Master thesis

Detecting malicious behaviour using system calls

Vincent Van Mieghem

Delft University of Technology

Master thesis

Detecting malicious behaviour using system calls

by

Vincent Van Mieghem

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on 14th of July 2016 at 15:00.

Student number:	4113640	
Project duration:	November 2, 2015 – June 30, 2016	
Thesis committee:	Prof. dr. ir. J. van den Berg,	TU Delft
	Dr. ir. C. Doerr,	TU Delft, supervisor
	Dr. ir. S. Verwer,	TU Delft, supervisor
	Dr. J. Pouwelse,	TU Delft
	M. Boone,	Fox-IT, supervisor

Acknowledgements

I would first like to thank my thesis supervisors Christian Doerr and Sicco Verwer for their supervision and valuable feedback during this work.

I would also like to thank Maarten Boone from Fox-IT. Without his extraordinary amount of expertise and ideas, this Master thesis would not have been possible.

I would like to thank several people in the information security industry. Pedro Vilaça for his work on bypassing XNU kernel protections. Patrick Wardle and Xiao Claud for their generosity in sharing OS X malware samples. VirusTotal for providing an educational account on their terrific service.

Finally, I must express my very profound gratitude to my parents, brother and my girlfriend for providing me with support and encouragement throughout my years of study and through the process of researching and writing this thesis.

This accomplishment would not have been possible without them.

Thank you.

Vincent Van Mieghem
Delft, June 2016

Contents

1 Abstract

The emergence of Apple's Macintosh computers' popularity introduces new threats and challenges for the security on the Mac. For a long time, OS X security has benefitted from the popularity of Microsoft Windows. The threat landscape for the Mac is rapidly changing as the marketshare of the Mac is approaching 15%[1]. Malware on Apple's OS X systems emerges to be an increasing security threat that is currently solely countered with ancient anti-virus (AV) technologies [18]. Current AV technologies pose a performance overhead on the entire system and have an inherent delayed effectiveness, due to their signature based detection [15][31]. In addition, current malware uses many forms of obfuscation to prevent detection by AV technologies, redering AV technologies useless against advanced threats [15][31]. Consequently, the need for more advanced detection and prevention techniques of malware is increasing. Detection of malicious behaviour instead of malicious signatures, ought to provide a more advanced form of protection. A system call is referred to as the request and service of specific, basic, functionality provided to applications by the operating system.

This Master thesis answers the research question: "*Is it possible to detect malicious behaviour performed by malware, based on monitoring system calls?*"

Presented is a novel, generic, behavioural detection and prevention mechanism for malware on OS X based on system calls. System call traces can be used to describe the behaviour of processes [11]. Much effort was put into the development of a kernel module that bypasses kernel security mechanisms and rewires one of the operating system's core functionalities; system call handling. The rewiring of system call handling provided the ability to log all of the system call invocations performed by processes running on the monitored system. A significant amount of OS X malware and benign applications were executed in a monitored environment of which system call traces were collected. Based on analysing heat map visualisations and manual sequential analysis of the system call traces of both malicious and benign processes, anomalies in the malicious traces could be observed. Subsequently, several malicious system call patterns and detection rules were extracted providing detection of malware on OS X. The most successful defined pattern is constructed around the executions of Unix shell processes performed by malware. It is shown that this detection pattern results in a 100% detection rate of *all* malware possible to obtain for this thesis. Even advanced malware in an infected OS X application, known as OSX.KeyRanger.A, was detected using this method. In order to evaluate the False Positive Rate (FPR) accurately in real world scenarios, three different user profiles were defined. Applications distributed via the Mac App Store do not generate false positives. In case of the developer user profile type, the FPR increases to 20%. Applications responsible for the false positives feature a cross-platform nature, such as MATLAB, R, LaTeX and interpreters for scripting languages. A conducted survey under Mac users verified these conclusions. However, the number of false positive generating benign applications is very limited and whitelisting solutions provided can reduce the FPR in this developer user profile. The results of this Master thesis have been composed in a paper *"Behavioural detection and prevention of malware on Mac OS X"* (appendix A) and submitted to the IEEE CNS 2016 conference[2].

[1] `https://www.idc.com/getdoc.jsp?containerId=prUS41176916`
[2] `http://cns2016.ieee-cns.org/`

2 Introduction

The emergence of the Internet over the last two decades has impacted our society and environment tremendously. Nowadays, the interconnected world seems inseparable from common practices. The Internet gave birth to entire new industries of which their absence in today's world would be unimaginable. The amount of Internet connected devices and appliances, the so-called Internet of Things (IoT), is growing exponentially. Gartner estimates that in 2020 over 20 billion devices will be connected to the Internet[1], creating a huge potential of functionality. However, as society shifted to the online world, so did crime. Cyberattacks and cybercrime profited from the growth of the Web and was in 2014 alone responsible for $445 billion in losses [42]. The threats presented by the interconnected world are varying and the characteristics of the malicious software used (known as malware) are changing over time. Around the mid-2000s, more than a million known computer worms circulated around the Internet. E-mail spam was becoming big business as luring recipients of spam e-mail into clicking on the content of the e-mail was often successful. In the years after, malware authors professionalised and new actors entered the scene. Currently, large companies primarily try to defend their networks against cyber espionage and state-actor attacks that use a type of threat named APT (Advanced Persistent Threat), due to its stealthy and persisting nature[2]. Since 2013 a new type of malware is disrupting the information security within personal computers and corporate networks[3]. Ransomware[4] is taking the prominent first place in 2016 as most problematic cyberthreat, according to a Kaspersky Labs report in 2016[2]. Many malware samples use obfuscation techniques, some even based on Microsoft Office macros, to prevent detection making attacks more difficult to defend against when using traditional anti-virus (AV) technologies [15][31]. Similar to many other advanced types of malware, ransomware authors put much effort into obfuscating the malware to construct a seemingly unique, new signature. Signature based malware detection technologies not aware of the new signature then fail to trigger on this new signature, resulting in an evasion of the traditional AV system [15][31].

From a security point of view, Apple's Macintosh computers benefitted from the popularity of Microsoft Windows for a long time. Due to its low market share, malware authors did not invest in targeting Macs[5]. However, as the Mac is gaining popularity and market share, the number of malware targeting the Mac increases. A study performed by security firm Bit9 + Cabron Black [4] indicates that over the last three years an increasing growth of malware targeting OS X (the operating system of the Mac) systems is observed. Five times more OS X malware appeared in 2015 than during the previous five years combined. Many types of malware previously only appearing on Microsoft Windows systems are now also emerging on OS X systems. Also the APT's and ransomware are not absent on Macs anymore.

[1] `https://www.gartner.com/newsroom/id/3165317`

[2] `https://securelist.com/files/2016/05/Q1_2016_MW_report_FINAL_eng.pdf`

[3] `https://blog.fox-it.com/2015/09/07/the-state-of-ransomware-in-2015/`

[4] Ransomware is a type of malware that restricts access to the infected computer system in some way, and demands that the user pays a ransom to the malware operators to remove the restriction. Some forms of ransomware systematically encrypt files on the system's hard drive, which become difficult or impossible to decrypt without paying the ransom for the encryption key, while some may simply lock the system and display messages intended to hoax the user into paying.

[5] `http://www.digitaltrends.com/computing/can-macs-get-viruses/`

Similar to Microsoft Windows systems, Mac anti-virus technologies still heavily rely on binary signature checking. Due to the reasons mentioned above, a need for more advanced malware detection methods arises.

2.1. A brief introduction to system calls

Operating systems form a platform of functionalities for applications running on a computer. Many of the functionalities are concerned with the availability of operating system and hardware services used by the applications running on the operating system [6]. The implementation of many of these functionalities reside in a part of the operating system called “the kernel”[6]. The kernel is the heart of the operating system that directly interacts with the hardware (USB ports, keyboards, hard disk drives etc.) of the computer. The kernel forms the core layer on which all other functionality layers are built. Core functionality to interact with hardware and other kernel services are implemented in the kernel. These core functionalities define the possible interactions that applications running on the operating system are able to perform. This set of core functionalities can be requested by other parts of the system through a mechanism named “system calls” [17]. When an application running on the operating system wishes to use functionality provided by the operating system, it will request that functionality by calling the particular system call that is responsible for the functionality or service. An example of this would be an application trying to write a file to the file system. In this scenario, the application uses a `SYS_open` system call to request a file handle and later a `SYS_write` system call to write a stream of data to the file handle it previously obtained from the `SYS_open` system call. Other system call examples are calls to execute processes, request memory or open an Internet connection.

Typically, system calls are not called directly by applications[7] [17]. Instead, the system call functions are wrapped in core libraries provided by the operating system that are used by applications. The most well known library is the GNU C library also known as glibc on Unix systems and `ntdll.dll` on Microsoft Windows systems. These libraries typically wrap the system call into a richer API (Application Programming Interface) function [17]. API library functions can be called by programs running on the operating system to perform specific tasks. Examples of such tasks are writing to a file handle or displaying an application window on the screen of the user. Operating system provide many standard libraries containing API functions aimed to provide standardised functionality to applications. The “Win32 API” on Microsoft Windows[8] and Core Services[9] API on OS X systems are examples of such API libraries.

The system calls are the key subject of this Master thesis. In particular, this thesis is interested in the invocations of system calls by applications. The ordered sequence of system calls performed by processes from start of the process to the end, called a system call trace, may be a description of the behaviour it is trying to perform. The following sections elaborate.

2.2. CTMp Endpoint module (FoxGuard)

Fox-IT is a Dutch IT security firm. One of the services Fox-IT provides to their clients is detecting and preventing cyber threats from damaging clients infrastructure. Their ‘Security Operations Centre’ (SOC) is a well advertised room in which employees of Fox-IT monitor clients’ infrastructure for cyber threats.[10] A complete new version of the monitoring framework that is used by the SOC is named ‘Cyber Threat Management platform (CTMp)’ and is currently being developed by Fox-IT itself. Part of this platform is a module named ‘Endpoint’ (formerly known as FoxGuard), which is software deployed on the computers of the clients’ employees (so called endpoints) to primarily prevent intrusions. FoxGuard takes a different approach in protecting the endpoints than traditional anti-virus (AV) software. Where AV-software is mainly concerned with detecting malicious binaries on a system that are marked as malicious by the AV-producer using characteristic binary signatures, FoxGuard aims to prevent malware

[6]`http://www.linfo.org/kernel_space.html`

[7]`http://man7.org/linux/man-pages/man2/syscalls.2.html`

[8]`https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx`

[9]`https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CoreServicesLayer/CoreServicesLayer.html`

[10]`https://www.fox-it.com/en/products/managed-security-services/`

from executing by hardening the system and reducing the attack surface of the system. FoxGuard enforces system- and user-wide rules that define the permissions certain applications and processes are granted and especially, what permissions are not granted. When a program or process, i.e. a browser, is exploited by malware, the malware will not be able to perform operations outside the permissions of the browser, defined and enforced by FoxGuard. For example, FoxGuard may enforce:

- a read-only operation by the browser in the `C:\Windows\` directory, or
- only allow the browser to use port 80 (HTTP traffic) and 443 (HTTPS traffic), or
- constrain the browser the ability to execute binaries.

The possibilities are arguably endless, depending on the creativity of the rule maker. Rules are FoxGuards charm, but simultaneously introduce a great challenge. If a rule fails to cover proper restrictions to certain component in the system, those components can be exploited by malware. If a client uses certain software, unknown to FoxGuard, this may cause security issues.

2.3. Thesis outline

This Master thesis conducts research to improve protection abilities for endpoints by systems like FoxGuard. In particular, it focusses on the detection and possibly prevention of malicious behaviour based on system calls.

2.3.1. Research question

The primary goal of this research is to design a method that allows for detection of malicious behaviour based on monitoring system calls performed by processes on an OS X system. Based on monitoring system call invocations, a process performing malicious behaviour should be detected and thereafter prevented from further harming the system.

The research question is thus formulated as:

Is it possible to detect malicious behaviour performed by malware, based on monitoring system calls?

In particular, recurring patterns in the system call traces that define malicious behaviour are a key research aspect with respect to answering the research question. The subquestions formulated to answer the principal research questions are:

1. *What are the current detection techniques (proposed in literature)?* Answering this question provides relevant orientation for a new and improved method of detection, based on techniques proposed in literature.
2. *How can all system calls performed by processes be monitored and collected?* In order to obtain an understanding of the performed system calls by processes, the system call traces have to be collected for further analysis.
3. *What is an efficient method to analyse system call traces for anomalies?* An efficient analysis method is important for finding anomalies in collected system call traces of malicious processes.
4. *How can the anomalies be defined as elementary detection rules?* Ultimately, the definition of malicious behavioural rules may improve the detection and thus prevention of malware on systems. The elementary rules can be adopted and implemented by anti-malware solutions such as FoxGuard.

The next section outlines the approach taken to answer the research questions.

2.3.2. Research outline

In Chapter 3, prominent OS X malware are investigated to obtain a high-level understanding of their characteristics. In addition, evolutions of malware are described and state of the art practical detection techniques of malware on OS X. Chapter 4 provides a comprehensive literature study on the detection techniques proposed in research. The chapter distinguish two types of malware analysis methods and categorises the methods of detection based on those two types of analysis. This literature study is used to draw conclusions that can be used for the research performed in this thesis. Chapter 5 describes the possible options of monitoring system call traces and the subsequent development decisions for a kernel module used to collect system call traces. The chapter finalises by explaining the structure of the dataset collected after monitoring system calls. The dataset containing system call traces will be analysed for anomalies. Chapter 6 and 7 describe observations and two analysis methods used to define malicious system call patterns. Four malware detection patterns are constructed based on the performed analysis techniques which are evaluated in Chapter 8 according to established evaluation metrics derived from the literature study. A discussion of the results and explanations for the observations made, is explained in Chapter 9. The conclusion in Chapter 10 finally ends this Master thesis and provides concrete directions for future research.

3

Malware on OS X

While not as commonly appearing as on Microsoft Windows, the targets of malware on OS X appear to be very similar to malware on Windows operating systems. Phishing, adware, spyware and backdoor trojans are the most commonly seen malware types as shown in this chapter. However, malware with fairly different targets has been found by the security industry as well. OS X malware named 'XcodeGhost' and 'Wirelurker' are an example of this more advanced type of malware. This chapter provides an overview of the characteristics of malware found on OS X. Subsequently, it explains the current practical malware detection and prevention methods for OS X. In addition, a brief overview is provided of more advanced infection techniques that may be adopted by malware on OS X in the future.

3.1. A brief study of malware for OS X

Malware is a general term for any kind of malicious software. Viruses, trojans, worms, spyware, keyloggers, remote access software all fall under the umbrella of malware. Differences in functionality and installation methods are relevant, as it implicates differences in the traces the malware leaves behind on a system. This section provides a brief overview of the most impactful or prominent malware samples found the OS X platform in terms of used techniques and scale of infection. The overview specifically focusses on the characteristics and methods used by malware leaving traces that may be detected. The concluding subsection describes the most commonly used techniques and in addition, provides more sophisticated techniques that may be used by malware on OS X in the future.

3.1.1. Spyware

Spyware is malware that aims to gather information about users without their knowledge and sends that information to the attacker without the users consent [13]. On Microsoft Windows, spyware is mostly used for the purposes of tracking and storing Internet users' movements on the Web and serving up pop-up ads to Internet users. Whenever spyware is used for malicious purposes, its presence is typically hidden from the user and can be difficult to detect [13].

Renepo (2004)
According to ESET, a Slovakian antivirus company, the first malware specifically written for OS X emerged in 2004 [8]. Renepo was a Bash shell script that required admin privileges or write access to system areas and utilities. Once installed, it adds itself as a Startup Item and it remains root privileged. Startup Items are programs that launch upon user login to the system. Later versions were reported to install a backdoor and spyware functionality, stealing a wide range of application configurations and information, including passwords.

Hovdy (2010)
Hovdy is a family of OS X malware implemented using bash and AppleScript scripts [8]. Hovdy weakens

security of the system, installs remote access (ARD, SSH, VNC), installs the open source LogKext keylogger and sends gathered information back to the Hovdy script creators.

LogKext (2010)
LogKext is an open source[1] keylogger for OS X that consists of a kernel extension and user space components daemon and client, which ensure communication with the kernel extension and user communication respectively. The kernel extension hooks onto a callback functionality in the `IOHIKeyboard` kernel component, which is responsible for driving the keyboard hardware.

3.1.2. Backdoor

In the context of this research, a backdoor is malware that opens unauthenticated (mainly remote) access to the victims computer, allowing the attacker to perform a variety of generic operations on the victims system[2].

Olyx (2011)
Olyx.A was found in a compressed file containing content that appeared to be from Wikipedia [30]. Among the files was an MS Word document containing a malicious binary that targeted Mac systems. Upon opening of the MS Word document the malicious binary is executed using the built in MS Word macro functionality, which installs and runs in the background without root privileges. It disguises itself as a Google application support file by creating a folder named ‘google’ in the `/Library/Application Support` directory, where the backdoor installs as under the name`startp`. It also keeps a copy in the temporary folder as `google.tmp`. It creates a Launch Agent `www.google.com.tstart.plist`, to ensure that the backdoor launches upon user login. The backdoor initiates a remote connection request to a static IP address, where it continues to make attempts until established. After a connection is established, Olyx allows remote access to the compromised system, allowing the attacker to perform file system manipulations, file exfiltrations and provides the attacker a remote shell to the compromised system.

XSLCmd (2014)
According to security firm FireEye, it is unclear how the XSLCmd binary was distributed [3]. A submission to VirusTotal caught attention of the binary, which contains large portions of code for a backdoor originally found in a Windows-based version of XSLCmd. The binary has an installation and backdoor routine. The installation routine is called upon first launch of XSLCmd which checks whether the binary is launched with root privileges. It stores a Launch Agent file into `/Library/LaunchAgents/` that launches the backdoor routine of XSLCmd upon user login and checks whether its parent process is launchd, indicating a launch by the operating system. The installation routine differs slightly depending on whether or not the process is running with root privileges. If run as root, the installation routine will also copy `/bin/ksh` to `/bin/ssh`. FireEye, who performed an investigation into XSLCmd, suspects that *“this is likely done to make it less obvious that a reverse shell is running on the system, since it may raise less suspicion to see an ssh process opening a network socket rather than a bash process, although the real ssh executable is actually located in /usr/bin/ssh, not /bin/ssh”* [3]. In case of root privileges, XSLCmd will also perform key logging. XSLCmd uses the CGEventTapCreate API call which requires root privileges, or ‘Assistive Devices’ to be enabled for the calling process. Enabling ‘Assistive Devices’ was easily achievable prior OS X 10.9. XSLCmd therefor crashes on versions later than OS X 10.8, where these API calls to enable ‘Assistive Devices’ were removed. XSLCmd uses a socket to communicate with its C&C (Command & Control[3]) server, with which it functions as a backdoor, the main purpose of XSLCmd.

3.1.3. Trojan horse

A Trojan horse, or Trojan, in computing is malware which misrepresents itself as useful or interesting in order to persuade a victim to install it. The term is derived from the Ancient Greek story of the wooden horse that was used to help Greek troops invade the city of Troy by stealth [9]. Although the payload of the trojan horse can be anything, oftentimes a backdoor is used to infect the system.

[1]`https://github.com/SlEePlEs5/logKext`
[2]`https://www.f-secure.com/v-descs/backdoor.shtml`
[3]`https://www.trendmicro.com/vinfo/us/security/definition/command-and-control-(c-c)-server`

Amphimix (2004)
According to ESET [8], Amphimix was the first (proof of concept) Mac Trojan seen early in 2004 that masqueraded as an MP3 media file, using an .mp3 icon. In fact, it is an application that, when clicked on, displays a message, launches iTunes and to play a 'wild laughter' audio clip.

RSPlug (2007)
This family of DNS changing malware is also closely related to the Zlob family, associated with similar malicious functionality on Windows platforms [8]. This type of malware was found in great numbers in the wild. It is predominantly found as a DMG file containing an installation package named install.pkg, appearing as a Codecs, an approach commonly found to be used by malware on other platforms.

RSPlug.A implemented DNSChanger functionality, which modifies the DNS namesevers to point to rogue DNS servers that are used to inject advertisements into webpages, generating fraudulent advertising profits. RSPlug.A installs itself in a system directory used for browser plugins [8].

Flashback (2011)
Flashback malware infected over 500.000 Apple computers, creating a large botnet [5]. Initially, OSX.Flashback.A masqueraded Adobe Flash Player to infect systems. Later versions used a 'drive-by-download', a technique used by malware that exploits a vulnerability in web browser components in order to install itself onto a system. Flashback exploited a Java vulnerability upon a visit to a malicious website by the victim. A self-signed Java Applet that impersonated Apple Inc. was served, asking the user for permission to run. The Java applet installed a hidden (indicated by a precursory dot) executable. Upon first execution, Flashback encrypts large portions of its main binary using the UUID of the system and RC4 encryption algorithm. Flashback checks if Xcode or anti-virus/firewall software is installed and uninstalls itself if predefined paths of this software is detected. Depending on permissions, Flashback installs a dynamic library that is used to intercept HTTP/HTTPS traffic and inject advertisements into the HTTP/HTTPS traffic that is then served to the user. New C&C servers are announced via Twitter hashtags, which Flashback checks to say updated.

Janicab (2013)
An analysis performed by ESET [8] describes that unlike most other malware on OS X, Janicab operates using scripts, allowing itself to be compliant with multiple systems. Janicab exploits a vulnerability in MS Word, allowing the execution of scripts. On OS X it used Python scripts to operate. Spyware functionality is implemented using these scripts, including audio recording via command line tool "Sound eXchange", capturing screenshots using command line tool "mt" (MouseTools), achieving persistence using the Unix command line utility "cron", a time-based Unix job scheduler.

Wirelurker (2014)
Wirelurker is a Trojan Horse that aims to infects both OS X and iOS systems, according to security firm Palo Alto Networks [55]. The malware was initially spread via Maiyadi, a third party app store, using a repackaged/torjanised cracked Mac applications. The cracked application functioned as a trojan horse to elevate privileges for Wirelurker. The main binary of the cracked application was replaced by a Wirelurker binary which called the cracked binary upon execution. In addition to the binary replacement, it includes a zip file and bash script that are hidden using an Apple specified hidden flag (`UF_HIDDEN`). Upon execution of the hijacked application, Wirelurker checks a system directory (`/usr/local/machook/machook`) whether a version of itself has already been installed. It then executes scripts with root privileges, acquired by the by Wirelurker hijacked application. The scripts install a Launch Daemon in the Launch Daemon specific directory (`/Library/LaunchDaemons/`) and decompress an executable to another system directory `/usr/bin/` that serves as an updater for Wirelurker. Wirelurker spawns two processes in the background, one to check for updates and one to check for iOS devices that are connected to the USB port. Wirelurker downloads an infected iOS application, stores this in system directories, and uses libimobiledevice library to interact and infect the connected iOS device by installing the infected iOS application onto the connected iOS device.

Wirelurker's main purpose it to infect iOS devices and exfiltrate user information contained on the iOS device. OS X is thereby only used as a host for this infection and is not targeted.

iWorm (2014)
In September 2014, a new 'multi-purpose backdoor' malware sample named iWorm was first sighted. Unlike the name suggests, according to Patrick Wardle [53], iWorm is a classic trojan that primarily

spread via pirated OS X applications that were shared via the torrent tracker “The Pirate Bay”. It used famous Adobe applications to grant itself access to the victims system. iWorms dropper[4] was designed in such a way that it modified the application install binary to first launch the iWorm malware and only afterwards the legitimate application to prevent user suspicion. This redirection of application launch to its own binary allowed iWorm to use the elevated privileges granted by the user for the installation of the torrented application. Normally, a modification in the application’s binary would be detected by OS X’s Gatekeeper (see Section 3.1.7). However, due to Gatekeeper’s design, applications that download executables to the system are responsible for enabling Gatekeeper signature checking. Torrenting applications typically do not enable Gatekeeper for the files they download [53]. The dropper installs iWorms binary in `/Library/Application Support/JavaW`, and uses a Launch Daemon to remain persistent after system reboots. iWorm provides basic backdoor functionality to the victim’s system, but contains no worm-like (i.e. self-spreading) capabilities [53]. The usage of Reddit.com for communication with its Command & Control servers, has not been seen before on OS X (Flashback used Twitter for communication with C&C servers).[53]

XcodeGhost (2015)
XcodeGhost (and variant XcodeGhost S) are modified versions of Apple’s Xcode development environment that are considered malware, targeting infection of iOS applications distributed through the iOS App Store [56]. The software first gained widespread attention in September 2015, when a number of iOS applications originating from China harboured the malicious code. XcodeGhost infected the Xcode compiler, forcibly linking compiled applications against libraries that contained malicious code objects. Specifically, a rogue version of CoreServices, a framework that is widely used by iOS applications includes malicious code objects that aim to exfiltrate iCloud passwords from iOS users. Developers that used this malicious version of Xcode to compile their iOS apps, would unknowingly produce malicious iOS applications containing this malicious code objects. XcodeGhost has no further interest in OS X itself, but merely functioned as injection vector for malicious code in iOS applications [56].

OceanLotus (2015)
In May 2015, researchers at Qihoo 360 published a report [25] on OceanLotus which included details about malware targeting Chinese infrastructure. OceanLotus for OS X is packaged as an application bundle pretending to be an Adobe Flash update. The application bundle when executed spawned a process named “EmptyApplication”. For obfuscation, the EmptyApplication binary uses XOR encryption to obfuscate strings contained within the binary. The EmptyApplication process in later stages spawns several other worker processes. One of those processes installs a Launch Agent `/Library/LaunchAgents/com.google.plugins.plist`. OceanLotus then continues to send user information about the victim machine to its C&C servers [25].

3.1.4. Worm

Malware that has de intent to replicate itself in order to spread other victims, is called a Worm [53]. Many worms that have been created are designed only to spread and do not attempt to change the systems they pass through [9]. However, this appears not to hold for OS X.

Leap (2006)
Leap.A used a graphic icon to hide a Unix executable as a JPG image [2]. The image claimed to be “the latest” OS X 10.5 screenshots and was spread through the iChat messenger client, using a file called `latestpics.tgz`.

The malware required user interaction in order to spread via iChat messenger. Upon infection it, based on the privileges of the current user, Leap.A extracted itself to either system locations or an unprivileged user location. After extraction, it used Spotlight to infect all applications stored on the hard disk drive, using an “apphook.bundle”. The apphook bundle does an `execv()` call on the resource fork of the executable (which is the original application) causing the application to launch normally. However, due to a memory allocation bug in the apphook bundle, infected applications would not launch anymore. It was believed that this behaviour was unintended [2].

[4]A dropper is a program (malware component) that has been designed to “install” some sort of malware (virus, backdoor, etc.) to a target system [53].

Leap.A did not have any malicious behaviour other than propagating itself via iChat messenger and local Bonjour (a protocol that Apple products use to establish a connection between each other) [2].

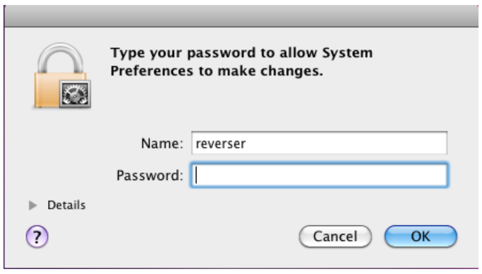
Inqtana (2006)
Inqtana.A is a Java based proof of concept bluetooth worm that affects OS X 10.4 (Tiger) systems that have not been patched against a bluetooth driver vulnerability, according to security firm Intego Security [41]. Inqtana.A arrives to victim system as an OBEX Push request, requiring users to accept the data transfer. Unlike Leap.A, Inqtana.A uses a LaunchAgent to gain persistency on the system. On a reboot, Inqtana.A will activate and try to propagate to devices that accept OBEX Push transfers.

Unlike Inqtana.A, a later variant (Inqtana.D) does not require user interaction and installs itself with root privileges using a self created account on the system. It then installs a backdoor that is accessible through the Internet [41].

3.1.5. Rootkit

A rootkit is malware designed to enable access to a computer or areas of its software that would not otherwise be allowed (for example, to an unauthorised user) while at the same time making extraordinary effort masking its existence on a system. Rootkits are particularly hard to detect and remove because they intent to integrate very tightly with the system, granting itself a large scala of functionality [9].

Crisis (2012)
Crisis, also known as DaVinci or Morcut, is a rootkit developed by the Italian company Hacking Team [49]. Crisis operates on both Windows and OS X versions 10.6 and 10.7 (which it specifically checks before it starts to operate on the system). Crisis is served by a self-signed Java applet dropper that detects the operating system type and installs the proper malware version accordingly. On OS X, the dropper installs two kernel extension binaries and two core component binaries. The dropper uses `INT80` system calls structure to directly call system calls instead of API or library functions. It installs a core component of the rootkit in a system reserved directory `/Library/Preferences/`. The rootkit gains persistency, after trying to trick the user into elevating privileges using an impersonated System Preferences pop-up, shown in figure 3.1.


Type your password to allow System Preferences to make changes.

Name: reverser
Password:

Figure 3.1: Impersonated request for elevated privileges executed by Crisis, by [49]

A LaunchAgent is installed and changes shared memory settings in the kernel for communication with the user space components when root privileges are obtained. Using the root privileges, a binary named `mdworker.flg` is being granted root privileges using the setuid/setgid system call. Crisis aims to infect certain communication applications with a spying module. The infection is achieved using a legitimate method provided by Apple, which allows to intercept events in an application using AppleScript. The infection allows Crisis to exfiltrate (communication) information about the victim, which is Crisis' main purpose.

The only functionality of Crisis' kernel extensions is to hide itself. It hooks three system calls related to the representation of the contents on the file system (`SYS_getdirentries`, `SYS_getdirentriesattr`, `SYS_getdirentries64`) to hide itself [34]. The same technique is used to hide itself from processes. More explanation about hooking system calls is provided in Section 5.3.2.

In February 2016, a new version of the famous Hacking Team rootkit was sighted. This new version used more advanced binary obfuscation techniques to prevent static analysis of the binary. In addition, it used Apple's binary protection feature which provides encrypted processes [50]. According to Perdo Vilaça [50], largely the same codebase of Crisis.I was used in Crisis.II.

The Crisis rootkit could be considered the most complex malware for OS X found to date, as it contains many features and makes great effort to hide itself on an infected system [49].

3.1.6. Ransomware

At the beginning of this Master thesis, ransomware was a type of malware only affecting Windows machines. This changed when ransomware KeRanger was sighted in March 2016 in an infected version of the BitTorrent client Transmission after the Transmission servers were hacked [6].

KeRanger (2016)
On March 4 2016, the official servers of the infamous BitTorrent client Transmission started to offer an infected version of the BitTorrent client. Security firm Palo Alto Networks [6] detected this malware several hours later which they marked as "the first functional ransomware for Mac". The Transmission version 2.90 was signed with a stolen Apple Developer certificate, used to bypass all security protections in the OS. Upon launch of the infected client, the process `/Library/kernel_info` is executed. This process remains in the background and communicates with the KeRanger C&C servers to request required information to initialise the encryption process, such as encryption keys. After 3 days, the KeRanger process will start encrypting document and media files on the file system of the infected machine. To encrypt each file, KeRanger starts by generating subkeys from the RSA key retrieved from the C&C server. It then uses file specific information to create random AES keys for each file it encrypts. In several locations, KeRanger stores instructions for the user to pay the ransom in Bitcoins and decrypt their files [6].

3.1.7. Malicious techniques and improvements

The discussed malware samples use many similar techniques. These techniques are summarised below. A 10-week study performed by security firm Bit9 & Carbon Black [4] confirms several of these findings.

- Many malware requires root privileges to operate as intended.
- Malware operates in system preserved file system directory.
- Almost all malware uses LaunchDaemons, LaunchAgents or Login Items to obtain persistency in OS X.
- Open source LogKext project is often repackaged and reused in malware, which relies on a kernel component that has to be loaded into the kernel.
- Backdoor functionality in malware requires frequently listening on a port.
- Malware tends to work with hidden files and folders to prevent detection.
- Some malware create new accounts on an OS X system.
- Rootkits use a kernel extension to implement system call hooking.

As Patrick Wardle, CTO of security firm Synack, points out — in his presentation at the Infiltrate and Blackhat 2015 conferences [54] — about malware on OS X; compared to malware on Microsoft Windows, the level of sophistication is significantly behind on OS X. In his talk, he suggests other, more advanced, techniques that may be used by malware in the future. The sections below summarise these proposed techniques.

Binary infection
Currently, persistency is achieved using LaunchDaemons and LaunchAgents. This technique is trivial to detect. Despite Apple's recommendation, the binary of many legitimate applications are still unsigned. This results in the execution of unchecked code in the binaries. Attackers can use these unsigned binaries to hide their malware inside and gain persistency. Each time the user launches an infected application, the malware will also be launched. Detection of injected malware is much harder than the detection of malicious daemons or agents.

Dynamic library hijacking
Similar to Microsoft Windows, applications on OS X can make use of dynamic libraries. Libraries are often packaged within an application and are currently not signature checked by the operating system upon execution. This can be misused by malware, replacing the legitimate library using an infected library containing malicious code. Upon execution of the application, the malicious library will be executed by the operating system, providing the ability for malware to operate.

Spotlight plugins
Spotlight is a system-wide search utility, built-in to OS X. Apple allows developers to develop plugins for Spotlight, extending the functionality of the famous search utility. These plug-ins are launched and executed each time the user uses Spotlight to search the system. Malware may use the plugins to gain persistency and collect data (as it may use the Spotlight scope on the file system).

Encrypted binaries
OS X natively supports encrypted Mach-O binaries. The Finder binary of OS X for example, uses this method to obfuscate its functionality and increase debugging difficulty. Encrypted binaries make it much harder for anti-virus engines to detect the malware. Especially if the malware only decrypts itself in the intended targeted environment, leaving itself encrypted on all other systems. This technique is heavily used by state-actor malware like Stuxnet, Duqu, Flame and others [51].

Gatekeeper signature check bypass
Gatekeeper is an anti-malware feature of the OS X operating system which verifies the signature of applications executed by the user. The signatures are checked against Apple's root certificate before they are allowed to execute on the system. However, Gatekeeper does not verify all applications before execution. Whether or not Gatekeeper should verify the signature is left to the application that download the application to the disk. Torrenting applications for example, do not set the file attributes necessary for Gatekeeper to perform signature checking resulting trojan horses to easily achieve execution rights the system.

Kernel extension signature checking
By default, OS X verifies the signature of kernel extensions that are loaded. Verification of a kernel extension (kext) is performed by a user space daemon named kextd. However, kextd is replaceable with a version that removes the signature verification, allowing unsigned kernel extensions to be loaded.

3.2. Evolution of malware: learn from Windows

Microsofts operating system Windows has been, and still is, a much greater target for malware than OS X[5]. This is largely due to the popularity of Windows. Figure 3.2 shows that over the last three decades, for every Mac sold, at least 5 to 60 times as many Windows PC's were sold.

Figure 3.3 shows the sales numbers of Windows PC's from 1984 to 2016. A peak is visible around 2011. Figure 3.4 shows the amount of new malware samples submitted to anti-virus testing firm AV-test. A very similar peak is observed. Arguably, there appears to be a strong relationship between the popularity of an operating system family and the number of malware samples that is targeting this family.

