Jarhead
Analysis and Detection of Malicious Java Applets

Johannes Schlumberger
University of California, Santa Barbara
js@cs.ucsb.edu

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Java applets have increasingly been used as a vector to deliver drive-by download attacks that bypass the sandboxing mechanisms of the browser's Java Virtual Machine and compromise the user's environment. Unfortunately, the research community has not given to this problem the attention it deserves, and, as a consequence, the state-of-the-art approaches to the detection of malicious Java applets are based either on simple signatures or on the use of honeyclients, which are both easily evaded. Therefore, we propose a novel approach to the detection of malicious Java applets based on static code analysis. Our approach extracts a number of features from Java applets, and then uses supervised machine learning to produce a classifier. We implemented our approach in a tool, called Jarhead, and we tested its effectiveness on a large, real-world dataset. The results of the evaluation show that, given a sufficiently large training dataset, this approach is able to reliably detect both known and previously-unseen real-world malicious applets.

1. INTRODUCTION

Malicious web content is a major attack vector on the Internet [32]. Typically, the attacker's goal is to install and run a piece of malware on the victim's computer, turning it into a member of a botnet. To this end, attackers try to lure users onto malicious web pages that contain malicious code [31]. 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 plugin. 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% in the first half of 2010 [15] alone. Interestingly, these exploits are widely ignored by security researchers so far [22]. This is despite the fact that commercial exploit toolkits such as "Blackhole" or "Bleeding Life" are using Java-based attacks to compromise large numbers of computers [22, 23].

A Java applet is a piece of software designed to be executed in a user's browser by a Java virtual machine (JVM [24]). Applets are embedded into web pages. They are automatically loaded and executed if the browser has a Java plugin installed (once the page has loaded), and graphical content is appropriately displayed by the browser. Applets predate cascading style sheets (CSS), and have typically been used for formatting and navigation purposes before more modern mechanisms became available (e.g., Flash, JavaScript, and HTML5). However, an applet can be any program and is not bound to a specific purpose.

Internet users are often not aware of the existence of applets and the Java plugin. Thus, they are not careful about keeping the plugin up to date, or care to disable it when visiting untrusted sites, even though multiple vulnerabilities exist and keep emerging with different versions of Java plugins [3, 4, 5, 6, 7]. As a result of this lack of awareness, Java plugins are still widely deployed, and about 85% [22] of all browsers today have the Java plugin installed and enabled. In addition, 42% of all browsers running with Java plugins have known vulnerabilities [21] and are thus susceptible to attacks [19, 22].

Traditionally, malicious content is recognized by matching programs (or data) against known malicious patterns, called *signatures* [2, 33]. Unfortunately, this approach is susceptible to obfuscation techniques that make malicious content look different and not match an existing signature [25]. Additionally, this approach can only detect already-known attacks, and new signatures need to be added to recognize new variations of an exploit.

Another approach to detect malicious web content is to use low interaction honeyclients [10, 30]. Honeyclients are built as instrumented, automated browsers that visit web pages and monitor changes to the system [1, 32]. 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 [34]. Moreover, malware authors attempt to fingerprint honeyclients, by testing for specific features in the

ACASC '12 Dec. 3-7, 2012, Orlando, Florida USA

environment it provides. When such mechanisms detect a honeyclient, the code will not behave maliciously, and therefore, evade detection.

To address the growing problem of malicious Java-applets, we have developed a system, called Jarhead, that statically analyzes Java applets. This system uses a set of features and machine learning techniques [17] to build a classifier that can decide whether an applet is malicious or benign. Our approach works independently of the attacked Java version and without recreating a realistic runtime environment. It allows us to analyze applets fast and efficiently and to categorize them as benign or malicious.

We have tested our system on real world data sets with more than 3,300 applets (collected both manually and from the public submission interface of Wepawet, a popular and publicly-available system for the analysis of web threats [8]). Our evaluation shows that Jarhead works with high accuracy and produces very low false positives.

