ANALYSIS AND DETECTION OF MULTIPLE MALICIOUS JAVA APPLETS BASED ON STATIC CODE ANALYSIS: A SURVEY

[1]SAPANA Y. SALUNKHE, [2]TAREEK M. PATTEWAR

[1]PG Student, Department of Computer Engineering, SES's R. C. Patel Institute of Technology, Shirpur, MS, India,
[2]Assistant Professor, Department of Information Technology, SES's R. C. Patel Institute of Technology, Shirpur, MS, India
E-mail: [1]sapana.salunkhe99@gmail.com, [2]tareekpattewar@gmail.com

Abstract- Malicious web content is a major attack vector on the Internet. Typically, the attacker's goal is to install and run a piece of malware on the victim's computer. The research community has not given to this problem the attention it deserves, and, the approaches to the detection of malicious Java applets are based either on simple signatures or on the use of honey-clients, which are both easily avoided. The proposed novel approach to the detection of malicious Java applets based on static code analysis. The approach extracts a number of features from Java applets, and uses supervised machine learning to produce a classifier. This approach is able to detect both known and previously-unseen real-world malicious applets. We can extend our analysis to consider multiple applets that appear on the same web page together.

Keywords- Code obfuscation, Disassembly, Honeyclients, Malicious Web Content.

I. INTRODUCTION

Internet is a technology that spreads faster than any other technology in the world and thus it has become part of our life. As now a day's most of our sensitive information is accessible through web browsers, securing these systems is of great concern, because there is a potential risk of attack to any computer hooks to networks. The risk is vast in an environment in which the software is downloaded across the network and executed locally as this is the case with the java applets. Applets are automatically (an applet may not give any visual signal of its existence or execution) downloaded across the network and run on the host machine when a user accesses in a browser a web page containing applet. A user coming across malicious applets is within the realms of possibility.

The Java Sandbox model provides you an environment, where you could welcome any code from any source. But when the code from an untrusted source runs Sandbox restricts the code from taking any actions that could possibly harm your system. It terms of applets the restrictions means that applets will be unable to determine information about each other; each applet is given its own memory space to run. Java has solved the problem of security and portability in advanced in this way that the output of a Java compiler is not executable code. It is the Java Virtual Machine (JVM) which helps to solve the major problems associated with downloading programs over Internet. The fact that a Java program is interpreted into an intermediate language (byte-code) is a part of Java language security. Because the execution of every Java program comes under the control of JVM, the JVM prevent it from modifying sensitive information in the user machine and harming the user machine. Malicious web content is a major attack vector on the Internet. Typically, the attacker's goal is to install and run a piece of malware on the victim's computer. The attackers try to lure users onto malicious web pages that contain malicious code. This code might trick the victim into downloading and running a malware program (through social engineering). Alternatively, the malicious code might try to exploit a vulnerable part of the victim's browser, such as an ActiveX control, the JavaScript engine, or the Java plug in. Attacks that exploit (browser or plugin) vulnerabilities when users visit malicious web pages are called drive-by download attacks. During the last two years, there has been a tremendous increase in Java applet-based attacks - more than 300. Applets that are digitally signed with a certificate (that is, certificates trusted by the user) run effectively outside the sandbox. The browser, encountering an applet with a signature, will usually display a dialog window asking the user if he trusts the applet's certificate or not. If the user accepts the certificate, the applet runs with full privileges, otherwise, it is sandboxed. An applet that is started from JavaScript remains always sandboxed.

Malicious applets try to escape the sandbox and install malware on a victim's computer. To achieve this, some applets try to trick careless users into trusting their certificates. Others target the JVM itself by trying to exploit a vulnerability in the Java plug-in, effectively disabling the sandbox and turning the applet into a full-privileged, non-restricted program with permissions equal to that of the user running the browser. We address the problem of malicious Java applets trying to exploit the Java virtual machine.

