Vulnerability of "Secure" Web Browsers *

F. De Paoli, A. L. Dos Santos, R. A. Kemmerer

Reliable Software Group
Computer Science Department
University of California
Santa Barbara, CA 93106

May 26, 1997

Abstract Today the World Wide Web is considered to be a platform for building distributed applications. This evolution is made possible by browsers with processing capabilities and by programming languages that allow web designers to embed real programs into HTML documents. Downloading and executing code from anywhere on the Internet brings security problems along with it. A systematic and thorough analysis of security flaws in the browsers and related technology is necessary to reach a sufficient level of confidence. This paper presents some preliminary results of ongoing research that has the final goal of developing properties for secure browsers and procedures for secure browsing. The research started by investigating features provided by the standard environment. The paper describes some experimental attacks that have been carried out by exploiting features of Java and JavaScript executed by Netscape Navigator and Microsoft Explorer browsers.

1 Introduction

The growth of the Internet and the World Wide Web (WWW) during the past few years has been phenomenal. The Internet is currently serving tens of millions of people connected through millions of computers. Most every business and government institution has a web page, and the web and web browsing are fast becoming the primary source of information for people of all ages.

Languages like Java and JavaScript have been developed to embed programs into HyperText Markup Language (HTML) documents (pages). Java applets, which are designed to be downloaded from the web and run directly by the Java virtual machine within a browser, are also increasingly being included in

*To be published in the Proceedings of the National Information Systems Security Conference, October 1997.

web pages to provide more sophisticated animation and other desirable features. Downloading and executing code from anywhere on the Internet brings security problems along with it. That is, the host computer is open to a variety of attacks, ranging from attacks that simply monitor the environment to export information, to attacks that change the configuration or the behavior of the host (changing files, consuming resources), and finally to attacks that open a back door to let intruders get into the host.

Attacks succeed either because the implementation of the browser is weak (from flaws in the specification and/or from poor implementation) or because the environment in which the browser is executed has flaws. Weaknesses in the environment are mainly due to the new open nature of software. Conventional computing paradigms assume that programs are installed and configured once on any and every machine and that these programs only exchange data. This means that a user can make all possible checks over a new program before running it. This assumption is no longer valid for open and mobile environments, such as Java and the web.

Many security holes in web browsers have been discovered and brought to the attention of the public [DFW96]. Although most of these holes have been patched, a systematic analysis of the features of both the Java and JavaScript languages and the web environment is needed to identify possible design weaknesses in order to avoid similar problems.

Most of the attacks carried out so far require the attacker to have substantial capabilities, such as taking over a web server or a domain name server or assuming that the user will navigate only through hyperlinks. These assumptions, therefore, often limit the likelihood of the attacks due to the low probability that the assumptions would be satisfied or because they require so much knowledge that only highly-skilled programmers could implement the attacks.

The purpose of this paper is to describe some experimental attacks that have been carried out by exploiting only features provided by the standard environment. What sets these attacks apart from most of the others is that the attacks presented in this paper can be realized without assuming that the attacker has any capabilities beyond those granted to every user on the net and without assuming that the attackers are highly skilled programmers with complex programs for carrying out the attacks. That is, the attacks reported in this paper are straight forward to set up and are of limited complexity.

This paper reports on two attacks. The first one deals with the violation of privacy, and it requires only that the victim downloads a page that includes an attacker applet, which collects and sends information back to its server. The second experiment is as powerful as a "man-in-the-middle" attack, but it too requires only that the victim downloads a honey-pot page that includes an applet that can detect when the victim visits a certain site and displays an imposter page for that site in order to steal sensitive information, such as credit card numbers or personal identification numbers.

The paper is structured in the following way. Section 2 summarizes the browser and language features that have been used. Section 3 briefly reviews some previously known attacks. Section 4 gives more details on applet capabilities and features to establish the background for the attacks. Section 5 describes the new attacks. Finally, future work and conclusions are presented.

2 Browsers and Languages

The World Wide Web is a hypertext of network-accessible information. It was designed to support static links to display static multimedia documents. The quick growth of the web and the related technology has changed this initial view. Today the web is considered a platform to build distributed applications. This evolution is made possible by browsers with processing capabilities and by programming languages that allow web designers to embed real programs into HTML documents.

This section outlines the characteristics of Java and JavaScript, which are the most popular languages for programming the web, and of Netscape Navigator and Microsoft Explorer, which are the most popular browsers.

2.1 Java

The Java language is a general-purpose object-oriented language that was introduced by Sun Microsystems in 1995 [GJS96a]. One of the major design goals for Java was portability. The result is that not only the Java source code, but also the binary code is executable on all processors. This is accomplished by compiling the source code into platform independent bytecode, which is then run by the Java virtual machine.

