virus BULLETIN Covering the global threat landscape

BEHAVIOURAL DETECTION AND PREVENTION OF MALWARE ON OS X

Vincent Van Mieghem
Delft University of Technology, The Netherlands

Malware on *Apple*'s *OS X* systems is proving to be an increasing security threat, and one that is currently countered solely with traditional anti-virus (AV) technologies. Traditional AV technologies impose a significant performance overhead on the computer system and there is an inherent delay in their effectiveness, due to their signature-based detection techniques. This paper presents a novel generic behavioural detection and prevention mechanism for malware on *OS X* that is based on system calls. A large number of system call traces are 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.

1. INTRODUCTION

Over the last three years, an increase in malware targeting *OS X* systems has been observed. Five times more *OS X* malware appeared in 2015 than during the preceding five years combined [1]. Many types of malware that previously appeared only on *Microsoft Windows* systems are now also emerging on *OS X* systems. Serious threats such as rootkits designed to exfiltrate valuable information from systems, or malware that encrypts personal documents such that they can only be decrypted in exchange for bitcoins are no longer absent from Macs. Traditional anti-virus technologies rely heavily on binary signature checking, a detection technique that often lags behind [2, 3], yet traditional detection techniques are still widely used, especially on *OS X*. Nowadays, many binary obfuscation and signature modification techniques are used by malware to evade AV detection. A need for more advanced malware detection methods has arisen.

This paper presents a novel malware detection method that has been shown to be able to prevent infection by every malware sample that could be found on *OS X* systems at the time, without prior knowledge of the malware samples. System calls are used to define and detect malicious behaviour of malware processes on *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 the system call usage of applications and processes on a system can reveal application behaviours, which can be used to identify malicious processes. By monitoring a large number of benign and malicious processes, a clear recurring pattern of system calls can be extracted from the malicious processes that is absent from the system call traces of benign processes on *OS X*. Heat map visualizations and sequential analysis of the system call traces are used to obtain the insights required to construct the malicious patterns. Several of these malicious patterns are provided and explained in this paper.

In Section 3, the paper describes the structure of the acquired system call traces and the utilities constructed. The process of collecting the system calls traces from malware samples is described in Section 4. Subsequently, the analysis and results are explained in Section 5. Finally, the results are evaluated and discussed in Sections 6 and 7, respectively.

2. RELATED WORK

The majority of research conducted into 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. Behavioural analysis is concerned primarily with deriving identifying behavioural 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 do not hinder 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 algorithms 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 security of the host 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: an 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 the 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 made by programs and use this information to construct a signature for a program. Nair *et al.* introduce their own classification algorithm and show an 80% detection rate for *Windows* malware.

Dehnert [13] argues that an IDS running on the host operating system itself is vulnerable to a direct attack and proposes an intrusion detection method based on a hypervisor. Dehnert uses system call usage and 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. However, due to the absence of any refinement, the detection technique performs very poorly and operates with a performance overhead of almost 300%.

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

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

3. MONITORING SYSTEM CALL USAGE

One way to separate the functionality provided by an operating system is a division between 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 the 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 the file handle of a file on the file system, and SYS_getsocket to obtain the handle of a socket to connect to the Internet. The implementation of calls resides in the XNU kernel of an *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 hooked into, which enables the logging functionality. The implementations of system calls in the sysent-table are then replaced with different implementations. This technique is called 'hooking' [15]. Figure 1 shows the rewiring of a system call implementation after the 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, while 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.