Initially, malware targeting Windows did not do as much harm, compared to todays malware samples [29]. From 1990 to 2000, the Windows malware starting to appear, featured a worm-like nature, occupying 'spreading' to other systems as a primary goal. Once their goal was accomplished, — with exception of a few malware samples — oftentimes the malware did not conduct more harm than annoyingly notifying the user their system has been infected [29]. Slowly but surely, the incentives of the malware creators changed. Where previously the incentives for creating malware were rather challenging skills and hobby projects, nowadays malware creators are financially-incentivised [29]. Malware creators are more sophisticated and organised than ever before [44]. Several underground services that increase accessibility and approachability cyber-crime emerge. Crimeware-as-a-Service (CaaS) and Exploit-as-a-Service (EaaS) (an analogous to Software-as-a-Service (SaaS) — where software services are available on request) are now a commodity in the service offerings on the underground

[5]`https://usa.kaspersky.com/internet-security-center/threats/mac-vs-pc-security`

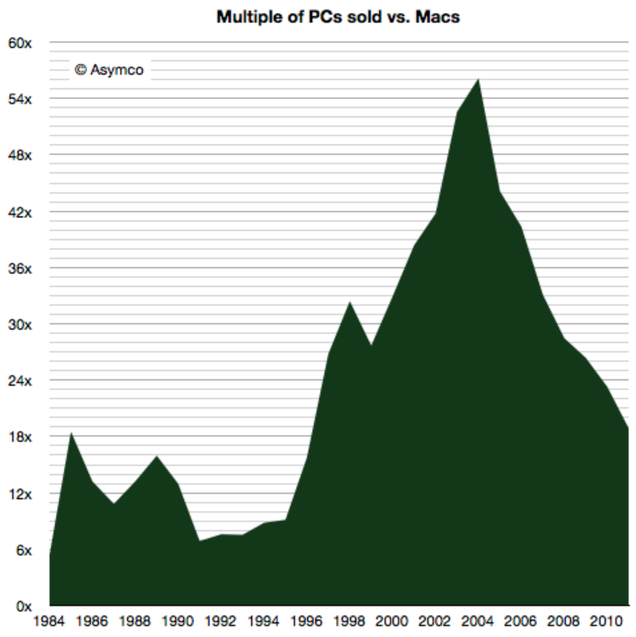


Figure 3.2: Amount of Windows PC's sold for every Mac, from 1984 to 2015.

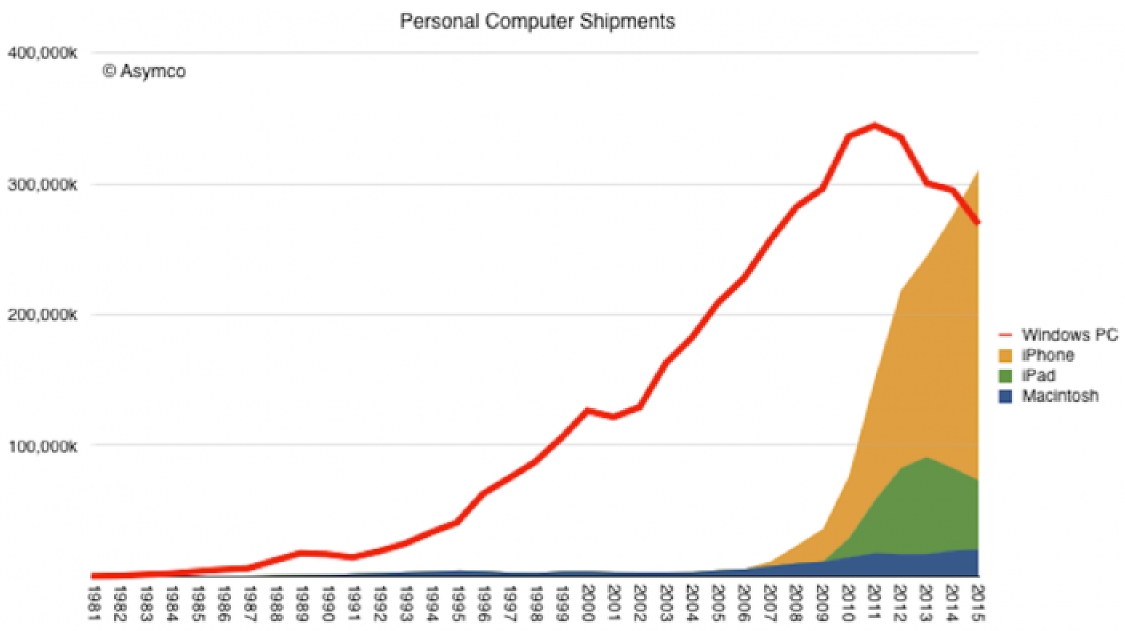


Figure 3.3: Personal computers sales, from 1984 to 2016, by Asmyco

market [44]. These evolutions, together with cryptocurrencies, facilitate the current biggest cyberthreat (according to the statement from the FBI in June 2015[6]) for Windows machines, both in a corporate and personal environment; ransomware. According to the FBI, estimated damages to corporates caused by ransomware CryptoWall (a single ransomware family) are over $18 million.

Currently on OS X, very recently the first functional ransomware sample has been found [6]. As discussed in Section 3.1.7, it can be argued that OS X malware lacks sophistication and is currently at the same level of sophistication as Windows malware was around 1995-2005 [54]. However, this is set to change with Macs gaining popularity. According to a IDC Worldwide Quarterly PC Tracker report in

[6]`https://www.ic3.gov/media/2015/150623.aspx`

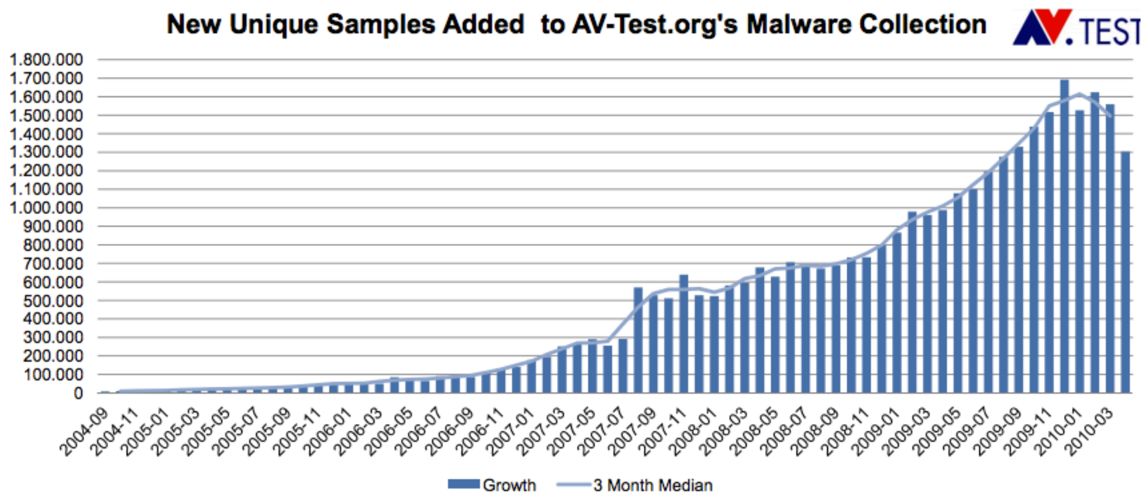


Figure 3.4: Number of unique malware samples signatures submitted to AV-test.

January 2015, Apple's Mac sales achieved a growth of 18% where the rest of the industry underwent a decrease in Windows PC's sales of 3%[7]. According to Asmyco[8] in figure 3.3, Apple has been able to consistently outperform the market the last years. Currently the marketshare of the Mac is approaching 15%[9]

As Macs gain popularity under consumers, Macs will gain popularity under malware creators. In October 2015, Bit9 + Carbon Black demonstrated the unprecedented growth in OS X malware [4]. In 2015 alone, the number of OS X malware samples has been five times greater than in 2010 to 2014 combined, the research found. The shift to more sophisticated malware for OS X could increase more rapidly than seen on Windows, since the cyber-crime infrastructure, ideas and experience are already in place to facilitate a malware roll-out to Macs.

3.3. Current OS X malware detection mechanisms

In general, detection methods for malware on OS X implemented in consumer products, can be divided into two types; signature- and path-based detection. The sections below elaborate on the detection types.

3.3.1. Signature-based detection

Traditional anti-virus (AV) engines implement signature-based detection techniques [40]. AV engines contain a database of all known malware samples and upon execution of any binary by the operating system or a write of a binary to the disk, the anti-virus checks its signature database against the signature of the particular binary. If a match is found, the anti-virus engine has detected the malware and will deny execution [40].

Signature-based detection technique can be very effective, but argues security firm Imperva [18], cannot defend against malware unless some of its samples have already been obtained by the anti-virus company, a proper signature has been generated and the anti-virus database is updated with the latest signatures. This approach is not really effective against zero-day or next-generation malware, i.e. malware that has not yet been encountered and analysed [18]. It also is ineffective against polymorphic and metamorphic malware. Polymorphic malware mutates its code while keeping the functionality intact [16]. This means that the signature changes, but the functionality (malware) remains intact. Metamorphic malware uses an interpreter to produce machine code [16]. Most common examples of metamorphic malware are samples of which certain parts (payloads) are encrypted and are begin

[7]`https://www.idc.com/getdoc.jsp?containerId=prUS25372415`
[8]`http://www.asymco.com/2015/04/14/personal-computer/`
[9]`https://www.idc.com/getdoc.jsp?containerId=prUS41176916`

decrypted at runtime. Signature-based detection renders ineffective against these techniques adopted by modern malware [18].

Heuristic analysis

A more advanced detection technique implemented in major anti-virus engines is a technique known as heuristic analysis [16]. In essence, heuristic analysis tries to detect malware by investigating and identifying specific (known as malicious) functionality or characteristics. Functionality or characteristics may be a program which:

- tries to inject a copy of itself into other programs.
- remains resident in memory after it has finished executing.
- binds to a TCP/IP port and listens for instructions over the network.
- is similar to other programs known to be malicious.

Heuristic analysis uses emulation and virtual machines to simulate the program in isolation and investigate the program for malicious patterns. It is often concomitantly implemented with a Multi-Criteria Analysis, where detected patterns are weighed to more accurately assess the program regarding malicious patterns [16].

Heuristic analysis appears to be very similar to behavioural analysis. The discrepancy however lies in the components they monitor and analyse. As explained above, heuristic analysis mainly focusses on analysing the binary for malicious parts prior to execution. Behavioural detection monitors and analyses the system for malicious events and aims to block those events [27].

3.3.2. Path-based detection

The previous section concluded that due to the relative absence of interest of malware creators in OS X, the amount of malware is relatively small and unsophisticated. In combination with the structure of the OS X operating system, this results in malware residing in specific locations [54]. It is relatively easy to scan these location and detect malware. Several of these detection programs exist for OS X.

BlockBlock

As stated in Section 3.1.7, an extremely common method for malware to acquire persistency, is to make use of LaunchDaemons. LaunchDaemons are contained in certain system location in order for the system to launch their corresponding programs upon boot of the system. BlockBlock (by Patrick Wardle)[10] is a program that consistently monitors file I/O events in these LaunchDaemon locations to detect ‘persistence attempts’. Upon such an event, it displays the event to the user together with some basic process information. BlockBlock uses the OS X FSEvents API to be alerted of specific file and directory changes. On top of that, DTrace is used to keep track of process creation to maintain a list of active processes and their information. This allows BlockBlock to be a userspace application that alerts the user of suspicious behaviour of processes running on the system.

KnockKnock

Unlike BlockBlock, which is a continuously monitoring background process, KnockKnock[10] (from the same creator as BlockBlock) is a system analysis tool that scans the system upon request of the user. It uses the same path based approach as BlockBlock, scanning for installed kernel extensions, login items, browser plugins, Launch Items, Spotlight importers and dynamic library insertions. Results are checksummed and checked against the VirusTotal malware definition database[11].

[10] `https://objective-see.com/`

[11] `https://www.virustotal.com`

osquery

Osquery[12] (by Facebook, Inc.) exposes an operating system as a high-performance relational database. This design allows for SQL-based queries to efficiently and easily explore operating systems. With osquery, SQL tables represent the current state of operating system attributes, such as:

- running processes
- loaded kernel modules
- open network connections

SQL tables are implemented via an easily extendable API. Examples of such queries are:

- `SELECT name, path, pid FROM processes WHERE on_disk = 0;`
  This shows all processes running on the system, but not maintaining a binary on the disk, which is perceived as typical malware behaviour.
- `SELECT DISTINCT process.name, listening.port, listening.address, process.pid FROM processes AS process JOIN listening_ports AS listening ON process.pid = listening.pid;`
  Using the SQL-join operation, multiple sources of information are merged together to provide an extended view of the context. This query provides a detailed description of processes that are listening on TCP/IP ports.

Osquery aims to be a valuable, flexible and interactive intrusion detection tool. However, detection intelligence is not implemented and very specific knowledge is required from the user in order to detect malware.

chrootkit

One of the main characteristics of rootkits, as explained in Section 3.1.5, is preventing detection by making great efforts to hide itself. chrootkit[13] (check rootkit) is a utility that tries to identify signs of rootkit hiding. Specifically, it checks for deletions in system, user login/logout, system status log files. It also includes checks for specific versions of the LKM rootkit. These checks are based on searching for discrepancies between the parent directory link count and the number of subdirectories, which would indicate hidden files and folders. Same checks are performed for running processes using `ps`-output.

3.4. Conclusions

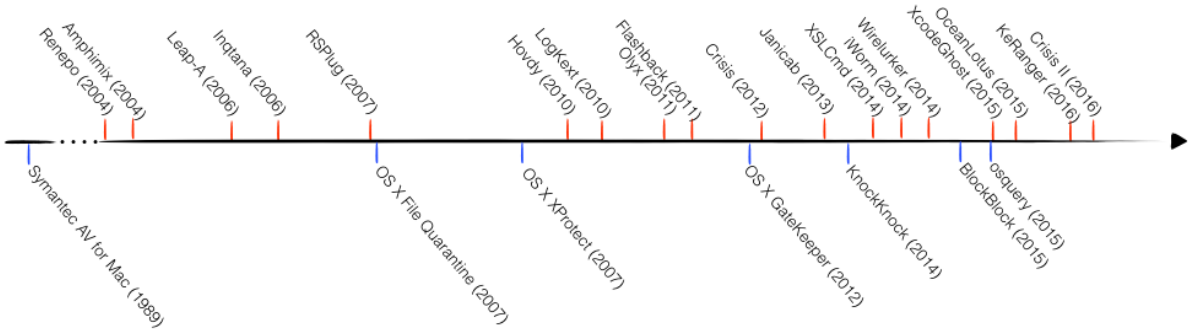


Figure 3.5: Timeline of impactful malware (red) and protection mechanisms from both Apple and third party producers. OS X File Qurantine was later improved and rebranded to XProtect, OS X built-in signature based malware detection.

This chapter has shown malware on OS X is on the rise and the level of sophistication is increasing. Lately, more frequently new OS X malware samples have been spotted. Threats only existing on Microsoft Windows, such as ransomware, are now also appearing on OS X. Section 3.2 showed there is a relationship between the popularity of an operating system and the amount of malware targeting

[12]`https://osquery.io/`
[13]`http://www.chkrootkit.org/`

the operating system. As OS X may continue to increase in market share, the numbers of malware samples will increase.

The timeline in figure 3.5 summarises the occurrences of new malware samples over the last 6 years, as well as the introduction of current protection and detection mechanisms. Relatively simple protection mechanisms currently suffice in preventing infection of OS X malware. Most of the detection mechanisms use path-based detection of malware.

Arguably, the majority of malware on OS X is still trivial to detect, hence the many path-based solutions. Malware on OS X uses specific file system directories to gain persistency on the system. Solutions such as BlockBlock monitor these locations, and in case of changes, inform the user. However, these solutions only detect certain characteristics used by some malware and do not provide a generic method of protection against malware, based on behaviour detection.

4 Literature research

In scientific research, many more methods and techniques to detect anomalies and malicious behaviour on operating systems have been proposed. A vast majority of the behavioural malware detection research is focused on anomaly detection by learning process behaviour. This chapter provides an overview of the conducted research related to malware detection. The first section briefly explains the machine learning methodology and popular algorithm types. The following sections explain different types of malware detection methods proposed by research. As stated earlier, this thesis is specifically interested in detecting malware using system calls. The first part describes several proposed broad behavioural detection methods, while the second part of the study focusses specifically on detection methods based on system calls.

4.1. Machine learning

Machine learning is a subfield of computer science that evolved from the study of pattern recognition and computational learning theory in artificial intelligence [37]. Algorithms used, are primarily based on statistics fundaments in order to make data-driven decisions and predictions. The machine learning algorithms can be categorised using the following categories [37]:

- **Supervised learning:** The computer is presented with example inputs and their desired outputs, given by a ‘teacher’, and the goal is to learn a general rule that maps inputs to outputs.
- **Unsupervised learning:** No labels are given to the learning algorithm, leaving the algorithm on its own to find structure in the input of the algorithm. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end.

Another categorisation of machine learning tasks arises when one considers the desired output of a machine-learned system:

- **Classification:** inputs are divided into two or more classes and the learner must produce a model that assigns unseen inputs to one or more of these classes. This is typically tackled in a supervised learning way. Spam filtering is an example of classification, where the inputs are email (or other) messages and the classes are ‘spam’ and ‘not spam’.
- **Regression:** also a supervised problem, the outputs are continuous rather than discrete.
- **Clustering:** a set of inputs is to be divided into groups. Unlike in classification, the groups are not known beforehand, making this typically an unsupervised task.

In the conducted research, anomaly detection for malicious behaviour of programs is concerned with supervised learning. The typical approach taken by the research training a classifier is based on data labeled as malicious or benign. The classifier is trained using a training dataset. In the training dataset the input and output results are known to the classifier algorithm. After the training phase, the classifier is then tested using a test dataset in which corresponding output is absent. The classifier will then behave to these input cases as it would in ‘real’ scenarios. This is different from anomaly detection in a networking environment, in which often unsupervised learning is used.

Oftentimes, the dataset used for training and testing undergoes some feature selection. Feature selection extracts relevant features (variables, predictors) from the dataset in order to improve the construction of the — in this case — classifier.

As shown in this Chapter, in the field of malware detection, the effectiveness of a detection technique is typically evaluated using two metrics, Detection Rate (DR) and False Positive Rate (FPR):

- Detection Rate (DR): the detection rate is defined as the number of intrusion instances detected by the system (True Positive) divided by the total number of intrusion instances present in the test set [43].
- False Positive Rate (FPR): An event signalling a detection system to produce an alarm when no attack has taken place [43].

4.2. Detecting malware

In general, there are two common approaches to analyse malware samples: statically and dynamically. Static analysis method is the most popular approach to identify whether a program is benign or malicious. This approach inspects executables to obtain a sequence of characteristics that identify the behaviour of the program. These characteristics are called signatures [26][33]. In dynamic analysis method, the malware binary is executed in a safe or virtual environment and the run-time behaviour of the malware binary is monitored. From this dynamic behaviour, signatures are extracted that are used to assess the nature of the program.

4.2.1. Static analysis and detection

Static analysis on binaries has been performed by many researchers [26][39][22][10][19]. Several different static analysis methods are proposed:

- **String occurrence, library and function calls**
  In 2006 Kolter et al. [23] pioneered in applying machine learning to the field of malware detection. Kolter et al. extracted strings, used in dynamic link libraries and API calls (explained in Section 2.1) from binaries, and used these features to test several machine learning algorithms. Boosted decision trees (J48) working on 500 n-grams (an n-gram is a contiguous sequence of n items from a given sequence of text or speech, in this case ranging from 1 to 4 items) were found to produce better results than both the Naive Bayes classifier and Support Vector Machines. True Positive rate (TPR) and False Positive Rate (FPR) are used to measure performance of the classifier.
- **Binary block comparison**
  A binary is disassembled and divided into blocks based on function structures (call and return instructions). These blocks are then compared against other known malware samples using an algorithm known as Hungarian algorithm[1], in order to find similarities. Kang et al. [22] claim to reduce the size of binaries that has to be scanned to 24% of the total binary size.
- **Constructing control-flow graphs**
  Faruki et al. [10] and Lee et al. [26] extracted API-call information in Control Flow Graph (CFG) structure from disassembled binaries. The CFG is then transformed to an API call-gram. In the proposed method, API-calls instead of system calls are used. The call-grams are then converted to feature vectors, which are then used to train four different classifier algorithms; Random Forest, J-48, SMO-SVM, Voted-Percetron. The train and test data is constructed using the K-fold cross validation, in Weka[2]. The measurements that are used to measure performance are True Positives, True Negatives and Accuracy. They claim to have a detection rate of 98,1%. Kazuki et al. [19] use Dice’s coefficient in combination with Hierarchical Cluster Analysis, however they acknowledge to have bad results.

[1]Hungarian algorithm explanation: `http://www.hungarianalgorithm.com/`

[2]Weka is a collection of machine learning algorithms for data mining tasks. `http://www.cs.waikato.ac.nz/ml/weka/`

- **API function usage of binaries**
  Sami et al. [39] extract API call information from the import address table in the header in the binary. They use the Fisher score to rate API calls that appear to be often occurring in malicious binaries and benign binaries. The authors claim to have achieved a detection rate of 99,7% while keeping accuracy as high as 98,3%. In addition, TPR and FPR are used as performance indication. Ye et al.[57] use Objective-Oriented Association (OOA) mining based classification on API call information extracted from Portable Executables (PE). J-48 appears to perform the best. Similar to other research, their performance is measured using the TPR, FPR, accuracy and detection rate metrics.

False Positive Rate and True Positive Rate appear to be a commonly used metric to evaluate the effectiveness of the proposed techniques.

Statically analysing malware can achieve a high detection speed but can be more easily defended against by malware authors using obfuscation techniques. Moser et al. [32] present an obfuscation scheme based on opaque constants “hidden” in processor registers that manipulate the control flow of a program. Opaque constants are constant values in source code generated by an obfuscated function that always generates the same constant. Their proposed technique also makes it difficult to locate control flow changes; call, jump and return instructions, indicating function changes. Mozer et al. show the proposed obfuscation approach is able to evade advanced semantics-based malware detectors and in addition, the opaque constant primitive can be applied in a way such that is provably hard to analyse for any static code analyser. Due to these limits of static analysis, a transition to dynamic analysis is research can be observed.

4.2.2. Dynamic analysis of malware

Unlike static analysis, dynamic analysis determines the behaviour of programs while executing in a safe and isolated environment. Several levels of abstraction are presented in research to define and analyse the behaviour of programs. This section provides an overview of proposed research in this field.

- **Function call monitoring**
  When a program is executed a trace is created from the order of operating system functions that is calls. These operating system functions consist of API functions of operating system libraries and system call functions. Sun et al. [46] use dynamic monitoring of Windows API calls to detect worms and other exploits. However, their approach is limited to detection of worms and exploits that use hard-coded addresses of API calls which is not the case if Address Space Layout Randomisation (ASLR) is activated on the operating system. Nair et al. [33] determine the frequency of critical API calls by programs and use this information to construct a signature of a program. Nair et al. introduce their own classification algorithm and claim to have a detection rate of 80% for Windows malware.

- **Function arguments monitoring**
  Tsyganok et al. [48] perform an analysis on API calls including arguments passed along with the function call to classify metamorphic and polymorphic malware. They cluster the behaviour of programs based on fuzzy clustering algorithm and reach a classification error of 21.4%. Salehi et al. [38] present a similar technique that monitors API calls and their arguments of malware samples that are ran in a virtual machine for 2 minutes and use several Weka classifiers to construct a new classifier. They use Relief feature selection and claim to get best performance using the Random Forest and J48 algorithms.

- **Information flow tracking**
  Yin et al. [58] present a system named Panorama which uses system wide information flow tracking to track the flow of critical information, based on taint graphs. A taint graph is a representation of information flow that shows the processes that access tainted data, how the data propagates through the system, and finally, to which file or network connection this data is written to. Special hardware and shadow memory is used to generate taint graphs. Yin et al. present a rule based model that classifies programs as benign or malicious.

4.3. Detecting malware based on system calls

Detecting malware based on system calls is considered a dynamic detection approach, since the malware has to be executed and inspected in order to analyse the its system call invocations. Where the previous section provided a broad overview of dynamic malware detection research, this subsection describes research purely using system calls as a detection method, the detection method this thesis is concerned with.

In 1996, Forrest et al. [11] were the first to introduce a simple intrusion detection method based on monitoring the system calls used by active, privileged processes. Each process is represented by its system call trace — the ordered list of system calls used by that process from the beginning of its execution to the end. This work showed that a program's normal behaviour could be characterised by local patterns in its traces and deviations from these patterns could be used to identify security violations of an executing process. Forrest methodology was based on lookahead pairs of system calls. Others [20] tried to improve on this methodology by using machine learning algorithms, however, these improvements came with a computational cost and were not able to perform real-time detection of malware.

4.3.1. Systrace

In 2002, Niels Provos [36] introduced a tool named Systrace which aims to improve the host security by enforcing system call policies based on interposing system calls. Several attempts in intrusion detection using interposing of system calls prior to Provos had been made. Cerb[3] and MAPbox are examples of these attempts. However, Provos and Garfinkel[14] point out that these attempts — based on ptrace (debugging functionality to trace a process) — are easily by-passable due to the limited traceable scope of ptrace. Systrace overcomes this problem by monitoring the direct system call usage (instead of a mirrored version by ptrace). Systrace consists of a kernel component that intercepts the system calls and a userspace component that enforces system call policies. The system call policies define for a process which system calls with corresponding arguments it is allowed to perform. Provos points out that, although powerful, policy enforcement at the system call level has inherent limitations. Monitoring the sequence of system calls does not provide complete information about an application's internal state. For example, some system services change the privileges of a process on successful authentication but deny extra privileges if authentication fails. A sandboxing tool at system call level can not account for such state changes and adapt policies to these state changes.

Kurchuk et al. [24] in 2004 improve upon Provos' Systrace by proposing two extensions; nested policies and dynamic policy generation. Nested policies enforce the same policies of the parent process on the child process. It also provides capability to keep track of the (un)privileged states and state transitions of programs. Performance tests of this proposed extension show a significant (2 to 10 times) speed decrease of the system. However, both Provos and Kurchuk do not elaborate on the effectiveness of intrusion detection by Systrace and its extensions. The effectiveness of intrusion detection will very likely largely depend on the restrictiveness of the enforced policies. These policies however, are not evaluated.

4.3.2. System calls and Machine Learning

Wagner et al. [52] in 2009 provide research for malware analysis with graph kernels and support vector machines. Described is a modelling framework capable of representing relationships among processes belonging to the same session, as well as the information related to the underlying system calls executed. The models that express relationships are based on process models, in which relations between process invocations are expressed. The `sys_execve` system call (process execution) plays a central role in this relationship creation. The process trees contain information on the execution of processes by other processes, their process names, process ID's and execution related system calls called. In order to accomplish the system call monitoring, Wagner et al. modify the source code of a Linux system to log the system call usage. In particular, they monitor behaviour of the default SSH (Secure Shell) that is claimed to be a high target for malware. Based on malicious and benign

[3] `http://cerber.sourceforge.net/`

process trees, a SVN classifier is trained. F_1-measure and the accuracy-score are used to measure the performance of the classifier.

Alazab et al. [1] propose a method that aims to detect obfuscated malware by investigating the structural and behavioural features of API calls. They developed a system that automatically extracts API calls. The system is based on IDA (a binary disassembler) to disassemble the binary and later store the contents in structured format using a SQLite database. The n-grams are constructed by first counting the frequency of each n-gram within the entire corpus. Once that has been completed, Alazab et al. reduce this list to the top 100 most frequent n-grams. The above procedure is replicated for n values between 1 and 5 inclusive. Then the SVM classification technique is used to construct an N-dimensional hyperplane which separates the dataset into two groups, malicious and benign. Alazab et al. achieve a 85% detection rate and 15% False Positive Rate (FPR) for n-grams larger than 4. For single API calls they achieve a 97% detection rate and 1.91% FPR. However, the reason for improvement in case of single API call inspections is not explained. Alazab's methods relies on existing unpacking tools. If those tools fail to unpack the binary, they are not able to analyse the binary. In addition, this technique could be considered static analysis, since the binary sample is not dynamically analysed.

4.3.3. Critics & alternatives

Dehnert [7] in 2013 argues that an IDS (Intrusion Detection System) running on the host operating system itself is vulnerable against a direct attack and proposes an intrusion detection method based on a hypervisor — a software component that virtualises an entire operating system and thereby entirely separating the IDS from the system on which it tends to detect intrusions. VMware VProbes are used for the proposed hypervisor implementation. VProbes are very similar to DTrace's probes (DTrace is explained in Section 5.2.1), and allow for specific detection of pre-defined patterns on the virtualised operating system on the host system that runs the hypervisor. Dehnert uses the VProbes to detect system call usage and their arguments. The detection methods implemented are manually defined patterns and system call patterns using 'Sequence time-delay embedding' (stide), implemented without any tuning. As expected by Dehnert, the detection with manual defined patterns was effective. The detection based on stide however, due to the absence of any refinement, performed poorly. When all the instrumentations of the proposed IDS are enabled, it operates with a performance overhead of almost 300%.

Nguyen et al. [35] in 2003 propose research for detecting insider threads on a system, based on file operations. Their research consists of constructing a large dataset of system call traces, defining the amount file system operations and process executions on the system originating from user processes. They show that 'normal' execute file operations very limitedly fluctuate around a steady amount, and malware or exploited system binaries would differentiate from this small fluctuation. In addition, they remark that the ability to monitor process executions using system calls allowed to detect exploitation of processes. A process should only spawn another process of itself. In a typical exploit situation, the exploited process would spawn a malicious process, instead of one of itself. However, they point out that malware can easily go undetected by this detection method by performing only very few file operations. Metrics used are the FPR and FNR to measure the performance.

Unfortunately, literature also suggests numerous weaknesses of detecting intrusions using system calls. Tan et al. [47] focus on argument-oblivious systems like Stide. They show that system call monitoring is viable in detecting intrusions of malware, but as a countermeasure show that an attacker can insert additional system calls that are common in the real system, to circumvent the detection. By spreading out the suspicious system calls, an attacker can hope to avoid triggering anything unusual. There is also a risk that a suspicious system call might be entirely replaced — while an `execve` system call might be unusual, an `open` and `write` might not be. On the other hand, if the file being opened and edited is `/etc/passwd`, the open/write may be no less dangerous. Garfinkel [14] suggests that Unix may be too complicated for system call techniques to work. Features like hardlinks and the `dup2` system call make it hard to do any sort of tracking based on arguments; there are too many different ways to name objects for it to be effective.

4.4. Conclusions

The majority of the detection methods proposed in literature are based on machine learning. The generation of classifiers is very similar to the general methods of training a machine learning classifier as in other fields where machine learning is applied. A classifier that is repeatedly reported to perform well is J-48, a Weka classifier. Other Weka classifiers based on decision trees also achieve good detection rates. Very few research provide an explanation, reasoning or assumption on why a certain machine learning algorithm is used or performs best in classifying their data set. There appears to be a consensus with regard to the metrics that ought to be used to measure performance of the classifier. TPR (True Positive Rate) and FPR (False Positive Rate) are the most commonly used metrics. Often accuracy and detection rate is also used as a metric.

Research points out that due to the code obfuscation techniques used by malware creators, static analysis appears to become less feasible and a move towards dynamic analysis is inevitable to determine the actual nature of a program. Very few real-time detection methods exist based on system calls. Systrace is a great example of an IDS that aims to detect and prevent malicious behaviour in real-time. It could be considered proven to be interesting to the security community, since it has been added to the security focussed OpenBSD Linux distribution.[4][5]

An important observation is that the literature concerned with machine learning techniques for detection of malicious behaviour, are heavily dependent on post-execution detection. In other words, the classifiers are trained to detect malicious patterns on complete system call traces of programs, which are only available after the programs have finished executing. This means that the potential damage of malicious programs is already accomplished and can not be prevented by these methods. This thesis strives for a real-time, ‘inline’ detection method that aims to detect malicious system call behaviour as early as possible in order to prevent potential harm. Research for such a detection method has not been found.

[4]http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man1/systrace.1
[5]http://www.openbsd.org/

5
Collecting system call traces

Similar to the approaches taken by the research discussed in Chapter 4, collecting data resembling the behaviour of processes is crucial for this research. This chapter explains the type of data the dataset consists of, the methods used to gather the data and elaborates on the construction of a kernel module that has been developed in order obtain the data.

5.1. System calls

Section 2.3.1 expressed the particular interest in system calls to define, detect and possibly prevent malicious behaviour based on system calls. Section 2.1 briefly discussed the concept and context in which system calls operate. This section will elaborate on system calls on OS X.

OS X features the XNU (X is Not Unix) kernel, a hybrid kernel that consists of a combination of components from the Mach microkernel and large portions of the FreeBSD kernel project and was originally designed by NeXT, before NeXT was acquired by Apple Inc. [28]. The Mach component in XNU is specifically responsible for process and memory management in OS X, where the BSD portion provides functionality for all other tasks that a kernel is expected to provide and take responsibility of [28]. This includes functionality on top of Mach to implement POSIX processes, File System, networking, security policies etcetera. The kernel defines functions (system calls) that can be invoked by other processes on the system in order for the processes to accomplish tasks on the system. Examples of tasks that can be performed would be to execute a process, I/O operations on files, create a socket for network communication, acquire memory. XNU provides a little over 430 system calls[1] — of which several reserved and deprecated — that are combination of BSD and Mach system calls and can only be called by userspace processes. The XNU kernel does not use system calls [21].

Processes cannot access memory outside of their memory space. Therefor, system calls are wrapped in programming libraries for programmers to access within their program's scope (the standard C library is an example of this). The C library function then calls the system call by their number, moving the number into the processor register EAX and calling interrupt 80. The kernel will take over from userspace and handles the system call, afterwards returning the result back to userspace. The system calls in essence define what processes can do on a system, which implies analysing system call invocations reveals the behaviour of processes [21].

5.1.1. Dataset

To analyse the system call invocations (and thus behaviour) of applications, a dataset consisting of every system call called is constructed. Hereafter, this thesis refers to the system call invocation order as a system call trace. Conceptually, the dataset can be interpreted as an extremely large log containing the behaviour of all processes running on the system while the log file was constructed. Every system

[1]XNU open source: `https://opensource.apple.com/source/xnu/xnu-1504.3.12/bsd/kern/syscalls.master`

call invoked during the period in which the log is constructed, has to be logged in the exact order in which the system calls occurred. Together with the order of system call invocation, metadata about the process is also logged to the dataset. Section 5.3.4, discusses the metadata of each system call that is captured and logged.

5.2. System call information gathering mechanisms

Several mechanisms and techniques exist in OS X that allow the retrieval of information concerning the invocation of system calls. Some of the mechanisms are supported and empowered by Apple itself, other techniques are not. This section elaborates on the possibilities and limitations of the available techniques.

5.2.1. DTrace

In November 2003, Sun Microsystems released a dynamic debugger tool for Sun Solaris 10 operating system, called DTrace[2]. DTrace is a dynamic tracing and debugging framework designed to real-time troubleshoot kernel and applications running on a system. DTrace was ported to FreeBSD and NetBSD, and thus is available under OS X by default. The framework contains a comprehensive set of tracing tools varying from tracing userspace applications to tracing system calls in the kernel. DTrace includes the ability to provide a global overview of the running system; memory, CPU, filesystem usage and network rescues by active processes. It also allows for more fine-grained inspection of the system using so called probes. Probes are defined using a DTrace specific programming language called D, which is inspired by the C programming language. Unlike C, D is a data driven language, meaning that it reacts on data patterns defined in a data stream, fed to DTrace. The data stream is extracted from the system, while the actions are defined by the user using probes. DTrace will react with an action defined by the user, upon detecting probes in the data stream that meet a specified condition.

Due to its power and comprehensiveness, DTrace seems an ideal choice for analysing system calls throughout the system. However, DTrace has its limitations with respect to the scope of its analysis. Based on the DTrace research performed, it appears to be impossible to gather every generic system call in a system including relevant process information. DTrace was designed to perform process specific analysis and therefor DTrace' scope is limited to user defined processes and does not work properly on an unknown, undefined set of processes, which is a requirement for analysing all system calls performed by, for example, malware.

