji, A. 2008. The evolution of system-call monitoring. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, 418–430, Washington, DC, USA. IEEE Computer Society.

[37] Garfinkel, T. 2003. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proc. Network and Distributed Systems Security Symposium*, 163–176.

[38] Sekar, R., Bendre, M., Dhurjati, D., & Bollineni, P. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *In Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 144–155.

[39] Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., & Kirda, E. 2012. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, 122–132, New York, NY, USA. ACM.

[40] TANDON, G. & CHAN, P. K. 2006. On the learning of system call attributes for host-based anomaly detection. *International Journal on Artificial Intelligence Tools*, 15(06), 875–892.

[41] Faruki, P., Laxmi, V., Gaur, M. S., & Vinod, P. 2012. Behavioural detection with api call-grams to identify malicious pe files. In *Proceedings of the First International Conference on Security of Internet of Things*, SecurIT '12, 85–91, New York, NY, USA. ACM.

[42] Pu, S. & Lang, B. 2007. An intrusion detection method based on system call temporal serial analysis. In *Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues*, Huang, D.-S., Heutte, L., & Loog, M., eds, volume 4681 of *Lecture Notes in Computer Science*, 656–666. Springer Berlin Heidelberg.

[43] Pungila, C.-P. 2009. A bray-curtis weighted automaton for detecting malicious code through system-call analysis. In *Proceedings of the 2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '09, 392–400, Washington, DC, USA. IEEE Computer Society.

[44] Ahmed, F., Hameed, H., Shafiq, M. Z., & Farooq, M. 2009. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, AISec '09, 55–62, New York, NY, USA. ACM.

[45] Qiao, Y., He, J., Yang, Y., & Ji, L. July 2013. Analyzing malware by abstracting the frequent itemsets in api call sequences. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, 265–270.

[46] Liu, S.-T., ching Huang, H., & Chen, Y.-M. 2011. A system call analysis method with mapreduce for malware detection. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, 631–637.

[47] Li, Z., Wang, X., Liang, Z., & Reiter, M. 2008. Agis: Towards automatic generation of infection signatures. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, 237–246.

[48] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., & Wang, X. 2009. Effective and efficient malware detection at the end host. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, 351–366, Berkeley, CA, USA. USENIX Association.

[49] Ammar Ahmed E. Elhadi, Mohd Aizaini Maarof, B. I. A. B. 2013. Improving the detection of malware behaviour using simplified data dependent api call graph. In *International Journal of Security and Its Applications Vol.7, No.5 (2013)*, 29–42.

[50] Salehi, Z., Ghiasi, M., & Sami, A. 2012. A miner for malware detection based on api function calls and their arguments. In *Artificial Intelligence and Signal Processing (AISP), 2012 16th CSI International Symposium on*, 563–568.

[51] Yan, G., Brown, N., & Kong, D. 2013. Exploring discriminatory features for automated malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Rieck, K., Stewin, P., & Seifert, J.-P., eds, volume 7967 of *Lecture Notes in Computer Science*, 41–61. Springer Berlin Heidelberg.

[52] Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., & Hamze, A. 2010. Malware detection based on mining api calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, 1020–1025, New York, NY, USA. ACM.

[53] Zhao, H., Xu, M., Zheng, N., Yao, J., & Ho, Q. 2010. Malicious executables classification based on behavioral factor analysis. In *e-Education, e-Business, e-Management, and e-Learning, 2010. IC4E '10. International Conference on*, 502–506.

[54] Tian, R., Islam, M., Batten, L., & Versteeg, S. 2010. Differentiating malware from cleanware using behavioural analysis. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, 23–30.

[55] Moffie, M., Cheng, W., Kaeli, D., & Zhao, Q. 2006. Hunting trojan horses. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, 12–17, New York, NY, USA. ACM.

[56] Park, Y., Reeves, D. S., & Stamp, M. 2013. Deriving common malware behavior through graph clustering. *Computers & Security*, 39, Part B(0), 419 – 430.

[57] Wagener, G., State, R., & Dulaunoy, A. 2008. Malware behaviour analysis. *Journal in Computer Virology*, 4(4), 279–287.

[58] Jang, M., Kook, J., Ryu, S., Lee, K., Shin, S., Kim, A., Park, Y., & Cho, E. H. 2013. An efficient similarity comparison based on core api calls. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, 1634–1638, New York, NY, USA. ACM.