Some of the features of the Java language that make it simpler and supposedly more secure are that it is strongly typed, there are no preprocessor statements (like C's #define and #include), there are no pointers, no global variables, and no global functions. By keeping the language simple and without many of the error prone features, such as multiple inheritance, it is expected that Java will be more secure.

A Java program is a collection of classes and instances of classes. Each class is compiled into an intermediate format, called *bytecode*, which is then interpreted to execute the program. A major characteristic of Java is that pointers are not supported; *object references* are provided instead. Java supports dynamic creation of instances and bindings. When a class instance (an object) is needed, it is created explicitly and a reference to it is returned; when a method is invoked on an object, the interpreter selects the method to be executed according to the class hierarchy and method overloading. Object destruction is automatically handled by a *garbage collector*, so that memory management is completely in the control of the interpreter.

Another feature of Java is the support for concurrent programming via *threads*. Threads allow programmers to associate an independent execution flow with each class. A class with a thread can be started, stopped, and suspended independently from the execution of the rest of the system of which it is part. Synchronization among thread executions can be accomplished by class *monitor*s.

As mentioned above, Java code was designed to run on any client; therefore, compiled Java programs are network and platform independent. The absence of physical pointers and automatic memory management help achieve this independence. Moreover, the bytecode has been designed to fully support the typing mechanism of Java so that dynamic code verification can be performed. This is a safety and a security feature designed to prevent one from executing corrupted or malicious code.

The *Java Virtual Machine* is emulated in software and can run on numerous platforms [LY96]. It could also be compiled or implemented directly in microcode or hardware, but currently it is mostly emulated in software. The virtual machine deals with class files, which contain Java virtual machine instructions, a symbol table, and a few other necessary items. Java virtual machine instructions are all one byte long, and that is why they are called bytecodes. Bytecode can also be generated from other high level languages, such as Ada or C, or it could be generated manually.

When Java classes are downloaded from the network it is necessary to use a class loader. Java supplies an abstract `ClassLoader` class for this purpose. Because abstract classes cannot be used directly, each browser needs to declare a subclass of this class to be used by the browser for downloading classes. Each subclass must include a customized implementation of the `loadClass()` method to retrieve and download a class from the net. A class is downloaded as an array of bytes that must be converted to an instance of class `Class`. That is, the array of bytes must be translated to the structure of a class. The `ClassLoader` method that actually does the conversion is `defineClass()`. Every class object contains a reference to the class loader that defined it, so related classes can be downloaded by the same class loader. These features make Java suitable for writing programs in a distributed, heterogeneous environment such as the web.

Besides supporting the design of complete applications, Java supports the implementation of small applications, called *applet*s, which can be embedded into HTML pages. A special tag in a downloaded HTML file tells the web server to download the bytecode for a Java applet, and the most popular web browsers, such as Netscape Navigator and Microsoft Explorer, include support for the execution of applets. Applets can be used for implementing small stand-alone applications, as well as for implementing clients of client/server applications.

A page may have more than one applet embedded in it. In this case, all of the applets are downloaded together. Because a thread is associated with each applet, more than one applet can run concurrently within the same web-page context. The class `Applet` defines a set of methods to control applet behavior. These methods can be overridden by programmers to get the desired behavior from their applets. The most important methods are `init`, `start` and `stop`. When an applet is downloaded it is automatically created, initiated and started: the methods `init` and `start` are invoked and the applet is labeled *active*. When the page that embeds an applet is not displayed (the browser is iconified or shows another page), the execution of that applet is suspended (i.e., the method `stop` is invoked and the applet is labeled *inactive*). If the page is shown again, the applet execution is resumed (i.e., the method `start` is invoked again and the applet is labeled *active*). It is possible to keep an applet running even when it is inactive, however. This is accomplished by overriding the `stop` method so that it never returns. One of the experiments described in this paper exploits this feature.

2.2 JavaScript

JavaScript is an object-based scripting language that has been specifically designed by Netscape to embed executable statements (scripts) into HTML pages. JavaScript resembles Java, but without Java's static typing and strong type checking. Scripts are embedded directly into an HTML page and interpreted by the browser. By invoking JavaScript functions, it is possible to perform actions (such as playing an audio file, executing an applet, or sending an e-mail message) in response to a user opening or exiting a page.

In the experiments described in this paper, the JavaScript support for forms has been exploited. The use of forms enables the browser to send information back to the server. A form is an HTML tag that includes an `ACTION` field, which specifies the action to be performed, an `INPUT` field, which specifies a simple input element, and other fields, which are not relevant to this discussion.