5.2.2. TrustedBSD

Mandatory Access Control (MAC) frameworks allow administrators to enforce policies for users and application behaviour. Every modern operating system uses resource control systems like MAC to provide operating system security. FreeBSD implements MAC in its TrustedBSD[3] module which is also adopted by OS X where it is used to implement system security services, including application sandboxing.

TrustedBSD provides a Kernel Programming Interface (KPI) that allows third party developers to reuse features and functionality provided by the TrustedBSD framework[4]. The KPI allows to filter events occurring in the OS, such as I/O access control, network activity, Mach ports and process launch. By filtering these events, TrustedBSD allows for implementation of security functionality. It also allows filtering for several critical system calls. It would be possible to system wide log events for when these critical system calls are executed, but due to the restriction of only critical (as defined by Apple) system function, this would limit the system call logging to only system calls that Apple considers useful for security purposes. This research requires analysis of broader set of system calls, which renders TrustedBSD in an unviable option.

[2]http://dtrace.org
[3]http://www.trustedbsd.org/mac.html
[4]http://sysdev.me/trusted-bsd-in-osx/

5.2.3. KAuth

OS X 10.4 introduced a new kernel subsystem; Kernel Authorization or Kauth for short, for managing authorization within the kernel. It was implemented primarily to simplify the implementation of access control lists (ACLs), a component within Mandatory Access Control that defines access rights for users of the system. While Kauth was originally designed to support ACLs, it is a general kernel authorization mechanism and can be used for a variety of other tasks. One such use is as a simple notification mechanism for anti-virus developers which is the KPI that Apple makes available to third parties. KAuth allows developers to define scopes in which their software operates. Within a scope, events occur for which actions can be defined by the developer. An action is implemented using a listener that listens in the scope for a specific event to occur. Upon an event, the function of the developer is called using a callback mechanism. A callback function is called by an operating system specific component, in this case KAuth, and has permission to interfere. An example would be a file change. In the filesystem scope, an event ‘file change’ occurs, for which third party software that uses KAuth can allow or deny this file change.

KAuth scopes allow for a wide variety of file system events monitoring, for example file modifications and executions. This could be used to monitor system call execution to some extend, but again limits the amount of system calls that can be monitored and logged. Similar to TrustedBSD, KAuth does not qualify to monitor system calls at large scale, as intended for this research.

5.2.4. System call hooking

The functions that implement the system calls, reside in memory owned by the kernel. A table with system calls and their corresponding implementing functions, which is a one-to-one mapping, also resides in kernel memory. This table is known as the ‘sysent table’. The sysent table is not available for developers and changes to this table is not supported and discouraged by Apple. However, since this table forms the core of the system calls, it is of interest for this research. System call hooking is a technique that allows for interference with system calls. By ‘replacing’ the original system call function in the sysent table by a function of the developer, the operating system will call the developer’s function instead of the original system call function. Typically, the developer performs the desired actions after which it calls the original kernel function to continue ‘normal’ execution of the system call as intended by the operating system. As explained in Section 3.1.5 system call hooking is a technique that is typically used by rootkits to hide their existence and manipulate the operating system at a very low level. The technique can also be used for this research since the performance overhead of monitoring the system calls would be minimal and has an unrestricted monitoring scope. The disadvantage however, is that protections that Apple has put in place to protect against system call hooking and other exploitations, have to be bypassed or defeated.

5.3. Kernel extension

To monitor real time, system wide execution of system calls, one has to reside very close to the core component in system call execution. The sysent table can be seen as this core component since every system call execution is routed using this table. System call hooking, as explained in the previous section, appears the most viable option. This section explains the creation of a kernel extension that has as its main purpose; creating a log of all executed system calls, which will later form the dataset as discussed in Section 5.1.1.

5.3.1. Defeating OS X kernel protections

Kernel memory is protected against tampering and malicious use. Two main mechanism are in place to perform these protections:

- **Kernel Address Space Layer Randomisation**
  KASLR is a mechanism that randomises the offsets at which processes are loaded into memory. It aims to protect memory against shellcode (malicious code) that loads itself at specific memory locations. Due to the randomising factor, the shellcode can not load itself at statically defined

memory locations that are available after exploitation of the kernel. This renders the shellcode unsuccessful. Upon boot of the operating system, the kernel is loaded at a random offset, defined by the bootloader, a layer connecting hardware and software.

- **Kernel memory write protection**
  Kernel memory by default is write protected. Disabling the write operation to kernel memory protects the kernel against modifications by malicious programs. Write protection is enforced by the CPU when it executes memory pages owned by the kernel. Write protection is indicated in 64-bit control register 0 (CR0) of the CPU. It is a register that defines several control parameters regarding the execution of memory pages. The CR0 contains a write protection (WP) bit that if set to 1, makes the memory page read-only. Anything on this memory page can not be modified.

Bypassing KASLR
In order to hook system call functions, the memory location of the `_sysent` table has to be determined. Due to KASLR, this is not a static address. In order to bypass KASLR and dynamically determine the `_sysent` memory location, the randomised offset has to be calculated at runtime of the operating system. Typically, this is performed using the binary of kernel and the memory location of exported functions, for example the famous `printf()` function or interrupt 80 (`INT80`)[5]. On a high level, one determines the memory location of an exported function in the kernel binary on the disk and the location of the same function loaded in memory. The delta of these addresses provides the (random) amount of memory locations by which the kernel memory has been shifted. Once the offset has been determined, the `_sysent` table can be located.

In the kernel extension built for this research, a more dynamic approach was implemented. Mach-O binaries contain a symbolic table in their header while loaded into memory. This symbolic table contains references to the memory locations of global variables and exported functions. Resolving the location of the `_sysent` table consists of 4 steps (visualised in Figure 5.1):

1. *Find the location of INT80 in the IDT.* The Interrupt Descriptor Table (IDT) is an array in the header of the kernel memory image containing the kernel handlers for the interrupts. This table is exported in the header and thus the memory location is resolvable. The memory location of interrupt 80 (`INT80`) can be obtained from the IDT table.

2. *Locate the base of the kernel header.* Using the interrupt 80 memory address obtained from the IDT, the start of the header can be found. The memory structure of interrupt 80 handler is known, which can be used to linearly “slide” over the memory locations and match its structure with the loaded memory .

3. *Locate the __DATA segment from the kernel header.* From the base of the memory, the __DATA segment can be resolved using the __DATA segment reference. From the __DATA segment, local variables and functions can be read from and written to[6].

4. *Locate the _sysent table from the __DATA segment.* Because the structure of the `_sysent` table is known, again a linearly brute-force technique is used to “slide” over the memory locations and match the `_sysent` structure with the structure found in memory.

Once the (dynamic) `_sysent` memory location as been resolved, system call handlers can be resolved and hooked.

The exact implementation of the resolution of the `_sysent` table is available at the open source project named “Onyx the Black Cat” by Pedro Vilaça[7].

Disabling Write-Protection
Disabling write protection involved less effort, since the CR0 register can be retrieved easily using global functions. These global functions are defined in `proc_reg` header file in the XNU kernel, the kernel contained in OS X. The WP-bit can then be changed to the value 0.

[5]`http://ho.ax/downloads/Defiling-Mac-OS-X-Ruxcon.pdf`
[6]`https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/`
[7]`https://github.com/gdbinit/onyx-the-black-cat`

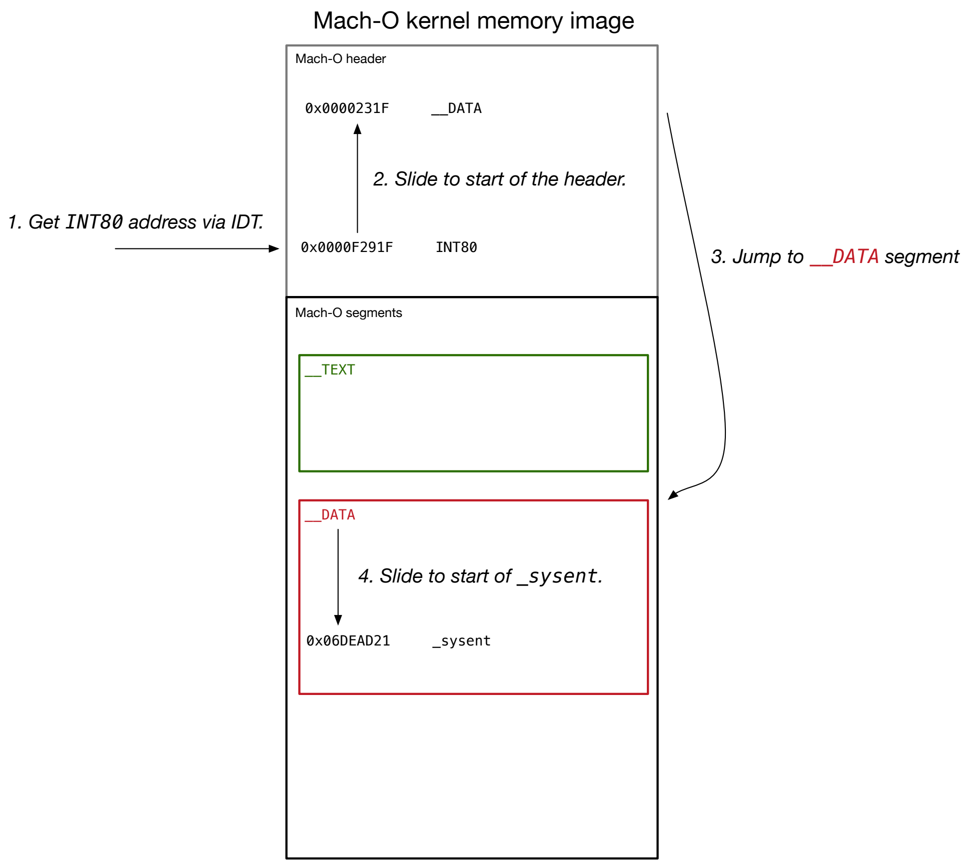

Mach-O kernel memory image
Mach-O header
0x0000231F __DATA
2. Slide to start of the header.
1. Get INT80 address via IDT.
0x0000F291F INT80
3. Jump to __DATA segment
Mach-O segments
__TEXT
__DATA
4. Slide to start of _sysent.
0x06DEAD21 _sysent

Figure 5.1: Process of retrieving the memory address of the `_sysent` table which defines the system call implementations. The memory addresses used are fictional.

5.3.2. Hooking system call functions

Once the sysent table has been located and the write protections are turned off, the defined system call functions can be hooked. Hooking technique replaces the original functions by self defined functions that perform their operation and call the original function as return value, assuring that the operating system remains functional. Figure 5.2 shows this structure.

Every system call in the sysent table (also presented in Figure 5.2), is contained in its own struct, shown in code block 5.1. This sysent system call struct contains among other information, a pointer to the system call implementation, the return type and the number of arguments.

```c
struct sysent_system_call {
    sy_call_t    *sy_call;
    sy_munge_t   *sy_arg_munge64;
    int32_t      sy_return_type;
    int16_t      sy_narg;
    uint16_t     sy_arg_bytes;
};
```

Listing 5.1: C struct defining required information regarding a system call.

The main purpose of hooking the system calls, is the ability to intercept the system call. Intercepting the system call allows for logging metadata of each individual system call. Every system call is passed a pointer to; the structure of the calling process (`struct proc *p`), a structure of arguments (system call specific) and a pointer to memory for the return value. The pseudo-code of a generic hooking function is shown in Listing 5.2. Specific metadata that are logged are described in Section 5.3.4.

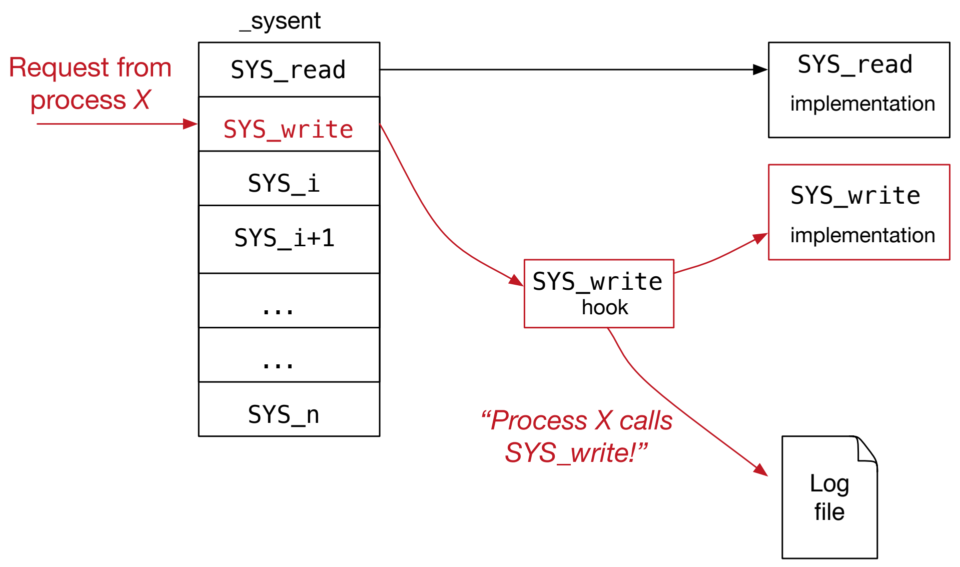


Figure 5.2: High level `sysent`-table representation in which all the system calls are defined. System call `SYS_read` represents the normal system call wiring to its implementation. System call `SYS_write` represents a hooked system call, which is logged to the log file to create a system call trace.

```
int hook_write(struct proc *p, struct write_args *a, user_ssize_t *r) {
        _log_metadata(p, a);
        return _kernel_fnctions[SYS_write](p, a, r);
}
```

Listing 5.2: Generic system call hook funtion

Since the structure of the arguments passed to a system call function differs for each system call, it is not possible to hook all functions with a generic system call hook-function. For every system call, a hook function is defined which are hooked into the sysent table (`_sysent[]`).

Code sample 5.3 provides the hook mechanism for hooking the write system call function. In essence, the original system call function is stored into an array (`_kernel_fnctions`) for use within the hook function. The original function in the `_sysent` table is then replaced with the hook-function.

```
_kernel_fnctions[SYS_write] = (void*)_sysent[SYS_write].sy_call;
_sysent[SYS_write].sy_call = hook_write;
```

Listing 5.3: Hooking a system call function

After hooking each system call of interest (see Appendix B for a list of all hooked system calls), kernel memory protections are enabled again and the system continues to operate, using the hooked-system calls. As explained in Appendix B, not all system calls are hooked, because several system calls are extremely frequently used by processes.

KAuth to monitor process execution

As explained in Section 5.2.3, KAuth is a framework that allows kernel extensions to be notified upon certain file system events, defined by Apple. Due to the relative difficulty in retrieving corresponding binary's file system path of the of a process, compared to the possibilities that KAuth provided, Kauth was used to gather information about the process creation. The KAuth scope `KAUTH_SCOPE_FILEOP` allows to filter for `KAUTH_FILEOP_EXEC` notifications of file system operations, which include process executions (`execve()` system call). KAuth calls the defined callback function which will log the path of the binary, process ID and parent process ID upon execution. This is useful information to obtain to gather more insights in possible process execution structures.

Table 5.1: Metadata of a system call that is captured.

Name	Abbreviation	Explanation
Uptime	time	Uptime of the system, in microseconds precision, to keep track of system call order.
Process ID	pid	Together with the parent process ID, this provides the ability to perform traces in the dataset.
Parent process ID	ppid	Together with the process ID, this provides the ability to perform traces in the dataset.
Privileges	is_root	Does the calling process have superuser privileges.
Process name	procname	Name of the process
Process execution path	ppath	File system path of the process
System call name	sys_call	Name of the system call
SYS_write location	SYS_write_loc	Location of the where the write system call writes to.

5.3.3. Preventing deadlocks

Hooking system call functions has implications regarding normal performance. Every additional operation performed inside the hook-function may invoke other system calls. Specifically, logging the system call data to a file invokes the `SYS_write` (the write system call). This can easily cause a deadlock. In a deadlock, resources are waiting for each other, bringing the system in a state in which it not able to perform any other operations anymore. To prevent deadlocks in system call logging, the hook-functions filter on the calling process. If the call appears to be invoked from the kernel (pid=0) or syslogd (the process that writes the system call data to a file), it will not log the system call data.

5.3.4. Gathering metadata of system calls

Specific information of interest is not only the calling process and system calls that were executed, but also metadata regarding the calling process and the arguments with which the system call was called. This enriches the dataset with possible valuable information, easily extractable from the process structure and arguments structure passed to a system call hooking-function. Table 5.1 provides an overview of the available properties logged for each system call.

Each system call will be represented according to the semicolon separated format shown in Listing 5.4. The first line presents the column descriptions, the rows below each represent a system call invocation with corresponding metadata.

```
time          ; process name; pid; ppid; syscall                ; is_root;
0:5:6,448659; pboard      ; 474; 1   ; SYS_pipe               ; 0;
0:5:6,448689; pboard      ; 474; 1   ; SYS_posix_spawn        ; 0;
0:5:6,450010; /bin/sh     ; 474; 1   ; NEW_PROCESS            ; 0;
0:5:6,450205; sh          ; 476; 474 ; SYS_shared_region_chk; 0;
```

Listing 5.4: Process pboard executes `/bin/sh` process and sh starts calling system calls (OSX.OceanLotus.A).

5.3.5. Constructing the log

In order to find patterns that can be used to describe malicious system call behaviour, the system call usage of malware has to be monitored and recorded. This kernel extension will create a record of every hooked system call made by every process (except for the processes explained in Section 5.3.3). The log is created by writing to both the `system.log` generic logging (located in the default location `/var/log/system.log`) and to the serial port. The advantage of logging over a serial port is its thread-safety, since a log operation over the serial port is fully synchronous[8]. When the serial port log is available (only in a virtual environment, or with a second computer connected over a FireWire port[8]), this log file is preferred. The log file will form the raw dataset of system calls, that will later be used to analyse malicious behaviour.

5.3.6. Load priority of the kernel extension at boot time

Collecting the system calls as early as possible in the boot process allows for a more detailed look of the behaviour of both the system and applications as well as the malware with which the system may be infected. Apple allows to specify the order in which kernel extensions are loaded upon boot time of the system. The property `OSBundleRequired` in the corresponding `Info.plist` of the extension informs the system that the kernel extension must be available for loading during early boot. This valid values for this property include `Root` (required to mount root), `Local-Root` (required to mount root on locally attached storage), `Network-Root` (required to mount root on network-attached storage), `Safe Boot` (required even in safe-mode boot) and `Console` (required to provide character console support, in single user mode).

This kernel extension would like to be loaded before the root of the file system is mounted, which means before any processes other than the kernel processes are started.

5.4. Environment setup and data gathering process

The (isolated) environment in which the system call data is gathered is explained in this section. First, the environment setup is described, thereafter the system call trace data collecting process is explained.

5.4.1. Environment setup

To prevent the malware from harming production systems, the malware samples are ran inside a Virtual Machine. OS X 10.6 Snow Leopard and 10.11 El Capitan are virtualised. Virtual Machines also provide the ability to rollback the state of a system to a previous saved (uninfected) state of the system. Due to OS compatibility issues of some malware samples (i.e. the Crisis rootkit, see Section 3.1.5), OS X 10.6 is also virtualised. Both systems are cleanly installed, OS X 10.11.3 from an App Store download of which the SHA-1 checksum is verified, OS X 10.6 is installed from an Apple installation DVD.

In the first phase, the systems are installed with basic, very commonly used office applications:

- Microsoft Office 2011 for Mac,
- OS X Mail client with an installed mail account,
- OS X Safari web browser, without Adobe Flash and Java Runtime,
- OS X Calendar application,
- OS X Address book/Contacts application.

5.4.2. Process of gathering system call data

To distinguish malicious system call behaviour from benign, two types of data set are required.

- A dataset in which only benign processes perform system calls: benign dataset.
- Datasets in which malware performs its system calls: malware datasets.

Benign dataset

The benign dataset is extracted from the cleanly installed virtual systems. Typical office behaviour is simulated using the installed applications. An exact workflow of the interactions with the system is

[8]Developer forum “Stack Overflow” question answered by well respected member Phil Dennis-Jordan: `https://stackoverflow.com/questions/36327605/printf-in-system-call-returns-malformed-output/`

Table 5.2: A list of OS X malware samples analysed in this research.

no.	Name	Type	Detection date	
1	OSX.Flashback	Trojan	09/30/2011	
2	OSX.Crisis.I	Rootkit	07/25/2012	
3	OSX.FakeCodec	Adware	02/03/2013	
4	OSX.LaoShu.A	Backdoor	01/21/2014	
5	OSX.CoinThief.A	Trojan	02/26/2014	
6	OSX.Xslcmd	Trojan	09/05/2014	
7	OSX.Wirelurker	Trojan	11/06/2014	
8	OSX.Janicab	Trojan	11/26/2014	
9	OSX.iWorm	Trojan	01/05/2015	
10	OSX.Kitmos.A	Backdoor	03/04/2015	
11	OSX.Genieo!gen1	Adware	05/18/2015	
12	OSX.Malcol	Adware	05/21/2015	
13	OSX.Downloader	Adware	07/29/2015	
14	OSX.Jahlav.A	Trojan	07/29/2015	
15	OSX.InstallCore	Adware	11/16/2015	*before*
16	OSX.EliteKeylogger	Keylogger	02/15/2016	*after*
17	OSX.OceanLotus	Trojan	02/19/2016	
18	OSX.Crisis.II	Rootkit	02/26/2016	
19	OSX.KeRanger.A	Trojan	03/06/2016	
20	OSX.Pirrit	Adware	04/06/2016	
21	OSX.Bundlore	Adware	04/11/2016	

described in Appendix C. To ensure that the simulation represented the behaviour of benign users, a survey under 30 real Mac user was conducted. Chapter 8 elaborates on the survey and evaluation.

Malware datasets
According to Symantec, 55 unique OS X malware samples have been found since 2010[9]. Obtaining functional malware samples is not trivial and in order to create a system call trace from a malware sample, the sample must be complete and functional. Often, only specific malicious parts of a OS X application that are not executable are uploaded to malware sample collecting services like VirusTotal. For this research, 21 functional OS X malware samples were obtained from different sources[10][11][12]. Table 5.2 provides an overview of the functional malware samples obtained and analysed in this research.

Every sample in Table 5.2 is ran in the exact same environment as the clean system for 10 minutes, after which the malware has performed its infection phase. Based on the literature study performed in Chapter 4, 10 minutes of recording appears to be the maximum amount of time a sample requires to infect a system. The infection phase is particularly interesting to study because if this stage could be prevented, malware itself would not have the ability to infect the system. A detection and prevention method for this phase would secure a system against malware. The kernel extension is launched and ran just before manual infection of malware samples, by executing the malicious binaries.

After 10 minutes, the log is captured and replaced by an empty log file, and the state of the system is rolled back to the ‘pre-malware’ state, ready for the next malware sample to monitor and record.

Eventually, after the process is completed, a large benign system call log is acquired, together with a log of system calls for every malware sample listed above. A sample of the log/raw dataset is provided in Appendix D. From this point in this research, the analysis of malicious behaviour starts.

[9]https://www.symantec.com/security_response/landing/azlisting.jsp?azid=0
[10]https://objective-see.com/
[11]https://www.virustotal.com/
[12]https://researchcenter.paloaltonetworks.com/

5.5. Conclusions

In order to analyse behaviour of processes based on system calls, a log of system calls performed by processes (system call traces) is required. Constructing a log of system call traces is not trivial. This chapter evaluated the possible tools currently available and concluded none of the tools was able to monitor every system call of every process, including unknown malicious processes. A kernel module was developed to facilitate the construction of a log. The kernel module “rewired” the system call implementation of the operating system by modifying protected kernel memory. Security technologies such as Address Space Layer Randomisation (ASLR) and memory write protection were bypassed to achieve the modifications in memory. The “rewiring” of the system call implementations allowed to include the logging functionality. When the kernel module is loaded, every system call invocation (including its metadata described in Table 5.1) logs to a pre-defined log file. The kernel module is open-source and available on GitHub[13].

As many as possible functional malware samples (shown in Table 5.2) were executed in an environment in which the kernel module was loaded. This process produced logs of system call traces for every malware sample executed. These log files are analysed in the next chapters.

[13] `https://github.com/vivami/grey_fox`

6 Heat map analysis

This chapter describes the initial analysis of the system call datasets performed in order to extract possible detection patterns. Heat maps are used to gain initial, general insights in collected system call traces datasets, with the aim to provide a starting point for further analysis. General observations and patterns are described and some initial malware detection system call patterns are presented at the end of this chapter.

6.1. Heat maps

A heat map is a graphical representation of data where the individual values contained in a two-dimensional matrix are represented as colours. The matrix can be extracted from a table, on which the columns and rows are represented in the matrix that is visualised using a heat map. Larger values are represented by dark colours and smaller values by lighter colours. A heat map can provide a good first general insight into a large dataset in which relations are not trivially extractable. In this chapter, a heat map is used to gain insights in a possible relation between processes and — the amount of — invoked system calls. From the raw dataset (described in Appendix D), a table has been constructed featuring system calls on the x-axis top row and process names on the y-axis first collum. Table 6.1 illustrates an example of the table format to which the raw system call dataset is converted.

	SYS_call_1	SYS_call_2	SYS_call_i+1	SYS_call_n
process1	6	8	11	4
process2	19	3	13	2
process3	6	9	7	12
process4	31	2	5	2

Table 6.1: Table format of the underlying system call data of the heat maps.

Several features of the raw dataset are not taken into account and left out in the analysis using heat maps. Time sequencing of system calls for example is not possible to combine in the current format of the heat map and will be discussed in the next chapter when more thorough analysis is applied.

R library `d3heatmap`[1] is used to create the heat map resulting in a — similar to Figure 6.1 — visualisation of the system call datasets.

[1]http://blog.rstudio.org/2015/06/24/d3heatmap/

6.2. Benign heat map

Figure 6.1 shows a heat map of the benign dataset.

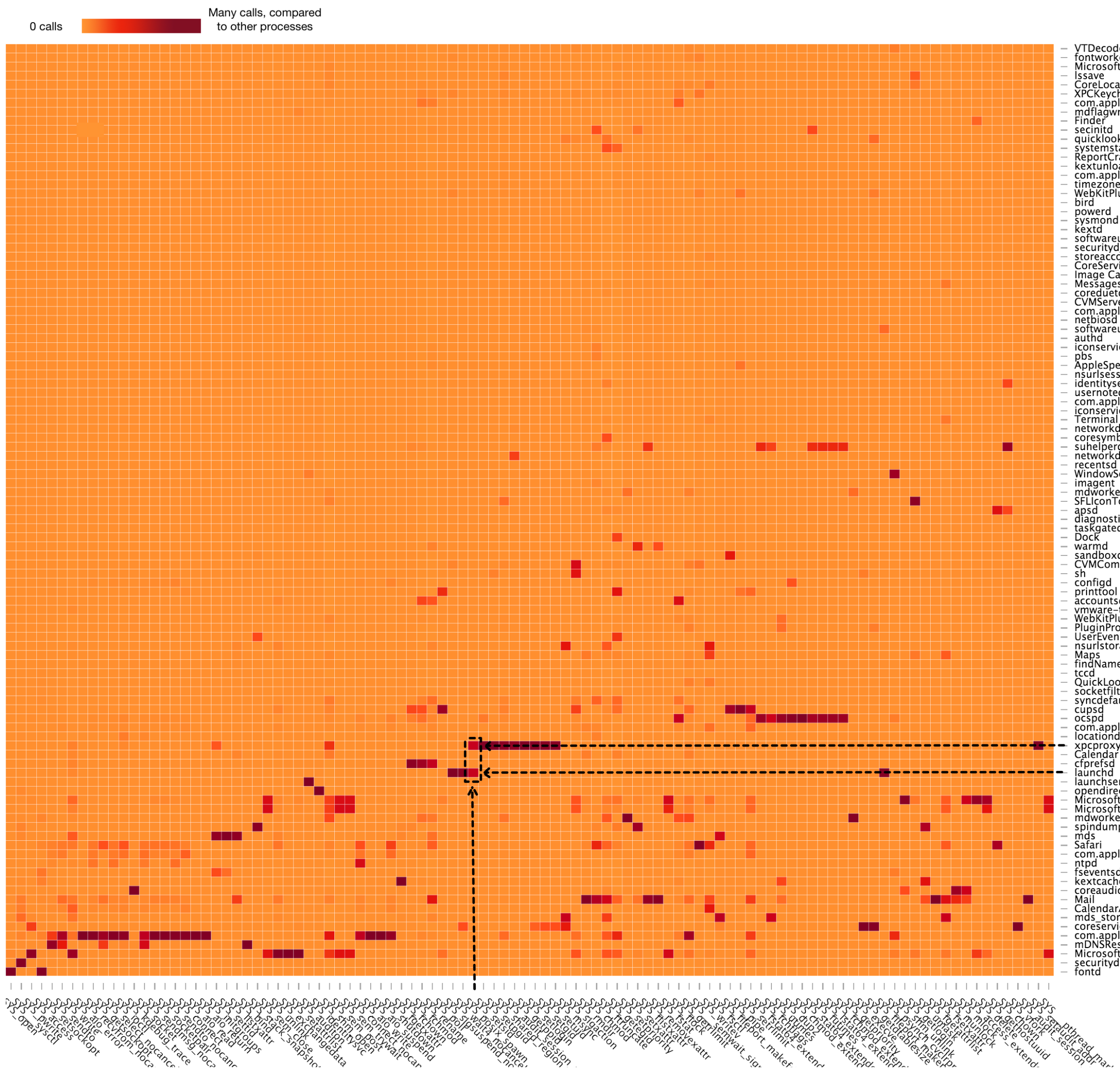


Figure 6.1: Heat map visualisation of the benign system call traces. The system calls called and processes are listed on the x-axis and y-axis respectively. Each square represents the number of system calls, as a normalised data point. Darker blocks represent an outlying number of system calls performed by a process, relative to other processes. The black arrows and square indicates `SYS_posix_spawn` being called by `launchd` and `xpcproxy` only.

A few observations are made:

1. The `SYS_posix_spawn` system call is only performed by `xpcproxy`[2] — an OS X process providing mechanisms for interprocess communications — and is tightly integrated with the `launchd`[3] process, responsible for starting and managing processes after the kernel has been loaded. These are described in more detail in Section 9.1. These calls are indicated in the benign heat map visualisation in Figure 6.1 by the black arrows.

2. The programs used the most, web browser and MS Word, perform the far majority of the system calls. The `ocspd`[4] process is related to SSL certificate validation, very likely used by the web browser to perform HTTPS connections.

[2]`https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man8/xpcproxy.8.html`
[3]`https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man8/launchd.8.html`
[4]`https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/ocspd.1.html`

3. The `fontd` process does an extraordinary amount of `SYS_open` calls. This is very likely due to MS Word loading and/or using the font files when opening a Word document.
4. MS Word does many `SYS_write` calls, very likely involved in the process of saving a Word document.
5. `com.apple.WebKit`, part of the Safari web browser, does many system calls related to socket operations. Sockets are used to communicate over Internet connections.
6. The shell process `sh` does exactly one `SYS_execve` call and several `SYS_sigaction` calls. This call should be ignored, since it used to launch `kextunload`, a process related to unloading the kernel extension from the kernel. This is not considered “ordinary” behaviour, as kernel extensions are typically loaded on boot of the system, as explained in Section 5.3.6.

The last observation however, is important to make. As assumed by Niels Provos [36] and as will be shown in the following sections, malware specifically performs many process execution (`SYS_execve` and `SYS_posix_spawn`) system calls to execute new processes. The next section will provide more observations regarding malware specific behaviour, that is not observable in the benign dataset.

6.3. Malware heat maps

From initial observations of the heat map visualisations of the malware samples listed in Section 5.4.2, some common patterns that are not existent in the benign dataset can be observed. The sections below explain both common, as well as malware specific system call patterns.

6.3.1. iWorm

The iWorm malware is a recent and relatively sophisticated OS X malware sample. iWorm also performs interesting behaviour as can be seen in the heat map of the iWorm dataset in Figure 6.2.

As explained in Section 3.1.3, iWorm is a backdoor that spawns two processes, ‘0’ and ‘1’. Process ‘1’ is a process that installs the iWorm backdoor binary ‘JavaW’ and creates a LaunchDaemon to gain persistence. Observed from the heat map in Figure 6.2 the following observations can be made:

1. DNS process `mDNSResponder` performes many socket operations. This may indicate iWorm trying to establish a connection with its C&C servers.
2. Process `1` does many system calls regarding semaphore operations; `SYS_semget` and `SYS_semop`. These operations and system calls are absent in the benign data set.
3. Process `JavaW` performs `SYS_fstatfs` and `SYS_getdirectories` system calls that are related to obtaining file system statistics[5], which is on par with iWorms backdoor nature.
4. The shell process `sh` is significantly more prominent compared to the benign dataset. It also performs many `SYS_fork`, `SYS_execve`, `SYS_dup2` and `SYS_pipe` calls. These calls are completely absent in the benign system call dataset.

6.3.2. Wirelurker

As described in Section 3.1.3, Wirelurker is a malware sample targeting iOS devices via OS X platform with the goal to exfiltrate data from the iOS device. Two patterns similar to the patterns observed in iWorms heat map, are also observed in Wirelurker’s heat map (Figure 6.3). A shell process is present, invoking process execution calls (`SYS_execve`).

The following sections show last observation will recur in many malware heat maps, and may be a good first indication of an infection process.