[59] Kim, D., Han, Y., Cho, S.-j., Yoo, H., Woo, J., Nah, Y., Park, M., & Chung, L. 2013. Measuring similarity of windows applications using static and dynamic birthmarks. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, 1628–1633, New York, NY, USA. ACM.

[60] Dunham, K. 2011. Malcode Context of API Abuse. `https://www.sans.org/reading-room/whitepapers/malicious/malcode-context-api-abuse-33649`. [Online; accessed 18-February-2014].

[61] Bailey, M., Oberheide, J., Andersen, J., Mao, Z., Jahanian, F., & Nazario, J. 2007. Automated classification and analysis of internet malware. In *Recent Advances in Intrusion Detection*, Kruegel, C., Lippmann, R., & Clark, A., eds, volume 4637 of *Lecture Notes in Computer Science*, 178–197. Springer Berlin Heidelberg.

[62] Kirda, E., Kruegel, C., Banks, G., Vigna, G., & Kemmerer, R. A. 2006. Behavior-based spyware detection. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA. USENIX Association.

[63] Rieck, K., Holz, T., Willems, C., Düssel, P., & Laskov, P. 2008. Learning and classification of malware behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, 108–125, Berlin, Heidelberg. Springer-Verlag.

[64] Saxe, J., Mentis, D., & Greamo, C. July 2013. Mining web technical discussions to identify malware capabilities. In *Distributed Computing Systems Workshops (ICDCSW), 2013 IEEE 33rd International Conference on*, 1–5.

[65] Christodorescu, M., Jha, S., & Kruegel, C. 2007. Mining specifications of malicious behavior. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, 5–14, New York, NY, USA. ACM.

[66] Mehdi, S. B., Tanwani, A. K., & Farooq, M. 2009. Imad: In-execution malware analysis and detection. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, 1553–1560, New York, NY, USA. ACM.

[67] Batra, R. 2014. API Monitor. `http://www.rohitab.com/apimonitor`. [Online; accessed 15-May-2014].

[68] Potier, J. 2014. WinAPIOverride. `http://jacquelin.potier.free.fr/winapioverride32/`. [Online; accessed 15-May-2014].

[69] Visicom Media Inc. 2014. Manycam. `http://manycam.com/`. [Online; accessed 15-May-2014].

[70] Confuser Team. 2012. Confuser - Home. `https://confuser.codeplex.com/`. [Online; accessed 14-March-2014].

[71] Lafortune, E. 2013. ProGuard. `http://proguard.sourceforge.net/`. [Online; accessed 15-May-2014].

[72] Lesueur, J.-P. 2012. DarkComet RAT Official Website. `http://darkcomet-rat.com/`. [Online; accessed 6-March-2014].

[73] Gonsalves, A. 2012. DarkComet shut-down shows law enforcement works. `http://www.csoonline.com/article/710379/darkcomet-shut-down-shows-law-enforcement-works`. [Online; accessed 6-March-2014].

[74] Kujawa, A. 2012. You Dirty RAT! Part 1 – DarkComet. `http://blog.malwarebytes.org/intelligence/2012/06/you-dirty-rat-part-1-darkcomet/`. [Online; accessed 3-March-2014].

[75] Villeneuve, N. 2012. Fake Skype Encryption Service Cloaks DarkComet Trojan. http://blog.trendmicro.com/trendlabs-security-intelligence/fake-skype-encryption-software-cloaks-darkcomet-trojan/. [Online; accessed 6-March-2014].

[76] Microsoft. 2014. IMediaSample::GetPointer method. http://msdn.microsoft.com/en-us/library/windows/desktop/dd407010%28v=vs.85%29.aspx. [Online; accessed 6-March-2014].

[77] Microsoft. 2014. waveInStart function. http://msdn.microsoft.com/en-us/library/windows/desktop/dd743851%28v=vs.85%29.aspx. [Online; accessed 6-March-2014].

[78] Microsoft. 2014. waveInClose function. http://msdn.microsoft.com/en-us/library/windows/desktop/dd743840%28v=vs.85%29.aspx. [Online; accessed 6-March-2014].

[79] Microsoft. 2014. MapVirtualKey function. http://msdn.microsoft.com/en-us/library/windows/desktop/ms646306%28v=vs.85%29.aspx. [Online; accessed 6-March-2014].

[80] Microsoft. 2014. ToAscii function. http://msdn.microsoft.com/en-us/library/windows/desktop/ms646316%28v=vs.85%29.aspx. [Online; accessed 25-May-2014].

[81] Solutions, F. C. 2013. “njRAT” Uncovered. http://threatgeek.typepad.com/files/fta-1009---njrat-uncovered-1.pdf. [Online; accessed 29-March-2014].