The main contributions in this paper are:

- We 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.
- We executed Jarhead on a large collection of malicious Java applets collected in the wild. We found that our system detected malicious programs with high accuracy, and we were able to identify a new (zero-day) exploit against a previously unknown vulnerability.

2. BACKGROUND

Before we describe our system, this section provides some brief background on applets and the Java sandboxing mechanism [16].

Applets usually come packaged as archives (called Jar files), which consist of individual class files containing Java bytecode (for brevity's sake, we refer to both individual class files and archives of class files as Jar files or Jars throughout the paper). In addition to the Java class files, a Jar archive can contain arbitrary additional files. If the Jar contains an applet, it has to hold at least the files containing the applet's code. Additional contents typically include a manifest file, describing the starting point for execution of this applet, version information, or other package-related data in name-value format. Additionally, Jars often contain additional data needed by the applet, such as media files or copyright information.

To protect against malicious applets, the JVM contains (sandboxes) an applet and heavily restricts its permissible operations when it is not loaded from disk. Applets loaded over the network can, for example, only access resources on the remote host they were loaded from, and they cannot read certain system properties on the client side (such as usernames or the current working directory). These protection mechanisms are part of the same-origin policy [9] of the browser, designed to prevent untrusted sites from interfering with the user's communication to trusted sites. Moreover, a sandboxed applet cannot access client resources, such as the file system. By restricting the abilities of untrusted mobile code, its abilities to infect an end user's machine or to tamper with her data are severely limited. Furthermore, by preventing the applet from loading native code that is not verifiable by the JVM, creating a class loader, or changing the Java security manager, the JVM provides a safe execution environment. In the absence of bugs in the described sandbox implementation, this enables end users to safely browse the web even in the presence of sites that serve malicious Java code.

Applets that are digitally signed with a certificate (that is, certificates trusted by the user) run effectively outside the sandbox. In such cases, the previously-described restrictions do not apply. 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 plugin [3, 5, 6, 7], effectively disabling the sandbox and turning the applet into a full-fledged, non-restricted program with permissions equal to that of the user running the browser.

In this paper, we address the problem of malicious Java applets trying to exploit the Java virtual machine. Even though Jarhead does not defend against social engineering techniques, our system is able to identify applets used as a necessary part of these attacks as malicious.

3. JARHEAD SYSTEM OVERVIEW

This section introduces our analysis system, called 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, similar to binary code for "bare metal" machines. 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. We pass these statistics and the disassembled file to our feature extraction tool, which derives concrete values for a total of 42 features. Our full set of features will be discussed in more detail in the next section.

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. This enables Jarhead to work with in-vitro samples, similar to anti-virus software. The main 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 shellcode strings) to the applet. When these parts are missing for an offline 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. More precisely, dynamic analysis systems are unable to start or even load the code successfully and cannot gain information from a program run.

Malicious code also uses fingerprinting to evade dynamic analysis, by not displaying malicious behavior while analyzed. Additionally, 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. The same program can execute quickly on real hardware, successfully infecting the users browser. By using static analysis, 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 aforementioned cipher keys or shellcode strings), one would need to run these samples in different versions of the JVM, using different environments. Jarhead does not suffer from this problem since its analysis is static and environment-independent.

Another limitation for 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. Additionally, 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.

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 [28] common to other binary program formats, such as computed jumps. Furthermore, 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 (such as the one proposed in this paper) can obtain sufficient information to correctly classify an unknown program (as our results demonstrate).

4. FEATURE DISCUSSION

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 evade.

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*.

4.1 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. These kits chop up string constants and introduce additional statements or declarations without changing the semantics of the original code. 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.

4.1.1 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 [27].

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.

To find semantically useless code, we measure the number of dead local variables and the number of unused methods and functions. For our analysis, a variable is dead if the

variable is declared and defined 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.

The code metric features measure either the size of the applet or the quality of the code. Changing the size of the applet will either remove parts of the applet that were specifically added to guard against signature-based detection or will increase the size of the applet, and hence, increasing the distribution cost (bandwidth, time) for campaigns. Changing the code quality requires better code obfuscation kits (that do not simply produce dead code and variables) or even manual obfuscation, effectively raising the bar for the effort necessary to avoid detection.