[5]`https://www.freebsd.org/cgi/man.cgi?query=fstatfs`

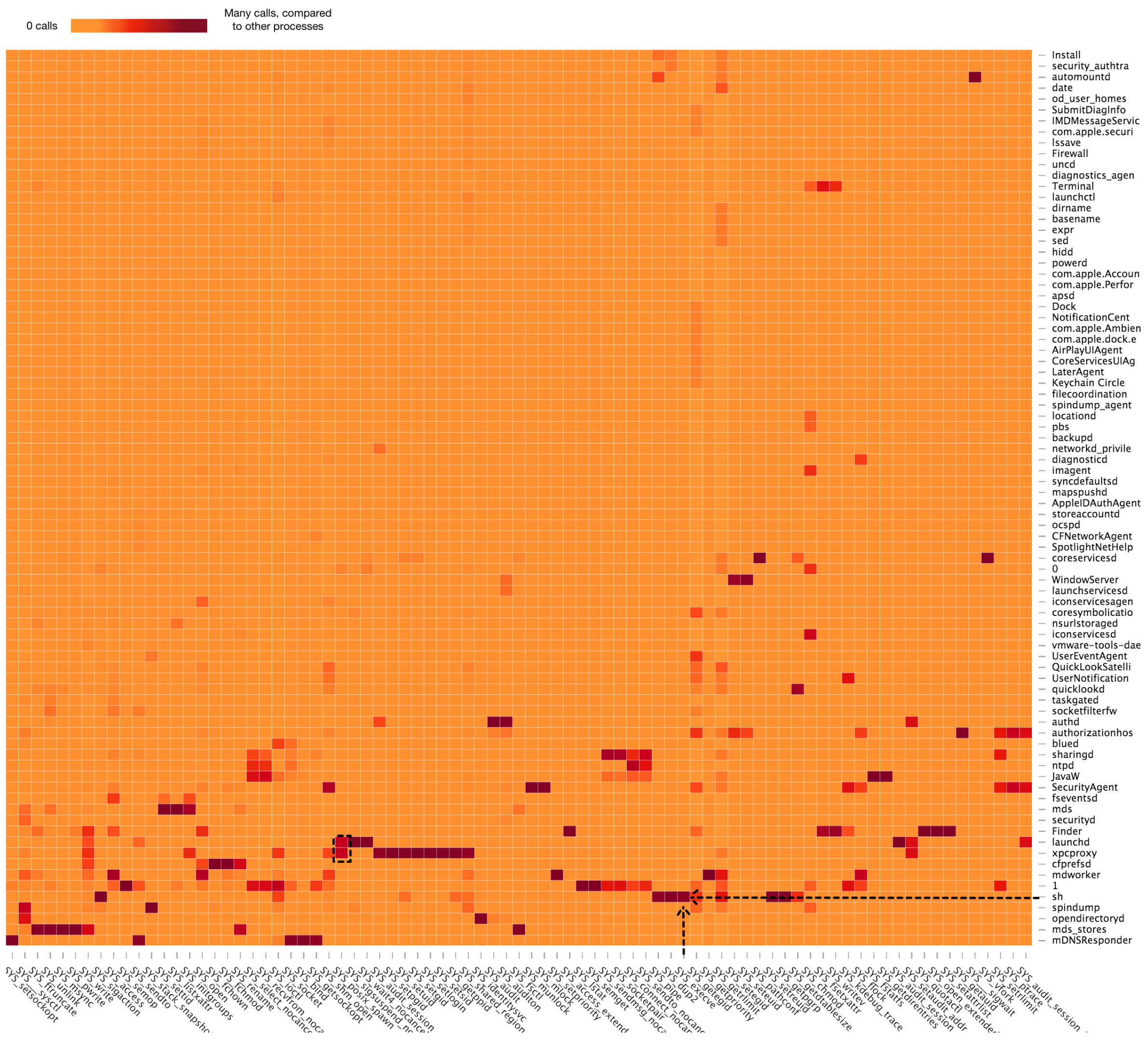


Figure 6.2: Heat map visualisation of the iWorm malicious system call trace. The system calls called and processes are listed on the x-axis and y-axis respectively. Each square represents the number of system calls, as a normalised data point. Darker blocks represent an outlying number of system calls performed by a process, relative to other processes. The black arrows indicate the `SYS_execve` calls performed by `sh`.

6.3.3. Common patterns

Appendix E shows the heat maps of other malware samples listed in Section 5.4.2. This section will describe and explain several patterns the heat maps of the malware samples have in common.

A majority of the malware samples is a Trojan Horse and uses an installer to infect an OS X system. Observed is that the installers heavily rely on a shell process (`sh`) used to install components of the malware onto the system. The `sh` process makes various system calls that are not existent in the benign dataset. In particular, the `SYS_execve` (used to execute a binary file), `SYS_fork` (creates a new (child) process[6]), `SYS_pipe` (used for interprocess communication[7]) and `SYS_dup2` (used for file I/O[8]). This pattern indicates execution of processes that communicate with each other over pipes.

Other behaviour prominently present in the malware is the information gathering `SYS_getpgrp` which is a system call used to obtain information about the group ID in which a process belongs[9].

[6]http://linux.die.net/man/2/fork
[7]http://linux.die.net/man/2/pipe
[8]http://linux.die.net/man/2/dup2
[9]http://linux.die.net/man/2/getpgrp

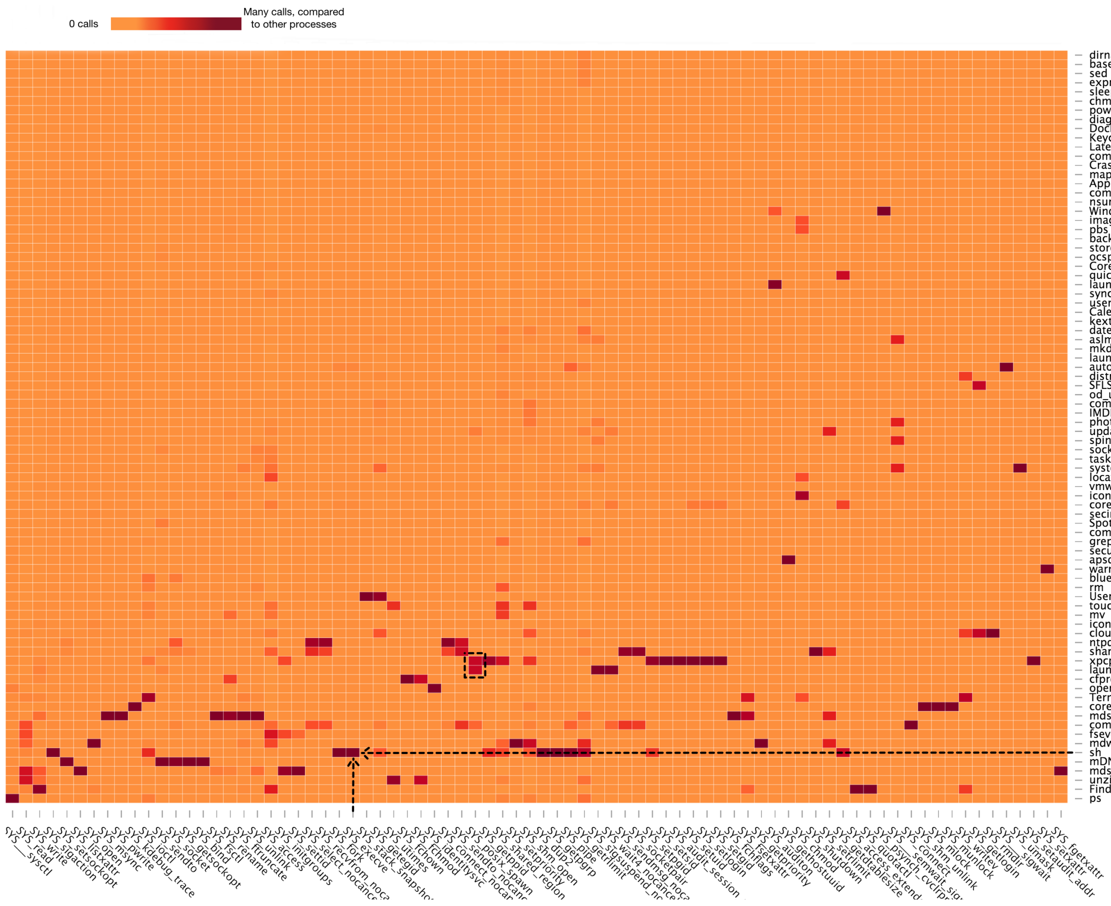


Figure 6.3: Heat map visualisation of OSX.Wirelurker malicious system call trace. The system calls called and processes are listed on the x-axis and y-axis respectively. Each square represents the number of system calls, as a normalised data point. Darker blocks represent an outlying number of system calls performed by a process, relative to other processes. The black arrows indicate the `SYS_execve` calls performed by `sh`, and the black square indicates `SYS_posix_spawn` being called by `launchd` and `xpcproxy`. Note: the `SYS_posix_spawn` calls by `com.apple.MailServer` and `update` are malicious calls performed by the OSX.Wirelurker malware.

Many occurrences (> 121) of `SYS_sigaction` in a very short period of time (< 5 minutes) are also a typical phenomenon observed by the malicious processes. The `SYS_sigaction` system call is used to change the action taken by a process on receipt of a specific signal by the OS. The signals are used for interprocess communication. In the benign dataset, the `SYS_sigaction` system call only occurred approximately 30 times by a single process.

Some of the samples that have a backdoor nature, show specific socket operations. Genieo malware for example, calls the `SYS_listen` system call, indicating that the malware marks a socket as a passive socket, that is, as a socket that will be used to accept incoming connection requests[10]. The `SYS_listen` is not used by any benign process.

Processes that connect to the internet are identifiable by their system calls related to sockets. In the benign dataset, processes that do not open a connection to the Internet, do not perform the `SYS_socket`, `SYS_setsockopt`, `SYS_socketpair`, `SYS_sendmsg_nocancel`, `SYS_sendto` and several others related to socket operations. The heat maps provide insights in the use of Internet by malicious processes, purely based on the aforementioned system calls.

The Genieo malware family (Genieo, MacVX, PaperPost, MacInstaller) is identifiable by significantly more `SYS_posix_spawn` calls. On a normal system, only the Apple processes `launchd` and `xpcproxy` appear to perform a certain amount of `SYS_posix_spawn` calls. The MacInstaller family tends to perform twice the amount of `SYS_posix_spawn` calls, compared to `launchd` and `xpcproxy`. Also the Flashback

[10]`http://linux.die.net/man/2/listen`

malware shows this behaviour. The SYS_posix_spawn is a system call providing similar functionality to the SYS_execve call[11].

Both MacInstaller and PaperPost show many similarities as explained above, also perform an extraordinary amount of SYS_open system calls to hidden files on the file system. In the two minutes that both samples ran, both made 5447 and 5352 SYS_open respectively. Providing a perspective: the benign dataset, where (apart from fontd process) the maximum amount of SYS_open calls was 1194 by Microsoft Word and the second largest consumer, mdworker (Apple owned), performed 487 open calls.

6.4. Conclusions

Heat maps are a valuable visualisation technique to provide initial insights into a raw data set, and this chapter has not shown differently. Many observations relating to both benign and malicious system call traces have been made in this chapter. Section 6.2 starts by presenting a visualisation of the benign system call traces (Figure 6.1). Observations made are the use of execution system calls only performed by OS X processes and Internet facing applications performing many system calls related to sockets. Section 6.3 visualises in Figures 6.2 and 6.3 malicious system call traces which shows the presence of shell processes and execution system calls performed by other processes than only OS X processes. After analysing all the heat maps of all malicious system call traces collected, execution system calls and shell processes appeared to be a common pattern present in malware and absent in benign processes.

The next chapter elaborates on these observations and provide more detailed patterns related to the detection of malicious behaviour.

[11]http://linux.die.net/man/3/posix_spawn

7 Manual sequencial analysis

After the initial analysis facilitated by the visualisation of the system call traces using heat maps in Chapter 6, some interesting patterns and behaviour that appear to belong to malicious processes were found. This chapter dives deeper into the the conclusions drawn in Chapter 6 by performing manual analysis for additional, more fine-grained and in particular sequential malicious patterns. In this section, all major malware samples are analysed by stepping through the raw system call trace, constructed by the kernel extension. Particularly interesting bits of these system call traces are listed in text blocks and is elaborated upon. Finally, this chapter is ended by providing a detailed conclusion in Section 7.9 of the findings and extracted common malicious patterns.

7.1. iWorm

The iWorm sample starts upon execution of the `Install` binary in the 'Trojaned' installer packed in the installer of an iWorm infected application.

The listing below shows the `Install` binary being launched.

```
Time; process name; PID; PPID; syscall; root privs; write/binary path;
0:1:43,82720; /Users/m/Desktop/Install.app/Contents/MacOS/Install; 362; 0;
     NEW_PROCESS; 0;
```

Listing 7.1: iWorms Install binary executed.

The `Install` binary performs several system calls seemingly unrelated to malicious behaviour and one set of (as defined in Section 6.3.3) possibly malicious calls: `SYS_shm_open`, `SYS_pipe`, `SYS_fork`, `SYS_dup2`, in that particular order. It finalises execution with two `SYS_execve` calls that launch 2 processes, `0` (executed with elevated privileges) and `1`.

```
Time; process name; PID; PPID; syscall; root privs; write/binary path;
0:1:43,249674; Install; 362; 1; SYS_shm_open; 0;
0:1:43,299625; Install; 362; 1; SYS_pipe; 0;
0:1:43,313445; Install; 362; 1; SYS_fork; 0;
0:1:43,339043; Install; 363; 362; SYS_dup2; 0;
0:1:43,352853; Install; 363; 362; SYS_execve; 0;
0:1:48,203333; /Users/m/Desktop/Install.app/Contents/MacOS/0; 363; 0;
    NEW_PROCESS; 1;
0:1:48,203846; Install; 362; 1; SYS_execve; 0;
0:1:48,208305; /Users/m/Desktop/Install.app/Contents/MacOS/1; 362; 0;
    NEW_PROCESS; 0;
```

Listing 7.2: iWorms Install binary performs its operations.

Process `0` executed by `Install` performs among some other frequently occurring system calls; two `SYS_posix_spawn` that launch a `sh` process. `SYS_posix_spawn`[1] appears to have very similar functionality to `SYS_execve`. Both are used to launch/spawn new (child) processes.

```
Time; process name; PID; PPID; syscall; root privs; write/binary path;
0:1:48,238454; 0; 363; 362; SYS_posix_spawn; 0;
0:1:48,239772; /bin/sh; 363; 0; NEW_PROCESS; 1;
0:1:48,326414; 0; 363; 362; SYS_posix_spawn; 0;
0:1:48,331568; /bin/sh; 363; 0; NEW_PROCESS; 1;
```

Listing 7.3: iWorms Install binary performs its operations.

These two `SYS_posix_spawn` calls seem to finalise the execution of `0`.

The other process executed by `Install` is named '`1`' and performs a large amount of system calls related to semaphores. According to Patrick Wardle [53], the '`1`' process is not a malicious process, but should be the process that installs the benign application. However, the iWorm sample used to create this system call trace dataset, did not contain a benign program to be installed. It is unclear what functionality resides inside the '`1`' binary, however semaphore operations could be considered uncommon behaviour, as it is not visible in any other dataset, both benign and malicious.

The `sh (367)` process is used to perform the malicious tasks. It launches `launchd` which appears to be responsible for the installation of a launch daemon (static analysis by Patrick Wardle can be used to confirm this behaviour [53]).

```
Time; process name; PID; PPID; syscall; root privs; write/binary path;
0:1:48,287125; sh; 367; 363; SYS_execve; 0;
0:1:48,288733; /bin/launchctl; 367; 0; NEW_PROCESS; 1;
```

Listing 7.4: sh process executes launchctl.

The `launchctl` process executes the `JavaW` process. A shell (`sh`) process interacting with the OS X process `launchctl` can be considered malicious behaviour, as the pattern is not visible anywhere in the benign dataset. In addition, `launchctl` is responsible to manage and control daemons/agents (automatic startup items), which is operating system functionality that malware is particularly interested in [59], also shown in the rest of this chapter. A large majority of the benign applications and processes are not interested in auto-run functionality, later shown in Chapter 8.

The `JavaW` process is mainly concerned with connecting to a socket (the iWorm backdoor). In its initial phase, it gathers some information about the file system and thereafter starts a socket connection and remains idle.

```
0:1:48,405641; JavaW; 368; 1; SYS_fstatfs; 0;
0:1:48,407694; JavaW; 368; 1; SYS_getdirentries; 0;
0:1:48,409801; JavaW; 368; 1; SYS_getdirentries; 0;
0:1:48,428004; JavaW; 368; 1; SYS_shm_open; 0;
```

Listing 7.5: JavaW binary performs file system information gathering.

7.2. Flashback

Similar to iWorm, the Flashback trojan horse installed a backdoor onto the victims machine. It is launched when the user clicks on an installer that appears to install Adobe Flash Player, but instead launches the malware. Note that the launch path seems odd since it resides in system preserved directories, but is due to Flashback using the default OS X installer framework, which is located in `/System/Library/CoreServices/Installer.app`[2].

[1]https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/posix_spawn.2.html
[2]http://www.mactech.com/articles/mactech/Vol.26/26.02/TheFlatPackage/index.html

```
0:1:51,911939; /System/Library/CoreServices/Installer.app/Contents/MacOS/
    Installer; 363; 0; NEW_PROCESS; 0;
```

Listing 7.6: Flashback installer launched.

After several read and access operations, the `Installer` calls `SYS_posix_spawn` and launches a process named `runner`. It continues to perform commonly seen system calls and finalises by removing itself.

The OS X itself (`xpcproxy`) appears to launch `installd` itself, which launches `install_monitor`, `runner` and `preinstall`. The first first two show benign behaviour. The latter is the most interesting.

preinstall
`preinstall` is a process featuring elevated privileges showing very similar behaviour to the iWorm `0` process. After several socket operations, `preinstall` launches an elevated `sh` process.

```
0:2:1,561297; preinstall; 375; 373; SYS_posix_spawn; 0;
0:2:1,562724; /bin/sh; 375; 0; NEW_PROCESS; 1;
```

Listing 7.7: preinstall spawns sh.

The `sh` process performs 8 times the exact same pattern of system calls, executing multiple file system management binary like `rm` in its last `SYS_execve` call (line 25 in the listing below).

```
 1  0:2:1,877144; /bin/sh; 375; 0; NEW_PROCESS; 1;
    0:2:1,878564; sh; 382; 375; SYS_shared_region_check_np; 0;
    0:2:1,881384; sh; 382; 0; SYS_write; 1; /dev/dtracehelper
    0:2:1,883866; sh; 382; 375; SYS_ioctl; 0;
 5  0:2:1,884948; sh; 382; 375; SYS___sysctl; 0;
    0:2:1,887464; sh; 382; 0; SYS_write; 1; /dev/tty
    0:2:1,888514; sh; 382; 375; SYS_ioctl; 0;
    0:2:1,889503; sh; 382; 375; SYS_getegid; 0;
    0:2:1,890524; sh; 382; 375; SYS_sigaction; 0;
10  0:2:1,891477; sh; 382; 375; SYS_sigaction; 0;
    0:2:1,892425; sh; 382; 375; SYS_sigaction; 0;
    0:2:1,893345; sh; 382; 375; SYS_sigaction; 0;
    0:2:1,894312; sh; 382; 375; SYS_sigaction; 0;
    0:2:1,895291; sh; 382; 375; SYS_sigaction; 0;
15  0:2:1,897127; sh; 382; 375; SYS_sigaction; 0;
    0:2:1,899107; sh; 382; 375; SYS___sysctl; 0;
    0:2:1,900137; sh; 382; 375; SYS_getppid; 0;
    0:2:1,902136; sh; 382; 375; SYS_shm_open; 0;
    0:2:1,905723; sh; 382; 375; SYS_getpgrp; 0;
20  0:2:1,906639; sh; 382; 375; SYS_sigaction; 0;
    0:2:1,907597; sh; 382; 375; SYS_getrlimit; 0;
    0:2:1,908787; sh; 382; 375; SYS_sigaction; 0;
    0:2:1,911487; sh; 382; 375; SYS_sigaction; 0;
    0:2:1,912428; sh; 382; 375; SYS_sigaction; 0;
25  0:2:1,913371; sh; 382; 375; SYS_execve; 0;
```

Listing 7.8: sh pattern performed by Flashback

These 8 iterations will eventually spawn several `sh` (shell) processes, that remain active until the system shuts down. It is expected that the idle shell processes are waiting for HTTP/HTTPS connections, however due to Flashback's C&C hashtags are not found on Twitter anymore, it may fail to continue to operate. This is not verifiable based on the system call trace.

7.3. Wirelurker

Wirelurker was one of the malware samples that was not gathered in installer form. The sample's launch instruction was executed via the OS X Terminal application (a bash shell), instead of launched by the kernel when it is packaged inside an installer/Trojan Horse.

```
0:1:56,519720; sh; 365; 351; SYS_execve; 0;
0:1:56,521053; /Users/m/Desktop/update; 365; 0; NEW_PROCESS; 1;
```

Listing 7.9: Launch of Wirelurker (update binary)

Wirelurkers `update` binary launches a shell process that launches several helper binaries to extract and store its malicious files into system reserved directories.

```
0:1:56,622014; sh; 367; 366; SYS_execve; 0;
0:1:56,626348; /usr/bin/unzip; 367; 0; NEW_PROCESS; 1;
```

Listing 7.10: sh launches unzip

```
0:1:57,243858; sh; 369; 366; SYS_execve; 0;
0:1:57,244945; /bin/mv; 369; 0; NEW_PROCESS; 1;
```

Listing 7.11: sh launches mv

```
0:1:57,441376; sh; 378; 366; SYS_execve; 0;
0:1:57,442641; /usr/bin/unzip; 378; 0; NEW_PROCESS; 1;
0:1:57,583832; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/
    libcrypto.1.0.0.dylib
0:1:57,742539; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/
    libiconv.2.dylib
0:1:57,769903; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/
    libimobiledevice.4.dylib
0:1:57,811844; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/libiodb
    .dylib
0:1:57,832382; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/liblzma
    .5.dylib
0:1:57,840472; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/
    libplist.2.dylib
0:1:57,894782; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/libssl
    .1.0.0.dylib
0:1:57,900817; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/
    libusbmuxd.2.dylib
0:1:58,44824; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/libz.1.
    dylib
0:1:58,55016; unzip; 378; 0; SYS_write; 1; /private/etc/manpath.d/libzip
    .2.dylib
```

Listing 7.12: sh launches unzip and unzips a dynamic library

This behaviour can be confirmed by the static analysis performed by Palo Alto Networks [55]. The `sh` process then uses `chmod` and `rm` to move files in the appropriate directories, as explained in PAN's Wirelurker's analysis [55].

Subsequently, the `sh` process will do a `SYS_execve` call to launch the `launchctl` process and create a startup item.

`xpcproxy` is then used to spawn another Wirelurker process: `com.apple.MailServiceAgentHelper`.

```
0:2:11,768285; xpcproxy; 391; 1; SYS_posix_spawn; 0;
0:2:11,777087; /usr/bin/com.apple.MailServiceAgentHelper; 391; 0;
    NEW_PROCESS; 1;
```

Listing 7.13: xpcproxy launches com.apple.MailServiceAgentHelper

The `com.apple.MailServiceAgentHelper` does not appear to do any maliciously identified calls, until it spawns another `sh` process using the `SYS_posix_spawn` system call.

```
0:2:11,840556; com.apple.MailSe; 391; 1; SYS_posix_spawn; 0;
0:2:11,841978; /bin/sh; 391; 0; NEW_PROCESS; 1;
```

Listing 7.14: MailServiceAgentHelper launches new sh process

That `sh` process is used to touch may files in system preserved directories:

```
0:2:11,894315; touch; 393; 0; SYS_write; 1; /usr/bin/periodicdate
0:2:11,918309; touch; 394; 0; SYS_write; 1; /usr/bin/systemkeychain-helper
0:2:11,941527; touch; 395; 0; SYS_write; 1; /usr/bin/com.apple.appstore.
    PluginHelper
0:2:11,964870; touch; 396; 0; SYS_write; 1; /usr/bin/com.apple.
    MailServiceAgentHelper
/System/Library/LaunchDaemons/com.apple.periodic-dd-mm-yy.plist
0:2:12,11968; touch; 398; 0; SYS_write; 1; /System/Library/LaunchDaemons/
    com.apple.systemkeychain-helper.plist
0:2:12,36063; touch; 399; 0; SYS_write; 1; /System/Library/LaunchDaemons/
    com.apple.appstore.plughelper.plist
0:2:12,60575; touch; 400; 0; SYS_write; 1; /System/Library/LaunchDaemons/
    com.apple.MailServiceAgentHelper.plist
0:2:12,84518; touch; 401; 0; SYS_write; 1; /usr/bin/stty5.11.pl
0:2:12,108955; touch; 402; 0; SYS_write; 1; /private/etc/manpath.d
```

Listing 7.15: sh uses touch to create several files.

Thereafter, the shell process launches two `grep`[3] and a `ps`[4] processes, presumably to gather system configuration parameters.

After `ps` has finished gathering process information, `com.apple.MailServiceAgentHelper` opens a socket connection and starts interacting with the socket.

Several OS X applications that have an iOS counterpart, like `photolibraryd`, `cloudphotosd`, `usernoted`, `com.apple.CloudP`, start to appear in the logs afterwards. This is interesting , as Wirelurker is specifically targeting iOS devices, exfiltrating user information.

Wirelurker processes exit and sleep after this behaviour, according to [55] waiting for an iOS device to connect. Since this research is focussed on OS X and detecting threats in an early stage, the iOS infection phase of Wireluker is out of the scope of this thesis.

7.4. MacInstaller

When a user clicks on what appears to be an MP3-file, it starts the `macLauncher` process which after several system calls that were previously identified as suspicious malicious behaviour, launches a shell process.

```
0:5:34,53365; /Volumes/Downloader/Ziggy MarleyBeach In Hawaii_mp3.app/
    Contents/MacOS/macLauncher; 462; 0; NEW_PROCESS; 0;
...
0:5:37,105410; macLauncher; 462; 1; SYS_pipe; 0;
0:5:37,106461; macLauncher; 462; 1; SYS_pipe; 0;
0:5:37,107471; macLauncher; 462; 1; SYS_fork; 0;
0:5:37,109000; macLauncher; 462; 1; SYS_wait4_nocancel; 0;
0:5:37,109238; macLauncher; 465; 462; SYS_dup2; 0;
0:5:37,111250; macLauncher; 465; 462; SYS_dup2; 0;
0:5:37,112239; macLauncher; 465; 462; SYS_dup2; 0;
```

[3] http://linux.die.net/man/1/grep
[4] http://linux.die.net/man/1/ps

```
0:5:37,113228; macLauncher; 465; 462; SYS_execve; 0;
0:5:37,114714; /bin/sh; 465; 0; NEW_PROCESS; 0;
```

Listing 7.16: macLauncher spawns its shell process.

The shell process gathers some information about the user groups, and eventually launches a `Downloader` executable.

```
0:5:37,137395; sh; 465; 462; SYS_getgroups; 0;
0:5:37,138423; sh; 465; 462; SYS_getpgrp; 0;
...
0:5:37,144097; sh; 465; 462; SYS_execve; 0;
...
0:5:37,178371; /Volumes/Downloader/.app/Downloader.app/Contents/MacOS/
    Downloader; 465; 0; NEW_PROCESS; 0;
```

Listing 7.17: shell process used to spawn Downloader.

`Downloader` performs many read and write operations, appearing to corrupt the indexing service `mds` databases.

```
0:5:37,304576; Downloader; 465; 0; SYS_write; 0; /private/var/folders/z2/
    ygg7tspj47j8bb2pxw21b8s80000gn/C/mds/mds.lock
0:5:37,309104; Downloader; 465; 0; SYS_write; 0; /private/var/folders/z2/
    ygg7tspj47j8bb2pxw21b8s80000gn/C/mds/mdsObject.db_
0:5:37,328211; Downloader; 465; 0; SYS_write; 0; /private/var/folders/z2/
    ygg7tspj47j8bb2pxw21b8s80000gn/C/mds/mdsDirectory.db_
```

Listing 7.18: Downloader is writing to mds's (file system indexing service) databases.

After many pairs of read and write operations, `Downloader` spawns `ioreg` and `awk`. They both live for a shot period, after which `Downloader` starts to perform many socket operations. It appears that processes connecting to the internet and listening for instructions or packets on a socket, are identifiable by the system call `SYS_getsockopt`. This call in combination with `SYS_setsockopt` are used to send and receive packets over a socket (usually connected to an Internet interface). This assumption is based on the many operations performed by malware samples, and being only performed by benign processes that require an internet connection, i.e. Safari web browser, Office applications (presumably for updates).

```
0:5:49,916084; Downloader; 465; 462; SYS_write; 0;
0:5:49,917210; Downloader; 465; 462; SYS_read; 0;
0:5:49,918266; Downloader; 465; 462; SYS_getsockopt; 0;
```

Listing 7.19: Downloader performs socket operations.

After many dozens of sockets operations, a typical malware pattern appears in the system call trace:

```
0:5:50,196141; Downloader; 465; 462; SYS_fork; 0;
0:5:50,207916; Downloader; 476; 465; SYS_dup2; 0;
0:5:50,211487; Downloader; 476; 465; SYS_dup2; 0;
0:5:50,214974; Downloader; 476; 465; SYS_dup2; 0;
0:5:50,217674; Downloader; 476; 465; SYS_execve; 0;
0:5:50,220998; /bin/sh; 476; 0; NEW_PROCESS; 0;
```

Listing 7.20: Downloader performs typical malicious system call pattern.

The shell process starts `hdiutil` which uses `SYS_posix_spawn` to launch `diskimages-helper`. iTunes is then launched, and fed the MacInstaller `*.mp3` file.

```
0:5:55,907747; Downloader; 465; 462; SYS_posix_spawn; 0;
0:5:55,936362; /Volumes/silentextension/extension.app/Contents/MacOS/mac.
    installer; 465; 0; NEW_PROCESS; 0;
```

Listing 7.21: Downloader spawns mac.installer.

`mac.installer` appears to perform no malicious behaviour, other than receiving and sending data over a socket. Eventually `mac.installer` exits and `Downloader` takes back control. `Downloader` executes a shell process that is used to launch another MacInstaller process `AppYS`, which could be considered the main binary of the malware. `AppYS` first performs several socket operations. Thereafter, it launches `osascript` to execute an AppleScript that is used to perform the actual malicious operations on the files system.

```
0:7:24,218401; AppYS; 519; 465; SYS_posix_spawn; 0;
0:7:24,221526; /usr/bin/osascript; 519; 0; NEW_PROCESS; 0;
```

Listing 7.22: AppYS launches AppleScript.

The AppleScript is mainly concerned with copying an application named ‘Royalbgood’ to a directory. `AppYS` then stores a LaunchAgent, and starts the `launchctl` process. It performs this last operation three times, shortly after each other.

```
0:7:26,118526; AppYS; 519; 465; SYS_posix_spawn; 0;
0:7:26,119820; /bin/launchctl; 519; 0; NEW_PROCESS; 0;
...
0:7:26,195429; AppYS; 519; 465; SYS_posix_spawn; 0;
0:7:26,196843; /bin/launchctl; 519; 0; NEW_PROCESS; 0;
...
0:7:26,281641; AppYS; 519; 465; SYS_posix_spawn; 0;
0:7:26,287504; /bin/launchctl; 519; 0; NEW_PROCESS; 0;
```

Listing 7.23: AppYS launches launchctl.

Afterwards, it launches several Unix `killall` processes calling an extraordinary amount of `SYS___sysctl`. After this phase, `AppYS` starts again a shell process that is used to copy/install a Safari plugin.

```
0:7:56,79491; AppYS; 519; 465; SYS_posix_spawn; 0;
0:7:56,81618; /bin/cp; 519; 0; NEW_PROCESS; 0;
0:7:56,92233; cp; 563; 0; SYS_write; 0; /Users/m/Library/Safari/Extensions
    /Royalbgood.safariextz
```

Listing 7.24: AppYS copies a malicious Safari extension.

After this operation, `AppYS` exits and MacInstaller appears to have achieved its target; injecting advertisements into webpages using a browser plugin.

7.5. Genieo

Genieo starts by launching the process `Installer`. Installer writes preferences in the default OS X preferences directory and performs several socket operations.

```
0:5:0,24931; Installer; 445; 1; SYS_posix_spawn; 0;
0:5:0,26475; /System/Library/Frameworks/JavaVM.framework/Versions/A/
    Commands/java_home; 445; 0; NEW_PROCESS; 0;
```

Listing 7.25: Genieo spawns Java process.

The Java process lives very short, and does not perform any noteworthy operations. Intstaller uses several `SYS_posix_spawn` system calls to spawn `cp` and `rm` processes that are used to prepare an application named ‘Application.app’.

After the copying process has finished, `Installer` spawns a `Python` process.

```
0:5:7,760922; Installer; 445; 1; SYS_posix_spawn; 0;
0:5:7,765287; /usr/bin/python; 445; 0; NEW_PROCESS; 0;
0:5:7,781434; python; 452; 445; SYS_posix_spawn; 0;
...
```

```
0:5:7,785041; /System/Library/Frameworks/Python.framework/Versions/2.7/
    Resources/Python.app/Contents/MacOS/Python; 452; 0; NEW_PROCESS; 0;
```

Listing 7.26: Genieo spawns Python process.

The `Python` appears to be used for setting the proper file attributes of the application. After Python has finished its job, `Installer` then launches the 'Application'.

```
0:5:8,406850; Installer; 445; 1; SYS_posix_spawn; 0;
0:5:8,408706; /private/tmp/Application.app/Contents/MacOS/Application;
    445; 0; NEW_PROCESS; 0;
```

Listing 7.27: Installer executes Application process.

`Application` process remains idle for a while. In the meantime, another Genieo process is launched, interestingly enough, by an OS X system process. `com.genieoinnova` is then used to spawn several Unix processes primarily used to manage file permissions.

```
0:5:13,531826; xpcproxy; 461; 1; SYS_posix_spawn; 0;
0:5:13,536276; /Library/PrivilegedHelperTools/com.genieoinnovation.
    macextension.client; 461; 0; NEW_PROCESS; 1;
...
0:5:13,596716; com.genieoinnova; 461; 1; SYS_posix_spawn; 0;
0:5:13,598613; /usr/sbin/chown; 461; 0; NEW_PROCESS; 1;
...
0:5:13,596716; com.genieoinnova; 461; 1; SYS_posix_spawn; 0;
0:5:13,598613; /usr/sbin/chown; 461; 0; NEW_PROCESS; 1;
...
0:5:13,733378; com.genieoinnova; 461; 1; SYS_posix_spawn; 0;
0:5:13,736387; /bin/chmod; 461; 0; NEW_PROCESS; 1;
...
0:5:13,875735; com.genieoinnova; 461; 1; SYS_posix_spawn; 0;
0:5:13,877273; /bin/cp; 461; 0; NEW_PROCESS; 1;
```

Listing 7.28: Genieo uses several chown and chmod operations on files.

The `cp` process appears to write a launch daemon to the reserved directory. After the Unix processes perform their task, `Application` process starts to interact with `launchctl`, confirming the behaviour previously performed by `cp`.

```
0:5:14,671492; Application; 453; 445; SYS_posix_spawn; 0;
0:5:14,673052; /bin/launchctl; 453; 0; NEW_PROCESS; 0;
```

Listing 7.29: Application process spawns launchctl.

After the interaction, `Installer` performs a large amount of socket operations. Thereafter, `Installer` again launches several Unix processes related to file system operations. Subsequently, two times a shell process is spawned to execute `echo`.

```
0:8:11,784965; Installer; 445; 1; SYS_posix_spawn; 0;
0:8:11,786575; /bin/sh; 445; 0; NEW_PROCESS; 0;
0:8:11,825330; sh; 518; 517; SYS_dup2; 0;
0:8:11,826892; sh; 518; 517; SYS_execve; 0;
0:8:11,829826; /bin/echo; 518; 0; NEW_PROCESS; 0;
```

Listing 7.30: Installer process launches a shell process, which spawns the echo process.

The Java application launched is mainly concerned with operations on a socket, but later uses `SYS_execve` to execute System Profiler, an OS X application that provides information about the OS X system. Eventually, `defaults`[5] is launched 6 times, appearing to adapt firewall settings.

[5]https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/defaults.1.html

```
0:8:32,635981; Genieo; 528; 445; SYS_posix_spawn; 0;
0:8:32,638064; /System/Library/Frameworks/JavaVM.framework/Versions/A/
   Resources/MacOS/JavaApplicationStub; 528; 0; NEW_PROCESS; 0;
...
0:8:35,288114; JavaApplicationS; 528; 445; SYS_pipe; 0;
0:8:35,289550; JavaApplicationS; 528; 445; SYS_pipe; 0;
0:8:35,291139; JavaApplicationS; 528; 445; SYS_pipe; 0;
0:8:35,292426; JavaApplicationS; 528; 445; SYS_pipe; 0;
0:8:35,298371; JavaApplicationS; 534; 528; SYS_dup2; 0;
0:8:35,299599; JavaApplicationS; 534; 528; SYS_dup2; 0;
0:8:35,300749; JavaApplicationS; 534; 528; SYS_dup2; 0;
0:8:35,301854; JavaApplicationS; 534; 528; SYS_dup2; 0;
0:8:35,306899; JavaApplicationS; 534; 528; SYS_execve; 0;
0:8:35,308138; JavaApplicationS; 534; 528; SYS_execve; 0;
0:8:35,309329; JavaApplicationS; 534; 528; SYS_execve; 0;
...
0:8:38,852865; JavaApplicationS; 542; 528; SYS_execve; 0;
0:8:38,854872; /usr/bin/defaults; 542; 0; NEW_PROCESS; 0;
```

Listing 7.31: JavaApplicationS performs SYS_execve.