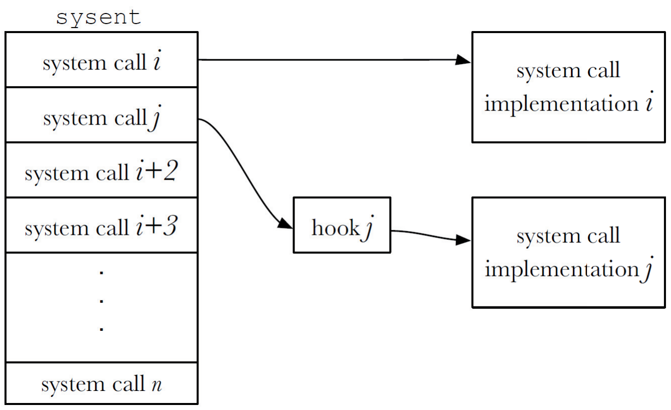

Figure 1: High level sysent-table representation in which all of the system calls are defined. System call i represents the normal system call wiring to its implementation. System call j represents a hooked system call.

A kernel module implementing the hooking of system calls was developed (the source code is available on *GitHub* [16]). The hook logs the system call invoked in the exact order of occurrence to a log file and subsequently calls the original system call implementation to continue normal functionality, as shown in Figure 1. Table 1 shows the metadata of a system call that is logged to the log file.

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 super-user 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 to which the 'write' system call writes.

Table 1: Metadata of a system call that is captured.

Subsequently, while the kernel module is loaded, system calls called by processes log their invocation to a log file, resulting in a large number of system call traces in the exact order in which the calls were invoked. The log file represents the 'raw dataset'. Listing 1 shows several records of the raw dataset. The records are sequential and ordered chronologically, 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 are called more than 500 times per second in an idle system state. Examples of such calls are SYS_read for reading files and SYS_setitimer, which is used by processes to set an interval timer. System call usage was observed on an idle system. The system calls generating more than 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 the *GitHub* repository [16].

4. COLLECTING SYSTEM CALL TRACES

System call traces can only be collected at the 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 a fully patched *OS X* 10.11.3 was used to run malware samples. After the kernel module was loaded, the malware sample was executed and monitored for five minutes, which was the 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 was much shorter. Subsequently, the log file was captured and the virtual machine reverted to its original state for the next malware sample to be monitored.

According to *Symantec*, 57 unique *OS X* malware samples have been found since 2010 [17]. 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 certain malicious parts of *OS X* applications that are not executable are uploaded to malware sample collecting services such as *VirusTotal*. For this research, 23 functional *OS X* malware samples were obtained from different sources [18–20]. Table 2 provides an overview of the functional malware samples obtained and analysed.

5. ANALYSIS

The analysis of the collected raw system call traces aims to provide insights into the discrepancies between system call traces in benign and malicious processes. The goal is to extract recurring patterns that are present in malicious system call traces, but absent from 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 into the usage of outlying system calls by a process. Figure 2 shows the heat map of OSX.Wirelurker's system call trace. The system calls called, and the processes, are listed on the x-axis and y-axis, respectively. The data points (number

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

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	
22	OSX.Eleanor	Backdoor	07/06/2016	
23	OSX.Keydnap	Backdoor	07/13/2016	

Table 2: A list of OS X malware samples analysed in this research.

of system calls) are normalized in the heat map, according to their occurrence in the system call traces of other processes. The normalization ensures that processes calling specific system calls that are called significantly less frequently by other processes show darker in the heat map and imply outliers. In the manual sequential analysis, consisting of linear traversal of the system call traces, these outliers were analysed to derive specific recurring patterns.

5.1 Execution calls

The heat map visualization of a dataset containing the system call traces of benign processes shows SYS_execve and SYS_posix_spawn system calls executed only 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 *OS X*'s sandboxing technology, launchd and xpcproxy [21] (XPC Services) are responsible for process executions and interprocess communication [22]. Figure 3 illustrates the XPC Services process execution request scheme on a high level. On a clean system, launchd and xpcproxy are the only processes that use SYS_execve and SYS_posix_spawn.

However, the heat map visualization of malware shows the presence of SYS_execve and SYS_posix_spawn performed by processes other than 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 traces of recurring patterns were searched for, a clear pattern appeared to be present in all malware samples listed in Table 2. At some point in their infection process, all malicious processes use either of these calls (SYS_execve or SYS_posix_spawn) to execute a shell process, without requesting XPC Services (illustrated in Figure 4). The possible reason is discussed in Section 7. Listings 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.