4.1.2 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 reconstructed 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. We have features to model the length of strings in the constant pool, their number, and the fraction of strings that contain characters that are not ASCII printable. For the length feature, we determine the length of the shortest and longest string in the pool as well as the average length of all strings.

4.1.3 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. In addition, we check if the `Java.io.Serializable` interface is implemented by any of the classes belonging to the applet. The reason is that some malicious applets load a serialized object at runtime that contains otherwise hidden parts of the exploit.

In addition to using reflection, some malicious applets avoid instantiating objects (instances) of specific types (subclasses) altogether. While there is some legitimate use for instantiating instances of the base class `java.lang.Object`, this is not very common in benign applets, as programmers do not want to lose the benefits of a strongly-typed language. We check if arrays or single instances of `java.lang.Object` or `java.lang.Class` are instantiated.

It is also possible to interpret JavaScript from within applets, via a Java library routine. This is highly uncommon in benign applets, but used by malicious ones to carry out general purpose computation outside the scope of most Java analysis tools. Therefore, we check if the JavaScript interface is used.

4.2 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.

4.2.1 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 plugin 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 `ClassLoader` class in an attempt to introduce its own class loader, or implements the `ClassLoaderRepository`, `SecurityPolicy`, or `PrivilegedAction` 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.

4.2.2 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.

4.2.3 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.

4.2.4 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.getSoundbank` and the constructor for `javax.management.remote.rmi.RMIConnectionImpl`. We will discuss the third function separately, since it needs some more explanation. The combination of functions is `MidiSystem.getSequencer`, and `Sequencer.addControllerEventListener`. Additionally, the `javax.management.MBeanServer` interface can be implemented by an applet to exploit a known vulnerability.

The third, individual function that we check for is usually used in MIDlet scams. MIDlets are applications that use the Mobile Information Device Profile (MIDP) of the Connected Limited Device Configuration (CLDC) for the Java Micro Edition, typically used on cellular phones. Most of these MIDlets implement games or display a specific web page to the user. A vulnerability existed in certain phones that allowed tricking the user into sending text messages to expensive pay-per-call numbers. To this end, attackers had to call the text message send function of Java ME: `MessageConnection.send`. Note that apart from that specific vulnerability, there are a lot of MIDlets trying to trick the user into sending SMS by social engineering means. Java ME is no longer used in modern smart phones, such as the iPhone or Android. Older phones are not very valuable targets for infection, except for the mentioned text message scam.

Exploits that target well-known, vulnerable Java library functions have to call the aforementioned functions to be successful. We introduced five Boolean features, each of which is set to true if we find at least one call to one of the three functions, the combination of the two functions, or a class that implements the vulnerable interface. Of course, these functions (and interfaces) also serve a legitimate purpose. However, we found that they are rarely used by benign applets (and our other features help in making this distinction). To assert this we looked at more than 200 benign applets manually, we decompiled them and inspected their source code to check for malicious behavior and whether the description (if available) of their intended purpose matched what we found in the code. The benign applets we looked at were from a random crawl of the Internet, and two sites offering collections of applets. These data sets are described in more detail in the evaluation chapter. Their rare usage probably also indicates that they were not well-tested in the first place and thus the vulnerabilities were overlooked. If more vulnerable functions were to be added to this set later on, we would expect them again to be somewhat obscure and rarely used by benign applets. These few functions provide a way to break out of the sandbox without user interaction in vulnerable Java versions.

4.3 Feature Discussion

Jarhead collects a total of 42 features: Six are numeric (real values), ten are integers, and the remaining 26 are Boolean values. In the following paragraphs, we argue that our features are robust, in the sense that they make it difficult for an attacker to evade our analysis, while still achieving his goal. Furthermore we examine the usefulness of individual feature groups and list our top ten features.