The `ACTION` field must be filled out with a URL, which usually refers to a Common Gateway Interface (CGI) program. CGI technology is a standard for interfacing external applications with Web servers, and it allows users to start an application on the server. Another valid URL for the `ACTION` field is `"mailto:any_e-mail_address"`, which causes an e-mail message to be sent by the browser to the specified address.

The `INPUT` field is composed of a set of fields like `NAME`, `VALUE` (to store the input string) and `TYPE`. One example of type is `"submit"`, which specifies a *pushbutton* that causes the current form to be packaged up into a query URL and sent to a remote server. JavaScript implements a set of objects to handle different types of forms. Of particular interest is the *submit object*, which has a pre-defined method `click()` that simulates a click on the submit button. By writing a JavaScript statement that invokes that method, it is possible to submit a form automatically, without obtaining the user's acknowledgment. If the specified `ACTION` is `"mailto:..."` the effect is to send an e-mail message from the user without the user knowing it.

2.3 Browsers

Both Netscape Navigator 3.01 and Microsoft Explorer 3.02 support the execution of Java applets and JavaScript. Their implementations of the support of the Java virtual machine and JavaScript interpreter differ, but they support the requirements of these languages. Moreover, they both support the "cookie" technology. Cookie technology was developed by Netscape to make up for the stateless nature of web communications. A cookie is a small piece of information that is stored by the browser on server request and that can be sent to the server at any time. A browser is sent a request to set a cookie the first time it asks the server for an HTML page. Cookies are useful to maintain related information during long browsing sessions, such as maintaining shopping baskets and remembering configuration preferences. Although cookies have been introduced to improve the quality of the service provided by the web, they can also be exploited for malicious purposes, as will be described in the next section.

Both browsers also give the user options to increase the security of the browser. These options are:

1. They support the option to turn the execution of Java and/or JavaScript on or off. This enables the user to trade the convenience of these languages for the security of not running any outside program in the user's machine.

2. They support the option of alerting the user before accepting a cookie and before sending a form in an insecure environment. This lets the user choose whether to continue the operation or not.

3. They support certificates that are digital signatures. These currently are not widely used, but they are likely to be used more in the future.

4. Netscape Navigator has an option to alert the user before sending forms by e-mail. The attacks reported in this paper were originally exercised on Netscape Navigator 3.0, and this option did not work properly there. This feature does work on Netscape Navigator 3.01.

5. Microsoft Explorer allows the user to set three levels of security (high, medium, low) against active contents (which is Microsoft's term for Java and JavaScript). Although this at first seems like a good feature, Microsoft Explorer fails in not explaining in an understandable way what the difference among the levels is.

In carrying out these experiments it was necessary to constantly be aware of the differences between Netscape Navigator and Microsoft Explorer. For instance, with Netscape Navigator, one instance of the Netscape Navigator program can show many windows. Microsoft Explorer, on the other hand, is able to show only one window for each instance of the program. The importance of this is that different program instances are isolated from each other by the features of the operating system that isolate two programs. As a result, these different instances do not share any environment or context. In the remainder of this paper the word browser is used to refer to the browser's windows that are spawned by the same instance of a browser program.

3 Brief Review of Some Already Known Attacks

There are many known security vulnerabilities in web browsers. The attacks that exercise these weaknesses vary in sophistication and in the amount of resources that the attacker is required to possess in order to carry out the attack.

In this section a few representative attacks are presented. First two very simple though effective attack types, which require only a browser and uses standard language features, are presented. Next, an attack that requires the attacker to control a Domain Name Server that advertises wrong addresses is described. This is followed by the discussion of an attack that relies on common assumptions for firewalls to force a connection to a protected port. Finally, the attack known as "web spoofing" is presented to show how an attack may require strong assumptions and a high level of sophistication.

One of the simplest forms of browser attacks and also one of the hardest to stop is a denial of service attack. The attacks reported here use Java features to exhaust resources of the host system or prevent the use of some feature. Two obvious examples of this attack are busy-waiting to consume cpu cycles and repeatedly allocating memory until all of the available memory is exhausted. Another attack in this category is locking a class to prevent access to particular resources (e.g., locking the InetAddress class in Netscape Navigator blocks host name lookups). The reason that the denial of service attacks are difficult to stop is that there is no regulation of how much of a resource an applet can consume. That is, an applet is either given as much as it wants or it is given nothing. More of the technical details, which help to understand this attack are given in the next section. Further information on this type of attack can be found in [DFW96].