II. RELATED WORK

A lot of research has been done to detect malware. In this section, we present different approaches and

compare them to Jarhead. Traditionally, malicious content is recognized by matching programs (or data) against known malicious patterns, called signatures. Find malware by matching it against previously selected code or data snippets specific to a certain exploit. Signature-based detection systems can be evaded by obfuscation, and cannot catch exploits they do not have signatures for. Jarhead complements signature-based techniques by identifying malicious samples based specifically on their obfuscation and behavior features. Jarhead is also able to detect previously-unknown families of exploits (since it uses anomaly detection). Unfortunately, this approach is susceptible to obfuscation techniques that make malicious content look different and not match an existing signature. Additionally, this approach can only detect already-known attacks, and new signatures need to be added to recognize new variations of an exploit. Honeyclients are built as instrumented, automated browsers that visit web pages and monitor changes to the system. If abnormal behavior, such as creation of files or processes, is detected, the page is deemed malicious. This approach can only detect malicious behavior that the system was explicitly set up to detect. In particular, if the software targeted by an exploit is not running on the honeyclient, it cannot be exploited, and as a result, no malicious intent can be detected. It is a very tedious, human-intensive task to keep a honeyclient running with all the different versions and combinations of software packages to ensure that no exploits are missed. Thus, honeyclients can have high numbers of false negatives. Moreover, malware authors attempt to fingerprint honeyclients, by testing for specific features in environment it provides.

When such mechanisms detect a honeyclient, the code will not behave maliciously, and therefore, avoid detection.

A broad range of low and high interactive honeyclients were proposed to identify malware. They cannot detect malware that targets vulnerable components that are not installed on the honeyclient. Specifically for Java applets, this means that the honeyclients need to have the correct version of the Java plug in installed, running in the correct configuration with the correct browser. Jarhead is able to detect malicious applets independent of this environment by using static analysis. Furthermore, the runtime environment of honeyclients can be fingerprinted by malware as part of evasion attempts. Since Jarhead relies purely on static analysis, fingerprinting the analysis system is not possible. Twelve years ago, Helmer suggested an intrusion detection system aimed at identifying hostile Java applets. Their system is geared towards the detection of applets annoying the user, rather than real malicious ones as we see today. Their approach is also based on machine learning combined with anomaly detection, but the features are very different. In particular, their system monitors the system call patterns emitted from the Java runtime system during applet execution. The system has not been tested on real malicious applets and requires dynamic execution, exposing it to similar problems as honeyclients. Jarhead has been tested on a large real-world dataset of modern, malicious applets, and it is not subject to the limitations that come with dynamic malware execution.

III. METHODOLOGY

A. Overview of jarhead

We will describe design choices and provide a high level overview of its operation. Jarhead relies on static analysis and machine learning to detect malicious Java applets. Jarhead operates at the Java bytecode level. That is, Jarhead analyzes the bytecode that is part of a class file.

Java bytecode is the encoding of the machine instructions for the JVM. In addition to the bytecode, we collect strings from the constant pool. The constant pool holds strings and other data values, and it is also part of a Java class file. Usually, there is one class file per class of the program. Java class files can be seen as the equivalent of binaries in a specific format (such as ELF on Linux or PE on Windows). To analyze a Jar file, we extract its contents and then disassemble all class files contained in the Jar into a single (disassembled) file. Furthermore, we collect certain statistics about the applet: The number and type of files contained, its total size, and the total number of class files. Jarhead operates in two steps: First, we use a training set of applets - each applet known to be benign or malicious to train a classifier based on the features Jarhead extracts from applets. After the training phase, we use this classifier in a second step to perform detection of malicious applets.