5.2 Persistency

Another behaviour derived from the heat map and the sequential analysis that appeared to be inherent to malware was gaining persistency using LaunchDaemons (auto-run items on system start-up). In *OS X*, processes are required

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

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

```
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;
```

Listing 3: Process transmission executes the /bin/sh process using the SYS_posix_spawn system call, and process sh executes the kernel_service process using a SYS_execve system call (OSX.KeyRanger.A).

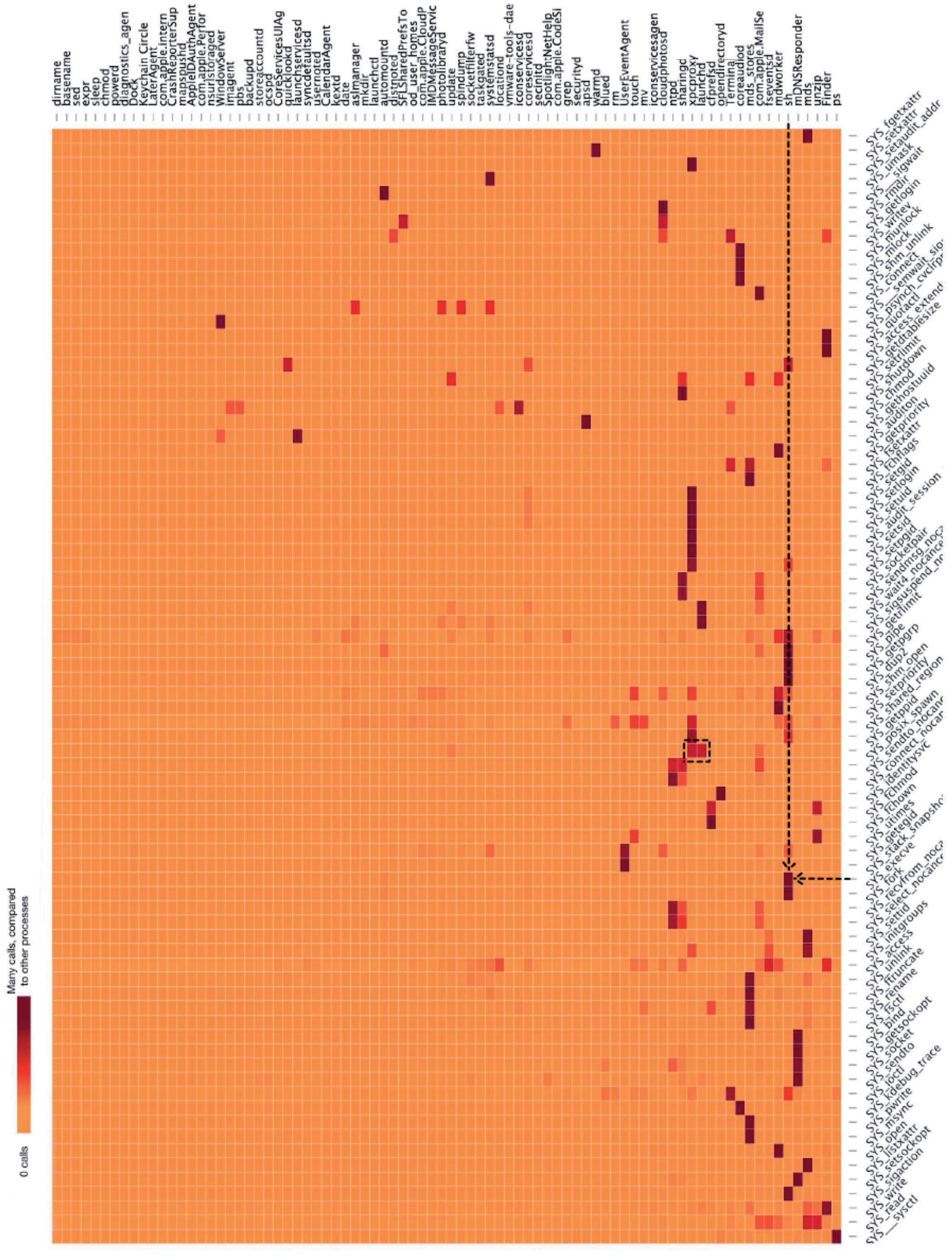

Figure 2: Heat map visualization of OSX.Wirelurker malicious system call trace. The system calls called and the processes are listed on the x-axis and y-axis, respectively. Each square represents the number of system calls, as a normalized data point. Darker blocks represent an outlying number of system 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.