Another simple attack deals with the use of cookies to keep track of a user's activities, which turns out to be a privacy violation. This attack is described in [Ste97] and reports a real situation that has been set up by DoubleClick Corporation. DoubleClick has a network of affiliated sites each of which agrees to having a space in their HTML pages for advertisements that points to the DoubleClick site instead of showing a local graphic file. The first time DoubleClick receives a request for an advertisement by a browser, it selects an advertisement from a list and sends it back along with a request for setting a cookie with a unique identification number. That number will be sent to DoubleClick at any further contact to let them build a user profile reporting every DoubleClick page and site that is visited. Even though the profile does not report any information about the user identity (only computer names are recorded), it allows DoubleClick to understand a user's tastes and interests. This information can be exploited for marketing purposes, such as sending each user (or computer) personalized advertisements and rating the effectiveness of the advertisements. To prevent this attack, a user should first check the presence of a DoubleClick cookie in his/her computer (its name is `ad.doubleclick.net`) and delete it if necessary. Next, the user should turn on the browser option that alerts the user whenever a cookie is to be placed, and the user should refuse DoubleClick cookies.

The Domain Name Server (DNS) attack assumes that the attacker has control of a DNS, which a trusting host (or even a firewall) uses to validate a server address. In this attack the applet requests a connection to the server by providing the name of the server. The malicious domain name server returns the address of a machine other than the one requested by the applet (e.g., the attacker's machine). The result is that the applet can then connect to an arbitrary host. Netscape fixed this problem in Navigator 2.01 and 2.02 by looking up the IP address for each site they process and refusing to listen to (possibly false) updates to that address [Ros96].

The next attack relies on the common assumptions that a firewall trusts all requests originating behind it and that only the first packet of a connection is checked. To start the attack the applet sends the originating server a connection request including a protected port number (e.g., the port for telnet connections). The firewall trusts the request, since it originates behind it. Subsequent messages from the server are also trusted, since they are related to an already approved connection. The effect is that the attacker can open any connection even though there are firewall controls. This attack highlights security flaws that derive from assumptions that are no longer valid for web-based computation. A simple though partial solution is to design browsers that prevent one from issuing requests related to well-known ports (e.g., port numbers below 1024). More sophisticated firewalls already implement more complex filtering rules to ensure better control. A complete description of the firewall attack can be found in [MRR97].

Another interesting attack is the net spoofing attack, which assumes that the attacker has a way to make the user connect to the attacker machine, which delivers rewritten pages in such a way that each URL refers to the attacker machine. The attacker machine works as a "man-in-the-middle." This requires the attacker to have control of the network such that everything that the browser thinks it is sending to the web actually goes to the attacker machine. The attacker may then forward the request to the real machine and relay the results back to the victim, while making copies of the interesting parts before passing them on. This attack is fully described in [FBD96].

The second experiment described in this paper was motivated by the description of the man-in-the-middle attack in [FBD96]. Section 5.2 describes the new attack in detail. The goal for the new attack was to achieve an attack with the same power as the attack in [FBD96], but without assuming that the user will follow only hyperlinks and with a much simpler approach, which targets only particular sites of interest. Although in [FBD96] it is claimed that JavaScript could be used to fake the location line of Netscape Navigator, efforts to verify this attack were unsuccessful, other than by opening a new browser as an empty window, as is described in the Future Work Section of this paper.

4 Applet Security

Java applets are designed to be downloaded from the web and to be run directly by a Java virtual machine within a browser. Since applets are automatically run by the browser just by accessing a web page that contains a reference to the applet, security concerns should be alleviated before users can be expected to accept the concept of running applets from untrusted sources. That is, special security measures should be taken to protect against security, privacy, and denial of service threats. Therefore, it is necessary for the browser that contains the Java virtual machine to limit the downloaded applet. These limitations are not normally placed on applets loaded from the local file system, because the local host is trusted.

This section examines some privacy and security features related to applets. The goal is to understand what interaction is allowed between an applet and its environment (i.e., between an applet and the host machine, the host browser, or other applets running in the same browser). Section 4.1 illustrates what system properties an applet can access, Section 4.2 describes what the *context* of an applet is, and Section 4.3 deals with thread-related features.

4.1 System properties

Since applets are programs to be executed in the host environment, the Java designers posed some restriction on the interaction between an applet and its environment. The intent is to prevent safety, security and privacy violations. The restrictions that apply to applets are:

- An applet is prevented from reading or writing files on the host that is executing it.
- An applet can make network connections only to the host that it came from.
- An applet cannot start any program on the host that is executing it.
- An applet is prevented from loading libraries or defining native methods.
- An applet can read only a restricted set of system properties. In particular, it can read the file separator (e.g., '/'), the path separator (e.g., ':'), the line separator, the Java class version number, the Java vendor-specific string and URL, the Java version number, and finally the operating system name and architecture.