Our features fall into two categories that capture both the behavioral and the obfuscation aspects of malicious applets. If an attacker tries to evade the obfuscation features, he will more likely be caught by traditional signature-based detection systems. An effort could be made to improve obfuscation techniques to match the code metrics of benign code very closely. However, in the presence of shellcode detection systems [12, 14], the shellcode within the applets, which is typically encoded in strings, needs to be well hidden. Furthermore, while active obfuscation techniques, such as reflection, are limiting static analysis, their use is not common in benign applications and hence, might even serve as indicators for maliciousness.

Even if an attacker finds a way to evade the obfuscation features, in order to achieve his purpose of infecting end users, his malware will necessarily implement functionality to achieve this goal. That is, the attacker needs to download and start malware on the victim's PC, and to do this, the code needs to break out of the Java sandbox. By capturing this behavior, intrinsic to the attacker's goal, we make it difficult for malicious code to evade our analysis and still succeed.

Previous systems [11, 13] for the detection of web-based exploits also use features to detect obfuscation and malicious behaviors. However, the concrete instances of these features are quite different. The main reason for these differences is that Jarhead uses static code analysis. Previous work on the detection of malicious JavaScript and Flash, on the other hand, relies on dynamic analysis. As a result, our novel features compute the presence and frequency of certain functions and resources, code patterns, and artifacts over the entire code base. Dynamic systems, on the other hand, typically compute the number of times certain events or activities occur during the execution of a program.

Answering the question how much impact do our top ten features (Table 1) have, compared to the full feature set we used a classifier with ten fold cross validation. We ran the classifier once with all our features and then with only the top ten features enabled on a total of 3372 applets, combining all our data sets. Again, the data sets are described in the evaluation chapter in detail along with more experiments. This experiment was run after the experiments in the evaluation section due to demand by our reviewers. Out of the 3372 applets the classifier working only with our top ten features misclassified a total of 122 (3.6%) applets, while our full feature set misclassified only 35 (1.0%) applets. This shows, that while our top ten features already perform reasonably well, our other features help to achieve even better results.

Merit	Attribute	Type
0.398	gets_parameters	behavior
0.266	functions_per_class	obfuscation
0.271	no_of_instructions	obfuscation
0.257	gets_runtime	behavior
0.254	lines_of_disassembly	obfuscation
0.232	uses_file_outputstream	behavior
0.22	percent_unused_methods	obfuscation
0.211	longest_string_char_cnt	obfuscation
0.202	mccabe_complexity_avg	obfuscation
0.197	calls_execute_function	behavior

Table 1: The ten most useful features, sorted by average merit.

In another experiment, we wanted to see how well the group of the obfuscation features performs without the behavioral features enabled and vice versa. To this end we used the same 3372 applets as before with ten fold cross validation, once with only the behavioral features and once with only the obfuscation features enabled. While the obfuscation features and behavioral features both performed reasonably well by themselves, with 119 (3.5%) respectively 150 (4.5%) misclassified applets they work best when used together. As stated before with all features enabled, we misclassify only 35 (1.0%) of the applets.

5. EVALUATION

In this section, we evaluate our system to measure how well it performs in detecting malicious applets. For our experiments, we used two different datasets: A manually-compiled dataset of applets (the manual set), and a set of samples collected and provided to us by the authors of the Wepawet system (the wepawet set).

5.1 Results: Manual Dataset

The manual dataset contains samples from four different sources: Two online collection of applets, an archive of mostly-malicious applets, and a number of manually collected samples. The two online collections[1] offer a selection of Java applets for web designers to use in their web pages. We crawled these two pages and downloaded a total of 1,002 Jar files. In addition, we obtained an archive of 357 Jar files with unknown toxicity from a malware research community site[2]. Finally, we manually searched for malicious applet samples on security sites[3], and we wrote a crawler that we seeded with Google search results on a combination of the terms "Java," "applet," and "malicious." This yielded another 1,495 Jar files, for a total of 2,854 samples. By combining applets from a broad range of different sources in this data set we tried to ensure it is diverse, representative of applets used in the wild and not biased to a specific kind of applet.