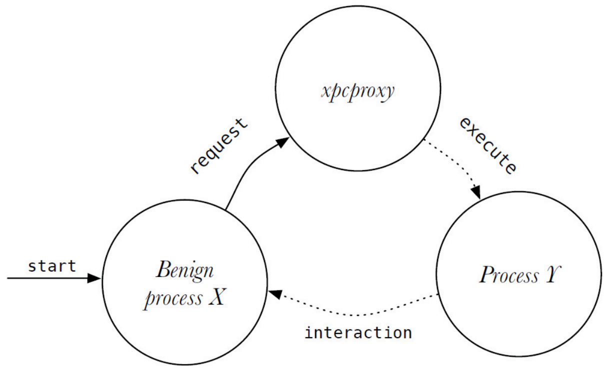

Figure 3: Benign (sandboxed) process X requests XPC Services to interact with process Y. XPC Services decides whether to allow or deny this request.

to store a configuration file in a specific LaunchDaemon directory and thereafter notify the OS (via launchctl) of the config's existence in order to (persistently) be started by the operating system upon start-up. 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 Listings 4, 5 and 6.

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

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

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

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

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

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

Some examples of *OS X* malware use 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.

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

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

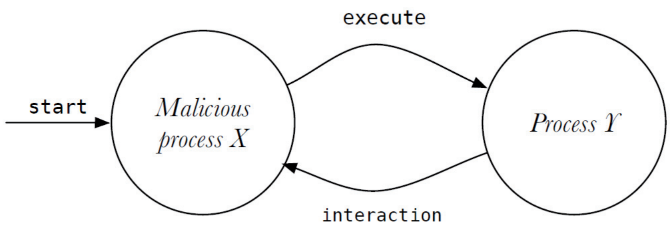

Figure 4: Malicious process X executes process Y without XPC interaction. Typically, process Y is a shell process.

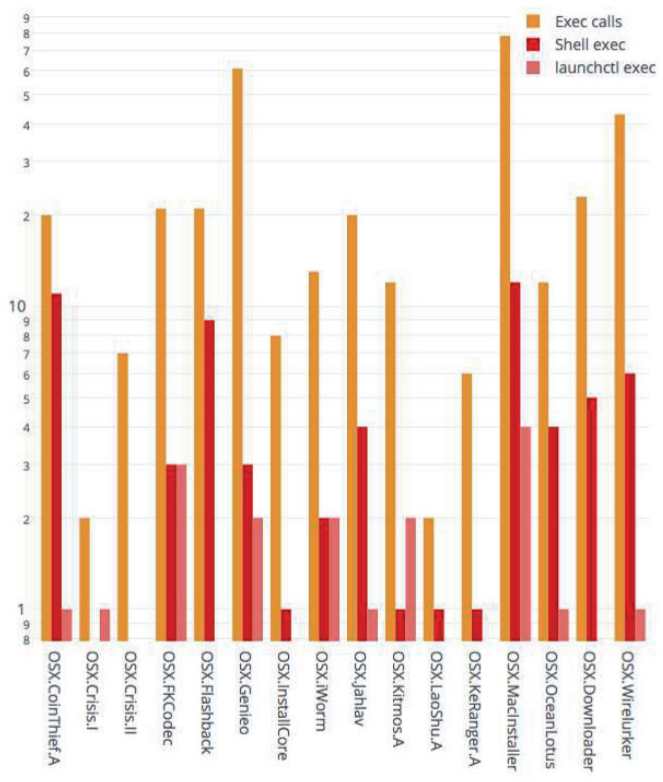

Figure 5: Number of execution calls, execution of shell processes and executions of launchctl by malicious processes. Note: y-axis has a logarithmic scale. Note: OSX.Crisis lacks the execution of a shell process due to the absence of a complete and functional sample of the rootkit.