The idea is that by placing these limitations on downloaded applets the applets are effectively placed in a "sand-box". The applet may do whatever it wants inside the sand-box, but it is limited as to what it can do on the outside.

These restrictions are enforced by the *class loader* and the *security manager*. The class loader implements the downloading rules for classes. Due to the above restriction, this means that an applet cannot download classes other than from the host it came from.

The Java virtual machine also limits the name space of downloaded applets. When a Java virtual machine tries to load a class from a source other than the local host machine a class loader, which cannot be overridden by the downloaded applet, is invoked. The class loader sets up a separate naming environment to accommodate the source. This will assure that name clashes don't occur between the names in this source and the names in classes loaded from other sources. Bytecode loaded from the local file system is set up in the system name space, which is shared by all other name spaces, and the system name space is always searched first to prevent system classes from being overridden.

After the name space is set up, the class loader calls a *bytecode verifier* to assure that the bytecode has proper structure. This is necessary because the bytecode could have been generated by an incorrect Java compiler, by a compiler altered to skip the compile time checks, or by a non-Java compiler. The bytecode also could have been altered after it was produced. The bytecode is verified in four passes and is claimed to satisfy the following conditions if it passes the verification.

- The downloaded bytecode has the proper format for a class file
- All "final" classes are not subclassed, and all "final" methods are not overridden.
- Every class except `Object` must have a superclass.
- All variable and method references have legal names and types.
- Methods are called with the appropriate arguments.
- Variables are assigned values of the appropriate type.
- There are no stack overflows or underflows.
- All objects on the stack are of the appropriate type.

Java provides an abstract `SecurityManager` class to enforce the browsers' security policy. Because `SecurityManager` is an abstract class it cannot be used directly; therefore, each browser must declare a subclass of the class `SecurityManager`, which acts as a reference monitor. The security manager is installed when the Java virtual machine within the browser starts up, and it cannot be overridden. The security manager is automatically consulted before any operation related to the environment is performed. In this way it can deny the authorization for any possibly insecure operation. It is important to note that the security policy is defined by the web browser and it is implemented in the browser's security manager. Therefore, the above restrictions are only advice and might not be followed by some browsers. In fact, Microsoft Explorer allows different configuration levels. One of these levels allows applets to download classes from anywhere. In contrast, Netscape Navigator strictly implements all of the restrictions.

The restrictions above cover many of the protection concerns, but there are additional concerns about security. The following subsections will discuss some of these.

4.2 The context of an Applet

A single HTML page may embed more than one applet. In both Netscape Navigator and Microsoft Explorer all applets in the same page define a single *context*. An applet can get a reference to its *context* by the method `getAppletContext()` of the class `Applet`. The context gives an applet access to a set of methods to collect references to other applets in the same page and to interact with the browser. These features could be exploited to make applets collaborate and communicate; for example, they could synchronize their behavior. Two methods of interest are `getApplets()` and `getApplet()`. `GetApplets()` returns a list of all applets *in the same context* (i.e., in the same page), and `getApplet()` returns a reference to an applet when given the applet's name. Each applet has a string name. Once an applet has a reference to another applet, it can invoke any of the methods available for the class `Applet` on the referenced applet.

An applet can interact with the browser to show (download) a new document in a window and to show a message in the status line of the current window. The methods of interest are `showDocument()` and `showStatus()`. `ShowDocument()` shows a new document in a window given its URL, and `showStatus()` shows a string in the status line of the browser.

The `showDocument` method can be asked to open a document in the current window, in a frame of the current window, or in a new window. Both Netscape Navigator and Microsoft Explorer define "window" as "browser window". So the two methods above refer to the browser itself. When `showDocument` opens a new window, it actually opens a new browser and downloads the requested HTML document in that browser.

In Netscape Navigator the new browser can be identified by a unique name that must be supplied to `showDocument` as a parameter. After being created, that browser can be accessed by name by *any active* applet, regardless of its context. This means that any applet that knows the name of that browser can make it download a document even without user acknowledgment and awareness.

In contrast, Microsoft Explorer creates an anonymous browser, even if a name is supplied when `showDocument` is invoked. This means that even the same applet that opened the browser cannot show another document in that browser.

This is an example of different implementations of the same language feature. Both implementations can be criticized. Netscape Navigator can be criticized because a name can be used in a different context as a reference to the same window. This could be a security (or a safety) flaw, since the behavior of two applets in two different contexts could interfere with each other (intentionally or not). Microsoft Explorer can also be criticized because one browser is prevented from opening another browser and then showing a sequence of documents in it.

The method `showStatus()` allows an applet to display any message on the status line of the browser showing the document. This method could be used to display misleading information to the user.

