A Hybrid Framework to Analyze Web and OS Malware

Vitor M. Afonso[1], Dario S. Fernandes Filho[1], André R. A. Grégio[1,2], Paulo L. de Geus[1], Mario Jino[1]
[1]University of Campinas (Unicamp), Campinas, SP, Brazil
Email: {vitor, dario, paulo}@las.ic.unicamp.br, jino@dca.fee.unicamp.br
[2]Renato Archer IT Research Center (CTI/MCT), Campinas, SP, Brazil
Email: argregio@cti.gov.br

***Abstract*—Malicious programs (malware) cause serious security issues to home users and even to highly secured enterprise systems. The main infection vector currently used by attackers is the Internet. To improve the detection rate and to develop protection mechanisms, it is very important to analyze and study these threats. To this end, several systems were developed to perform malware analysis, which support operating system (OS) programs or Web codes, but they all suffer from limitations. Also, the existing systems focus only on one type of malware, those that target the OS or that require a Web browser. In this article, we propose a framework that is able to analyze Web and OS-based malware, which provides better detection rates and a broader range of malware types analysis. We have also evaluated and compared our analysis results to the state-of-the-art systems, presenting the advantages of the developed framework over them when regarding Web and OS-based malware.**

I. Introduction

Malicious programs, such as trojans, worms and javascript exploits, are a great threat to computer security. They cause several problems to the infected system, its users and to other computers and networks. Users of infected systems may have their private data stolen (e.g. credit card numbers and passwords) and their systems may be remotely controlled to send spam messages or to launch DDoS attacks.

Currently, the Web is the main vector to install malware in attacked systems. Web malware, i.e. malicious codes inside Web pages, exploit the browser or some of its components and perform drive-by downloads that install malicious programs on the compromised system. They can also be used to steal personal cached information and even control the browser.

Two methods are often used to have the victims' browser load malicious content, either by injecting malicious codes in benign pages and waiting for users to unwittingly access it, or by sending phishing messages containing malicious files or links. Infecting benign sites is performed through the exploitation of some vulnerability in the Web application (server-side), followed by the insertion of the malicious content on that server. On the other hand, phishing messages are used to lure victims into executing files or into accessing links (client-side) that lead them to the malicious content.

To develop and improve protection mechanisms deployed on the client-side, it is necessary to study and more deeply understand malicious pages and programs. There are several systems that perform this kind of analysis, but they are focused either on Web or operating system (OS) malware.

One of the major problems toward malware analysis is the use of obfuscation techniques through packers. These packers change the code in a way that keeps the main functionality, but turns the manual or static analysis into a very hard task. Moreover, some packers insert blocks of code inside the obfuscated file to verify whether they are being executed on an emulated or virtual environment, thus hiding their malicious behavior in such cases [1], [2].

In this article, we propose a framework that obtains URLs and files from spam crawlers and malware collectors, and transparently analyzes them. This framework is capable of analyzing both Web and OS-based malware, while its modular design makes it possible to extend it to other file types or languages. We deployed a prototype and tested it against actual malware from our collected dataset, presenting results showing that our framework has some advantages over the existing systems that perform Web or OS-based malware analysis. The evaluation performed using Web malware shows that our detection rate is much better when compared to existing systems, due to our more effective selection of attributes and the dynamic execution of Web pages in a non-emulated system. Also, due to the way we capture the behavior of the OS malware, we can deploy the monitoring system in emulated, virtual or real environments, providing an advantage over existing malware analysis systems. In summary, the main contributions of this article are:

- We present a hybrid framework to analyze both Web and OS-based malware;
- Our tests show that our analysis of Web malware produce better detection rates than existing systems;
- The deployed OS behavioral monitor can operate in emulated, virtual or real environments, allowing our framework to correctly analyze samples that detect virtual or emulated environments.