We first cleaned the manual dataset by removing duplicates, broken Jar files, and non-applets; i.e., we removed all Jar files that, after extraction, did not contain at least one class file that was derived from one of the applet base classes (`java.applet.Applet`, `javax.swing.Japplet` or `javax.microedition.midlet.MIDlet`). After this initial filter step, there were 2,095 files (of unknown toxicity) left.

To obtain ground truth for building and evaluating our classifier, we submitted the entire manual dataset to Virustotal. Virustotal is a website that provides free checking of suspicious files using 42 different anti-virus products. Virustotal found 1,721 (82.1%) of the files to be benign and 374 (17.9%) to be malicious (we counted a sample as benign if none of the Virustotal scanners found it to be malicious and as malicious if at least one scanner found it to be malicious).

Using the results from Virustotal as a starting point, we built an initial classifier and applied it to the manual dataset. In other words, we performed training on the manual data set using the results from Virustotal as ground truth. We then manually inspected all samples for which the results from our classifier and the results from Virustotal were different. We decompiled the samples and inspected their source code checking for known exploits that use certain functions, download and execute behavior. In the cases were documentation was available for a (probably) benign applet, we checked if the described behavior was found in the source code. Often checking the checksum of a jar file on Google provided a good indication if this was a known piece of malware. If after a while of manually inspecting the applet we were still unsure, we stuck with the verdict assigned by Virustotal. In this process, we found that Virustotal has actually misclassified 61 (2.9%) applets. In particular, Virustotal classified 34 (1.6%) benign applets as malicious, and deemed 27 (1.3%) malicious applets as benign. We manually corrected the classification of these 61 applets, and used the resulting, corrected classification as the ground truth for our dataset (with 381 malicious and 1,714 benign samples).

In the next step, we trained different classifiers on the manual dataset. More precisely, the manual dataset was split into a training set and a test set, using ten-fold cross validation. We evaluated C4.5 decision trees, support vector machines, and Bayes classification, which all showed comparable results. Eventually, decision trees turned out to be the most reliable classifiers, and they also provide good explanatory capabilities.

The detection results for the decision tree classifier were very good. With ten-fold cross validation, the classifier only misclassified a total of 11 (0.5%) samples. The false positive rate was 0.2% (4 applets), the false negative rate 0.3% (7 applets).

If we compare our detection results to the results produced by Virustotal, we find that our results are both better in terms of false positives and false negatives, and we see a reduction in the total number of misclassifications by a factor of six. This is in spite of the fact that we used Virustotal to build our initial labels, before manually adjustments (and hence, our ground truth might be biased in favor of Virustotal).

An overview of the results for the manual dataset are given in Table 2.

Discussion.

We examined the 11 instances (4 false positives, 7 false negatives) in more detail: One false positive was a potentially benign MIDlet, which had the ability to send text messages to arbitrary numbers. Two false positives were triggered by applets that did show suspicious behavior by trying

[1] `http://echoecho.com` and `http://javaboutique.internet.com`
[2] `http://filex.jeek.org`
[3] `http://www.malwaredomainlist.com`

	Virustotal (42 AVs)	Jarhead (10x cross-val.)
False pos.	1.6%	0.2%
False neg.	1.3%	0.3%

Table 2: Comparison of Jarhead and Virustotal misclassifications - note that our ground truth is biased towards Virustotal.

to write local files and executing commands on the windows command shell, but they were probably not intended to be malicious. The last false positive is an obfuscated applet that is performing calculations based on the current date, without ever displaying the results; its purpose remains unclear to us.

Two false negatives were applets that contained incomplete (broken) exploits. While these applets would not successfully execute exploits, we still count them as false negatives, since their intent is malicious (and we classified them as benign). We missed three additional malicious applets that were using reflection or loading additional code at runtime to protect against static analysis and that used an instance of CVE-2010-0094 [5]. CVE-2010-0094 is a flaw in the object deserialization that allows attackers to call privileged Java functions without proper sandboxing. The last two misclassified applets were instances of CVE-2009-3869 [4]. This particular stack-based buffer overflow is hard to catch for our system, since the vulnerable function it exploits within the native JVM implementation is not directly triggered by a specific function called from the bytecode, but, instead, is reachable from a large set of widely-used Java library functions.