Based on the heat map visualization of the entire system call trace, processes that connect to the Internet can be observed. Internet-connecting processes invoke many SYS_getsockopt and SYS_setsockopt operations, which are 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. The large number of execution system calls to shell processes in all malware samples can clearly be seen.

6. EVALUATION

The most prominent malicious pattern derived from the analysis phase to detect presence of malware on a system is the extraordinary number of execution calls (to shells) performed by malicious processes. Initially, the pattern was observed in malware samples 1 to 15 in Table 2. At that time, malware samples 16 to 23 were still undiscovered by the security industry. After obtaining malware samples 16 to 23, the defined malicious patterns from the earlier malware samples were found to be present in system call traces of the new samples 16 to 23 as well, and can thus be used to detect the newer malware samples. The horizontal line in Table 2 indicates the moment of the malicious pattern definition.

To verify this pattern as unique to malware, several different experiments were performed. The presence of the execution patterns was expected in benign processes as well. To present an accurate false positive rate (FPR) evaluation, three different user profiles are defined. The user profiles define the type of user, hence the type of applications and processes used by the user.

Three user profiles are defined:

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 *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 define the FPR more accurately by differentiating between the type of applications triggering false positives.

The evaluation consists of two phases:

1. ***Determining application usage of Mac users***. A survey of 25 real Mac users was conducted to gain insights into application usage by users fitting certain profiles. Each user was asked to select the profile that best suited them, and to upload their installed applications.
2. ***Testing the top X applications for false positives***. In roughly the same environment as described in Section 4, 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 on *GitHub* [16].

6.1 App Store user profile

Apart from the operating-system-owned processes, all processes on the system of the *App Store* user profile are sandboxed. This implies that all the process executions are performed through XPC Services, and SYS_execve and SYS_posix_pawn are performed only by launchd and xpcproxy. 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. Their system call traces show 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%.

6.2 Typical user profile

A typical user is more likely to occasionally use applications distributed outside of the *App Store*. Applications distributed outside of the *App Store* neither have to comply with the strict *App Store* rules, nor do they have to be sandboxed. In this case, processes may perform execution calls, bypassing XPC Services. The 30 applications distributed outside the *App Store* that were used by 10 users fitting the 'typical user profile' were tested. In the evaluation phase, applications that require closer 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* (the equivalent of *Windows Explorer*) behaviour. The *Google Chrome* browser, due to its own sandbox security behaviour, spawns many helper processes to isolate web pages and plug-in 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 shell execution calls, the FPR is still 0%, while the detection rate remains 100%.

6.3 Power user/developer profile

The power user profile describes a user using many development tools, compilers and interpreters like Python and JavaScript (NodeJS/Electron). Of the users surveyed, 15 described themselves as developer/power users. The top 70 applications most used by these 15 developers were analysed. 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 (e.g. /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 (e.g. /sbin/route and /sbin/ifconfig). *GPGTools* (a GPG encryption toolset) also performs execution calls to its own binaries.

A feature that 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 3 shows the detection rates (DR) and false positive rates (FPR) for both shell executions and solely execution calls for each type of user profile. Clearly, the malware detection is most effective on the 'App Store user' and 'Typical user' profiles.

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

Table 3: Detection rate (DR) and false positive rate (FPR) of the detection patterns per user profile.

7. DISCUSSION & FUTURE WORK

The heat map and sequential analysis techniques used in this research resulted in the extraction of 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 described in this paper. In addition, it is observed that malware for *OS X* is not yet as advanced as some *Microsoft Windows* malware families, as Patrick Wardle also explains in [23].