II. Related Work

There are several analysis systems designed to monitor the behavior of Web or OS malware, as described below. However, each of them focus solely on one of the mentioned malware types. Below, we present the main systems and techniques that are used to analyze malware, to produce informative reports about them and, in the case of Web malware analyzers, to tell if the analyzed matter is malicious or benign.

A. *OS Malware analysis*

A malware behavior can be defined by the set of activities performed on the operating system by the execution of a malware instance. These activities include modifying files, writing registry values, performing network connections, creating processes etc. Malware analysis is the process applied to a malicious program in order to extract features that can characterize it.

Malware analysis can be performed in a static way, i.e. without executing the sample, or dynamically, by monitoring its execution. The use of packers makes static analysis a quite difficult and slow [3] process. Therefore, the most used systems for malware analysis follow a dynamic approach. Common techniques to dynamically extract malware behavior are: Virtual Machine Introspection (VMI), System Service Dispatch Table (SSDT) Hooking and Application Programming Interface (API) Hooking.

In the case of VMI, a virtual environment is used to execute the malware and restore the system after the analysis. Monitoring is performed in an intermediary layer, called Virtual Machine Monitor (VMM), which is interposed between the virtual system and the real one. This approach allows the extraction of low-level information, such as system calls and the state of memory. However, it also requires a virtual environment. Moreover, some types of malware try to realize whether they are under analysis, through attempts to detect a virtual environment. Thus, a particular malware instance might decide not to perform malicious actions [2] if a virtual environment is detected. VMI is used by the Anubis [4] system.

SSDT is a Windows kernel structure that contains the addresses of native functions. SSDT hooking is performed at kernel level by a specially crafted driver that modifies some of the SSDT addresses to point to functions inside this driver. These functions can change both the execution flow and values returned to programs. This technique can be used either in virtual, emulated or real environments as its flexibility is linked to the driver's mobility. However, there may be some issues related to rootkits' analysis, as they also operate at the kernel level and possess the same privileges of the monitoring driver. The framework presented in this article uses this technique to capture malware behavior at the OS level.

Another technique to monitor malware execution behavior is API hooking. It modifies the binary under analysis to force the execution of certain functions that are in the monitoring program before calling selected system APIs. As this technique is deployed at a level that is closer to the analyzed sample, it is possible to easily obtain higher-level information. However, this feature also makes it easy for a malware sample to detect the monitoring through integrity checking. Furthermore, malware can issue system calls directly from memory-mapped addresses, evading this kind of monitoring. This approach is used by CWSandbox [5].

B. *Web malware analysis*

Web malware analysis is usually performed through a component located in the operating system or in the browser. In both cases, the monitoring system verifies whether the analyzed Web page contains malicious codes or not and also provides some information about the captured behavior. The three most used (and publicly available) systems are JSand, PhoneyC and Capture-HPC, which are described below.

JSand [6] is a low-interaction honeyclient that uses a browser emulator to obtain the behavior of the JavaScript code present in a Web page. Then, the system extracts some features from the obtained behavior and applies machine learning techniques to classify the analyzed page as benign, suspicious or malicious. The main problems related to this approach are its limitation to JavaScript-only analysis and its inability to detect attacks that steal information from the browser.

PhoneyC [7] is another low-interaction honeyclient that uses a browser emulator to process the analyzed Web page and is able to analyze JavaScript and VBScript codes. To detect exploits in those codes, it emulates certain vulnerable components. The limitations are the same of JSand's, except for the added VBScript analysis.

Capture-HPC [8] is a high-interaction honeyclient that uses a full-featured browser and a kernel driver inside a virtual environment to extract the system calls performed by the browser as it accesses the analyzed page. Then, it performs a classification step (benign or malicious) based on these system calls. Capture-HPC can detect attacks independently of the script language that is used, but only those that generate anomalous system calls.