Despite a few misclassified instances, we found that our system performs detection with high accuracy. In particular, some of the incorrect cases are arguably in a grey area (such as possibly benign applets that try to execute commands directly on the Windows command line and malicious applets that contain incomplete, half-working exploits).

We also examined the decision tree produced by our classifier in more detail. We found that our most important features, i.e., the ones that are highest up in the decision tree, include both features from our obfuscation set and our behavioral set. The top ten features include features capturing the interaction with the runtime, the execute feature, and the feature monitoring local file access. We also find the text message send function to be important. On the obfuscation side, we find size features, string features, and the number of functions per class to be the most effective at predicting malicious behavior.

We also tried to understand the impact of removing the feature that checks for known vulnerable functions (since this could be called signature-based detection). If this feature is removed, the misclassification rate increases only marginally, by 0.17%. This confirms that many features work together to perform detection, and the system does not rely on the knowledge of specific, vulnerable functions. To further emphasize this point, we draw the attention to the ability of Jarhead to detect zero-day attacks. After we finished the first set of experiments on the manual dataset, a new vulnerability[7], which was not part of our training set, began to see widespread use. When we tested our classifier on these completely new exploits, it was able to identify all five samples we acquired that implement this exploit, without any adjustment.

5.2 Results: Wepawet Dataset

To further test our classifier on real-world data, we collected Jar files from the Wepawet system. Wepawet is a public portal that allows users to upload suspicious URLs. Wepawet would then fetch the content at these URLs and attempt to detect drive-by download exploits. Since certain drive-by download exploit kits make use of malicious Java code, we expected to find some applets that Wepawet users on the Internet might have submitted to the system. Unfortunately, Wepawet does currently not support the analysis of Jar files, and hence, would consider all such files as benign.

The authors of Wepawet provided us with 1,551 Jar files that the system had collected over the past months. We removed duplicates, broken archives, and non-applets (as before) and ended up with a set of 1,275 applets. Again, we used Virustotal to obtain some form of initial classification. Virustotal found 413 (32.4%) applets to be benign and 862 (67.6%) applets to be malicious. We then ran our classifier on this Wepawet set. Compared to Virustotal, we assigned the same verdict to 1,189 (93.3%) samples, while 86 (6.7%) samples were classified differently. More precisely, we classified 59 (4.6%) malicious applets (according to Virustotal) as benign and 27 (2.1%) benign applets (according to Virustotal) as malicious.

Manual examination of the instances with different classifications revealed interesting results, both for the false positives and the false negatives. For the false positives, we found that 19 of the 27 were clearly errors in the initial classification by Virustotal (that is, these 19 applets were actually malicious but falsely labeled as benign by Virustotal). That is, Jarhead was correct in labeling these applets as bad. Interestingly, one of these 19 samples was a malicious applet that was stealing CPU cycles from the user, visiting a web page to mine bitcoins [29]. While we had not seen such behavior before, Jarhead correctly classifies this applet as malicious.

The remaining eight samples were properly classified by Virustotal as benign, and hence, false positives for Jarhead. Seven of our eight false positives had the potential to download files to the user's disk, and five of these seven would even execute these files after downloading! Except for intended use of these applets as software installers for potentially benign programs, their behavior (and code patterns) are essentially identical to malware. The last false positive was a (likely benign) MIDlet that can send text messages to arbitrary numbers.

We then inspected the 59 applets that Virustotal labeled as malicious (while Jarhead labeled them as benign). Four programs were partial exploits that did not implement actual malicious behavior. The remaining nine were false positives by Virustotal. We do not consider these partial exploits (which are essentially incompletely packed archives) and the nine benign programs to be properly labeled by Virustotal.