4.3 Playing with Threads

Since every applet is implicitly associated with a thread, an applet can access another applet by using an instance of the class `Thread`. The thread class defines the method `getThreadGroup`, which returns a `ThreadGroup` object. Using this object one can retrieve a list of references to all threads (i.e., applets) that belong to the *same group or subgroup of this thread group.* Both Netscape Navigator and Microsoft Explorer define a single thread group that contains all threads associated with all active applets in all of the browsers. The `getParent` method can be recursively called to get the thread group that is the parent of all thread groups in the browser. This means that an applet can get a reference to, as well as access to, any applet that is embedded in any displayed document through its thread object.

This feature allows the design of applets that continuously monitor the browser to get access to applets embedded in downloaded pages. Once the applet has this information, it can interfere with the execution of other applets in several ways. The denial of service attacks described in Section 3 are based on this feature. The attacker applet may affect performance by delaying, suspending and resuming threads, or by hanging their priority. It is sometimes possible to kill a thread (and therefore the associated applet).

Every thread has a name associated with it, which is the name of the class that defines the applet. Names in combination with thread references can be exploited to understand what the user is doing. The attacks described in the next section are based on this feature.

5 The Experiments

In this section some experiments that were carried out by the Reliable Software group at the UCSB are described. These experiments led to the definition of two new attacks. One attack deals with privacy violation; its goal is to build dossiers on Internet users. The second deals with confidentiality; its goal is to steal access information (e.g., user's PIN).

What is interesting is that both attacks are based on features of the browser or of the language exploited. This demonstrates that even a good implementation can lead to an insecure environment if it is possible to combine "features" that have been designed independently of each other.

In particular, the JavaScript capability of sending e-mail from within a script, the Java capability of opening a new window and downloading a new HTML page in it, and the object-oriented nature of Java that associates every applet with a thread have been exploited.

5.1 Privacy Violation: Building User Dossiers

The goal of this attack is to collect and send information about the user back to the server. This is accomplished by JavaScript statements and a spy applet embedded in a honey-pot page.

The attack starts with the victim downloading the honey-pot page. The attack itself is carried out in two steps. First, when the victim downloads the honey-pot page from the attacker server, JavaScript silently fills out a form and sends it from the unsuspecting user back to an address on the server. In this way the server can uniquely identify the victim by his/her e-mail address. Next, the spy applet monitors the victim's activity by collecting information on the victim's running threads. When the spy applet detects that a new thread is running, it retrieves the thread's name and sends it back to the server using a UDP/IP message. The stop method has also been redefined to keep the spy applet alive even when the document that embeds it is not being shown. This allows the spy applet to monitor all the open browsers at any time.

The blueprint of the attack is:

1. Build a honey-pot HTML document that embeds some JavaScript statements to send the e-mail message and that also contains a spy applet with the stop method overridden.

2. When the honey-pot page is downloaded from the attacker server, an e-mail message is sent back to the server to identify the victim, and the spy applet starts monitoring the victim's activity.

3. The spy applet sends the attacker server any information about threads that are embedded in the documents downloaded by the victim. Since the `stop` method of the spy applet has been overridden, it continuously runs even if the victim closes the honey-pot document.

On the server side there is a daemon that receives and collects the information and builds up a dossier on the victim. The daemon knows the identity of the victim and a list of names of threads run by the victim. These thread names are searchable by any web search engine. The web is searched to get references to the pages that include the thread executed by the victim browser. The result of the attack is a user profile with a reasonable degree of accuracy.

There are two technical problems with this attack. The first is that only pages including threads can be monitored. This is considered to be a minor problem, since the number of pages that embed applets is increasing every day. The second problem is that it may happen that the same applet can be embedded in more than one page. Thus, when searching on a thread name the thread name query may result in several HTML pages. In this case, more sophisticated techniques may be required to correlate the collected information.

The attack presented here is likely to succeed since it does not require any special preparation. To be more affective it could be focused on a specific target. For example, the honey-pot page could be a collection of links and information about a particular subject, such as fishing. It is likely that web surfers who like to fish will go to that page and then follow links to look for information interesting to them. The user profiles that can be built may be used for commercial and marketing studies. For example manufacturers might be interested in deducing what items are more appealing, advertising companies might use this information to better focus their advertisements, or commercial organizations might better understand the interests of customers from different geographic areas, allowing them to offer different items in different stores.

5.2 "Man-in-the-middle" Attack: Stealing Account and Password Information

The goal of this attack is to monitor a victim's activities to understand when he/she visits a target site (e.g., a bank site) and then to replace the original pages with fake pages in order to retrieve sensitive information. The attack requires that there are two open browsers: one with the attacking applet, and one to be used by the victim. This is necessary because the attack is based on the `showDocument` method and only *active* applets can execute it.