III. System Description

A. *Architecture*

The architecture of our proposed framework can be seen in Figure 1. Spam crawlers, malware collectors and manual input are the source of the samples, which are then forwarded to the *Selector* module. *Selector* is responsible for identification of the sample type and sending it to the appropriate module. Windows executable files—PE32 and DLL file formats—are sent to the OS module, whereas Web-related content, such as URLs, HTML and JavaScript files, are sent to the Web module. The OS module extracts the sample behavior and send it to the *Parser*, which processes it, extracts the high-level information and produces a report containing the analysis results. As for the Web module, the extracted behavior is fed to four different processors, each of them responsible for one type of malicious behavior detection. The *General classifier* collects the results of these four processors and produces a summarized information report. When the Web module detects an executable file, for example, due to a drive-by download, this is forwarded to the OS module, which returns the corresponding observed behavior.

B. *Collection*

Apart from manual insertion, malicious content is obtained by spam crawlers and malware collectors. The spam crawlers

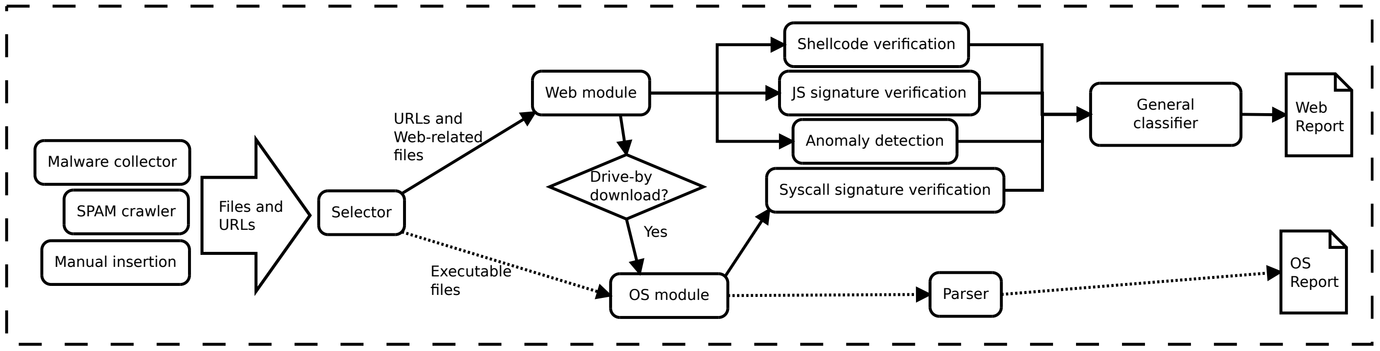

Fig. 1. Framework's architecture. Dark lines indicate the analysis flow from the Web related files and URLs whereas dotted lines indicate the analysis flow of OS executable files.

periodically fetch emails from purposely created accounts on collaborating sites. When a crawler finds a link or an attached file, it sends such file to *Selector*. Malware collectors are low-interaction honeypots (systems that emulate some vulnerable services) that collect samples by downloading them after attempted exploits.

C. OS module

The OS module is based on a Windows kernel driver and contains a pool of emulated and real environments. The SSDT hooking technique is used to monitor system calls performed by the analyzed sample and its children-processes. The captured actions are related to file, registry, sync (mutex), process, memory, driver loading and network operations.

When it detects the use of some packer that is known to cause problems in emulated environments or when the analysis in the emulated environment finishes with error, the sample is sent to analysis on a real system, i.e. neither emulated nor virtual.

D. Parser

The *Parser* processes the behavior extracted by the OS module and selects only relevant actions to feed into the analysis report. An action is considered relevant if it either causes a modification in the system state or incurs in sensitive data leakage.

E. Web module

The Web module performs its monitoring process through a Windows library (DLL - Dynamic Link Library) that hooks some functions from libraries that are required by the Internet Explorer browser. When one of the monitored functions is called, the execution flow is changed to a function inside the monitoring DLL. It then logs all the needed information and redirects the execution flow back to the original function.

We monitor actions that are executed by JavaScript code due to its wide usage in attacks as a client-side script language [9]. The monitored actions are related to string, ActiveX, decoding and array operations, DOM (Document Object Model) modifications, dynamic code execution and manipulation of personal information, such as cookies.