After this behaviour, `JavaApplicationS` mainly connects and interacts with sockets. Other malicious behaviour is not found in its system call trace.

7.6. InstallCore

InstallCore is one of the most recent malware samples, detected in February 2016. It distinguished itself from other malware due to its binaries being signed with a valid Apple Developer certificate. This allowed InstallCore to bypass Gatekeeper= which — as explained in Section 3.1.7 — verifies the signatures of binaries before execution.

InstallCore tries to trick a user into thinking it is installing a new version of Adobe Flash. Once the user launches the installer, InstallCore takes off.

```
0:6:10,781945; xpcproxy; 370; 1; SYS_posix_spawn; 0;
...
0:6:10,810977; /Volumes/Installer/Installer.app/Contents/MacOS/tirocinium;
    370; 0; NEW_PROCESS; 0;
```

Listing 7.32: tirocinium process started.

After many `SYS_ioctl`/`SYS_read` system call pairs, `tirocinium` eventually executes a shell process using `SYS_posix_spawn`.

```
0:6:13,165176; tirocinium; 370; 1; SYS_posix_spawn; 0;
0:6:13,166494; /bin/sh; 370; 0; NEW_PROCESS; 0;
```

Listing 7.33: tirocinium launches shell process.

The shell process performs several known patterns:

```
0:6:13,190134; sh; 374; 370; SYS_pipe; 0;
0:6:13,191222; sh; 374; 370; SYS_fork; 0;
0:6:13,192630; sh; 374; 370; SYS_pipe; 0;
0:6:13,193588; sh; 374; 370; SYS_fork; 0;
...
0:6:13,196035; sh; 374; 370; SYS_fork; 0;
0:6:13,198652; sh; 375; 374; SYS_dup2; 0;
```

Listing 7.34: Shell process performs process execution related system calls.

Then the `codesign` and `xargs` processes are launched by the shell process using `SYS_execve`. Both processes do not perform any seemingly interesting behaviour.

The `tirocinium` process thereafter performs an extraordinary amount (over 60.000) of `SYS_ioctl` system calls. It is unclear why the malware shows this behaviour. Patrick Wardle in an static analysis of InstallCore claims that these calls are performed to obfuscate behaviour of the process: "*All these calls are 'useless' - they're simply present to change the signature (hash) of identical samples and/or perhaps thwart simple AV emulators and/or hinder analysis. However, in terms of manual analysis, while somewhat annoying, in reality they don't stop us.*"[6]

In its final phase, `tirocinium` starts to perform socket operations, and presumably downloads Adobe Flashplayer to a temporary directory.

```
0:7:36,443582; tirocinium; 370; 0; SYS_write; 0; /private/var/folders/z2/
    ygg7tspj47j8bb2pxw21b8s80000gn/T/adobe_flashplayer_e2c7b.dmg
```

Listing 7.35: tirocinium writes DMG file to the disk.

Eventually, Safari is launched by `xpcproxy` (described in more detail in Section 9.1), presumably caused by `tirocinium`.

7.7. Crisis

In late February 2016, a new version of the HackingTeam Crisis rootkit for OS X was sighted by the information security industry[8]. This later version will be referred to as Crisis.II. Both versions are analysed in this section.

7.7.1. Crisis.I

The OS X Crisis rootkit by HackingTeam consists of many components. The components are different binaries that need to be assembled in order to function according to the intends of its authors. In this research, only one component of the rootkit has been obtained and can be analysed. This component is likely the a part of the userspace component of the rootkit[7].

Crisis was launched by executing `OSX_Crisis_A32E0`. This process writes another binary to a System Preferences directory.

```
0:15:13,310759; OSX_Crisis_A32E0; 1044; 0; SYS_write; 1; /private/var/root
    /Library/Preferences/2Md1ctl2/WaAvsmZW.EMb
0:15:13,319438; OSX_Crisis_A32E0; 1044; 893; SYS_fork; 0;
0:15:13,319993; OSX_Crisis_A32E0; 1046; 1; SYS_execve; 0;
0:15:13,320172; /private/var/root/Library/Preferences/2Md1ctl2/WaAvsmZW.
    EMb; 1046; 0; NEW_PROCESS;
```

Listing 7.36: OSX_Crisis_A32E0 process executes WaAvsmZW.EMb.

The `WaAvsmZW.EMb` process then performs some well-known calls to write a file named `mdworker.flg`. According to an analysis performed by SANS [34], `mdworker.flg` is a file used to mark a system as compromised by Crisis and is checked by Crisis before it starts to perform its malicious functionality.

```
0:15:13,367869; WaAvsmZW.EMb; 1046; 1; SYS_pipe; 0;
0:15:13,367878; WaAvsmZW.EMb; 1046; 1; SYS_fork; 0;
```

Listing 7.37: WaAvsmZW.EMb performs process execution related system calls.

```
0:15:13,435378; WaAvsmZW.EMb; 1046; 1; SYS_chmod; 0;
0:15:13,435432; WaAvsmZW.EMb; 1046; 1; SYS_chown; 0;
```

[6] `https://objective-see.com/blog/blog_0x0C.html`
[7] SHA-1: 41e6edd798979be2bdfc87e293d00c54d793a340

```
0:15:13,435623; WaAvsmZW.EMb; 1046; 1; SYS_fsetxattr; 0;
0:15:13,435629; WaAvsmZW.EMb; 1046; 0; SYS_write; 1; /private/var/root/
    Library/Preferences/2Md1ctl2/mdworker.flg
```

Listing 7.38: WaAvsmZW.EMb writes mdworker.flg to disk.

```
0:15:13,438177; WaAvsmZW.EMb; 1049; 1046; SYS_dup2; 0;
0:15:13,438182; WaAvsmZW.EMb; 1049; 1046; SYS_dup2; 0;
0:15:13,438186; WaAvsmZW.EMb; 1049; 1046; SYS_dup2; 0;
```

Listing 7.39: WaAvsmZW.EMb SYS_dup2 calls.

Eventually, `WaAvsmZW.EMb` uses `SYS_posix_spawn` to spawn `launchctl`, an operation previously indicated as malicious behaviour.

```
0:15:13,439772; WaAvsmZW.EMb; 1049; 1046; SYS_posix_spawn; 0;
0:15:13,439945; /bin/launchctl; 1049; 0; NEW_PROCESS; 1;
```

Listing 7.40: AppYS copies a malicious Safari extension.

Crisis.I terminates after this call. However, it is unknown what behaviour is performed by the other components, which could not be obtained for this research.

7.7.2. Crisis.II

Crisis.II has major differences in terms of binary obfuscation complexity compared to Crisis.I. The binary is encrypted using Apples encryption scheme (as explained by Patrick Wardle in Section 3.1.7). It is also packed with a custom-made[8] packer[9] to obfuscate the binary and prevent detection by signature checking anti-virus software. This behaviour has not been seen before by malware on OS X[8].

The sample is started using a shell process, since the sample was not obtained in 'installer-form'.

```
0:5:4,783342; /Users/m/Desktop/58
    e4e4853c6cfbb43afd49e5238046596ee5b78eca439c7d76bd95a34115a273; 370;
    354; NEW_PROCESS; 1;
...
0:5:4,812105; 58e4e4853c6cfbb4; 370; 354; SYS_write; 1; /private/var/root/
    Library/Preferences/8pHbqThW/_9g4cBUb.psr
0:5:4,841386; 58e4e4853c6cfbb4; 371; 370; SYS_execve; 0;
0:5:4,842744; /private/var/root/Library/Preferences/8pHbqThW/_9g4cBUb.psr;
     371; 1; NEW_PROCESS; 1;
```

Listing 7.41: After Crisis.II is launched, it writes a binary to disk and executes that binary.

After the `58e4e4853c6cfbb4` Crisis.II process is started, it writes a binary `_9g4cBUb.psr` to the file system and executes that binary using a `SYS_execve` call.

The `_9g4cBUb.psr` process then writes a LaunchAgent to the LaunchAgent respective directory.

```
0:5:5,546923; _9g4cBUb.psr; 371; 1; SYS_open; 0; /Users/root/Library/
    LaunchAgents/.dat0173.001;
```

Listing 7.42: After Crisis.II is launched, it writes a binary to disk and executes that binary.

During `_9g4cBUb.psr`'s lifetime, it spawns several binaries: two times `sysctl`, `chown`, four times `system_profiler`.

```
0:5:21,53072; system_profiler; 376; 371; SYS_posix_spawn; 0;
0:5:21,54709; /usr/sbin/system_profiler; 376; 371; NEW_PROCESS; 1;
...
0:5:5,331064; _9g4cBUb.psr; 371; 1; SYS_posix_spawn; 0;
```

[8]https://reverse.put.as/2016/02/29/the-italian-morons-are-back-what-are-they-up-to-this-time/
[9]http://www.kaspersky.com/en/internet-security-center/threats/suspicious-packers

```
0:5:5,333974; /usr/sbin/sysctl; 371; 1; NEW_PROCESS; 1;
```

Listing 7.43: Crisis.II binary executes multiple OS X processes.

After these calls, the OS X processes perform their actions, and `_9g4cBUb.psr` goes in an idle state. It is important to note that this Crisis.II sample is not complete and lacks functionality that is not analysed in this analysis. The described patterns however, should be enough to identify Crisis.II's behaviour.

7.8. Ransomware

Two functional ransomware samples were obtained for this research: OSX.KeyRanger.A and Gopher. The latter is a proof of concept built by Pedro Vilaça[10].

7.8.1. KeRanger

The infected Transmission BitTorrent client executes a malicious shell process before any UI elements are shown to the user.

```
0:13:58,523214; Transmission; 413; 1; SYS_posix_spawn; 0;
0:13:58,524348; /bin/sh; 413; 1; NEW_PROCESS; 0;
```

Listing 7.44: Transmission process spawns a shell process using the `SYS_posix_spawn` system call.

This shell process is then later used to spawn the ransomware process `kernel_service` [6].

```
0:13:58,530059; sh; 414; 413; SYS_execve; 0;
0:13:58,530631; /Users/m/Library/kernel_service; 414; 413; NEW_PROCESS; 0;
```

Listing 7.45: sh process executes the malicious `kernel_service` process.

`kernel_service` starts encrypting files after 3 days, given it has received encryption keys from its C&C, located at `lclebb6kvohlkcml.onion.link`. At the moment of the conducted research, this C&C was still operational. To speed up the infection process, the epoch time in `.kernel_time` was set back to an earlier moment of time.

Upon encryption of files by the `kernel_service`, many `SYS_write` calls are performed. Files on the file system are presumably encrypted and written back to the file system featuring a `*.encrypted` postfix. The `SYS_read` calls are expected to be performed by `kernel_service` also, but are not visible in our dataset since the `SYS_read` system calls are not hooked in the kernel extension, as explained in Appendix B. Listing 7.46 shows a small sample this behaviour. This behaviour is observed for every file in the user directory.

```
1:115:3,980185; kernel_service; 721; 1; SYS_access; 0;
1:115:3,989580; kernel_service; 721; 1; SYS_write; 0; /Users/m/Desktop/
    prep_kext.sh
1:115:3,994398; kernel_service; 721; 1; SYS_access; 0;
1:115:4,10534; kernel_service; 721; 1; SYS_chmod; 0;
1:115:4,14923; kernel_service; 721; 1; SYS_write; 0; /Users/m/Desktop/
    prep_kext.sh.encrypted
1:115:4,75076; kernel_service; 721; 1; SYS_access; 0;
1:115:4,83673; kernel_service; 721; 1; SYS_write; 0; /Users/m/Desktop/
    response.txt
1:115:4,89416; kernel_service; 721; 1; SYS_access; 0;
1:115:4,94868; kernel_service; 721; 1; SYS_chmod; 0;
1:115:4,99715; kernel_service; 721; 1; SYS_write; 0; /Users/m/Desktop/
    response.txt.encrypted
```

Listing 7.46: `kernel_service` encrypts files on the file system.

[10] https://github.com/gdbinit/gopher

7.8.2. Gopher

Gopher ransomware is a proof of concept and not featuring an ‘injection phase’. To run the ransomware sample, it was launched manually from a shell process controlled by the Terminal.app. Directly after, gopher starts encrypting the files on the file system. A similar pattern as OSX.KeyRanger is observed: many `SYS_write` calls touching many files on the system. Listing 7.47 shows a short sample of the gopher behaviour.

```
0:6:44,477232; gopher_encrypt; 624; 566; SYS_write; 0; /Users/m/Documents
    /Presentation1.pdf
0:6:44,486060; gopher_encrypt; 624; 566; SYS_write; 0;
0:6:44,491274; gopher_encrypt; 624; 566; SYS___sysctl; 0;
0:6:44,497652; gopher_encrypt; 624; 566; SYS_writev; 0;
0:6:44,503237; gopher_encrypt; 624; 566; SYS_write; 0; /Users/m/Documents
    /Presentation1.pptx
0:6:44,511840; gopher_encrypt; 624; 566; SYS_write; 0;
0:6:44,516091; gopher_encrypt; 624; 566; SYS___sysctl; 0;
0:6:44,531157; gopher_encrypt; 624; 566; SYS_writev; 0;
0:6:44,540171; gopher_encrypt; 624; 566; SYS_write; 0; /Users/m/Documents
    /test.docx
0:6:44,550778; gopher_encrypt; 624; 566; SYS_write; 0;
0:6:44,555413; gopher_encrypt; 624; 566; SYS_write; 0; /Users/m/Desktop/
    session_pub.key
```

Listing 7.47: `gopher_encrypt` encrypts files on the file system.

7.9. Conclusions

This chapter performed an in depth manual, sequential analysis of the system call traces of the collected malware samples for OS X. The analysis identifies various patterns that recur in many of the analysed malware samples, some of which recur in every sample. In addition to the patterns found in Chapter 6, the following system call patterns were identified:

1. **Execution of (child) processes**
   Every malware sample analysed in this chapter consists of multiple processes. The samples are launched by a single process execution, which then spawns multiple other processes. The other processes are used to perform specific actions that together resemble the behaviour of the malicious payload. In order to accurately identify malicious behaviour, it is important that the behaviour of all the processes in the malware’s process tree is observed.

2. **Execution of shell processes**
   In a far majority of the analysed samples, the most common pattern that recurred was the shell (`sh`) processes used to facilitate the malicious intent of the malware. The shell process was launched and used to launch and interact with other processes performing the malicious action. `SYS_posix_spawn` and `SYS_execve` are the two calls very frequently used by malware, but are absent in benign processes, apart from `launchd` and `xpcproxy` (both part of the OS X operating system).

3. **Interaction with launchctl**
   Every malware sample using persistency based on LaunchDaemon/Agents (auto-run items), eventually starts to interact with `launchctl` (an operating system process that is used to manage and control daemons and agents). Nowhere in the benign system call traces is behaviour seen of processes interacting neither with nor without using a shell with `launchctl`. In general, a shell process will use `SYS_execve` to launch `launchctl`, where any other process will use `SYS_posix_spawn` to launch `launchctl`. Typically, in case of a benign application requiring LaunchDaemon functionality to function, this daemon/agent configuration occurs at install time and does not recur afterwards.

4. **Ransomware performing write calls touching many files**
   Two OS X ransomware samples currently publicly known, show the same behaviour. `SYS_write` calls are performed touching all files in a user directory. The `SYS_write` are performed in a very short time interval.

The next chapter will evaluate these patterns as well as suggest detection patterns for OS X malware, based on system call traces.

8 Evaluation

The previous two chapters described a variety of malicious behaviour observed in OS X malware's system call traces. This chapter constructs detection patterns based on the observations and subsequently evaluates the detection patterns. The evaluation phase consists of metrics commonly used in the anomaly detection field, discussed in Chapter 4. Typically used metrics in this field of science are the Detection Rate (DR) and the False Positive Rate (FPR). The patterns constructed in Section 8.1 are evaluated in Section 8.2 and 8.3 respectively.

8.1. Detection patterns

Based on the observations and conclusions drawn in Section 7.9, the following detection patters are derived:

- **Execute system call usage by non-OS X processes (pattern 1)**
  A process other than `xpcproxy` or `launchd` performing a `SYS_posix_spawn` or `SYS_execve`. Pattern 2 refines this pattern to a more precise malicious pattern.
- **Execution of shell processes (pattern 2)**
  A `SYS_execve` or `SYS_posix_spawn` executing a shell process (i.e.: `/bin/bash`, `/bin/sh`, `/bin/python`), is a generic malicious pattern.
- **Interaction with launchctl (pattern 3)**
  A shell process (i.e.: `/bin/bash`, `/bin/sh`, `/bin/python`) launching the `/bin/launchctl` or `/usr/bin/crontab` binary, is a malicious pattern to gain persistency on the system using LaunchDaemons or cronjobs.
- **Ransomware performing write calls touching many files (pattern 4)**
  A process that is performing `SYS_write` calls (presumably in combination with `SYS_read` calls) touching many files on the file system in a very short period of time, is considered a malicious pattern shown by ransomware encrypting files on the file system.

These three core derived patters will be used as 'malware detection patterns'. Ideally, these patterns should only be observable on systems infected with malware and not visible on clean systems. An interim hypothesis could be formulated as: *'The three patterns described above are only visible on a malware infected OS X system.'* To evaluate and validate the hypothesis, two metrics commonly used in other anomaly detection literature are used:

- Detection Rate (DR): the detection rate is defined as the number of intrusion instances detected by the system (True Positive) divided by the total number of intrusion instances present in the test set [43].
- False Positive Rate (FPR): An event signalling a detection system to produce an alarm when no attack has taken place [43].

8.2. Detection rate

The detection rate is an important metric to measure the accuracy of the detection patterns. In order to measure the detection rate as accurately as possible, the number of malware samples tested against the detection rules should be as high as possible. In the following sections, all malware samples described in Table 5.2 are analysed.

8.2.1. Detecting patterns

The process of detecting patterns consisted of creating a system call trace of the malware samples in Table 5.2 by executing the malware samples on in a Virtual Machine (fully patched and updated OS X 10.11.3 system). The system call traces were logged in the exact same way as described in Section 5.4.2. The malicious system call traces were then manually inspected for containing the detection patterns stated in Section 8.1. Table 8.1 shows the amount of detected patterns for each malware sample.

The horizontal line in Table 8.1 indicates the moment of pattern construction. After the detection patterns were derived and constructed, several other newer malware samples surfaced (listed below "*after*"). In other words, the patterns were constructed based on system call traces from samples 1 to 15 and verified to be existent in patterns 16 to 21.

no.	Name	Pattern 1	Pattern 2	Pattern 3	Pattern 4	
1	OSX.Flashback	21	11	1	–	
2	OSX.Crisis.I	2	0	5	–	
3	OSX.FakeCodec	21	6	3	–	
4	OSX.LaoShu.A	2	1	–	–	
5	OSX.CoinThief.A	20	10	1	–	
6	OSX.Xslcmd	14	8	–	–	
7	OSX.Wirelurker	43	8	1	–	
8	OSX.Janicab	32	19	3	–	
9	OSX.iWorm	13	4	2	–	
10	OSX.Kitmos.A	12	1	2	–	
11	OSX.Genieo!gen1	61	5	3	–	
12	OSX.Malcol	23	11	–	–	
13	OSX.Downloader	73	2	4	–	
14	OSX.Jahlav.A	21	6	2	–	
15	OSX.InstallCore	8	3	–	–	*before*
16	OSX.EliteKeylogger	20	5	0	–	*after*
17	OSX.OceanLotus	12	4	1	–	
18	OSX.Crisis.II	7	0	–	–	
19	OSX.KeRanger.A	6	2	0	PRESENT	
20	OSX.Pirrit	401	35	8	–	
21	OSX.Bundlore	242	110	8	–	

Table 8.1: Lists all occurrences of patterns in analysed malware samples. The entries containing a '–' mean the criteria was not applicable. No persistency was gained or the malware sample was not ransomware.

Table 8.1 shows that all malware samples analysed trigger the detection patterns and in many cases, multiple times. Note: as previously stated, Crisis samples obtained are incomplete and lack large portions of functionality, hence the lack of pattern 2. OSX.KeyRanger ransomware was a malware sample that used persistency by 'backdooring' a BitTorrent client. It would be started when the user started the BitTorrent client and thus be persistent on the system, hence no usage of LaunchDaemons or cronjobs.

Based on the results in Table 8.1 it can be stated that the detection patterns feature a 100% detection rate for OS X malware.

8.3. False Positive Rate

Determining the False Positive Rate is more difficult compared to determining the Detection Rate of the patterns. The main difficulty originates from the different types of nature existing in benign applications. A far majority of the applications running on OS X do not perform tight interaction with the underlying system. Many of those applications honour the sandboxing restrictions that Apple directs and while it is hard to gain an overview of the Mac App Store (MAS) usage, currently over 16,230[1] Mac applications are available in the MAS. On the ofter end of the spectrum are the applications featuring a nature that tightly integrates and interacts with the underlying system. In some cases, these applications also perform modifications to the underlying system. Examples of these applications are:

- **GPGTools:** A toolset related to GPG encryption (e-mail in particular). It features a plug-in that nestles inside the default OS X Mail application to provide it with extra functionality.
- **Flux:** A utility that changes the color scheme of the display to a warmer color, decreasing the eyestrain of a user sitting for long periods of time behind the display. It is a system plug-in that hooks into the display settings of the system to automatically modify the display colour behaviour.
- **Electron:** Electron is a developer toolset that allows developers to develop JavaScript apps to run locally. It is based on NodeJS and uses many cross-platform binaries. The cross-platform binaries are often interacted with in a similar way on all Unix-like systems.

Based on these observations, the initial hypothesis was that the applications with a Linux native nature ported to OS X system or an OS X native application tightly integrating with the system or aim to extend the functionality of OS X features, will with a high probability generate false positives. Early inspections for false positives showed applications relying on shell processes to execute standalone binaries. It is expected the shell process is used as an interface to interact with standalone binaries used to provide cross-platform compatibility (elaborated on in Section 9.2). The next subsection explains the validation of this hypothesis.

8.3.1. User profiles

In order to provide a more accurate analysis of the False Positive Rate three different types of users are described and used for the evaluation. Expected is that a user only using applications downloaded from the Mac App Store will not experience false positives, where a more advanced user will more often trigger false positives with its application usage. Three user profiles are defined:

1. *App Store user:* a user only using applications downloaded from the App Store.
2. *Typical user:* occasionally uses applications distributed outside the App Store.
3. *Power user/developer:* a user who uses advanced features of the system or uses developer environments to develop software.

8.3.2. Measuring false positives

The user profiles allow for more accurate definition of the FPR by differentiating between the type of applications triggering false positives. The evaluation consists of two phases:

1. *Collecting application usage under Mac users.* A survey under 25 real Mac user was conducted to gain insights in application usage by users fitting a profile. A user was asked to select its best suiting profile of the profiles described above and upload their installed applications.
2. *Testing the top X applications for False Positives.* In the same environment as described in Section 5.4.1 the top 90 applications derived from the survey were analysed for Detection Rate (DR) and False Positive Rate (FPR). Malicious patterns in the benign applications' system call traces imply false positives.

Ninety benign applications were picked based on the popularity among the survey participants. Only applications at least used by two participants or more were analysed and formed in total 90 applications. A table of all the applications and their corresponding categorised user profile is available in Appendix

[1]`http://appshopper.com/mac`

F. The 90 applications were installed in the monitored environment described in Section 5.4.2 and their system call traces were collected. Manually, the system call traces were inspected inside a text editor and was searched for the malicious detection patterns defined in Section 8.1.

App Store user profile

One of the strict rules[2] the Mac App Store dictates, is the requirement of an application to be sandboxed. This implies apart from the operating system owned processes, all applications in the Mac App Store profile are sandboxed. In this user profile it is observed the process executions `SYS_execve` and `SYS_posix_pawn` are only performed by `launchd` and `xpcproxy`. This behaviour is explained in Section 9.1. Sixty applications from the App Store were installed and their system call traces analysed. The nature of the applications was diverse, varying from unzip utilities to photo editors. The list of tested applications is available in Appendix F.

The conducted analysis of the App Store user profile shows `launchd` and `xpcproxy` as the only processes performing the execution calls and did not execute shell processes. This results in a DR of malware of 100% and a FPR of 0%.

Typical user profile

A typical user is more likely to occasionally use applications distributed outside of the Mac App Store. Applications distributed outside of the Mac App Store do not have to comply with the strict App Store rules, neither do they have to be sandboxed. Concluded from the survey, the 30 used applications distributed outside the Mac App Store from 10 users fitting the ‘typical user profile’ were tested. In the evaluation, applications that require more close interactions with the underlying operating system appeared as false positives. Examples of these processes are: Dropbox and Tresorit, both file syncing cloud services that need processes to modify default OS X Finder (equivalent of Microsoft Windows Explorer) behaviour. Google Chrome browser, due to its own sandbox security[3] behaviour, spawns many helper processes to isolate Web pages and plugin elements. Other browsers do not show this behaviour. However, none of the process executions spawns a shell process, a prominent feature of OS X malware. The FPR is 0% while the detection rate is 100%, meaning the defined malicious patterns were absent in all the tested applications.

Power user/developer profile

The power user profile describes a user using many development tools, compilers and interpreters like Python and JavaScript (NodeJS/Electron). Under the users of the held survey, 15 of them describe themselves as developer/power user. The top 90 most used applications under these 15 developers were analysed. As expected, the interpreters and compilers for the scripting languages in particular showed executions of shell processes. The shell processes in particular call binaries related to the scripting language (i.e.: `/usr/bin/python`).

Git, a widely used source code version control utility, also performs many execution calls to its own binaries through the use of shell processes. Xcode, Apple’s IDE, also performs execution calls to git binaries.

OpenVPN (VPN software) performs execution calls to OS X system binaries (i.e. `/sbin/route` and `/sbin/ifconfig`). GPGTools (GPG encryption toolset) also performs execution calls to its own binaries.

A feature all the false positive-triggering applications have in common is their cross-platform compatibility. Python, R, git, OpenVPN etc. all consist largely of binaries that are available cross-platform, meaning they have to be functional on a variety of (Unix based) platforms. As expected earlier in Section 8.3, a shell process creates a generic method to interface and interact with these binaries, since the shell is a powerful component available on all Unix based systems. The next chapter provides reasons for this behaviour. Of the 90 analysed applications, 18 applications (20%) categorised as developer tool resulted in a false positive. The FPR under power users thus increases to roughly 20% based

[2] https://developer.apple.com/app-store/review/guidelines/mac/

[3] https://tools.google.com/dlpage/res/chrome/en-GB/more/security.html

Table 8.2: Detection Rate (DR) and False Positive Rate (FPR) of the detection patterns per user profile.

Profile	DR	FPR (Shell execution)	FPR (Execute call)
App Store user	100%	0%	0%
Typical user	100%	0%	25%
Developer/Power user	100%	20%	–

on the tested developer tools derived from the survey. It should be noted however that this number is solely an indication and is completely dependent on the amount of developer tools and App Store applications used by a particular user.

Table 8.2 shows the detection rates (DR) and false positive rates (FPR) for both shell executions and solely execution calls for every type of user profile. Clearly, the malware detection is most effective on the *App Store* and *Typical* user profile.

8.4. Conclusions

This chapter described the process of evaluating the effectiveness of the defined malicious detection rules in Chapter 7. Similar to the literature studied in Chapter 4, the Detecting Rate (DR) and False Positive Rate (FPR) were used to define the effectiveness of the detection rules. From all the malware samples of which the system call traces were obtained, the system call traces were searched for occurrences of malicious patterns. Every malware sample analysed (shown in Table 8.1) was detected by malware pattern 2, which resulted in a DR of 100%. Every malware sample would be detected when detection rule 2 was implemented into an Intrusion Detection System (IDS). Determining the FPR is much more difficult. The FPR is largely dependent on the type of user: some users only use App Store applications, other users mainly use software developer tools. Three different user profiles were defined, “App Store user”, “Typical user” and “Developer/Power user”. A survey among 25 real Mac users was conducted and obtained was their own opinion to which user profile they belong and a list of applications installed on their system. Almost all the obtained applications were used in the FPR, of which the App Store and Typical user resulted in a FPR of 0% and the Developer user profile in an estimation of 20%. This is an estimation, because this percentage is completely dependent on the amount of developer tools and App Store applications used by a particular user. The results are shown in Table 8.2. A majority of the false positive generating applications are applications featuring a cross-platform nature, typically originating from a Linux environment, ported to OS X. Such applications use a shell process to interact with their underlying cross-platform binaries.

9 Discussion

Based on the results observed in Chapter 8 it can be concluded that for several user profiles the detection – which in this case also implies prevention – of malware on OS X is effective. However, the observations also raise important questions. Why is this behaviour observed and why is it never before published in research? How difficult is it for an attacker to 'bypass' the detection rules and what can we expect from more advanced OS X malware in the future? This chapter aims to provide an answer to these questions. It starts by explaining the important sandboxing components OS X and shell processes in general. Subsequently, the questions above are answered.

9.1. XPC services

As described in Section 8.3.2, a phenomenon observed when analysing the Mac App Store applications is the execution of execute calls by the two OS X processes `launchd` and `xpcproxy`. In OS X's sandboxing technology, `launchd` and `xpcproxy`[1] (XPC Services) are responsible for process executions and interprocess communication. `launchd` forms the equivalent of `init` on Linux [28], the first userspace process that is started by the kernel and is responsible for the execution of other processes in userspace. Many of the operating system specific processes initiated by the kernel or the user, are executed by `launchd` [28].

XPC services[2] provide privilege separation and interprocess communication (IPC) for sandboxed applications. XPC services are managed by `launchd` which launches on demand of other processes permitted to make use of the XPC service. By default, XPC services are run in the most restricted environment possible – sandboxed with minimal filesystem access, network access, and so on. Elevating a service's privileges to root is not supported. Furthermore, an XPC service is private and is available only to the main application that contains it. `xpcproxy` is a service functioning as an execution trampoline that configures the environment for an XPC service's execution[2]. This means that `xpcprocy` is responsible for the execution of sandboxed processes. The execution of sandboxed processes explains the execution calls performed by `xpcproxy` for App Store applications.

Figure 9.1 illustrates XPC services process execution request scheme.

Apple also notes that *"use of `NSTask` and `posix_spawn`, do not let you put each part of the application in its own sandbox, so it is not possible to use them to implement privilege separation"*[3]. `posix_spawn` is by now well-known and `NSTask()` appears to be a wrapper for `execve()` [28], exactly the calls that allow malware to perform its behaviour.

[1]`https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man8/xpcproxy.8.html`
[2]`https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man8/xpcproxy.8.html`
[3]`https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingXPCServices.html`

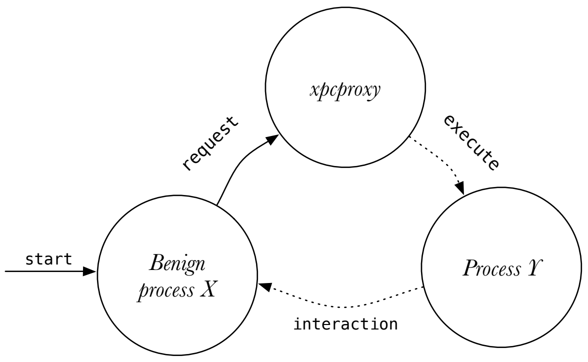


Figure 9.1: Benign (sandboxed) process X requests XPC services to interact with process Y. XPC services decides whether to allow or deny this request.

9.2. Shell processes

The Unix shell has been around longer than most of its users have been alive [45]. It has survived so long because it is a power tool that allows user to perform complex tasks. The shell is an interpreter of commands to interact with processes and the file system. Many of the available shells come with their own scripting language that allows users to automate shell tasks and ‘glue’ components together [45]. The shell’s main functionality is abstracting the executing binaries by the kernel and providing a standardised method to interact with the processes.

In the hacker community, shells are often seen as the “holy grail” after a successful exploitation of a process [12]. A majority of the shellcode (135 as of writing) in the Exploit Database[4] aims to execute the shell process (`/bin/sh`). Shellcode is typically executed after an attacker has successfully exploited a vulnerability in a process to further perform malicious interactions with the underlying system , due to its broad powers and flexibility [12].

9.2.1. Shell process execution

Shell processes are supported in a majority of the programming languages and so do Swift and Objective-C, OS X primary programming languages. Some tasks a programmer is trying to achieve may not be possible using library functions and API’s provided by either the language itself or external libraries. In these cases, the language provides an API to interact with system components (in this case the shell) that allows for less restrictions and more flexibility. Listing 9.1 and 9.2 show a typical code snippet in Apple’s programming language Swift and the C programming language respectively used to interact with a shell process.

```swift
shell("ls -ail")

func shell(args: String...) -> Int32 {
    let task = NSTask()
    task.launchPath = "/bin/sh"
    task.arguments = args
    task.launch()
    task.waitUntilExit()
    return task.terminationStatus
}
```

Listing 9.1: Hooking a system call function

The code sample in Listing 9.1 uses `NSTask()`, a standard Objective-C and Swift API call. `NSTask()` can run another program as a subprocess and can monitor that program’s execution. An NSTask object creates a separate executable entity that does not share memory space with the process that creates

[4] `https://www.exploit-db.com/`

it[5]. `NSTask()` uses the `posix_spawn` system call to execute a process[6].

A shell process by default uses the `execv` C library function (wrapper for system call `SYS_execve` to execute commands[7].

```
int main () {
    char command[50];
    strcpy(command, "ls -ali");
    system(command);
    return 0;
}
```

Listing 9.2: Hooking a system call function

In C, the API call `system()` can be used to pass commands to a shell process. On OS X, `system()` is implemented in the LibC library and uses the `posix_spawn` system call[8].

Listing 9.3 and 9.4 show the system call trace of samples 9.1 and 9.2 respectively. The exact same pattern as in the malware samples can be observed. Both shell executing processes use the `SYS_posix_spawn` system call to execute a shell process (see `/bin/sh` in line 2). The shell process then uses the `SYS_execve` to execute the command.

```
0:20:38,66317; ExecuteShell_Swift; 701; 1; SYS_posix_spawn; 0;
0:20:38,68877; /bin/sh; 701; 1; NEW_PROCESS; 0;
0:23:18,8105; sh; 777; 776; SYS_execve; 1;
0:23:18,42350; /bin/ls; 777; 776; NEW_PROCESS; 1;
```

Listing 9.3: Swift program "ExecuteShell_Swift" executes binary `ls`.

```
0:23:17,487526; ExecuteShell_C; 776; 657; SYS_posix_spawn; 1;
0:23:17,506033; /bin/sh; 776; 657; NEW_PROCESS; 1;
```

Listing 9.4: C program "ExecuteShell_C" executes sh.

9.2.2. Static detection of shell usage in malware

Currently, the behaviour analysis process conducted in this research to extract system call usage involves some manual operations. The malware has to be copied into the virtual machine, the kernel extension has to be loaded and the malware sample has to be executed. After the malware sample has finished executing the virtual machine has to be rolled back to it initial state and the system call traces log created on the host has to be analysed for malicious patterns.

Even though this process may be more automated, it would be interesting to know if the patterns can be found using static analysis of the binary sample. Many of the infrastructure is already in place at AV companies and as such, it would not involve the setup of an isolated behavioural detection environment as set-up in this research.