Like the first attack, this one also starts with the victim downloading a honey-pot page that includes a spy applet. When the victim asks for the honey-pot page, he/she gets a first document on the open browser and a second document on another, new browser. This second browser is opened by the spy applet embedded in the first document. The purpose of the second browser is to keep the first one displaying the document including the spy applet, so that it never gets suspended. The spy applet can then monitor the activities carried out by the victim and have access to the second browser through the `showDocument` method.

The same technique of monitoring thread names, which directly translate to applet names, is used to discover when the user is looking at the target page. For this to be successful, the target page must contain an applet that does not have a common name. When the victim has downloaded a page from the target site, the spy applet downloads a fake page from the applet's server with the same appearance as the real page. At this point the spy applet is the man-in-the-middle. Now, the attacker has complete control of the victim's interaction with hyperlinks and forms. The fake page has forged links to make the victim interact with the attacker server instead of with the real server for the target site. In the experiment, when the victim chooses the log-on page for the bank he/she actually downloads a pseudo log-on page from the attacking server. Thus, the filled out form, which contains the victim's account number and pin, is sent to the attacker server instead of to the bank. After collecting the wanted information, a bad connection message is displayed and the real page is downloaded, so that the victim can continue his/her activity without understanding what was going on. For the experiment this was accomplished by first downloading a page with an error message "Connection refused, please try again" and then downloading the real log-on page for the bank.

The blueprint of the attack is:

1. Build a Honey-Pot page that includes the spy applet.
2. When downloaded, the spy applet spawns a new named browser with the same or a possibly different page.

3. It is assumed that the user uses the new browser so that the spy applet is never stopped and it can monitor the documents downloaded by the victim.

4. When the user retrieves a page that embeds an applet that signals that he/she is looking at the target page that the attacker is interested in (e.g., the bank page), the spy applet calls the `showDocument` method to replace the real page with a forged page downloaded from the attacker server.

5. The forged page sends information back to the attacker server by means of JavaScript support for forms. This causes a CGI script to run on the attacker server.

6. The CGI script stores the information and answers with a page that shows an error message and embeds an applet that downloads the real page to conclude the attack.

As mentioned above, this attack uses the `showDocument` method and requires that there are two open browsers: one with the attacking applet, and one to be used by the victim. The question is how can one convince the victim to keep the first browser open and use the second for further downloading? In one experiment a honey-pot page that provides up-to-date information with auto refresh every few minutes was implemented. It is likely that the victim will take advantage of this feature by keeping that page active (e.g., for updated sports scores). In another experiment a second browser, which overlaps the first, is opened. Having the second browser overlap the first is something that most users are willing to accept, especially when browsing new pages. Furthermore, for users that use full-screen browsers the new browser completely overlaps the existing one.

A weakness of this attack is that the victim may notice that something is wrong by checking out the URL displayed by the status line and the location line. The former problem was easily overcome by embedding JavaScript statements to overwrite the status line. When the victim points the mouse to a link with a forged URL, the status line still shows the original URL. The JavaScript statement used is `onMouseOver="self.status='faked link';return true"`. This is added to the tag that has the hyperlink for which the status line is to display the desired link instead of the real link whenever the mouse is over it. The latter problem is more subtle. If the victim carefully reads the location address, he/she will surely understand that the contacted server is not the right one. A partial solution is to build up URLs that are similar to the real ones, except for a small detail. It is likely that the victim will only look carefully at the last part of the URL, skipping the first part. One possibility, which was suggested in [FBD96], is to register a domain name with the DNS that is close to the name of the site where the page is to be replaced. For instance, one could replace the letter 'b' in bank with the letter 'd' in the attacker site.

6 Conclusions

In this paper two powerful attacks that can be accomplished by the monitoring of thread names, which translates to applet and page names, have been presented. In the first attack the use of applets with distinct names in web pages makes it possible to identify the site the user is browsing. The second attack assumes that the target page has an applet, which will be identified at the time the user requests the targeted page. The current trend on the Internet is towards increasing the number of pages with embedded applets and the number of applets on a page is also increasing. As a result, the set of applets in a page is more likely to uniquely identify the page. As Java applets become more widely used the two attacks become even more of a threat.

The two attacks are very easy to implement, and they do not require special resources. The dossier attack only needs a place that serves the World Wide Web, and the man-in-the-middle attack needs the same plus it needs to be able to set a CGI for collecting the form information.