This research and its results show that the techniques used are durable and reusable to extract other patterns from system call traces. As shown in Section 5, 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 this research, system call function arguments may be of value in successive system call research. This research focused 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 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 are presumably therefore 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 *OS X* API functions, which sandboxes and isolates the process from the underlying system. XPC Services is an example. Such restrictions provide protection to a system generically.

This research focused primarily 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 into malicious behaviour.

8. CONCLUSION

This research has shown that malware on *OS X* is detectable based on the system call traces of malware in which fundamental dependencies of malware surface. A kernel extension was developed to construct the system call traces of processes. Based on heat map visualization and sequential analysis, specific system call patterns were 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 20%. It is shown that, in contrast with 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 visualization techniques.

ACKNOWLEDGEMENT

In particular, I would like to thank Maarten Boone and *Fox-IT* for their threat intelligence and cooperation; *VirusTotal* for its 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; and Pedro Vilaça for his shared knowledge and open-source techniques to bypass kernel ASLR and memory write protection in the XNU kernel [24].

REFERENCES

[1] Bit9 + Carbon Black. 2015: The Most Prolific Year in history for OS X malware. 2015. https://www.documentcloud.org/documents/2459197-bit9-carbon-black-threat-research-report-2015.html.

[2] Gaudesi, M.; Marcelli, A.; Sanchez, E.; Squillero, G.; Tonda, A. Challenging Anti-virus Through Evolutionary Malware Obfuscation. Applications of Evolutionary Computation. Proceedings of the 19th European Conference, EvoApplications 2016, Part II.

[3] Moser, A.; Kruegel, C.; Kirda, E. Limits of Static Analysis for Malware Detection. Annual Computer Security Applications Conference, ACSAC, 2007.

[4] Forrest, S.; Hofmeyr, S.; Somayaji, A.; Longstaff, T. A Sense of Self for UNIX Processes. Proceedings of the 1996 IEEE Symposium on Security and Privacy.

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

[6] Provos, N. Improving Host Security with System Call Policies. http://niels.xtdnet.nl/papers/systrace.pdf.

[7] Kurchuk, A.; Keromytis, A. D. Recursive Sandboxes: Extending Systrace to Empower Applications. Security and Protection in Information Processing Systems, 2004, Springer.

[8] Kang, D. K.; Fuller, D.; Honavar V. Learning Classifiers for Misuse and Anomaly Detection Using a Bag of System Calls Representation. Annual Information Assurance Workshop, 2005.

[9] Mehdi, B.; Ahmed F.; Khayyam S.; Farooq M. Towards a Theory of Generalizing System Call Representation for In-execution Malware Detection. International Conference on Communications, 2010.

[10] Xiao H.; Stibor T. A Supervised Topic Transition Model for Detecting Malicious System Call Sequences. Workshop on Knowledge discovery, Modeling and Simulation, 2011.

[11] Sun, H. M.; Lin, Y. H.; Wu, M. F. API Monitoring System for Defeating Worms and Exploits in MS-Windows Systems. Information Security and Privacy, 2006, Springer.

[12] Nair, V. P.; Jain, H.; Golecha, Y. K.; Gaur, M. S.; 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, A. W. Using VProbes for Intrusion Detection. 2013. Massachusetts Institute of Technology.

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

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

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

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

[18] https://objective-see.com/.

[19] https://www.virustotal.com/.

[20] https://researchcenter.paloaltonetworks.com/.

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

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

[23] Wardle, P., Writing bad ass malware for OSX. BlackHat Conference 2015.

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

Editor: Martijn Grooten

Chief of Operations: John Hawes

Security Test Engineers: Scott James, Tony Oliveira, Adrian Luca, Ionuţ Răileanu, Chris Stock

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

Developer: Lian Sebe

Consultant Technical Editor: Dr Morton Swimmer

© 2016 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

Tel: +44 (0)1235 555139 Fax: +44 (0)1865 543153

Email: editorial@virusbtn.com Web: https://www.virusbulletin.com/