Our analysis system has to be robust to obfuscation, so that attackers cannot easily evade detection by small changes to their exploits. It should ideally also not require reconfiguration or retraining when new vulnerabilities are exploited, to avoid the window of "blindness" when a new exploit surfaces. Furthermore, the system should be fast and not require human interaction. Jarhead's analysis focuses only on the Jar file (and the Java classes within), without looking at the surrounding web page or the interaction between the applet and its environment (and the browser). Thus, Jarhead does not require access to the parameters stored in the web page containing the applet, nor to other parts of the environment in which the Jar was deployed.

The advantage of this approach is that we do not require parts of the applet that are sometimes hidden by the attacker through obfuscation. In fact, often, parameters are used to provide shellcode (or a key used to decipher shell code strings) to the applet. When these parts are missing for an often sample, or they are successfully hidden from a dynamic analysis system during online processing, detection fails. We found that, very often, malicious Java applets require input from the surrounding environment. Unfortunately, this information is often missing from samples shared between security analysts and anti-virus companies. In these cases, a dynamic analysis approach (which is often used for analyzing web and script content) simply fails.

B. Limitations of Dynamic Analysis

1. Dynamic analysis systems are cannot start or even load the code successfully and cannot gain information from a program run.
2. Malicious code also uses fingerprinting to evade dynamic analysis, by not displaying a malicious behavior while analyzed.
3. Dynamic analysis through emulation works at a slower speed than execution on real hardware. Therefore, a malicious program can be written in such a way that it delays dynamic analysis by executing slowly within the analysis system. When an analysis run is terminated before the malicious activity starts, the malware will be falsely deemed benign.
4. Dynamic analysis systems is that they can only learn about the part of the program that is executed. Usually, the code coverage for one run of a program is not complete.
5. Due to time constraints, dynamic analysis systems often do not wait for even one full program run to finish. Jarhead does not suffer from this limitations and analyzes the entire code.

Jarhead is able to analyze malware samples that resist dynamic analysis systems in an efficient way, while being robust to fingerprinting. Furthermore, many Java vulnerabilities only apply to specific versions of the JVM. To be able to execute samples, even those that are available to the analysis system together with their external input data (i.e., the a forementioned cipher keys or shellcode strings), one would need to run these samples in different version, versions of the JVM, using different environments. Jarhead does not suffer from this problem since its analysis is static and environment-independent. Java bytecode was specifically designed to be verifiable and easy to parse. Static analysis, therefore, works well on bytecode, and does not suffer from a lot of the limitations common to other binary program formats, a Java program does not have the ability to dynamically generate code at runtime, which is usually a problem for static analysis techniques. Thus, even when attackers make use of code obfuscation, a static analysis approach can obtain sufficient information to correctly classify an unknown program.

IV. FEATURES OF JARHEAD

This section describes the features that we extract from a potentially-malicious Jar file and that we use in our classification process. We categorize our features and argue why they are difficult to avoid. Our features can be divided into seven categories. The first three categories address obfuscation; they form the group of the obfuscation features. The other four categories aim at exposing the purpose of an applet by statically examining its potential behavior. These four categories form the group of behavioral features.

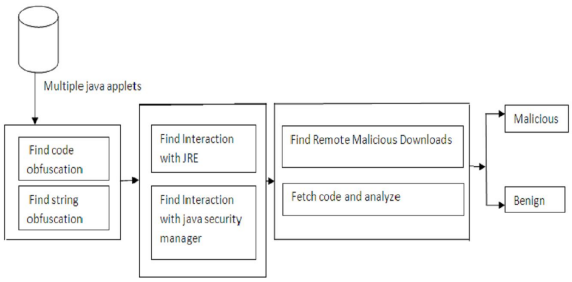


Figure 1: Malicious Java Applets Detection

A. Obfuscation features

Obfuscation is an important aspect for all malware today. Obfuscated code differs from other code because it is generated by obfuscation kits in an automated way. We use different code metrics as features to find differences between obfuscated and non-obfuscated code. Of course, obfuscation alone is not sufficient to identify a Java program as malicious, since obfuscation kits are also used by benign code for legitimate purposes. However, while manually examining many Java applets collected from the Internet, we found that obfuscation is overwhelmingly used by malicious applets. Furthermore we do not rely purely on the obfuscation features but our other features help in distinguishing benign obfuscated applets from malicious ones.