The remaining 46 samples were actual malware. They were largely made up of two families of exploits for which we had no samples in our (manual) training set, which we used to build the classifier. 28 samples belonged to a family of MIDlet scam applets, and another 15 belonged to a new, heavily obfuscated version of a new vulnerability (CVE-2011-3544) that was not present in our training set. We

also missed one instance of CVE-2009-3869 [4] (discussed previously) and one other malicious MIDlet. Moreover, one exploit used a new method to introduce code that was not in our training set.

Of the 46 false negatives, we missed 44 samples (or 96%) because of limitations with the manual dataset used to train our classifier. While we were collecting features useful for identifying these malicious applets, our classifier did not learn that they were important, because it was missing samples triggering these features in its training set. To show that we can achieve better results with a better training set, and to demonstrate that our features are indeed well-selected and robust, we trained and tested a new classifier on the Wepawet dataset using ten-fold cross validation. For that experiment, we found a total misclassification count of 21 (1.6%), the total false positive rate was 0.9% (12 applets), and the false negative rate 0.7% (9 applets).

The results for the Wepawet dataset are presented in Table 3.

	Original classifier	10x cross validated
False positives	2.1%	0.9%
False negatives	4.6%	0.7%

Table 3: Jarhead's performance on the Wepawet dataset.

We also collected performance numbers during our experiments. On average our analysis takes 2.8 seconds per sample, with a median of 0.6 seconds. This shows that the majority of samples is very fast to analyze, although there is a narrow longtail for which the analysis takes longer (specifically, 2% of the samples take longer than 10 seconds and 0.3% took longer than a minute). For the slower 50% of the samples, more than 98% of the total running time was spent in the disassembler. Thus, these numbers could be significantly improved simply by a more efficient implementation of the disassembler or by implementing our feature collection directly on the bytecode so disassembling the code becomes unnecessary.

6. POSSIBLE EVASION

We have seen that Jarhead performs well on real-world data. In this section, we will discuss the limitations of Jarhead, i.e., possible ways for malicious applets to avoid detection by our system.

A lot of the usual limitations for static analysis [28, 25, 26] do not apply to Java bytecode. However, for a trusted applet, it is possible to use the Java native interface (JNI) to execute native code on the machine. This is not covered by our analysis. If the parts of the malware that are implementing the malicious behavior are in the Java bytecode parts of the applet, it is likely that we will detect them, otherwise, there exist many analysis tools for native malware that would be able to detect such malicious behavior.

Static analysis is also limited by the use of reflection in a language. Interestingly, we found that reflection is not in widespread use by benign Java applets. Malicious applets, however, use it in an attempt to evade systems such as ours. While we do not completely mitigate this problem, we have features that aim to precisely detect this kind of evasion. Moreover, other features that target the Jar file and its code as a whole, such as code metrics and Jar content features, are unaffected by reflection.

We examine each applet individually. Applets on the same web page can communicate with each other, by calling each others public methods. Applets can also be controlled from the surrounding JavaScript in a similar fashion. If malicious behavior is distributed among multiple applets within a single page, or partly carried out by the surrounding JavaScript, our analysis scope is too limited, and we might misclassify these applets. Fortunately, to the best of our knowledge, malicious applets that use JNI and malicious code splitting behavior between multiple applets (or interacting with the surrounding JavaScript) do currently not exist in the wild. Moreover, we can extend our analysis to consider multiple applets that appear on the same page together. We already combine all class files within a single Jar archive, so such an extension would be rather straightforward.

While we have shown that today's malicious applets are very well covered by our features, a completely new class of exploits or vulnerabilities could bypass our detection either because we do not collect good features to capture the new exploit or because the classifier was unable to learn this exploit pattern from the training set. In these cases, it might be necessary to add new features or extend the set of known vulnerable functions. This would be straightforward to do. In other cases, simply retraining the classifier on a dataset containing the new exploits might suffice.

Since we operate on the Java bytecode, identifying vulnerabilities in the underlying native implementation of the JVM itself (such as CVE-2009-3869 [4]) is difficult. The reason is that corresponding exploits target a heap overflow vulnerability by displaying a specially crafted image. The set of possible functions within the Java API that can lead to execution of the vulnerable function is very large, and the API functions are widely used. Moreover, there is not obvious malicious activity present in the Java class file when this vulnerability is triggered.