To investigate in the static detection possibility, all the malware samples analysed in this research were also statically analysed using the Hopper[9] disassembler. Consulting Maarten Boone (reverse engineer and malware analyst at Fox-IT) led to the search for shell paths (e.g. `/bin/sh` and `/bin/bash`) in the __TEXT segment of the binary (also shown in Figure 5.1. In this __TEXT segment, the strings used in the binary are stored. As shown in the previous section, a shell execution consists of an `execve()` or `posix_spawn()` call where one argument is a string containing the path of the executable to be launched. This string is included in the __TEXT segment of the binary. Searching for the system call execution in the binary is more difficult, but not impossible. Knowledge about the instructions used in the system call prologue is needed in order to find the `SYS_execve` and `SYS_posix_spawn` system call

[5]https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSTask_Class/
[6]https://github.com/apple/swift-corelibs-foundation/blob/master/Foundation/NSTask.swift
[7]https://opensource.apple.com/source/bash/bash-29/bash/execute_cmd.c
[8]https://opensource.apple.com/source/Libc/Libc-763.11/stdlib/system-fbsd.c
[9]http://www.hopperapp.com/

executions. In several malware samples the `/bin/sh` and `/bin/bash` paths are indeed included in the __TEXT segment.

However, in many cases the malware samples were packed using packers. Packers obfuscate the binary in order to make it harder for reverse engineers and malware analysts to analyse the binary sample. Many of the samples analysed used the infamous UPX[10] packer to obfuscate the binary sample. As such, it is not possible to find plain strings in the __TEXT segment anymore. Table 9.1 shows obfuscated and non-obfuscated samples.

no.	Name	Packer	Shell strings
1	OSX.Flashback	?	?
2	OSX.Crisis.I	Custom	No
3	OSX.FakeCodec	Custom	No
4	OSX.LaoShu.A	?	?
5	OSX.CoinThief.A	No	Yes
6	OSX.Xslcmd	No	Yes
7	OSX.Wirelurker	No	Yes
8	OSX.Janicab	No	Yes
9	OSX.iWorm	UPX	No
10	OSX.Kitmos.A	No	Yes
11	OSX.Genieo!gen1	No	Yes
12	OSX.Malcol	?	?
13	OSX.Downloader	No	Yes
14	OSX.Jahlav.A	Custom	No
15	OSX.InstallCore	Custom	No
16	OSX.EliteKeylogger	No	Yes
17	OSX.OceanLotus	XOR[11]	No
18	OSX.Crisis.II	Custom	No
19	OSX.KeRanger.A	No	Yes
20	OSX.Pirrit	?	?
21	OSX.Bundlore	UPX	No

Table 9.1: Lists all occurrences of packers in analysed malware samples and visible shell paths in the .TEXT segment. A question mark indicates obfuscation type could not be determined.

9.3. Literature: never before has this behaviour been spotted

Why is this research the first to spot a rather prominent behavioural feature of malware? This is a question to which I can only guess. First and foremost, few OS X malware has been statically analysed by the research community, let alone behavioural analysis was performed of this OS specific malware. I assume it is known malware samples use shells to perform malicious tasks on a system, since in some static analyses also referred to in this thesis, the characterising shell strings pop up in the disassembly. However, what is not known, is that *all* OS X malware samples perform executions to shell processes. In addition, I assume that it is underestimated how few benign processes make use of shell processes. This research has shown that only applications with a cross-platform compatibility requirement rely on shell processes to connect to its core (cross-platform) components.

9.4. Bypasses and alternatives

Comprehending the possible ways of malware to go is difficult and remains the main reason why malware continues to be a major threat. If the possibility for malware to use shell processes would be subducted, none of the tested samples would work anymore. Removing the ability for malware to

[10] `http://upx.sourceforge.net`

spawn shell processes would pose a serious obstruction for malware to perform its malicious tasks. Malware on OS X would have to “step up its game” to still be able to perform malicious behaviour.

The Sandbox API’s provided by Apple’s OS X development framework, as explained in Section 9.1, are restrictive and limiting to ensure security. Apple’s “App Sandbox Design Guide”[12] explains the very few system resources available to sandboxed processes. `root` privileges are absent for sandboxed processes, which form additional restrictions for (sandboxed) malware. The intentions of the restrictions and limitations of sandboxed processes are mainly to improve the security of the system[2]. Current malware as observed in this thesis is hopeless in a sandboxed environment.

However, processes on OS X are not required to be sandboxed, yet. OS X provides a rich set of API’s to “un-sandboxed” processes, which can be used to mimic some of the shell functionality. The next section addresses into more detail which types of functionality malware typically needs to perform malicious tasks and which functionality is still available to malware using API’s, in case a shell process cannot be used.

9.4.1. Malware without a shell, using purely API calls

Section 3.1 described a variety of malware samples with very different malicious functionality. Based on the functionality analysis performed by malware, described in Section 3.1, the following basic (malicious) operations can be extracted:

1. **Process execution**
   As seen throughout this thesis, malware specifically performs many process executions. Malware contains many different processes (as shown in Chapter 7), each performing its own functionality.
2. **File system manipulations**
   Malware typically stores files into specific file locations on the file system.
3. **Remote shell**
   Backdoor malware provides a remote shell to the attacker to access victim machine.
4. **Collecting system information**
   Spyware often gathers system, hardware and application information.
5. **LaunchDaemons and cronjob**
   Malware obtaining persistency on a system mainly uses LaunchDaemons or cronjobs to be automatically or periodically started, without user interaction.
6. **Modifying file attributes**
   Many malware samples change the attributes of the files they store onto the system, ensuring the files and binary own the proper rights to perform their malicious tasks. This is typically achieved by setting file attributes ‘owner’ and ‘group’, indicating the privileges of the file or binary.
7. **Loading a kernel extension**
   Some malware samples consist of a kernel extension. This kernel extension has to be loaded into the kernel in order to perform tasks.

As explained in Chapters 6, 7 and 8, every OS X malware sample obtained for this thesis relied on a shell process to achieve the asks described above. In the listing below, the feasibility of using standard OS X API’s provided by Apple to achieve such a malicious task, is evaluated.

1. **Process execution**
   On demand process execution is not possible using purely API’s. `NSTask()` is typically used to execute a process, but as explained in Section 9.2, NSTask relies on a shell process. XPC services could be used, but the process would be sandboxed[2]. Processes started by the operating system (started by `launchd`) are executed with root privileges[13]. This would be possible by creating a LaunchDaemon, but would only allow the process to start on system start.

[12] `https://developer.apple.com/library/mac/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html`

[13] `https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Articles/AccessControl.html`

2. **File system manipulations**
   The OS X `NSFileManager`[14] API's allows for file system manipulations. However, no method was found to achieve modifications in system directories. Privileged processes may be able to achieve privileged file system operations using an OS X framework named "Authorization Services"[15]. However, important API's in this framework required to execute with system privileges, are deprecated and will likely be removed by Apple. Another method to perform privileged file system modifications is using `SMJobBless`[16].

3. **Remote shell**
   A remote shell process is not possible as the execution of a shell process would be detected by the defined rules in section .

4. **Collecting system information**
   Collecting information about the current process is possible using the `NSProcessInfo` class. Various API's are available to obtain unique ID's for devices, an example is the `NSUUID` class[17].

5. **LaunchDaemons and cronjob**
   Creating LaunchDaemons is possible by storing a proper plist file into the LaunchDaemons system directory. On startup, OS X will pick up the plist file and launch the corresponding binary[18]. The plist file can be stored using `NSFileManager`, as described above. There is no API available to set cronjobs[18].

6. **Modifying file attributes**
   Modifying file attributes is possible using the NSFileManager API[14].

7. **Loading a kernel extension**
   There is no API available to load a kernel extension using kextload. However, using `NSFileManager` (only as a privileged process), the kernel extension binary can be stored into the kernel extension directory and automatically be launched on system start.

It is shown many of the functionality malware currently uses a shell process for, is also available using the standard OS X API's. However, the pure API variant is more constrained, especially in the execution of processes, where it is dependent on startup of the system. When the malware has obtained root privileges, it can achieve virtually unrestrictive file system modifications. Arguably, one could question the effectiveness of *any* protection mechanism when malware has obtained root privileges, since the highest possible rights allow for direct attacks by the malware against the protection mechanisms themselves.

While it is still possible to perform malicious behaviour on a system when a shell process is not available, malware authors will have to revise and seriously improve the current architecture of their malware. Some functionality such as a backdoor remote shell, would become impossible to achieve for malware.

9.5. Conclusions

This chapter elaborates on important concepts touched upon in this Master thesis. Understanding the possibilities and restrictions of process execution mechanisms in OS X is important in the study of blocking malware. Shown in previous chapters is the dependency of malware on execution of shell processes. Sandboxed technologies (XPC Services) in OS X prevent processes from doing harm to the underlying system by imposing serious restrictions, including the ability to execute privileged processes. App Store applications have to comply with these restrictions, which is why their FPR is 0%.

[14]https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSFileManager_Class/

[15]https://developer.apple.com/library/mac/documentation/Security/Conceptual/authorization_concepts/01introduction/introduction.html//apple_ref/doc/uid/TP30000995-CH204-TP1

[16]https://developer.apple.com/reference/servicemanagement/1431078-smjobbless

[17]https://developer.apple.com/library/mac/documentation/Foundation/Reference/NSUUID_Class/

[18]https://developer.apple.com/library/prerelease/content/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/ScheduledJobs.html

Shells processes are extremely powerful due to their flexibility and broad interactivity with the underlying system. Almost every programming language supports usage of shells processes. As described in Section 3.1.7, a majority of the malware on OS X is still fairly unsophisticated and unprofessional. The extraordinary use of shell scripts confirm this statement.

When the binary of a malware sample is not obfuscated, static analysis of the binary (using reverse engineering) indicates the usage of a shell process, based on the strings used in the binary. This technique can be used when gathering system call traces from a malware sample is not possible.

This chapter explains malware in its current form is catastrophically crippled to the extend it is not able to function anymore when a shell process is not available. Some of the functionalities provided by shell processes can be substituted using the standard OS X API's and libraries. However, this poses restrictions and forces malware authors to seriously revise their malware architecture.

10 Conclusion

This final chapter finalises this Master thesis by answering the research questions stated in Section 2.3.1 and presents the results of the conducted research. Subsequently, the potential future work continuing on this research is presented.

10.1. Reflection of the research process

This research started by addressing the emerging malware threats on Macs. An ongoing growth of popularity of Apple's Macintosh computers is observed and malware creators benefit from the success of the Mac. Over the last few years, more Mac specific malware is observed and due to the many obfuscation techniques of malware samples, traditional anti-virus systems (AV) more often result to be ineffective. To improve the detection rates and overcome the obfuscation techniques used by malware, research now focusses on behavioural detection of malware. This Master thesis follows the trend by defining the following research question:

Is it possible to detect malicious behaviour performed by malware, based on monitoring system calls?

This work shows the answer to that research question is: **Yes.**

System calls are low level requests for functionality and services from the operating system, performed by processes running on the operating system. Many of these requests are performed by processes and it is widely assumed monitoring the system call traces of processes reveals process behaviour.

In order to create system call traces of all processes on the system, a kernel module was developed. This kernel module is required to bypass kernel security restrictions enforced by Apple in order to achieve hooking of a core component of the kernel; system calls. While this kernel module is loaded into the kernel, system call traces are constructed by logging system call invocation. In the subsequent phases of this research, the system call traces of both malware and benign processes are analysed in order to find anomalies and system call patterns defining malicious behaviour. Over 20 different malware samples and 90 different benign Mac applications were analysed. Chapters 6 and 7 explain the heat map and manual sequential analysis of the system call traces, respectively. The heat map analysis visualised system calls very rarely used by showing prominent colours in the heat map. The manual sequential analysis involved a thorough investigation for the recurring patterns and defining the malicious behaviour in terms of system calls. Subsequently, Chapter 8 described the extracted patterns and anomalies from the malicious datasets. These patterns can be used to detect malware.

10.2. Results

This research shows that system call traces are viable research material in deriving new methods to detect and prevent malware. The system call traces collected after monitoring processes form a viable perspective to describe behaviour of processes, including malicious processes. This research also shows that heat map visualisations of the system call traces form an effective analysis technique to gain insights in the behaviour of processes and form a technique that can be recurrently used to extract more patterns from the system call traces.

The research question of this Master thesis can be answered positively and motivated by multiple patterns that define malicious behaviour on OS X, extracted from the system call traces:

1. **Execute system call usage by non-OS X process (pattern 1)**
   A process other than `xpcproxy` or `launchd` performing a `SYS_posix_spawn` or `SYS_execve` system calls. Pattern 2 refines this pattern to a more precise malicious pattern.
2. **Execution of shell processes (pattern 2)**
   A `SYS_execve` or `SYS_posix_spawn` system call executing a shell process (i.e.: `/bin/bash`, `/bin/sh`, `/bin/python`), is a generic malicious pattern. This pattern is used to achieve a 100% detection rate with a 0% — 20% False Positive Rate.
3. **Interaction with the OS X launchctl process (pattern 3)**
   A shell process (i.e.: `/bin/bash`, `/bin/sh`, `/bin/python`) launching the `/bin/launchctl` or `/usr/bin/crontab` binary, is a malicious pattern to gain persistency on the system using Launch-Daemons or cronjobs.
4. **Ransomware performing write calls touching many files (pattern 4)**
   A process that is performing `SYS_write` calls in combination with `SYS_read` calls touching many files on the file system in a very short period of time, is considered a malicious pattern shown by ransomware encrypting files on the file system.

Pattern 2 is used to achieve a detection rate of 100% for *all* malware currently known on OS X systems. In order to accurately evaluate the False Positive Rate (FPR) of the detection patterns in real circumstances, three different user profiles were defined varying from Mac App Store application users to a developer user profile. Only in case of the developer user profile, the FPR increases to roughly 20%. In the other user profiles, the FPR is 0% or very close to 0%. The FPR is dependent on the type of applications used. Typical false positives are generated by applications with a cross-platform compatibility nature. These type of applications use shell processes to interact with cross platform binaries/processes. Examples of these applications are: MATLAB, R, IDE's and LaTeX compilers.

This research has shown the dependency of OS X malware on shell processes. Shell processes create a universal and generic method to interact with the underlying system. If the ability for processes to use a shell processes is subverted, current malware would have revise its architecture and become much more advanced. The implications of the absence of shell processes are discussed in Chapter 9.

10.3. Future research

The heat map and sequential analysis techniques used in this research resulted in extraction of simple, but very powerful detection patterns for malware on OS X. Obviously, sufficient knowledge regarding the OS X system internals is required in order to perform an effective analysis using the techniques in this research. In addition, it is observed that malware for OS X is not yet as advanced as some Microsoft Windows malware families, also noted by Patrick Wardle [54].

This research and results show that the techniques used are durable and reusable to extract other patterns from system call traces. As shown above and in Chapter 8, multiple independent malicious patterns were extracted using the same analysis technique. More of these patterns may be extracted by analysing the system call traces of malware. In addition to the feature set used in the dataset of this research, system call function arguments may be of value in successive system call research. This research focussed in particular on Apple's OS X operating system, but similar observations may be present in system call traces on Microsoft Windows systems. I believe other elementary detection patterns — such as those provided in this research — may be derived using machine learning methods on system call traces data set. In a passive fashion, the machine learning algorithm may be trained on malware system call traces to construct simple patterns which are then used in the implementation of an anti-virus solution. This method of extracting detection rules for malware would increase the detection rate of malware, while significantly limiting performance overhead posed by the AV solution.

This research primary focussed on the infection phase of malware. In this stage, the traces of malware appeared to be most prominent and prevention of this phase provides the most effective protection against malware infection. System call analysis of the successive phase of malware may provide other insights in malicious behaviour.

To further evaluate the possibility of evading the detection rules presented in this thesis, a shell independent malware sample should be constructed — if possible — and its system call trace should be analysed. New valuable malicious patterns, which can result in malicious behaviour detection rules, may be derived from such research.

A
CNS IEEE paper

The results of this Master thesis have been composed in a paper *“Behavioural detection and prevention of malware on Mac OS X”* which was submitted to the 2016 IEEE CNS conference in Philadelphia (`http://cns2016.ieee-cns.org/`) on 27th of April 2016. The submitted paper has been added to this thesis document and starts on the next page.

Behavioural detection and prevention of malware on Mac OS X

Vincent Van Mieghem
Delft University of Technology
The Netherlands

Christian Doerr
Delft University of Technology
The Netherlands

Sicco Verwer
Delft University of Technology
The Netherlands

Abstract—**Malware on Apple's Mac OS X systems emerges to be an increasing security threat that is currently solely countered with ancient anti-virus (AV) technologies. Current AV technologies pose a performance overhead on the entire system and are inherent with a delayed effectiveness, due to their signature based detection. This paper presents a novel, generic, behavioural detection and prevention mechanism for malware on Mac OS X based on system calls. A large amount of system call traces is analysed from which certain malicious system call patterns are defined. These patterns are based on execution system calls, executing Unix shell processes. Three types of user profiles are defined to evaluate the detection patterns, resulting in a 100% detection rate and a 0 to 20% False Positive Rate, depending on the type of user profile.**

I. Introduction

Over the last three years an increasing growth of malware targeting Mac OS X systems is observed. Five times more OS X malware appeared in 2015 than during the previous five years combined [1]. Many types of malware previously only appearing on Microsoft Windows systems are now also emerging on Mac OS X system. Serious threats like rootkits designed to exfiltrate valuable information from systems or malware that encrypt personal documents that can only be decrypted in exchange for bitcoins are not absent on Macs anymore. Current anti-virus technologies still heavily rely on binary signature checking, a detection technique that often lacks behind [2] [3]. Nowadays, many binary obfuscation and signature modification techniques are used by malware to evade AV-detection. A need for more advanced malware detection methods arises.

This paper presents a novel malware detection method able to prevent infection of every malware sample we currently found on Mac OS X systems without preceding knowledge regarding the malware samples. System calls are used to define and detect malicious behaviour of malware processes on Mac OS X. In addition, the proposed techniques are performance efficient and not based on any computationally intensive machine learning algorithms. System calls are requests for specific functionality from applications to an operating system. This paper shows that monitoring system call usage of applications and processes on a system reveals the application's behaviour which can be used to identify malicious processes. Monitoring a large amount of benign and malicious processes, a clear recurring pattern of system calls can be extracted from the malicious processes that are absent in system call traces of benign processes on Mac OS X. Heat map visualisations and sequential analysis of the system call traces were used to obtain the insights required to construct the malicious patterns. Several of these malicious patterns are provided and explained in this paper.

We describe the structure of the acquired system call traces and the utilities constructed in section III. The process of collecting the system calls traces from malware samples is described in section IV. Subsequently, the analysis and results are explained in section V. Finally, the results are evaluated and discussed in sections VI and VII respectively.

II. Related work

A majority of the research in detecting malware focuses on static analysis. Static analysis of malware is a technique in which the machine code contained in the malware binary file is interpreted to understand actions that are supposed to be performed by the binary file. Typically, the disassembly of the malware binary is used to obtain an understanding of its intended functionality.

On the other end of the malware analysis spectrum, dynamic analysis of malware aims to interpret functionality and behaviour by running the malware sample on a particular system and analysing the systems resources used by the malware sample. Behavioural analysis is primarily concerned with deriving behavioural identifying features from the malware. The advantage of this technique is that binary obfuscation techniques that are applied by malware creators to hinder analysis by malware analysts are not hindering dynamic analysis, because functionality is derived from actions performed by malware on the system [3].

In 1996, Forrest et al. [4] introduced an intrusion detection method based on monitoring the system calls used by active, privileged processes. This work shows that a program's normal behaviour can be characterised by local patterns in its traces, and deviations from these patterns could be used to identify security violations of an executing process. Others tried to improve this work by using machine learning algorithms, however, these improvements came with a computational cost and were not able to perform real-time detection of malware. [5]

Niels Provos [6] introduced a tool named Systrace, which aims to improve the host security by enforcing system call policies based on interposing system calls. Systrace monitors the direct system call usage to detect and prevent processes from violating policies. Provos points out that, although powerful, policy enforcement at the system call level has inherent limitations regarding the interpretation of an application's internal state. Kurchuk et al. [7] improve upon Provos' Systrace by proposing two extensions; nested policies and dynamic policy generation.

Kang et al. [8] Medhi et al. [9] and Xiao et al. [10], propose a malware detection method based on system calls using established machine learning algorithms. System calls are represented by a 'bag' data structure where elements are integer frequencies of system call occurrences. Medhi uses multidimensional n-grams to perform 'in-execution' detection on Linux systems. The proposed approaches in terms of detection rate and false positive rate are promising, a 85% to 95% detection rate is achieved.

Sun et al. [11] use dynamic monitoring of Windows API calls to detect worms and exploits. However, their approach is limited to detection of worms and exploits that use hard-coded addresses of API calls, which is not the case when Address Space Layout Randomization (ASLR) is activated on the operating system. Nair et al. [12] determine the frequency of critical API calls by programs and use this information to construct a signature of a program. Nair et al. introduce their own classification algorithm and show a 80% detection rate for Windows malware.

Dehnert [13] argues that an IDS running on the host operating system itself is vulnerable against a direct attack and proposes an intrusion detection method based on a hypervisor. Dehnert uses system call usage and their arguments to detect malware. The detection methods implemented are manually defined patterns and system call patterns using 'Sequence time-delay embedding' (stide), a time sequence machine learning algorithm. The detection however, due to the absence of any refinement, performs very poorly and operates with a performance overhead of almost 300%.

Canzanese et al. [14] propose another machine learning approach for Microsoft Window systems using logistic regression trained by the Stochastic gradient descent of system call 3-grams. Their detection method achieves a TPR of 92%.

The majority of the research focusses on detection algorithms based on established machine learning algorithms. In many cases the detection algorithms are computationally intensive and not real-time. In addition, the detection rate allows for improvement. This research uses the very similar fundamentals – system call traces – but does not use machine learning as a detection technique.

III. Monitoring system call usage

One perspective to separate the functionality provided by an operating system is a division in user space and kernel space. Applications that run in user space are less privileged in terms of permissions than kernel space processes and are controlled by the kernel. This separation mainly facilitates security of an operating system. A common way for processes in user space to interact with the kernel is by means of system calls. System calls are used to request specific functionality from the operating system. Examples of typical system calls are `SYS_open` to obtain a file handle of a file on the file system and `SYS_getsocket` to obtain a handle to a socket to connect to the Internet. The implementation of calls resides in the XNU kernel of a Mac OS X system. The latest XNU kernel features 489 system calls which are defined in kernel owned memory named `sysent`. To monitor the system call usage of processes, system call implementations have to be modified to log themselves upon invocation. To accomplish system call logging, system call implementations are intervened to enable the logging functionality. The implementations of system calls in the `sysent`-table are replaced with our own implementations. This technique

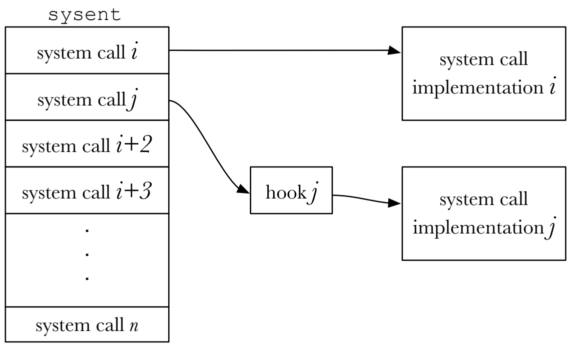


Fig. 1. High level `sysent`-table representation in which all the system calls are defined. System call i represents the normal system call wiring to its implementation. System call j represents a hooked system call.

TABLE I. METADATA OF A SYSTEM CALL THAT IS CAPTURED.

Name	Description
Uptime	Uptime of the system, in microseconds precision, to keep track of system call order.
Process ID	Together with the parent process ID, this provides the ability to perform traces in the dataset.
Parent process ID	Together with the process ID, this provides the ability to perform traces in the dataset.
Privileges	Does the calling process have superuser privileges.
Process name	Name of the process
Process execution path	File system path of the process
System call name	Name of the system call
SYS_write location	Location of the where the write system call writes to.

is called "hooking"[1]. Figure 1 shows the rewiring of a system call implementation after the system call is "hooked". The `sysent` entry points to the hooked implementation system call instead of the original system call implementation. System call i represents a normal system call wiring, where system call j represents a hooked system call, in which the j-hook block logs system call j and eventually returns back to the original j system call implementation.

We developed a kernel module that implements hooking of system calls. The hook logs the system call invoked in the exact occurring order to a log file and subsequently calls the original system call implementation to continue normal functionality as shown in figure 1. Table I represents the metadata of a system call that is logged to the log file.

[1]https://en.wikipedia.org/wiki/Hooking

Listing 1. Process pboard executes `/bin/sh` process and sh starts calling system calls (OSX.OceanLotus.A).

```
time          ; process name; pid; ppid; syscall              ;
      is_root;
0:5:6,448659; pboard      ; 474; 1   ; SYS_pipe             ; 0;
0:5:6,448689; pboard      ; 474; 1   ; SYS_posix_spawn      ; 0;
0:5:6,450010; /bin/sh     ; 474; 1   ; NEW_PROCESS          ; 0;
0:5:6,450205; sh          ; 476; 474 ; SYS_shared_region_chk; 0;
```

Subsequently, while the kernel module was loaded, system calls called by processes log their invocation to a log file resulting in a large amount of system call traces in the exact order in which the system calls were invoked. The log file represents our 'raw dataset'. Listing 1 shows several records of the raw dataset. The records are sequential and ordered by time, where each record represents a single operation (system call invocation). The records are semicolon separated, representing the attributes presented in the first line (header). In listing 1 process `pboard` executes the `/bin/sh` binary, using the `SYS_posix_spawn` system call.

Not all system calls are hooked. Some system calls are called over 500 times per second in an idle system state. Examples of such calls are `SYS_read` for reading files or `SYS_setitimer` used by processes to set an interval timer. System call usage was observed on an idle system. The system calls generating over 500 log records per second were excluded from the list of hooked system calls to prevent pollution of the dataset. A list of the hooked system calls is available at our Github repository[2].

IV. COLLECTING SYSTEM CALL TRACES

Collecting system call traces is only possible at runtime of processes. Malware has to be executed while the kernel module is loaded into the kernel of the operating system. A virtual machine featuring the latest and fully patched Mac OS X 10.11.3 was used to run malware samples. After the kernel module is loaded, the malware sample is executed and monitored for 5 minutes, which was our initial estimate of the amount of time required for infection of a system by the malware samples. In the analysis phase, it appeared the infection phase is much shorter. Subsequently, the log file is captured and the virtual machine is reverted to its original state for the next malware sample to be monitored.

According to Symantec, 55 unique OS X malware samples have been found since 2010[3]. Obtaining functional malware samples is not trivial

[2]https://github.com/vivami/grey_fox

[3]https://www.symantec.com/security_response/landing/azlisting.jsp?azid=O

TABLE II. A LIST OF OS X MALWARE SAMPLES ANALYSED IN THIS RESEARCH.

no.	Name	Type	Detection date	
1	OSX.Flashback	Trojan	09/30/2011	
2	OSX.Crisis.I	Rootkit	07/25/2012	
3	OSX.FakeCodec	Adware	02/03/2013	
4	OSX.LaoShu.A	Backdoor	01/21/2014	
5	OSX.CoinThief.A	Trojan	02/26/2014	
6	OSX.Xslcmd	Trojan	09/05/2014	
7	OSX.Wirelurker	Trojan	11/06/2014	
8	OSX.Janicab	Trojan	11/26/2014	
9	OSX.iWorm	Trojan	01/05/2015	
10	OSX.Kitmos.A	Backdoor	03/04/2015	
11	OSX.Genieo!gen1	Adware	05/18/2015	
12	OSX.Malcol	Adware	05/21/2015	
13	OSX.Downloader	Adware	07/29/2015	
14	OSX.Jahlav.A	Trojan	07/29/2015	
15	OSX.InstallCore	Adware	11/16/2015	*before*
16	OSX.EliteKeylogger	Keylogger	02/15/2016	*after*
17	OSX.OceanLotus	Trojan	02/19/2016	
18	OSX.Crisis.II	Rootkit	02/26/2016	
19	OSX.KeRanger.A	Trojan	03/06/2016	
20	OSX.Pirrit	Adware	04/06/2016	
21	OSX.Bundlore	Adware	04/11/2016	

and in order to create a system call trace from a malware sample, the sample must be complete and functional. Often, only specific malicious parts of a Mac OS X application that are not executable are uploaded to malware sample collecting services such as VirusTotal. For this research, 21 functional OS X malware samples were obtained from different sources[4][5][6]. Table II provides an overview of the functional malware samples obtained and analysed in this research.

V. ANALYSIS

The analysis of the collected raw system call traces aims to provide insights in the discrepanties in system call traces between benign and malicious processes. The goal is to extract recurring patterns present in malicious system call traces, which are absent in the benign system call traces. Such patterns may then be used to identify a malicious process. The analysis is based on two simple techniques: heat maps and manual sequential analysis. A heat map, representing the number of calls per system call per process, was used to gain insight in the outlying system call usage by a process. Figure 2 shows the heat map of OSX.Wirelurker's system call trace. The system calls called and processes are listed on the x-axis and y-axis respectively. The heat map in which the data points (number of system calls) are normalised, according to their occurrence in other processes' system call trace. The normalisation ensures processes calling specific system calls that are significantly less called by other processes show darker in the heat map and imply an outlier. In the conducted manual sequential analysis, consisting of linear traversal of the system call traces, these outliers were analysed to derive specific recurring patterns.

A. Execution calls

The heat map visualisation of a dataset containing system call traces of benign processes showed `SYS_execve` and `SYS_posix_spawn` system calls only executed by two processes specific to the OS X operating system: `launchd` and `xpcproxy`. The black dotted square in figure 2 shows their usage of `SYS_posix_spawn`. In Mac OS X's sandboxing technology `launchd` and `xpcproxy`[7] (XPC Services) are responsible for process executions and interprocess communication [15]. Figure 3 illustrates XPC services process execution request scheme. On a clean system, `launchd` and `xpcproxy` are the only processes that use `SYS_execve` and `SYS_posix_spawn`.

However, the heat map visualisation of malware showed the presence of `SYS_execve` and `SYS_posix_spawn` performed by other processes than only `launchd` and `xpcproxy` (also observable in figure 2, `SYS_posix_spawn` calls by `com.apple.MailServer` and `update` are malicious calls performed by the OSX.Wirelurker malware). Malicious processes appeared to be responsible for these supplementary `SYS_execve` and `SYS_posix_spawn` calls. In addition, the execution calls appeared to execute shell processes (i.e. `/bin/sh` and `/bin/bash`) that again are responsible for many of the additional execution calls. After more in-depth manual sequential analysis of the system call traces, in which we searched for traces of recurring patterns, a clear pattern appears to be present in *all* malware samples in table II. All malicious processes at some point in their infection process use either of these calls (`SYS_execve` and `SYS_posix_spawn`) to execute a shell process, without requesting XPC services (illustrated in figure 4). We discuss the reason for this behaviour in section VII. Listing 1, 2 and 3 show a sample of the records in the system call trace in which the malicious processes use the execution calls to spawn a shell process.

Listing 2. Process 0 executes `/bin/sh` process using `SYS_posix_spawn` system call (OSX.iWorm).

```
0:1:27,174721; 0; 358; 357; SYS_posix_spawn; 0;
0:1:27,176143; /bin/sh; 358; 0; NEW_PROCESS; 1;
```

[4]https://objective-see.com/
[5]https://www.virustotal.com/
[6]https://researchcenter.paloaltonetworks.com/

[7]https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man8/xpcproxy.8.html

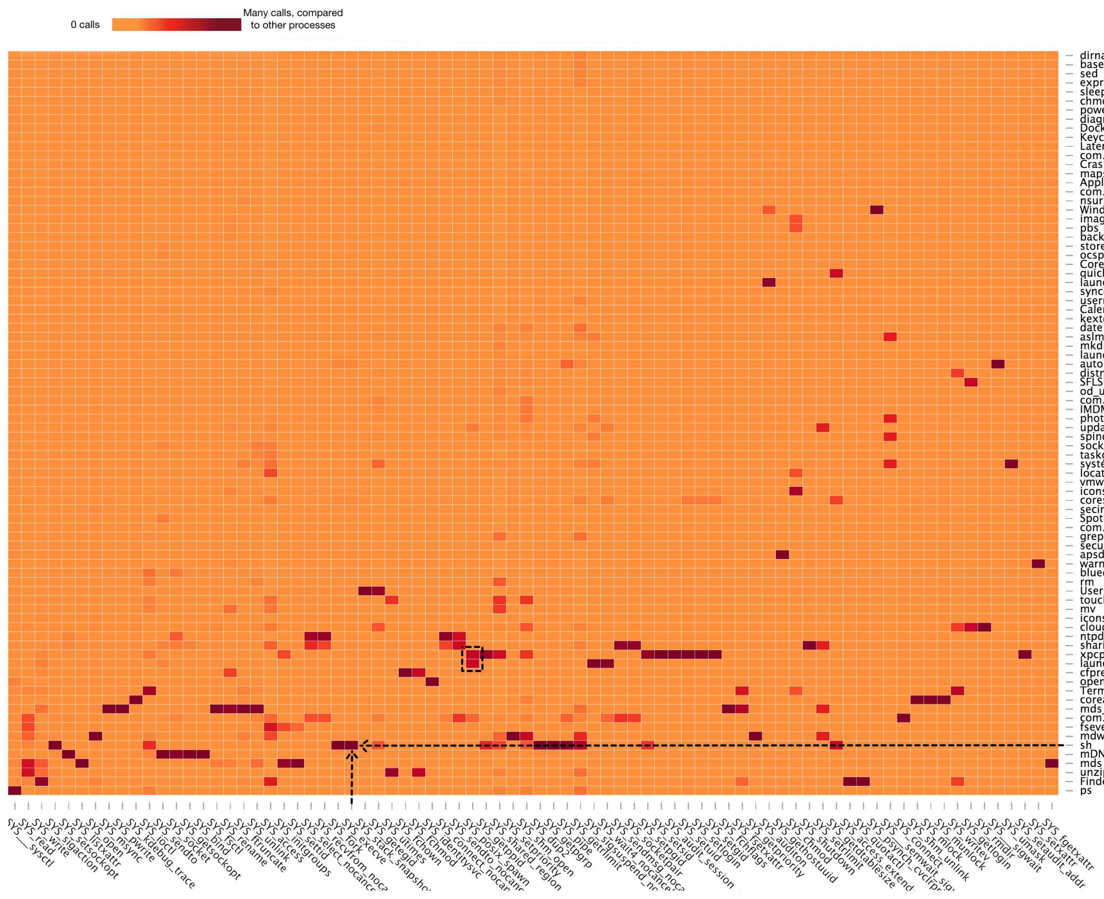


Fig. 2. Heat map visualisation of OSX.Wirelurker malicious system call trace. The system calls called and processes are listed on the x-axis and y-axis respectively. Each square represents the number of system calls, as a normalised data point. Darker blocks represent an outlying number of system calls performed by a process, relative to other processes. The black arrows indicate the `SYS_execve` calls performed by `sh`, and the black square indicates `SYS_posix_spawn` being called by `launchd` and `xpcproxy`. Note: the `SYS_posix_spawn` calls by `com.apple.MailServer` and `update` are malicious calls performed by the OSX.Wirelurker malware.

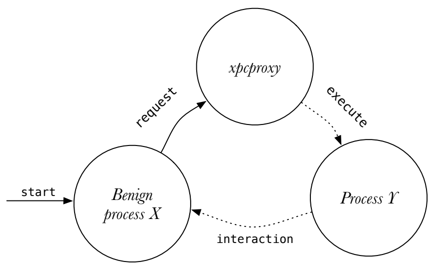


Fig. 3. Benign (sandboxed) process X requests XPC services to interact with process Y. XPC services decides whether to allow or deny this request.