The actions that the Web module captures are then sent to the four detection modules available, each one responsible for one type of detection. The windows executable files obtained during the Web module's monitoring process are sent to the OS module, which then forwards the returned system calls the system call signature detector. Its results are used to classify the analyzed sample.

F. General classifier

Classification is performed in four steps: anomaly detection of JavaScript behavior, shellcode detection, JavaScript and system call signatures matching.

1) Anomaly detection: We extract eight features from the JavaScript behavior and use machine learning techniques to find malicious patterns. These eight features are the number and size of string definitions and strings inserted into arrays, the number of dynamic code execution calls and DOM modifications, the size of dynamically executed code, the number and size of possible shellcodes (explained later), the number of ActiveX objects created and the size of parameters passed to ActiveX functions.

We use the Weka framework [10]—the metaclassifier ThresholdSelection and the RandomForest classifier algorithm—to generate the anomaly detection classifier. This classifier, when used as a detection mechanism, can detect most of the attacks performed using the JavaScript language, even when the attack is not successful.

2) Shellcode detection: The results of JavaScript string operations, the strings embedded in array objects and the strings returned from decoding operations are verified by their mime-type. The ones with a mime-type that does not contain the string *text* are considered possible shellcodes. These possible shellcodes are verified using the libemu tool (http://libemu.carnivore.it) and, if positive, the page is considered malicious.

3) JavaScript signatures: JavaScript signatures are sets of regular expressions used to detect certain JavaScript operations and parameters. These signatures are used to detect known patterns of malicious actions. In the current version of our system they are only used to detect information stealing attacks, such as navigation history information.

4) System call signatures: System call signatures are used to match actions that should not be performed without the user's consent. As the dynamic analysis is performed in an automated way, without any human interaction, all system calls that should require user confirmation are considered malicious.

These signatures are formed by regular expressions that ultimately define whether a system call is considered allowed or not. This verification can detect successful attacks that result in malware installation, regardless of the script language used to carry the attack.

IV. Tests and Results

In this section, we present the Web and OS malware analysis results that validate our approach and compare it to similar available systems.

A. OS Malware Tests

We performed tests to demonstrate the effectiveness of the OS module in monitoring malicious behavior. Our module was compared to Anubis [4] and CWSandbox [5]. We chose those systems because they use different monitoring techniques, have a public submission interface and are among the most used and referenced systems for dynamic malware analysis.

For our tests we used 1,744 malware samples obtained from the collection mechanisms described earlier. We normalized the reports to a common format so we could compare them, as each system formats its results in a different way. The normalization was accomplished by the generalization of some parts of the reports and the removal of some actions that are not relevant to find malicious behavior, such as the listing containing all processes executing inside the analysis environment, which is returned by CWSandbox. The similarity between two reports is then calculated as the percentage of the smaller report contained in the larger one.

The tests are divided regarding malware that use or not packers embedded with anti-emulation features. In Figure 2, we present the similarity between reports for malware samples that do not make use of anti-emulation packers, whereas in Figure 3, the comparison is performed on malware that are packed with the anti-emulation packers *tElock!*, *Armadillo* and *ASProtect*. The identification of packers was performed with the PeID tool (http://www.peid.info).

A fast growth of the curves in the figures indicates greater similarity among the reports. By comparing them, we notice that our analysis reports are more similar to Anubis'. Also, the initial value of the Y axis is greater in these cases, indicating that a larger number of reports presents 100% of similarity. From the results with malware samples that do not contain anti-emulation packer, it is not possible to observe which system (ours or Anubis) has more reports that are similar to those produced by CWSandbox, because the curves are very similar.