7. RELATED WORK

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

Signature-based approaches [33, 2] 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).

A broad range of low and high interactive honeyclients were proposed to identify malware [30, 10, 11]. 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 plugin 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 [18]. 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.

8. CONCLUSIONS

We address the quickly growing problem of malicious Java applets by building a detection system based on static analysis and machine learning. We implemented our approach in a tool called Jarhead and tested it on real-world data. We also deployed our system as a plugin for the Wepawet system, which is publicly accessible. Our tool is robust to evasion, and the evaluation has demonstrated that it operates with high accuracy.

In the future, we plan to improve our results by using more sophisticated static analysis techniques to achieve even higher accuracy. For example, we would like to use program slicing [20] to statically determine whether a downloaded file is indeed the one that is executed later in the program or whether a suspicious image file is actually passed to a vulnerable function.

9. ACKNOWLEDGMENTS

This work was supported by the Office of Naval Research (ONR) under Grant N000140911042, by the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and by Secure Business Austria.

10. REFERENCES

[1] Capture hpc. http://nz-honeynet.org.

[2] Clamav. http://www.clamav.net.

[3] CVE-2009-3867. National Vulnerability Database.

[4] CVE-2009-3869. National Vulnerability Database.

[5] CVE-2010-0094. National Vulnerability Database.

[6] CVE-2010-0842. National Vulnerability Database.

[7] CVE-2012-0507. National Vulnerability Database.

[8] Wepawet. http://wepawet.iseclab.org.

[9] Same origin policy. http://www.w3.org/Security/wiki/Same_Origin_Policy, 2010.

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

[11] 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.

[12] 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.

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

[14] Y. Fratantonio, C. Kruegel, and G. Vigna. Shellzer: a tool for the dynamic analysis of malicious shellcode. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*.

[15] Mike Geide. 300% increase in malicious jars. http://research.zscaler.com/2010/05/300-increase-in-malicious-jars.html, 2010.

[16] Li Gong and Marianne Mueller e.a. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. USITS, 1997.

[17] Hall and Mark e.a. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1), 2009.

[18] Guy G. Helmer and Johnny S. Wong e.a. Anomalous intrusion detection system for hostile java applets. *Journal of Systems and Software*, 55(3), 2001.

[19] Stefanie Hoffman. Microsoft warns of unprecedented rise in java exploits. http://www.crn.com/news/security/227900317/microsoft-warns-of-unprecedented-rise-in-java-exploits.htm, 2010.

[20] Susan Horwitz and Thomas Reps e.a. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12, 1990.

[21] Wolfgang Kandek. The inconvenient truth about the state of browser security. http://laws.qualys.com/SPO1-204_Kandek.pdf, 2011.

[22] Brian Krebs. Java: A gift to exploit pack makers. http://krebsonsecurity.com/2010/10/java-a-gift-to-exploit-pack-makers, 2010.

[23] Brian Krebs. Exploit packs run on java juice. http://krebsonsecurity.com/2011/01/exploit-packs-run-on-java-juice/, 2011.

[24] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[25] 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 '03, 2003.

[26] Douglas Low. Java control flow obfuscation. Technical report, 1998.

[27] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4), 1976.

[28] Andreas Moser and Christopher Kruegel e.a. Limits of static analysis for malware detection. In *ACSAC*, 2007.

[29] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.

[30] Jose Nazario. Phoneyc: a virtual client honeypot. LEET'09, 2009.

[31] Niels Provos and McNamee e.a. The ghost in the browser analysis of web-based malware. In *Proceedings*

of the first conference on First Workshop on Hot Topics in Understanding Botnets, HotBots'07, 2007.

[32] Niels Provos and Panayiotis Mavrommatis e.a. All your iframes point to us. *Google Inc*, 2008.

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

[34] Christian Seifert and Ian Welch e. a. Identification of malicious web pages through analysis of underlying dns and web server relationships. In *LCN*, 2008.