Listing 3. Process Transmission executes `/bin/sh` process using `SYS_posix_spawn` system call and process `sh` executes process `kernel_service` using a `SYS_execve` system call (OSX.KeyRanger.A).

```
time; process name; pid; ppid; syscall; is_root;
0:13:58,523214; Transmission; 413; 1; SYS_posix_spawn; 0;
0:13:58,524348; /bin/sh; 413; 1; NEW_PROCESS; 0;
...
0:13:58,530059; sh; 414; 413; SYS_execve; 0;
0:13:58,530631; /Users/m/Library/kernel_service; 414; 413;
      NEW_PROCESS; 0;
```

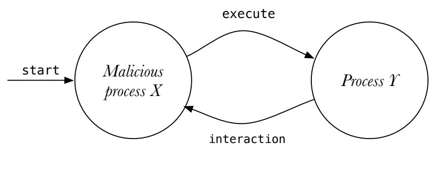


Fig. 4. Malicious process X executes process Y without XPC interaction. Typically, process Y is a shell process.

B. Persistency

Other behaviour derived from the heat map and the sequential analysis appeared to be in-

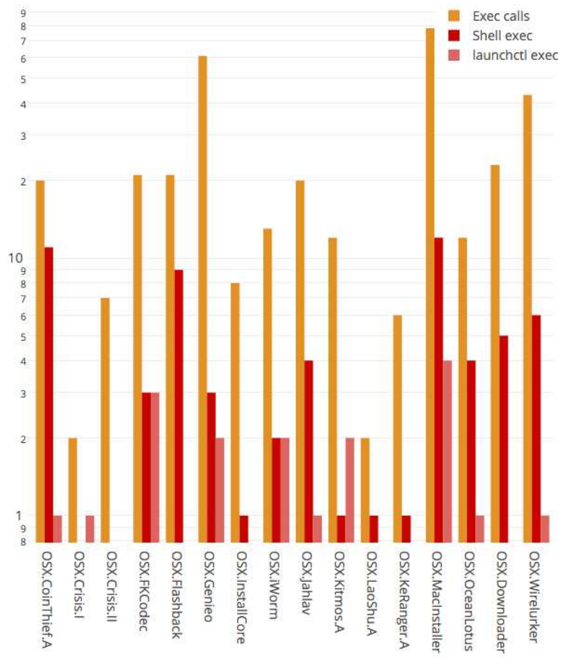


Fig. 5. Amount of execution calls, execution of shell processes and executions of `launchctl` by malicious processes. Note: y-axis has a logarithmic scale. Note: OSX.Crisis lack the execution of a shell process due to the absence of a complete and functional sample of the rootkit.

herent to malware was gaining persistency using LaunchDaemons (auto-run items on system startup). In OS X, processes are required to store a configuration file in a specific LaunchDaemon directory and notify the OS (via `launchctl`) of the config's existence in order to be persistently started by the operating system upon startup. Notifying `launchctl` is clearly visible through the execution of `launchctl` by the malicious process. Every process analysed that uses LaunchDaemons for persistency on the system executes `launchctl` as shown in listing 4, 5 and 6.

Listing 4. Shell process `0` executes process `launchctl` (OSX.iWorm).

```
0:1:48,287125; sh; 367; 363; SYS_execve; 0;
0:1:48,288733; /bin/launchctl; 367; 0; NEW_PROCESS; 1;
```

Listing 5. Shell process executes process `launchctl` (OSX.Wirelurker.A).

```
0:2:11,670527; sh; 390; 387; SYS_execve; 0;
0:2:11,671598; /bin/launchctl; 390; 0; NEW_PROCESS; 1;
```

Listing 6. Process `WaAvsmZW.EMb` executes process `launchctl`. (OSX.Crisis.A).

```
0:15:13,439772; WaAvsmZW.EMb; 1049; 1046; SYS_posix_spawn; 0;
0:15:13,439945; /bin/launchctl; 1049; 0; NEW_PROCESS; 1;
```

Some OS X malware uses cronjobs, a Unix utility used to execute a script at certain defined moments of time, to gain persistency on a system. This is visible (listing 7) based on the execution of `crontab`, a process responsible for managing cronjobs.

Listing 7. Shell process launching `crontab`. (OSX.Jahlav).

```
0:1:33,92905; sh; 388; 386; SYS_execve; 0;
0:1:33,94207; /usr/bin/crontab; 387; 0; NEW_PROCESS; 0;
```

Based on the heat map visualisation of the entire system call trace, we observed the processes that connect to the Internet. Internet connecting processes invoke many `SYS_getsockopt` and `SYS_setsockopt` operations, responsible for receiving and sending data over a socket respectively.

Figure 5 shows the number of execution system calls, execution to shell processes and executions of `launchctl` per malware sample. Prominent in the figure is the high amount of execution system calls to shell processes in all malware samples.

VI. Evaluation

The most prominent malicious pattern derived from the analysis phase to detect presence of malware on a system is the extraordinary amount of execution calls (to shells) performed by malicious processes. Initially, the pattern was observed in malware samples 1 to 15 in table II. At that time, malware samples 16 to 21 were still undiscovered by the security industry. After obtaining malware sample 16 to 21, our defined malicious pattern from the earlier malware samples was existent in system call traces of the new samples 16 to 21 as well and can thus be used to detect the newer malware samples. The horizontal line in table II indicates the moment of our malicious pattern definition.

To verify this pattern to be unique to malware, several different experiments were performed. We expected the presence of the execution patterns in benign processes also. To present an accurate False Positive Rate (FPR) evaluation, 3 different user profiles are defined. The user profiles define the type of user, hence the type of applications and processes used by the user. We define 3 user profiles:

1) *App Store user:* a user only using applications downloaded from the App Store. One of the requirements for App Store applications is the use of Mac OS X's sandbox technology.
2) *Typical user:* occasionally uses applications outside the App Store that are not sandboxed.

3) *Power user/developer:* a user who uses advanced features of the system or uses developer environments to develop software.

The user profiles allow us to more accurately define the FPR by differentiating between the type of applications triggering false positives. The evaluation consists of two phase:

1) *Collecting application usage under Mac users.* A survey under 25 real Mac user was conducted to gain insights in application usage by users fitting a profile. A user was asked to select its best suiting profile and upload their installed applications.
2) *Testing the top X applications for False Positives.* In roughly the same environment as described in section IV the top 90 applications derived from the survey were analysed for Detection Rate (DR) and False Positive Rate (FPR). Malicious patterns in the applications' system call traces imply a false positive.

The applications tested are available at our Github repository[2].

A. App Store user profile

Apart from the operating system owned processes, all processes on the system of the Mac App Store profile are sandboxed. This implies that all the process executions are performed through XPC services and `SYS_execve` and `SYS_posix_pawn` are only performed by `launchd` and `xpcproxy`. Sixty applications from the App Store were installed and their system call traces analysed[2]. The nature of the applications was diverse, varying from unzip utilities to photo editors.

In our results, `launchd` and `xpcproxy` are the only processes performing the execution calls. This results in a DR of malware of 100% and a FPR of 0%.

B. Typical user profile

A typical user is more likely to occasionally use applications distributed outside of the Mac App Store. Applications distributed outside of the Mac App Store do not have to comply with the strict App Store rules, neither do they have to be sandboxed. In this case, processes may perform execution calls, bypassing XPC services. Concluded from the survey, the 30 used applications distributed outside the Mac App Store from 10 users fitting the 'typical user profile' were tested. In our evaluation, applications that require more close interactions with the underlying operating system appeared as false positives. Examples of these processes are: Dropbox and Tresorit, both file syncing cloud services that need processes to modify default OS X Finder (equivalent of Windows Explorer) behaviour. Google Chrome browser, due to its own sandbox security behaviour, spawns many helper processes to isolate Web pages and plugin elements. Other browsers do not show this behaviour. However, none of the process executions spawns a shell process, a prominent feature of OS X malware. When filtering purely on execution calls, the FPR increases to 30% (depending on the amount of 'system nesting' applications). However, when refining the filter to the execution of a shell process, the FPR is still 0% while the detection rate remains 100%.

C. Power user/developer profile

The power user profile describes a user using many development tools, compilers and interpreters like Python and JavaScript (NodeJS/Electron). Under the users of the held survey, 15 of them describe themselves as developer/power user. The top 70 most used applications under these 15 developers were analysed[2]. As expected, the interpreters and compilers for the scripting languages in particular showed executions to shell processes. The shell processes in particular call binaries related to the scripting language (i.e.: `/usr/bin/python`).

Git, a widely used source code version control utility, also performs many execution calls to its own binaries through the use of shell processes. Xcode, Apple's IDE, also performs execution calls to git binaries.

OpenVPN (VPN software) performs execution calls to OS X system binaries (i.e. `/sbin/route` and `/sbin/ifconfig`). GPGTools (GPG encryption toolset) also performs execution calls to its own binaries.

A feature all the false positive-triggering applications have in common is their cross-platform compatibility. Python, R, git, OpenVPN etc. all consist largely of binaries that are available cross-platform, meaning they have to be functional on a variety of (Unix based) platforms. A shell process creates a generic method to interface and interact with these binaries, since the shell is a powerful component available on all Unix based systems. The FPR under power users increases to roughly 20% based on the tested developer tools derived from the survey.

TABLE III. Detection Rate (DR) and False Positive Rate (FPR) of the detection patterns per user profile.

Profile	DR	FPR (Shell)	FPR (Exec call)
App Store user	100 %	0 %	0 %
Typical user	100 %	0 %	25 %
Developer/Power user	100 %	20 %	–

Table III shows the detection rates (DR) and false positive rates (FPR) for both shell executions and solely execution calls for every type of user profile. Clearly, the malware detection is most effective on the *App Store* and *Typical* user profile.

VII. Discussion & Future work

The heat map and sequential analysis techniques used in this research resulted in extraction of very powerful detection patterns for malware on Mac OS X. Obviously, sufficient knowledge regarding the Mac OS X system internals is required in order to perform an effective analysis using the techniques in this research. In addition, it is observed that malware for Mac OS X is not yet as advanced as some Microsoft Windows malware families as Patrick Wardle also explains in [16].

Our research and results show that the techniques used are durable and reusable to extract other patterns from system call traces. As shown in section V, multiple independent malicious patterns were extracted using the same analysis technique.

The use of execution system calls and interactions with shells and auto-run services appears to be an accurate indication of malware on a system. Similar conclusions were drawn by Niels Provos in [6]. More of these patterns may be extracted by analysing the system call traces of malware. In addition to the features in the dataset of our research, system call function arguments may be of value in successive system call research. This research focussed in particular on Apple's Mac OS X operating system, but similar observations may be present in system call traces on Microsoft Windows systems. We believe other detection patterns may be derived using machine learning methods on system call traces.

Malware functionality appears to be largely dependent on shell processes, however it is difficult to grasp the level of shell dependency. Shells are an extremely powerful and effective way to interact with the operating system and is presumably therefor used in extraordinary numbers by malware. Arguably, the absence of a shell for processes may significantly reduce (malicious) interaction capabilities with the underlying system. Processes are forced to use Mac OS X API functions which sandboxes and isolates the process from the underlying system. XPC Services is an example. Such restrictions generically provide protection to a system.

This research primary focussed on the infection phase of malware. In this stage, the traces of malware appeared to be most prominent and prevention of this phase provides the most effective protection against malware infection. System call analysis of the successive phase of malware may provide other insights in malicious behaviour.

VIII. Conclusion

We have shown that malware on Mac OS X is detectable based on system call traces of malware in which fundamental dependencies of malware surface. We developed a kernel extension to construct system call traces of processes. Based on heat map visualisation and sequential analysis, specific system call patterns are identified as malicious. The detection patterns form a detection and prevention rate of 100%, where depending on the type of applications running on a user's system the FPR varies between 0 and roughly A%. We have shown that in contrary to many other malware detection research based on complex machine learning algorithms, it is possible to construct powerful malware detection patterns and efficient prevention mechanisms using simple analysis and visualisation techniques.

Acknowledgment

The authors would like to specifically thank Maarten Boone and Fox-IT for their threat intelligence and cooperation, Virus-Total for their generosity in providing a research account used to obtain malware samples, Patrick Wardle from Objective-See/Synack and Claud Xiao from Palo Alto Networks for providing malware samples. Pedro Vilaça for his shared knowledge and open-source techniques to bypass KASLR and memory write protection in the XNU kernel [17].

References

[1] Bit9 + Carbon Black, *2015: The Most Profilic Year in history for OSX malware*, 2015.

[2] Gaudesi, M. and Marcelli, A. and Sanchez, E. and Squillero, G. and Tonda, A., *Challenging Anti-virus Through Evolutionary Malware Obfuscation*, Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part II.

[3] Moser, A. and Kruegel, C. and Kirda, E., *Limits of static analysis for malware detection*, Annual Computer Security Applications Conference, ACSAC, 2007.

[4] Forrest, S. and Hofmeyr, S. and Somayaji, A. and Longstaff, T. and others, *A sense of self for unix processes*, Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on, 1996, IEEE.

[5] Jewell, B. and Beaver, J., *Host-Based Data Exfiltration Detection via System Call Sequences*, The Proceedings of the 6th International Conference on Information Warfare and Security, 2011.

[6] Provos, N., *Improving Host Security with System Call Policies.*,

[7] Kurchuk, A. and Keromytis, AD., *Recursive sandboxes: Extending systrace to empower applications*, Security and Protection in Information Processing Systems, 2004, Springer.

[8] Kang, D.K. and Fuller, D. and Honavar V., *Learning classifiers for misuse and anomaly detection using a bag of system calls representation*, Annual Information Assurance Workshop, 2005.

[9] Mehdi, B. and Ahmed F. and Khayyam S. and Farooq M., *Towards a theory of generalizing system call representation for in-execution malware detection*, International Conference on Communications, ICC, 2010.

[10] Xiao H. and Stibor T., *A supervised topic transition model for detecting malicious system call sequences*, in workshop on Knowledge discovery, modeling and simulation, 2011.

[11] Sun, HM. and Lin, YH. and Wu, MF., *API monitoring system for defeating worms and exploits in MS-Windows system*, Information Security and Privacy, 2006, Springer.

[12] Nair, VP. and Jain, H. and Golecha, YK. and Gaur, MS. and Laxmi, V., *MEDUSA: MEtamorphic malware dynamic analysis using signature from API*, Proceedings of the 3rd International Conference on Security of Information and Networks, 2010, ACM.

[13] Dehnert, AW., *Using VProbes for intrusion detection*, 2013, Massachusetts Institute of Technology.

[14] Canzanese R. and Mancoridis S. and Kam M., *System Call-Based Detection of Malicious Processes*, Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference 2015.

[15] Levin, J., *Mac OS X and iOS Internals: To the Apple's Core*, Wrox, 1st edition, 2012.

[16] Wardle, P., *Writing bad ass malware for OSX*, Blackhat Conference 2015.

[17] Vilaça, P. (a.k.a. gdbinit), *Onyx the Black Cat*, https://github.com/gdbinit/onyx-the-black-cat

B Hooked system calls

Some system calls provide services that are very frequently used by applications. Changing the working directory for a process, for example, is such a very extensively used system call. Some system calls are called more than 500 times per second. This generates an overwhelming amount of logging data, and the kernel will intervene the logging of the kernel extension, which will result in logged behaviour gabs in the dataset. To prevent gabs, some of these system calls are not hooked/logged. Table B.1 provides an overview of the calls that are not hooked.

Table B.1: Table of system calls hooked status.

No.	System call	Hooked	No.	System call	Hooked
0	SYS_syscall	NO	263	SYS_shmctl	YES
1	SYS_exit	NO	264	SYS_shmdt	YES
2	SYS_fork	NO	265	SYS_shmget	YES
3	SYS_read	YES	266	SYS_shm_open	YES
4	SYS_write	YES	267	SYS_shm_unlink	YES
5	SYS_open	YES	268	SYS_sem_open	NO
6	SYS_close	NO	269	SYS_sem_close	YES
7	SYS_wait4	NO	270	SYS_sem_unlink	YES
9	SYS_link	YES	271	SYS_sem_wait	YES
10	SYS_unlink	YES	272	SYS_sem_trywait	YES
12	SYS_chdir	NO	273	SYS_sem_post	YES
13	SYS_fchdir	NO	274	SYS_sem_getvalue	NO
14	SYS_mknod	YES	275	SYS_sem_init	YES
15	SYS_chmod	YES	276	SYS_sem_destroy	YES
16	SYS_chown	YES	277	SYS_open_extended	YES
18	SYS_getfsstat	YES	278	SYS_umask_extended	YES
20	SYS_getpid	NO	279	SYS_stat_extended	YES
23	SYS_setuid	YES	280	SYS_lstat_extended	YES
24	SYS_getuid	NO	281	SYS_fstat_extended	YES
25	SYS_geteuid	YES	282	SYS_chmod_extended	YES
26	SYS_ptrace	YES	283	SYS_fchmod_extended	YES
27	SYS_recvmsg	NO	284	SYS_access_extended	YES
28	SYS_sendmsg	NO	285	SYS_settid	YES
29	SYS_recvfrom	NO	286	SYS_gettid	NO
30	SYS_accept	NO	287	SYS_setsgroups	YES
31	SYS_getpeername	NO	288	SYS_getsgroups	YES
32	SYS_getsockname	NO	289	SYS_setwgroups	YES
33	SYS_access	YES	290	SYS_getwgroups	YES
34	SYS_chflags	YES	291	SYS_mkfifo_extended	YES
35	SYS_fchflags	YES	292	SYS_mkdir_extended	NO

36	SYS_sync	NO	293	SYS_identitysvc	YES
37	SYS_kill	NO	294	SYS_shared_region_check...	YES
39	SYS_getppid	YES	296	SYS_vm_pressure_monitor	YES
41	SYS_dup	NO	297	SYS_psynch_rw_longrdlock	YES
42	SYS_pipe	YES	298	SYS_psynch_rw_yieldwrlock	YES
43	SYS_getegid	YES	299	SYS_psynch_rw_downgrade	YES
46	SYS_sigaction	YES	300	SYS_psynch_rw_upgrade	YES
47	SYS_getgid	NO	301	SYS_psynch_mutexwait	NO
48	SYS_sigprocmask	NO	302	SYS_psynch_mutexdrop	NO
49	SYS_getlogin	YES	303	SYS_psynch_cvbroad	NO
50	SYS_setlogin	YES	304	SYS_psynch_cvsignal	NO
51	SYS_acct	YES	305	SYS_psynch_cvwait	NO
52	SYS_sigpending	YES	306	SYS_psynch_rw_rdlock	NO
53	SYS_sigaltstack	NO	307	SYS_psynch_rw_wrlock	NO
54	SYS_ioctl	YES	308	SYS_psynch_rw_unlock	NO
55	SYS_reboot	YES	309	SYS_psynch_rw_unlock2	YES
56	SYS_revoke	YES	310	SYS_getsid	YES
57	SYS_symlink	YES	311	SYS_settid_with_pid	YES
58	SYS_readlink	NO	312	SYS_psynch_cvclrprepost	YES
59	SYS_execve	YES	313	SYS_aio_fsync	YES
60	SYS_umask	YES	314	SYS_aio_return	YES
61	SYS_chroot	YES	315	SYS_aio_suspend	YES
65	SYS_msync	YES	316	SYS_aio_cancel	YES
66	SYS_vfork	YES	317	SYS_aio_error	YES
73	SYS_munmap	NO	318	SYS_aio_read	YES
74	SYS_mprotect	NO	319	SYS_aio_write	YES
75	SYS_madvise	NO	320	SYS_lio_listio	YES
78	SYS_mincore	YES	322	SYS_iopolicysys	NO
79	SYS_getgroups	YES	323	SYS_process_policy	YES
80	SYS_setgroups	YES	324	SYS_mlockall	YES
81	SYS_getpgrp	YES	325	SYS_munlockall	YES
82	SYS_setpgid	YES	327	SYS_issetugid	NO
83	SYS_setitimer	NO	328	SYS___pthread_kill	YES
85	SYS_swapon	YES	329	SYS___pthread_sigmask	NO
86	SYS_getitimer	YES	330	SYS___sigwait	YES
89	SYS_getdtablesize	YES	331	SYS___disable_...	NO
90	SYS_dup2	YES	332	SYS___pthread_markcancel	YES
92	SYS_fcntl	NO	333	SYS___pthread_canceled	NO
93	SYS_select	NO	334	SYS___semwait_signal	NO
95	SYS_fsync	NO	336	SYS_proc_info	NO
96	SYS_setpriority	YES	337	SYS_sendfile	YES
97	SYS_socket	YES	338	SYS_stat64	NO
98	SYS_connect	YES	339	SYS_fstat64	NO
100	SYS_getpriority	YES	340	SYS_lstat64	NO
104	SYS_bind	YES	341	SYS_stat64_extended	YES
105	SYS_setsockopt	YES	342	SYS_lstat64_extended	YES
106	SYS_listen	YES	343	SYS_fstat64_extended	YES
111	SYS_sigsuspend	NO	344	SYS_getdirentries64	NO
116	SYS_gettimeofday	NO	345	SYS_statfs64	NO
117	SYS_getrusage	NO	346	SYS_fstatfs64	NO
118	SYS_getsockopt	YES	347	SYS_getfsstat64	NO
120	SYS_readv	YES	348	SYS___pthread_chdir	NO
121	SYS_writev	YES	349	SYS___pthread_fchdir	NO
122	SYS_settimeofday	YES	350	SYS_audit	YES
123	SYS_fchown	YES	351	SYS_auditon	YES
124	SYS_fchmod	YES	353	SYS_getauid	YES

126	SYS_setreuid	YES	354	SYS_setauid	YES
127	SYS_setregid	YES	357	SYS_getaudit_addr	NO
128	SYS_rename	YES	358	SYS_setaudit_addr	YES
131	SYS_flock	YES	359	SYS_auditctl	YES
132	SYS_mkfifo	YES	360	SYS_bsdthread_create	NO
133	SYS_sendto	YES	361	SYS_bsdthread_terminate	NO
134	SYS_shutdown	YES	362	SYS_kqueue	NO
135	SYS_socketpair	YES	363	SYS_kevent	NO
136	SYS_mkdir	NO	364	SYS_lchown	YES
137	SYS_rmdir	YES	365	SYS_stack_snapshot	YES
138	SYS_utimes	YES	366	SYS_bsdthread_register	NO
139	SYS_futimes	YES	367	SYS_workq_open	NO
140	SYS_adjtime	NO	368	SYS_workq_kernreturn	NO
142	SYS_gethostuuid	YES	369	SYS_kevent64	NO
147	SYS_setsid	YES	370	SYS___old_semwait_signal	NO
151	SYS_getpgid	YES	371	SYS___old_semwait_sig...	NO
152	SYS_setprivexec	YES	372	SYS_thread_selfid	NO
153	SYS_pread	NO	373	SYS_ledger	NO
154	SYS_pwrite	YES	380	SYS___mac_execve	YES
155	SYS_nfssvc	YES	381	SYS___mac_syscall	NO
157	SYS_statfs	YES	382	SYS___mac_get_file	YES
158	SYS_fstatfs	YES	383	SYS___mac_set_file	YES
159	SYS_unmount	YES	384	SYS___mac_get_link	YES
161	SYS_getfh	YES	385	SYS___mac_set_link	YES
165	SYS_quotactl	YES	386	SYS___mac_get_proc	YES
167	SYS_mount	YES	387	SYS___mac_set_proc	YES
169	SYS_csops	NO	388	SYS___mac_get_fd	YES
170	SYS_csops_audit...	NO	389	SYS___mac_set_fd	YES
173	SYS_waitid	YES	390	SYS___mac_get_pid	YES
180	SYS_kdebug_trace	YES	391	SYS___mac_get_lcid	YES
181	SYS_setgid	YES	392	SYS___mac_get_lctx	YES
182	SYS_setegid	YES	393	SYS___mac_set_lctx	YES
183	SYS_seteuid	YES	394	SYS_setlcid	YES
184	SYS_sigreturn	NO	395	SYS_getlcid	YES
185	SYS_chud	YES	396	SYS_read_nocancel	NO
187	SYS_fdatasync	YES	397	SYS_write_nocancel	NO
188	SYS_stat	YES	398	SYS_open_nocancel	NO
189	SYS_fstat	YES	399	SYS_close_nocancel	NO
190	SYS_lstat	YES	400	SYS_wait4_nocancel	YES
191	SYS_pathconf	YES	401	SYS_recvmsg_nocancel	YES
192	SYS_fpathconf	YES	402	SYS_sendmsg_nocancel	YES
194	SYS_getrlimit	YES	403	SYS_recvfrom_nocancel	YES
195	SYS_setrlimit	YES	404	SYS_accept_nocancel	YES
196	SYS_getdirentries	YES	405	SYS_msync_nocancel	YES
197	SYS_mmap	NO	406	SYS_fcntl_nocancel	NO
199	SYS_lseek	NO	407	SYS_select_nocancel	YES
200	SYS_truncate	YES	408	SYS_fsync_nocancel	YES
201	SYS_ftruncate	YES	409	SYS_connect_nocancel	YES
202	SYS___sysctl	YES	410	SYS_sigsuspend_nocancel	YES
203	SYS_mlock	YES	411	SYS_readv_nocancel	YES
204	SYS_munlock	YES	412	SYS_writev_nocancel	YES
205	SYS_undelete	YES	413	SYS_sendto_nocancel	YES
216	SYS_open_dprot...	NO	414	SYS_pread_nocancel	YES
220	SYS_getattrlist	NO	415	SYS_pwrite_nocancel	YES
221	SYS_setattrlist	YES	416	SYS_waitid_nocancel	YES
222	SYS_getdirent...	YES	417	SYS_poll_nocancel	YES

223	SYS_exchangedata	YES	418	SYS_msgsnd_nocancel	YES
225	SYS_searchfs	YES	419	SYS_msgrcv_nocancel	YES
226	SYS_delete	YES	420	SYS_sem_wait_nocancel	YES
227	SYS_copyfile	YES	421	SYS_aio_suspend_nocancel	YES
228	SYS_fgetattrlist	YES	422	SYS___sigwait_nocancel	YES
229	SYS_fsetattrlist	YES	423	SYS___semwait_signal...	YES
230	SYS_poll	YES	424	SYS___mac_mount	YES
231	SYS_watchevent	YES	425	SYS___mac_get_mount	YES
232	SYS_waitevent	YES	426	SYS___mac_getfsstat	YES
233	SYS_modwatch	YES	427	SYS_fsgetpath	NO
234	SYS_getxattr	NO	428	SYS_audit_session_self	YES
235	SYS_fgetxattr	YES	429	SYS_audit_session_join	YES
236	SYS_setxattr	YES	430	SYS_fileport_makeport	YES
237	SYS_fsetxattr	YES	431	SYS_fileport_makefd	YES
238	SYS_removexattr	YES	432	SYS_audit_session_port	YES
239	SYS_fremovexattr	YES	433	SYS_pid_suspend	YES
240	SYS_listxattr	YES	434	SYS_pid_resume	YES
241	SYS_flistxattr	YES	438	SYS_shared_region_map...	YES
242	SYS_fsctl	YES	439	SYS_kas_info	YES
243	SYS_initgroups	YES	440	SYS_memorystatus_control	NO
244	SYS_posix_spawn	YES	441	SYS_guarded_open_np	NO
245	SYS_ffsctl	YES	442	SYS_guarded_close_np	NO
247	SYS_nfsclnt	YES	443	SYS_guarded_kqueue_np	NO
248	SYS_fhopen	NO	444	SYS_change_fdguard_np	NO
250	SYS_minherit	YES	446	SYS_proc_rlimit_control	NO
251	SYS_semsys	YES	447	SYS_connectx	NO
252	SYS_msgsys	YES	448	SYS_disconnectx	NO
253	SYS_shmsys	YES	449	SYS_peeloff	NO
254	SYS_semctl	YES	450	SYS_socket_delegate	NO
255	SYS_semget	YES	451	SYS_telemetry	NO
256	SYS_semop	YES	452	SYS_proc_uuid_policy	NO
258	SYS_msgctl	YES	453	SYS_memorystatus_get_...	NO
259	SYS_msgget	YES	454	SYS_system_override	NO
260	SYS_msgsnd	YES	455	SYS_vfs_purge	NO

Note: deprecated system calls are not displayed in Table B.1, hence the missing system calls. `syscalls.master`[1] provides an overview of all the XNU implemented system calls.

[1] https://opensource.apple.com/source/xnu/xnu-1504.3.12/bsd/kern/syscalls.master

C

Workflow of a typical office user

This appendix describes the workflows performed to simulate an office user in the virtualised environment using a virtual machine, described in Section 5.4.2. Monitoring of the system calls only started after the kernel extension was loaded into the kernel, using the `kextload` bash command.

The office user simulation consisted of the following workflow, which does not include installations of software. The following applications are pre-installed, before the simulation was performed:

- Microsoft Office 2011 for Mac,
- OS X Mail client with an installed mail account,
- OS X Safari web browser, without Adobe Flash and Java Runtime,
- OS X Calendar application,
- OS X Address book/Contacts application.

Workflow
The following actions were performed after the kernel extension was loaded. A Gmail account was used to provide E-mail, contacts and calendar synchronisation.

1. Safari web browser was started to visit a web page.
2. Mail client was started.
3. E-mail was sent to the account itself (simulating both sending and receiving of e-mail).
4. Several HTTP and HTTPS websites were visited using the browser.
5. An MS Excel file was downloaded from a website.
6. The downloaded MS Excel file was opened and several macro's were edited to make Excel perform new calculations
7. The Excel file was saved to a user owned directory and sent in an e-mail.
8. An MS Word file was created and some text was typed.
9. The MS Word file was saved to a user owned directory.
10. Several more websites were visited.
11. An MS Powerpoint file and several Powerpoint slides were was created.
12. The Powerpoint file was saved to a user owned directory.
13. Another email containing plain text was sent to the account itself.
14. An event was created in the Calendar application for which several email addresses were invited.
15. New contacts were created in the OS X Contacts application.

16. Several more HTTP and HTTPS websites were visited.

The described workflow is a simplification of the workflow of an office user, but contains all the operations that such an office user could perform, if the application described would be used.

D
Sample of raw system call log

In Listing D.1, a sample of the log file created by the kernel extension is shown. The sample below shows the system calls that occur two seconds before the iWorm malware sample (see Section 3.1.3) is executed. The different features are separated by a `;`. The first line of the sample describes the features. The `Time` feature represents the uptime of the system, in `hh:mm:ss:microsecs` format. The last feature represents the write path or binary path, for a `SYS_write` system call or process execution respectively.

```
Time; process name; PID; PPID; syscall; root privs; write/binary path;
0:1:26,831618; xpcproxy; 363; 0; SYS_write; 1; /dev/null
0:1:26,833227; /System/Library/Frameworks/Security.framework/Versions/A/
    XPCServices/authorizationhost.xpc/Contents/MacOS/authorizationhost;
    363; 0; NEW_PROCESS; 1;
0:1:26,836205; authorizationhos; 363; 1; SYS_shared_region_check_np; 0;
0:1:26,839357; authorizationhos; 363; 0; SYS_write; 1; /dev/dtracehelper
0:1:26,840835; authorizationhos; 363; 1; SYS_ioctl; 0;
0:1:26,841960; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,846707; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,848642; authorizationhos; 363; 1; SYS_ptrace; 0;
0:1:26,849740; authorizationhos; 363; 1; SYS_setrlimit; 0;
0:1:26,850927; authorizationhos; 363; 1; SYS_getegid; 0;
0:1:26,852150; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,854152; authorizationhos; 363; 1; SYS_access; 0;
0:1:26,855806; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,857219; authorizationhos; 363; 1; SYS_access; 0;
0:1:26,859729; authorizationhos; 363; 0; SYS_write; 1; /dev/dtracehelper
0:1:26,861268; authorizationhos; 363; 1; SYS_ioctl; 0;
0:1:26,862974; securityd; 72; 1; SYS___sysctl; 0;
0:1:26,864014; securityd; 72; 1; SYS___sysctl; 0;
0:1:26,865187; authorizationhos; 363; 0; SYS_write; 1; /private/var/db/mds
    /system/mds.lock
0:1:26,866958; authorizationhos; 363; 1; SYS_shm_open; 0;
0:1:26,868090; authorizationhos; 363; 1; SYS_ftruncate; 0;
0:1:26,869670; authorizationhos; 363; 1; SYS_flock; 0;
0:1:26,873489; authorizationhos; 363; 1; SYS_access; 0;
0:1:26,875394; authorizationhos; 363; 1; SYS_shm_open; 0;
0:1:26,878479; authorizationhos; 363; 1; SYS_access; 0;
0:1:26,879684; authorizationhos; 363; 1; SYS_audit_session_join; 0;
0:1:26,883441; authd; 119; 1; SYS_auditon; 0;
0:1:26,884521; authd; 119; 1; SYS_audit; 0;
0:1:26,885557; authd; 119; 1; SYS_auditon; 0;
```

```
0:1:26,886628; authd; 119; 1; SYS_audit_session_port; 0;
0:1:26,889150; authorizationhos; 363; 1; SYS_auditon; 0;
0:1:26,890293; authorizationhos; 363; 1; SYS_getauid; 0;
0:1:26,891729; authorizationhos; 363; 1; SYS_getrlimit; 0;
0:1:26,895919; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,897280; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,898625; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,900075; authorizationhos; 363; 1; SYS_open; 0; /Library/Managed
    Preferences/.GlobalPreferences.plist;
0:1:26,904476; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:26,906393; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:26,908022; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:26,910367; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,911635; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,913521; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:26,914839; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,916093; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,917367; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,918668; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:26,920113; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:26,922008; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:26,923712; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:26,926088; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:26,927834; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:26,929299; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,32131; com.apple.Accoun; 208; 1; SYS___sysctl; 0;
0:1:27,32277; opendirectoryd; 66; 1; SYS_unlink; 0;
0:1:27,34536; opendirectoryd; 66; 1; SYS_auditon; 0;
0:1:27,35149; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,35824; opendirectoryd; 66; 1; SYS_audit; 0;
0:1:27,38122; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,39914; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,41116; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,42509; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,44225; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,45920; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,48135; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,49752; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,51078; SecurityAgent; 361; 1; SYS_shm_open; 0;
0:1:27,51104; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,52257; SecurityAgent; 361; 1; SYS_shm_open; 0;
0:1:27,55045; com.apple.Accoun; 208; 1; SYS___sysctl; 0;
0:1:27,55370; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,57522; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,58898; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,60161; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,61368; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,62724; authorizationhos; 363; 1; SYS_open; 0; /Library/Managed
    Preferences/.GlobalPreferences.plist;
0:1:27,64743; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,65905; authorizationhos; 363; 1; SYS_getegid; 0;
0:1:27,67065; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,68222; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,69427; authorizationhos; 363; 1; SYS_setegid; 0;
0:1:27,70561; authorizationhos; 363; 1; SYS_seteuid; 0;
0:1:27,71736; authorizationhos; 363; 1; SYS___sysctl; 0;
```

```
0:1:27,73005; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,74214; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,75421; authorizationhos; 363; 1; SYS_seteuid; 0;
0:1:27,76655; authorizationhos; 363; 1; SYS_setegid; 0;
0:1:27,77776; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,78991; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,80244; authorizationhos; 363; 1; SYS_getegid; 0;
0:1:27,81493; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,82910; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,84822; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,86523; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,88662; authorizationhos; 363; 1; SYS___sysctl; 0;
0:1:27,90077; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,91654; authorizationhos; 363; 1; SYS_auditon; 0;
0:1:27,92846; authorizationhos; 363; 1; SYS_getegid; 0;
0:1:27,94000; authorizationhos; 363; 1; SYS_auditon; 0;
0:1:27,95116; authorizationhos; 363; 1; SYS_audit; 0;
0:1:27,96896; authd; 119; 1; SYS_auditon; 0;
0:1:27,97902; authd; 119; 1; SYS_audit; 0;
0:1:27,98894; authd; 119; 1; SYS_auditon; 0;
0:1:27,99864; authd; 119; 1; SYS_audit_session_port; 0;
0:1:27,101886; authorizationhos; 363; 1; SYS_open; 0; /Library/Managed
    Preferences/.GlobalPreferences.plist;
0:1:27,104733; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,107599; cfprefsd; 118; 1; SYS_open; 0; /var/root/Library/
    Preferences/ByHost/.GlobalPreferences.564DCE16-BB58-FD4B-00A9-
    DD4178FC5DAF.plist;
0:1:27,110589; authorizationhos; 363; 1; SYS_shm_open; 0;
0:1:27,112502; authd; 119; 1; SYS_auditon; 0;
0:1:27,113479; authd; 119; 1; SYS_audit; 0;
0:1:27,114464; authd; 119; 1; SYS___sysctl; 0;
0:1:27,115568; authd; 119; 1; SYS_auditon; 0;
0:1:27,116623; authd; 119; 1; SYS_audit; 0;
0:1:27,117697; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,119075; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,120421; opendirectoryd; 66; 1; SYS___sysctl; 0;
0:1:27,121666; authd; 119; 1; SYS_auditon; 0;
0:1:27,122643; authd; 119; 1; SYS_setaudit_addr; 0;
0:1:27,123805; authd; 119; 1; SYS___sysctl; 0;
0:1:27,124905; authd; 119; 1; SYS_auditon; 0;
0:1:27,125912; authd; 119; 1; SYS_audit; 0;
0:1:27,126875; authd; 119; 1; SYS_auditon; 0;
0:1:27,127832; authd; 119; 1; SYS_audit; 0;
0:1:27,129349; security_authtra; 358; 357; SYS_dup2; 0;
0:1:27,130900; security_authtra; 358; 357; SYS___sysctl; 0;
0:1:27,132442; security_authtra; 358; 357; SYS_execve; 0;
0:1:27,136163; /Users/m/Desktop/malware/iWorm/Install.app/Contents/MacOS
    /0; 358; 0; NEW_PROCESS; 1;
0:1:27,136599; Install; 357; 1; SYS_execve; 0;
```

Listing D.1: Sample of the raw dataset of system calls of an iWorm infection.