1. Code Metrics

We collect a number of simple metrics that look at the size of an applet, i.e., the total number of instructions and the number of lines of disassembled code, its number of classes, and the number of functions per class. In addition, we measure the cyclomatic complexity of the code. Cyclomatic complexity is a complexity metric for code, computed on the control flow graph (CFG). It counts the number of linearly independent paths through the code. A code that does not have branches has cyclomatic complexity 0. If the code has exactly one branch, the cyclomatic complexity would be one; if one of the two branches

contains another branch before the two paths merge, the cyclomatic complexity would be three, and so on. A measurement is computed per function, and we use the average of all measurements as one feature. For our analysis, a variable is dead if the variable is declared and dened at least once, but its value is never read (loaded) after the definition. A method or function is unused if it is defined within the applet, but never invoked by any code within the applet.

2. String Obfuscation

Strings are heavily used by both benign and malicious applets. However, the way these strings are used differs according to their purpose. While strings in benign applications are mainly used to display text to the user, strings in malicious applets are used for obfuscation and to store machine-readable shellcode. Such strings are often chopped up and stored in the data section of the applet, and then re- constructed at run time, to be used as function arguments, class names, function names or file names. The reason for string obfuscation is to defend against signature-based systems.

3. Active Code Obfuscation

The features in this category characterize code constructs that are often used to thwart static code analysis. These constructs are different in that they do not attempt to hide strings (or code fragments) from manual analyst or signature based systems, but, instead, try to mislead static analysis. To counter code analysis techniques that check for the invocation of known vulnerable library functions within the Java library, malicious applets frequently use reflection, a part of the Java language that allows for a program to modify its structure and behavior at runtime, to invoke these functions indirectly. Other (malicious) applets use reflection to instantiate objects whose type cannot be determined statically. If an object type is not known, it is generally impossible to statically decide which method is called on it. Additionally, malicious applets sometimes dynamically load parts of the exploit at runtime or use a key, passed in as a parameter, to decrypt shellcode. To detect such activity, we count the absolute number of times reflection is used in the bytecode to instantiate objects and to call functions.

B. Behavioral Features

The overwhelming majority of applet malware has a specific purpose: Escaping the Java sandbox to infect the end user's machine and make it a member of a botnet. The features in this section aim at statically identifying this behavior.

1. Interaction with Runtime and Security System As described previously, the JVM sandbox restricts applets by limiting the accessible resources. Several vulnerabilities in different versions of the Sun Java plug in have led to exploits that bypass the sandboxing mechanisms. Most of these exploits need to interact with the Java security mechanisms in different ways to be successful. Therefore, we collect features related to these security mechanisms. More precisely, we check if an applet calls into the Java runtime system (by using the Runtime class), interacts with the system security manager, or accesses system properties. We also check if the applet uses the System class to load additional code from the client system. Furthermore, we check whether the applet extends the Class Loader class in an attempt to introduce its own class loader, or implements the Class Loader Repository, Security Policy, or Privileged Action interfaces. These mechanisms are used to elevate applet privileges at runtime and can introduce code that runs without sandbox restrictions. Features that cover these sensitive core classes of the Java security mechanisms allow for anomaly detection of new exploits. The reason is that it is likely that new exploits will need to use these mechanics in some way to break out of the sandbox.

II. DOWNLOAD AND EXECUTE

We characterize the download behavior of applets by checking whether java.net.URL objects are instantiated or sockets are used. We also check for the use of functions that are able to write files. For a successful exploit, it is necessary to execute a file after it has been downloaded. There exist a number of different ways in which the Java library API can be used to spawn a new process from a binary, e.g., by using the java.awt. Desktop class, or, again, the system interface.