In Figure 3, we observe that the curves that involve CWSandbox are different in their initial portion. Most of those 100% similar reports between Anubis and CWSandbox are due to malware packed with *tElock!*. Thus, in Figure 4 we see

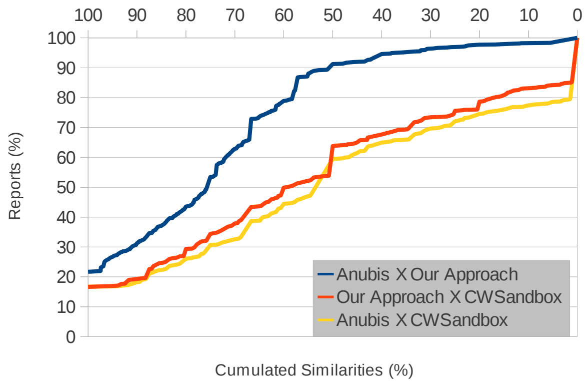

Fig. 2. Comparison using malware with no anti-emulation packers

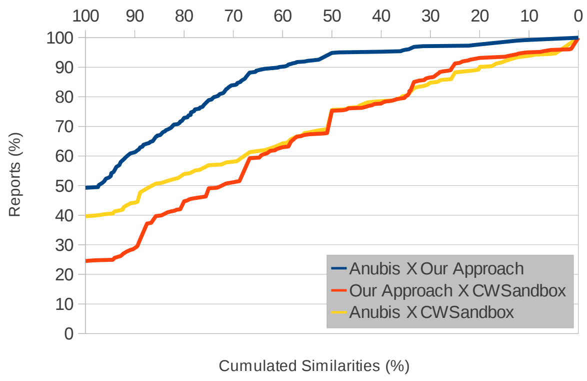

Fig. 3. Comparison using malware with anti-emulation packer

the similarity between reports related to malware packed only with *tElock!*. Also, most of the reports are in the 90–100% zone. We manually verified these reports and realized that Anubis did not correctly analyze them, i.e. there was only one action reported, representing a mutex creation, after which the analysis crashed. Only one of those Anubis reports produced extra information beyond the mutex creation; that was also present in the equivalent report from our module and from CWSandbox.

With this comparison tests, we showed that in cases in which malware samples do not contain anti-emulation packers, our OS module produced results similar to Anubis'. However, if the malware samples were packed with *tElock!*, Anubis could not successfully analyze them, whereas our module and CWSandbox produced valid results. Furthermore, CWSandbox produces very noisy reports, i.e. containing a great amount of irrelevant information toward analyzing malicious behaviors.

B. Web Malware Tests

We compared our Web module to three of the most widely used and publicly available honeyclients—JSand, PhoneyC and Capture-HPC— so as to demonstrate its effectiveness. In this test, we used 1,400 malicious HTML files and 6,781 benign URLs. We obtained the malicious files from domains hosting Web malware lists and from the VxHeaven database (http://vx.netlux.org). The benign URLs were obtained from the Alexa (http://www.alexa.com) site. Furthermore, we sent

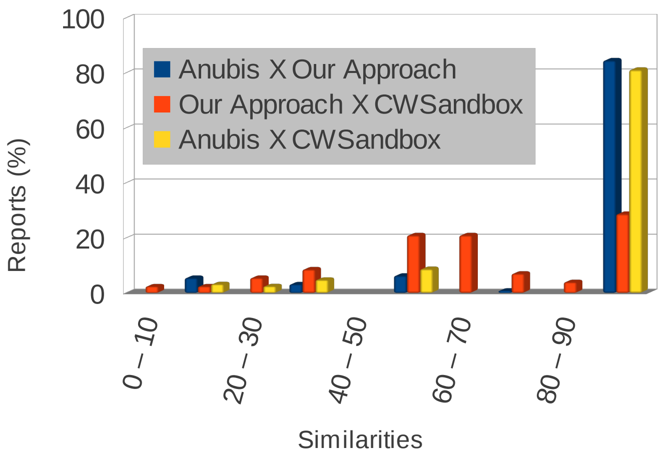

Fig. 4. Comparison using malware with the *tElock!* packer

	TP	FP	TN	FN	P	R	Hm
Our approach	76.6	0.4	99.6	23.4	99.4	76.6	86.6
JSand	17.1	2.2	97.8	76.2	88.5	18.3	30.3
PhoneyC	11.3	0	100	88.7	100	11.3	20.3
CaptureHPC	5.8	0.20	99.8	94.3	96.6	5.8	10.9