E

Heat maps of malware

Heat maps of malware samples discussed in Section 6.3.3.

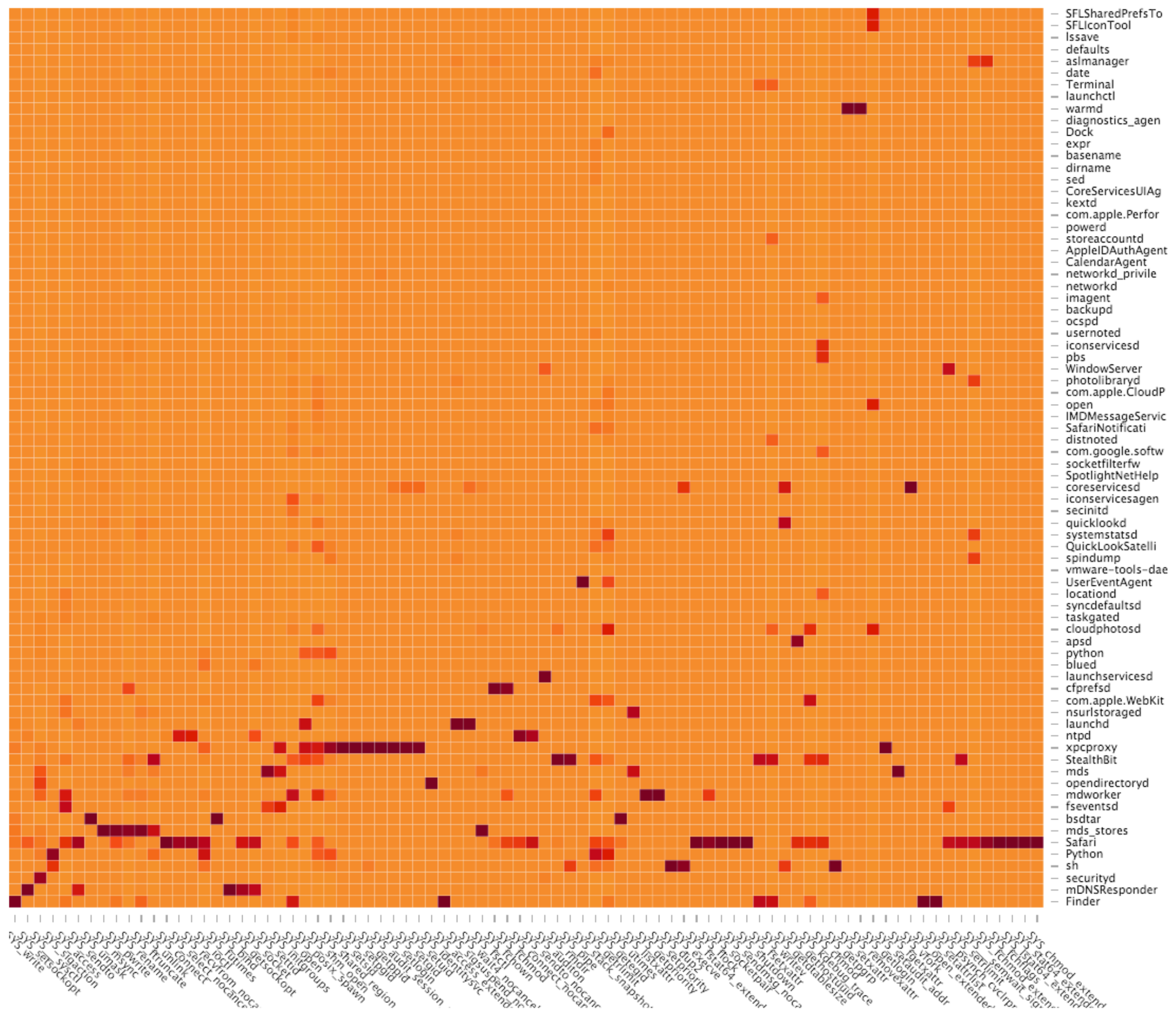


Figure E.1: Heat map of the Cointhief dataset, showing processes on y-axis and system calls on the x-axis.

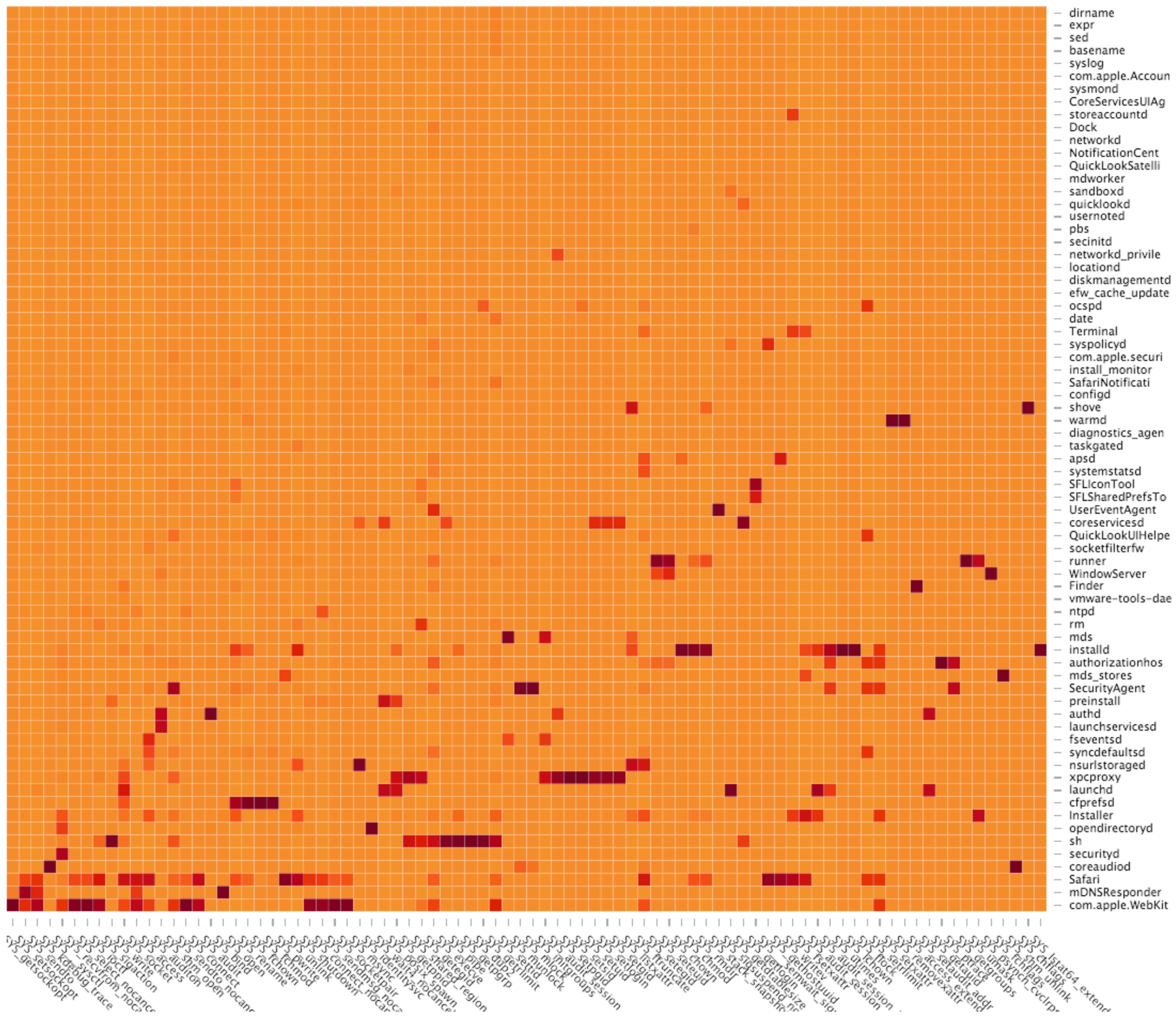


Figure E.2: Heat map of the Flashback dataset, showing processes on y-axis and system calls on the x-axis.

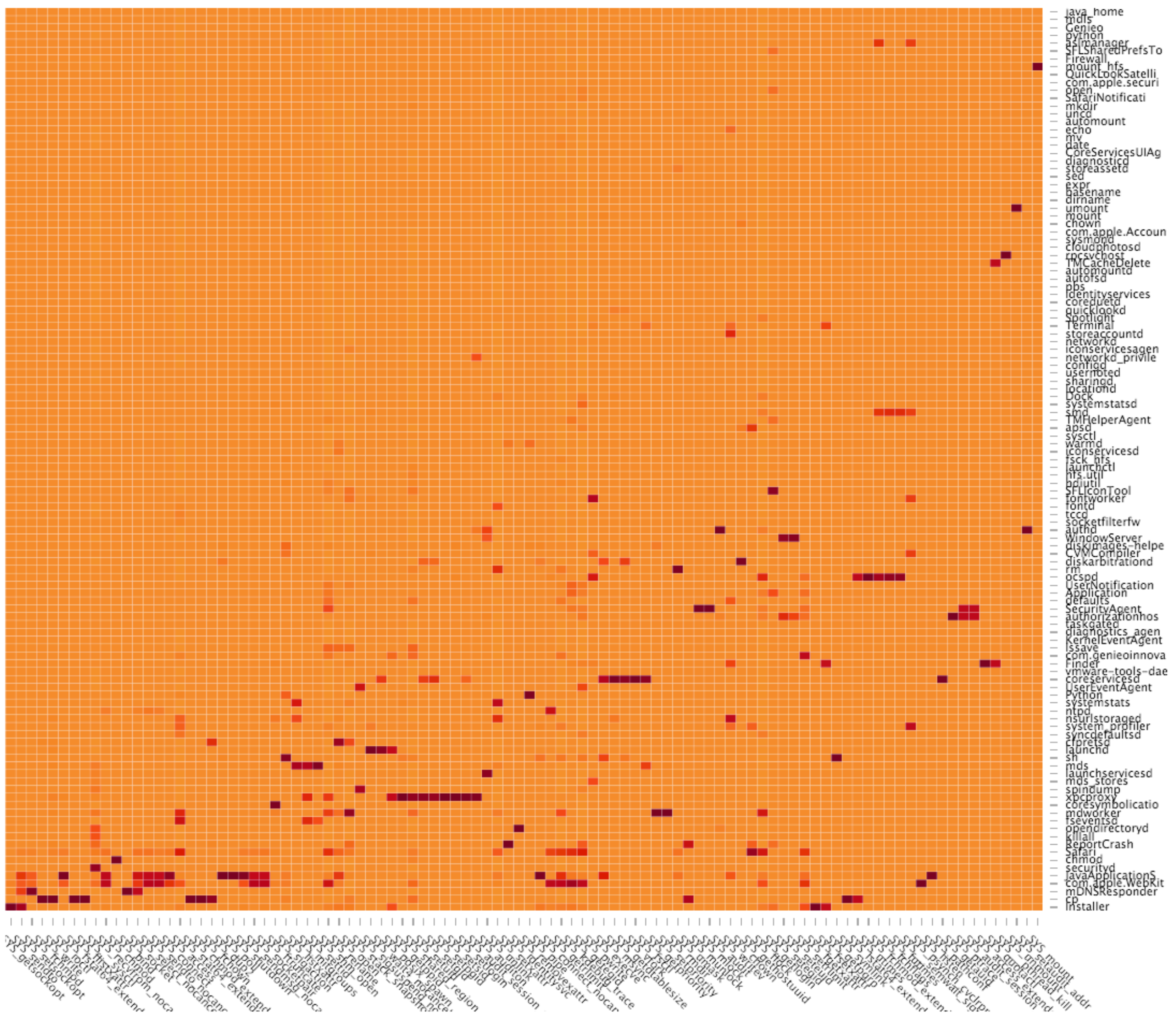


Figure E.3: Heat map of the Genieo dataset, showing processes on y-axis and system calls on the x-axis.

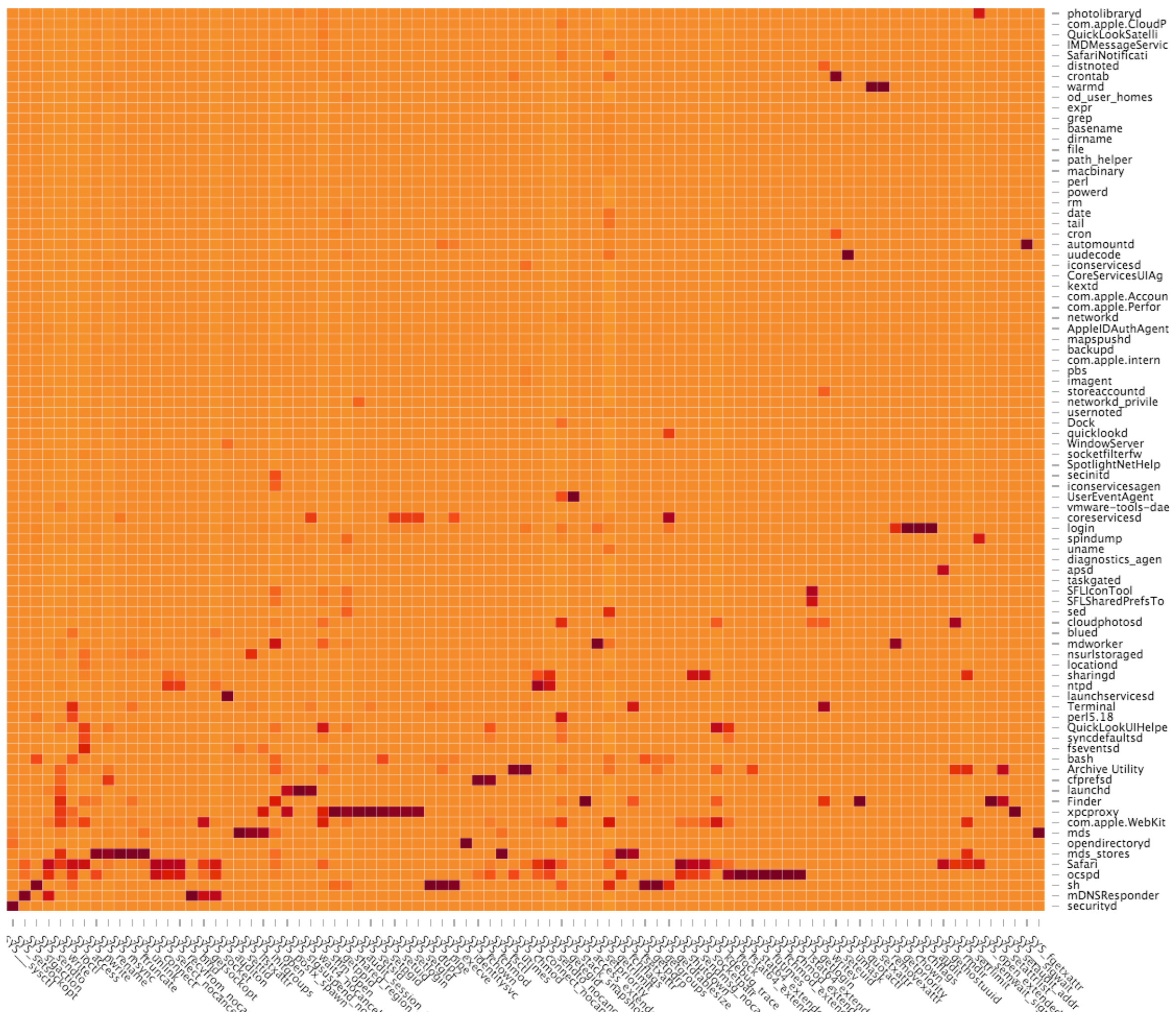


Figure E.4: Heat map of the Jahlav dataset, showing processes on y-axis and system calls on the x-axis.

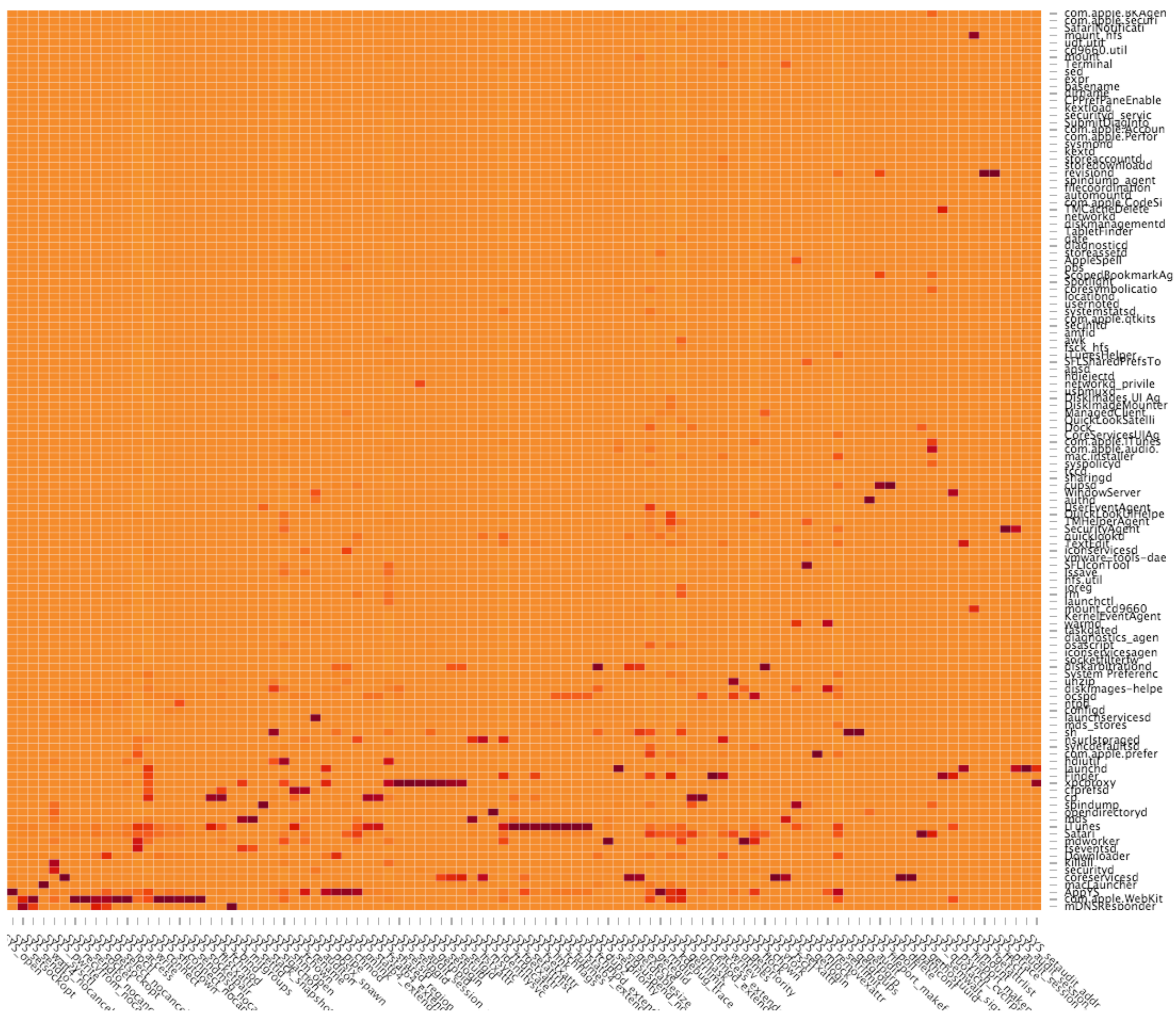


Figure E.5: Heat map of the MacInstaller dataset, showing processes on y-axis and system calls on the x-axis.

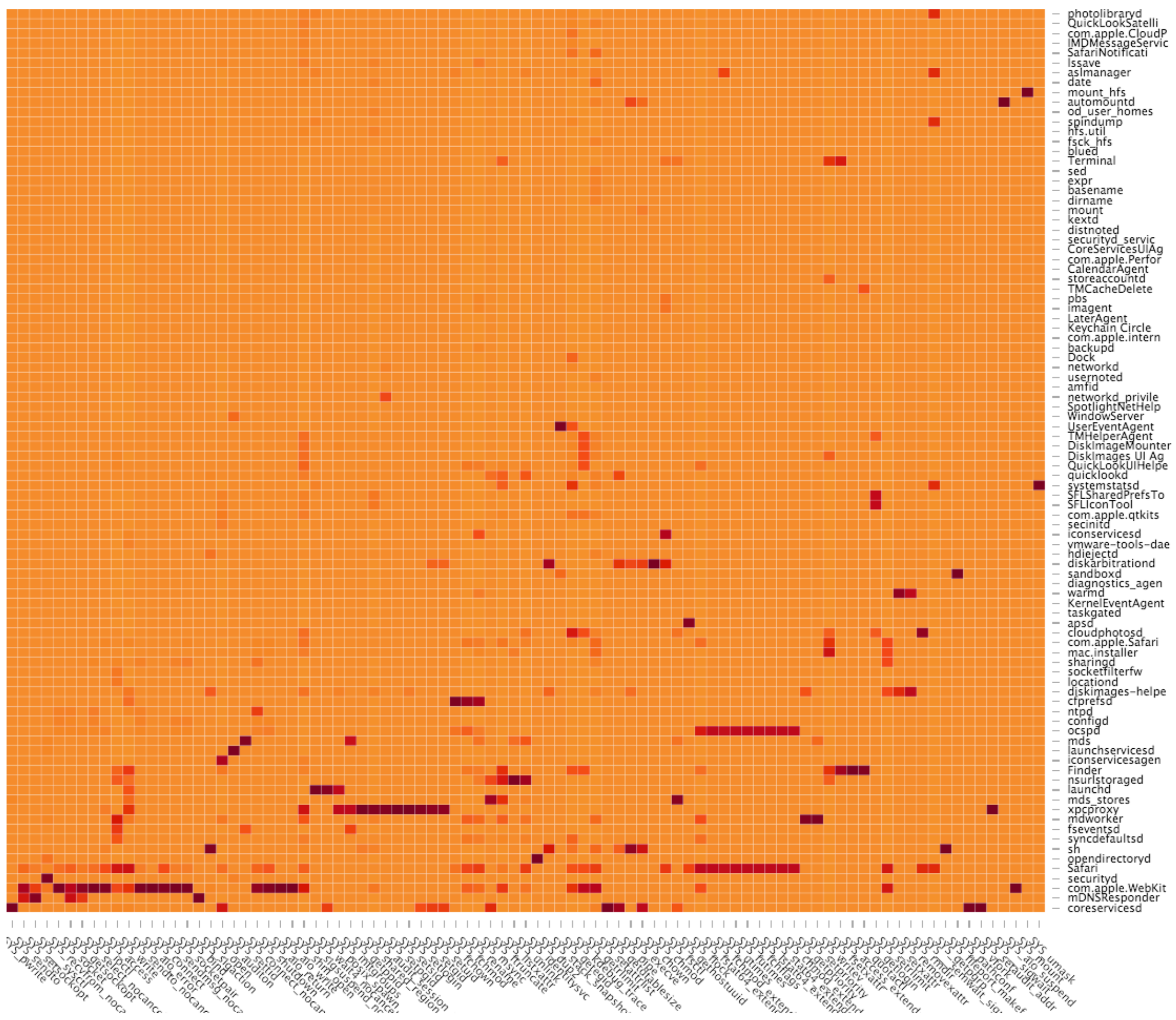


Figure E.6: Heat map of the MacVX dataset, showing processes on y-axis and system calls on the x-axis.

F

Analysed benign applications

Table F.1 shows the benign applications that were used by the participants of the survey. A majority of the applications can be categorised as "Developer application", marked as "D" in the table. The "App Store user profile" and "Typical user profile" are "A" and "T" respectively. The application in this table were tested for false positives, which is shown in column "FP".

Table F.1: Table of applications tested for false positives.

Name	FP	Profile	Name	FP	Profile
CodeRunner	NO	D	Sunrise Calendar	NO	A
Fantastical	NO	A	Kaleidoscope	NO	D
Messages for WhatsApp	NO	A	Textual	NO	A
Adium	NO	T	Picasa	NO	T
GitHub	YES	D	FirefoxNightly	NO	D
Minecraft	NO	T	CleanMyMac	NO	T
EasyFind	NO	A	Transmit	NO	T
MATLAB_R2012b	YES	D	LastPass	NO	T
TextWrangler	NO	D	Latexian	YES	D
HandBrake	NO	T	Pixelmator	NO	A
TextMate	NO	D	AppCleaner	NO	T
Coda	NO	D	Mathematica	NO	D
LanScan	NO	D	PyCharm	YES	D
ForkLift	NO	D	MacDown	NO	D
WeChat	NO	A	Little Snitch Configuration	NO	D
Texpad	YES	D	iTerm	YES	D
TorBrowser	YES	D	BetterZip	NO	T
Microsoft Remote Desktop	NO	T	BlockBlock	NO	D
TeXShop	YES	D	Tunnelblick	YES	T
Google Earth	NO	A	Microsoft Communicator	NO	T
1Password	NO	A	CMake	YES	D
Microsoft OneNote	NO	A	Google Drive	NO	T
Sublime Text	NO	D	GPG Keychain	YES	D
Flux	YES	D	Steam	NO	T
Microsoft Outlook	NO	T	AppCleaner	NO	T
Microsoft Messenger	NO	T	Atom	YES	D
GitHub Desktop	YES	D	RStudio	YES	D
OmniGraffle	NO	A	Telegram	NO	A
Twitter	NO	A	TeamViewer	NO	T
Microsoft PowerPoint	NO	T	Microsoft Word	NO	T
Cyberduck	NO	T	Adobe Acrobat Reader DC	NO	T
R	YES	D	iPhoto	NO	T

Sublime Text	NO	D	Transmission	NO	T
FileZilla	NO	D	uTorrent	NO	T
VMware Fusion	YES	D	Microsoft Excel	NO	T
The Unarchiver	NO	A	Slack	NO	T
Remote Desktop Connection	NO	T	Spotify	NO	T
GarageBand	NO	A	Numbers	NO	A
iMovie	NO	A	Firefox	NO	T
Keynote	NO	A	Pages	NO	A
VirtualBox	NO	D	Wireshark	NO	D
Xcode	YES	D	VLC	NO	T
Dropbox	NO	T	Photos	NO	A
Google Chrome	NO	T	Skype	NO	T

Bibliography

[1] M. Alazab, R. Layton, S. Venkataraman, and P. Watters. Malware detection based on structural and behavioural features of api calls. 2010.

[2] Andrew. New mac os x trojan/virus alert. `http://www.ambrosiasw.com/forums/index.php?showtopic=102379`, 2006. Accessed on: 25-11-2015.

[3] Scott M. Bennet J. Backdoor olyx - is it malware on a mission for mac?, 2014.

[4] Bit9 + Carbon Black. 2015: The most profilic year in history for osx malware, 2015.

[5] I. Broderick. Flashback os x malware. *Virus Bulletin Conference September 2012*, 2012.

[6] Xiao C. and Chen J. New os x ransomware keranger infected transmission bittorrent client installer, 2016.

[7] A. Dehnert. *Using VProbes for intrusion detection*. PhD thesis, Massachusetts Institute of Technology, 2013.

[8] ESET. Straight facts about mac malware. `https://www.eset.com/int/mac-malware-facts/`, 2015. Accessed on: 25-11-2015.

[9] F-Secure. F-secure threat description. `https://www.f-secure.com/v-descs/`.

[10] P. Faruki, V. Laxmi, M. S. Gaur, and P. Vinod. Mining control flow graph as api call-grams to detect portable executable malware. In *Proceedings of the Fifth International Conference on Security of Information and Networks*. ACM, 2012.

[11] S. Forrest, S. Hofmeyr, A. Somayaji, T. Longstaff, et al. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE, 1996.

[12] J. Foster. *Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals: Reverse Engineering Exploits and Tool Coding for Security Professionals*. Syngress, 2005.

[13] FTC. Spyware workshop: Monitoring software on your personal computer: Spyware, adware, and other software: Report of the federal trade commission staff. `https://www.ftc.gov/reports/spyware-workshop-monitoring-software-your-personal-computer-spyware-adware-other-software`, 2005.

[14] T. Garfinkel et al. Traps and pitfalls: Practical problems in system call interposition based security tools. 2003.

[15] M. Gaudesi, A. Marcelli, E. Sanchez, G. Squillero, and A. Tonda. Challenging anti-virus through evolutionary malware obfuscation. *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016*, 2016.

[16] D. Harley and A. Lee. Heuristic analysis — detecting unknown viruses. `http://www.welivesecurity.com/media_files/white-papers/Heuristic_Analysis.pdf`. Accessed on: 28-11-2015.

[17] J. He. Linux system call quick reference.

[18] Imperva. Assessing the effectiveness of antivirus solutions. `http://www.imperva.com/docs/hii_assessing_the_effectiveness_of_antivirus_solutions.pdf`, 2012.

[19] K Iwamoto and K. Wasaki. Malware classification based on extracted api sequences using static analysis. In *Proceedings of the Asian Internet Engineeering Conference*. ACM, 2012.

[20] B. Jewell and J. Beaver. Host-based data exfiltration detection via system call sequences. In *ICIW2011-Proceedings of the 6th International Conference on Information Warfare and Secuirty: ICIW*. Academic Conferences Limited, 2011.

[21] Michael K. How system calls work on linux/i86, 1996.

[22] B. Kang, T. Kim, H. Kwon, Y. Choi, and E. Im. Malware classification method via binary content comparison. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*. ACM, 2012.

[23] J. Kolter and M. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 2006.

[24] A. Kurchuk and A. Keromytis. Recursive sandboxes: Extending systrace to empower applications. In *Security and Protection in Information Processing Systems*. Springer, 2004.

[25] E. Lee. Oceanlotus for os x – an application bundle pretending to be an adobe flash update. `https://www.alienvault.com/blogs/labs-research/oceanlotus-for-os-x-an-application-bundle-pretending-to-be-an-adobe-flash-update`, 2016.

[26] J. Lee, K. Jeong, and H. Lee. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10. ACM, 2010.

[27] K. Lee. Behavioral vs. heuristic antivirus.

[28] J. Levin. *Mac OS X and iOS Internals: To the Apple's Core*. Wrox, 2012.

[29] N. Milosevic. History of malware. *arXiv preprint arXiv:1302.5392*, 2013.

[30] mmpc2. Backdoor olyx - is it malware on a mission for mac? `http://blogs.technet.com/b/mmpc/archive/2011/07/25/backdoor-olyx-is-it-malware-on-a-mission-for-mac.aspx`, 2011. Accessed on: 25-11-2015.

[31] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. *Annual Computer Security Applications Conference*, 2007.

[32] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[33] V. Nair, H. Jain, Y. Golecha, M Gaur, and V. Laxmi. Medusa: Metamorphic malware dynamic analysis using signature from api. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, pages 263–269. ACM, 2010.

[34] H. Nayyar. An opportunity in crisis, 2014.

[35] N. Nguyen, P. Reiher, and G. Kuenning. Detecting insider threats by monitoring system call activity. Citeseer, 2003.

[36] N. Provos. Improving host security with system call policies.

[37] S. Russell and P. Norvig. Artificial intelligence: a modern approach. 1995.

[38] Z. Salehi, M. Ghiasi, and A. Sami. 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*, pages 563–568. IEEE, 2012.

[39] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze. Malware detection based on mining api calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10. ACM, 2010.

[40] M. Schmall. Building an anti-virus engine. `https://www.f-secure.com/v-descs/`, 2010.

[41] INTEGO SECURITY. Inqtana.d bluetooth exploit. `http://www.mactech.com/content/inqtanad-bluetooth-exploit-0`, 2006. Accessed on: 25-11-2015.

[42] Intel Security. Net losses: Estimating the global cost of cybercrime, 2014.

[43] S. Şen, J. Clark, and J. Tapiador. Power-aware intrusion detection in mobile ad hoc networks. In *Ad hoc networks*, pages 224–239. Springer, 2009.

[44] A. Sood and R. Enbody. Crimeware-as-a-service—a survey of commoditized crimeware in the underground market. *International Journal of Critical Infrastructure Protection*, 2013.

[45] R. Stevens and S. Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.

[46] H. Sun, Y. Lin, and M. Wu. Api monitoring system for defeating worms and exploits in ms-windows system. In *Information Security and Privacy*. Springer, 2006.

[47] K. Tan, K. Killourhy, and R. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection*. Springer, 2002.

[48] K. Tsyganok, E. Tumoyan, L. Babenko, and M. Anikeev. Classification of polymorphic and metamorphic malware samples based on their behavior. In *Proceedings of the Fifth International Conference on Security of Information and Networks*. ACM, 2012.

[49] P. Vilaça. Tales from crisis, ch. 1-3. `https://reverse.put.as/tag/crisis/`, 2012.

[50] P. Vilaça. The italian morons are back! what are they up to this time? `https://reverse.put.as/2016/02/29/the-italian-morons-are-back-what-are-they-up-to-this-time/`, 2016.

[51] N. Virvilis and D. Gritzalis. The big four - what we did wrong in advanced persistent threat detection? In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 248–254. IEEE, 2013.

[52] C. Wagner, G. Wagener, R. State, and T. Engel. Malware analysis with graph kernels and support vector machines. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, 2009.

[53] P. Wardle. Invading the core: Iworm's infection vector and persistence mechanism. *Virus Bulletin*, 2014.

[54] P. Wardle. Writing bad ass malware for osx, 2015.

[55] C. Xiao. Wirelurker: A new era in ios and os x malware, 2014.

[56] C. Xiao. More details on the xcodeghost malware and affected ios apps, 2015.

[57] T. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang. An intelligent pe-malware detection system based on association mining. *Journal in computer virology*, pages 323–334, 2008.

[58] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 2007.

[59] Lenny Z. Categories of common malware traits, 2010.