We have manually inspected all classes known to us that are able to spawn a new process and collected the API functions implementing this functionality. Since this kind of functionality is offered only by the Java library, we consulted its documentation for the relevant classes and listed the functions. We check if any of these functions are potentially used. By detecting all known possible ways for an applet to spawn a new process, we make it impossible for malicious applets to achieve this without triggering the corresponding detection feature.

III. JAR CONTENT

Benign applets are usually written with the goal of displaying something to the visitor of a web page or to play audio. Typically, the files necessary to do so (media files, images) are included in the Jar archive. We count the number of files in the Jar that are not Java class files. Furthermore, we check all files in the archive to see if any of them contain binary machine code, i.e., if there is an executable or library. In

addition, we use the total size of the Jar archive in bytes as a feature.

These features characterize benign properties rather than malicious ones. An attacker who attempts to mimic the structure of legitimate Jar archives is forced to increase the size of the exploit, which raises the distribution costs for successful campaigns.

IV. KNOWN VULNERABLE FUNCTIONS

Finally we also compiled a set of five well known vulnerable components of the Java library: Three individual functions, a combination of two functions, and one interface. The first two functions are MidiSystem.get Sound bank and the constructor for javax. management. remote. rmi. MIConnectionImpl. The combination of functions is MidiSystem. Get Sequencer, and Sequencer. add Controller Event Listener. Additionally, the javax. management. MBean Server interface can be implemented by an applet to exploit a known vulnerability.

V. OBJECTIVES

Our objective is to address the problem of malicious Java applets, a problem on the rise that is currently not well addressed by existing work. To this end, we have developed a reliable detector for malicious Java applets, which we call Jarhead. Jarhead uses static analysis and machine learning techniques to identify malicious Java applets with high accuracy. Our detection is fast and robust to obfuscation. It also requires little maintenance compared to signature-based detection systems and honeyclients; we do not need to collect signatures or maintain a realistic and complete runtime environment.

CONCLUSION

The proposed reliable detector for malicious Java applets called Jarhead. It can be used for overcome the problems of malicious Java applets by building a detection system based on static analysis and machine learning. A number of features extracts from Java applets, and would use supervised machine learning to produce a classifier. The detection would be fast and robust to obfuscation. This approach is able to reliably detect both known and previously-unseen real-world malicious applets. We can extend the analysis to consider multiple applets that appear on the same web page together.

REFERENCES

[1] Christopher Kruegel, Giovanni Vigna, "Jarhead Analysis and Detection of Malicious Java Applets", ACM, 978-1-4503-1312-4/12/12, 2012.

[2] Yaser Alosefer and Omer Rana, " Honeyware: A web-based low interaction client honeypot",ICSTW, vol. 10, 2010.

[3] Marco Cova, Christopher Kruegel, and Giovanni Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code", In World-wide web conference (WWW) , 2010.

[4] Cullen Linn and Saumya Debray, "Obfuscation of executable code to improve resistance to static disassembly", In Proceedings of the 10th ACM conference on Computer and communications security,CCS, vol.03, 2003.

[5] Jose Nazario, " Phoneyc: a virtual client honeypot", LEET, Vol.9, 2009.

[6] Martin Roesch, "Snort - lightweight intrusion detection for networks" , LISA 99, 1999.

[7] Christian Seifert and Ian Welch, "Identi_cation of malicious web pages through analysis of underlying dns and web server relationships" , in LCN, 2008

[8] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda, "Defending browsers against drive-by downloads: mitigating heap-spraying code injection attacks" in DIMVA 09, 2009.

[9] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna, "Analyzing and Detecting Malicious Flash Advertisements" ,In Annual Computer Security Applications Conference (ACSAC), 2009.

[10] Niels Provos and McNamee, "The ghost in the browser analysis of web-based malware", In Proceedings of the _rst conference on First Workshop on Hot Topics in Understanding Botnets., Vol 07, 2007.

★★★