TABLE I
RESULTS OF THE TEST USING THREE WEB MALWARE ANALYSIS SYSTEMS AND OUR WEB MODULE, SHOWING TRUE-POSITIVES (TP), FALSE-POSITIVES (FP), TRUE-NEGATIVES (TN), FALSE-NEGATIVES (FN), PRECISION (P), RECALL (R) AND THE HARMONIC MEAN (HM), ALL IN PERCENTAGE (%)

the benign URLs to Google's safe browsing service and those reported as malicious were removed from the dataset.

We divided the malicious and benign datasets into "training" and "testing". The training datasets were used to train the anomaly detection engine, whereas the testing ones were used for classification. The ten-fold cross-validation of the training dataset resulted in 1.08% of false-positives (benign samples classified as malicious) and 22.83% of false-negatives (malicious samples classified as benign).

As it is hard to evaluate the systems based solely on the false-positive, false-negative, true-positive and true-negative rates, we also calculated the harmonic mean for quality measuring purposes. The harmonic mean considers the precision and the recall of the result. The precision represents the amount of samples classified as malicious that are really malicious and is calculated by $P = \frac{TP}{(TP+FP)}$. The recall represents the amount of samples previously classified as malicious that were correctly classified by the system and is calculated by $R = \frac{TP}{(TP+FN)}$. Finally, the harmonic mean is calculated by F-*measure* $= \frac{2 \times R \times P}{R+P}$. The results are presented in Table I and they show that our Web module has a much superior quality when compared to the other evaluated systems. Our harmonic mean value is about five times closer to the ideal 100% than the second highest scored system's. We credit most of the benefits in our approach to the use of more than one detection method and of a real browser, as emulated browsers can be easily detected and evaded [11].

V. Conclusion and Future Work

The analysis of Web and OS malware is very important to a better understanding of these threats and to the development of counter-measures. To this end, many systems were developed, but they all have some problems and do not analyze both types of malware. In this article, we proposed a framework that is able to analyze both traditional OS-based and Web-based malware, whose test results show the effectiveness of the approach against existing systems over the same malware samples.

The framework's Web module was compared to three of the most popular Web malware analysis systems. The results of this test showed that our module has a very good detection rate, several times closer to the ideal result than the second place's. In addition, our OS module was compared to two of the most popular malware analysis systems and the results showed that it produces reports that are more similar to Anubis' results in case of malware that are not equipped with anti-emulation packers. Moreover, our module can capture more behavioral information in cases of malware samples packed with *tElock!*.

We plan to expand the Web module to monitor other script languages, such as VBScript, and also to expand the OS module to analyze rootkits in a more adequate fashion.

References

[1] T. Raffetseder, C. Krugel, and E. Kirda, "Detecting System Emulators," in *ISC*, 2007, pp. 1–18.

[2] D. Quist and V. Smith, "Detecting the Presence of Virtual Machines Using the Local Data Table," http://www.offensivecomputing.net/files/active/0/vm.pdf.

[3] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," *ACSAC*, pp. 421–430, 2007.

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

[5] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, pp. 32–39, 2007.

[6] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 281–290.

[7] J. Nazario, "Phoneyc: A virtual client honeypot," in *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*. USENIX Association, 2009, pp. 6–6.

[8] C. Seifert and R. Steenson, "Capture - honeypot client (capture-hpc)," pp. Available at https://projects.honeynet.org/capture–hpc, 2006.

[9] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 88–106, 2009.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[11] F. Howard, "Malware with your mocha: Obfuscation and antiemulation tricks inmalicious javascript," 2010.