[82] hackforums.net. 2014. njRAT v0.7 Setup/No-Ip/Detailed/Pictures. http://www.hackforums.net/showthread.php?tid=4019465. [Online; accessed 17-May-2014].

[83] jRAT. 2014. jRAT. http://jrat.su. [Online; accessed 6-March-2014].

[84] Payet, L. 2014. JRAT Targets UK and UAE in Payment Certificates Spam Campaign. http://www.symantec.com/connect/blogs/jrat-targets-uk-and-uae-payment-certificates-spam-campaign. [Online; accessed 7-March-2014].

[85] jRAT. 2014. jRAT Changelog. http://jrat.su/api/changelog.php. [Online; accessed 18-May-2014].

[86] jSpy. 2014. jSpy RAT - Developed in Java. http://jstealth.co.uk/jspy/. [Online; accessed 18-May-2014].

[87] Coding4Noobs. 2014. Cloud Net - Beta. http://cloudnetbeta.blogspot.ca/. [Online; accessed 13-March-2014].

[88] XilluX. 2014. LuxNET | Alpha 2.0 | Free Remote Administration Tool | A must to check out! http://www.hackforums.net/showthread.php?tid=4082457. [Online; accessed 18-May-2014].

[89] Skype. 2014. Skype - Free internet calls and online cheap calls to phones and mobiles. `http://www.skype.com`. [Online; accessed 19-May-2014].

[90] Microsoft. 2014. About WASAPI. `http://msdn.microsoft.com/en-us/library/windows/desktop/dd371455%28v=vs.85%29.aspx`. [Online; accessed 19-May-2014].

[91] Google. 2014. Google+ Hangouts – Google Hangouts. `https://www.google.com/hangouts/`. [Online; accessed 19-May-2014].

[92] Microsoft. 2014. GetForegroundWindow function. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms633505%28v=vs.85%29.aspx`. [Online; accessed 6-March-2014].

[93] Krebs, B. 2014. Antivirus is Dead: Long Live Antivirus! `http://www.krebsonsecurity.com/2014/05/antivirus-is-dead-long-live-antivirus/`. [Online; accessed 9-May-2014].

[94] VirusTotal. 2014. VirusTotal - Free Online Virus, Malware and URL Scanner. `https://www.virustotal.com/`. [Online; accessed 19-May-2014].

[95] Microsoft. 2014. IMediaSample::GetPointer method. `http://msdn.microsoft.com/en-us/library/windows/desktop/dd407010%28v=vs.85%29.aspx`. [Online; accessed 26-May-2014].

[96] Microsoft. 2014. waveInStop function. `http://msdn.microsoft.com/en-us/library/windows/desktop/dd743852%28v=vs.85%29.aspx`. [Online; accessed 26-May-2014].

[97] Microsoft. 2014. waveInReset function. `http://msdn.microsoft.com/en-us/library/windows/desktop/dd743850%28v=vs.85%29.aspx`. [Online; accessed 28-May-2014].

[98] Microsoft. 2014. IAudioClient::Start method. `http://msdn.microsoft.com/en-us/library/windows/desktop/dd370879%28v=vs.85%29.aspx`. [Online; accessed 26-May-2014].

[99] Microsoft. 2014. IAudioClient::Stop method. `http://msdn.microsoft.com/en-us/library/windows/desktop/dd370880%28v=vs.85%29.aspx`. [Online; accessed 26-May-2014].

[100] Microsoft. 2014. ToUnicodeEx. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms646322%28v=vs.85%29.aspx`. [Online; accessed 26-May-2014].

[101] Microsoft. 2014. SetFocus function. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms646312%28v=vs.85%29.aspx`. [Online; accessed 6-March-2014].

[102] Microsoft. 2014. GetFocus function. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms646294%28v=vs.85%29.aspx`. [Online; accessed 25-May-2014].

[103] Microsoft. 2014. send function. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms740149%28v=vs.85%29.aspx`. [Online; accessed 28-May-2014].

[104] Microsoft. 2014. RpcStringBindingCompose function. http://msdn.microsoft.com/en-us/library/windows/desktop/aa378481%28v=vs.85%29.aspx. [Online; accessed 28-May-2014].

[105] Microsoft. 2014. RpcBindingFromStringBinding function. http://msdn.microsoft.com/en-us/library/windows/desktop/aa375590%28v=vs.85%29.aspx. [Online; accessed 28-May-2014].