The dossier attack is similar to the DoubleClick cookie scenario. Because of the limited pages (or sites) being logged, they both can only build partial dossiers. The dossier attack reported in this paper is able to identify and log pages only when they have embedded applets with meaningful names. The cookie attack logs only pages of sites that are affiliated with DoubleClick; that is, it logs only pre-marked pages. Thus, the cookie attack has a specific target and each of the target pages need to be marked, which is more difficult to implement. In contrast, the dossier attack is general purpose and requires no changes to the pages that our logged. Moreover, the dossier attack is more personalized, since the user is identified by his/her e-mail address, rather than by only an identification number.

The man-in-the-middle attack is much easier to implement than the web spoofing attack [FBD96], since it is not as intrusive as the web spoofing attack. Also, it does not require a special server to understand and dynamically process the requested URLs. Network spoofing is a very powerful technique, but one of the goals for the experiments reported in this paper were that the attacks would be, although very threatening, very easy to implement so that a programmer could implement them without the need for sophisticated tools or special resources.

The two attacks described in this paper and other experiments that we have run lead us to conclude that although much continues to be done to improve the security of browsers, much more has to be done to reach a sufficient level of confidence. Java is a reasonable starting point, since it has overcome many of the basic security flaws of other conventional languages. At the same time the experiments reported in this paper have demonstrated that even a "secure" language cannot suffice to solve all security problems. Every component of the environment has to be designed to address security. More importantly, each component needs to be aware of the features of the other components in the environment. In this paper it has been shown that the combined use of not-so-bad features of different tools can support the design of very dangerous attacks.

In addition to Java, these experiments have examined JavaScript and cookie technologies by Netscape. JavaScript was designed to give an easy to program environment for Internet developers both to the client side embedded in HTML language and to the server side, where it is called LiveWire, which is used like a CGI. This requirement of being flexible and easy to program subjects JavaScript to many security flaws as per [Dig97]. Cookies were introduced to store and retrieve information on the client side. As discussed in [Ste97] it is possible to use this information to cross reference HTML pages and build dossiers that violate user privacy. The lesson learned is that adding even small features without thorough analysis can open serious security flaws in the whole environment.

For these experiments the latest versions of the two most popular browsers: Netscape Navigator 3.01 and Microsoft Explorer 3.02 were used. Both are aware of the security problems, and both have many optional features that provide better security. However, because these options result in not enabling Java and JavaScript at all or in popping up many annoying warning windows, we believe that most users are likely to disable the security options. This is a typical example of the trade off between ease of use and security. If the security features are not user friendly, they will not be used.

7 Future Work

This paper has presented the preliminary results of ongoing research that has the final goal of developing properties for secure browsers and procedures for secure browsing.

We are currently designing and running other experiments. Some are addressing new areas and others are improving on the experiments described in this paper. The man-in-the-middle attack could be improved by taking over the browser instead of just monitoring threads in downloaded pages. This improved attack exploits the ability of JavaScript to open a new browser without any menu, toolbar, status bar, etc. This is accomplished by using the method `window.open()` with the option `toolbar=no`. The blueprint of the attack is similar to the one described in Section 5.2, except that the second browser is opened as an empty window that shows a page that includes an applet that paints the entire browser window to simulate the browser that is being used. Thus, for Netscape browsers the applet will paint a menu, a location field and toolbars on the top of the window, a scroll bar if necessary, and status bar and security key, broken or not, on the bottom of the window. The result is that the user will see this new window as a new complete browser. From that time on, any interaction between the user and the faked browser will be processed locally by the spy applet and any information will be forwarded to the attacker server through a socket connection. This represents a bigger threat since the victim is completely in the control of the attacker. The attacker applet can act on behalf of the user to perform any transaction. Of course this attack requires much more work on the part of the spy applet to follow the mouse and respond appropriately to slider moves, etc.

Future plans are to continue collecting information on known attacks and on countermeasures for these attacks and to continue with the preliminary experiments and develop new attacks. The objective is to categorize the browser attacks to determine information such as what resources are needed to exercise the attack, what browsers it works on and why, and what countermeasures could detect and possibly thwart the attack. The process of determining what classification schemes to use when categorizing the attacks is a critical part of the research, for it is only by working with the actual attack scenarios that the appropriate classification schemes become evident. In carrying out some of the preliminary experiments it was discovered that some attacks that were possible when using the Netscape Navigator browser were not possible when using the Microsoft Explorer browser. This is not very surprising, but on further examination it was discovered that the reason that the attack was not successful was due to an implementation error in the Microsoft Explorer browser. Thus, sometimes interesting results are revealed when classifying the attacks.

Finally, based on the understanding gained by analyzing and categorizing the browser attacks and countermeasures, it is expected that properties for secure browsers and procedures for secure browsing will be developed. These two work together, for sometimes by following secure procedures one can work as securely with a less secure browser as with one with a higher level of security assurance. That is, there may be many tradeoffs between implementing secure features or enforcing the user to follow secure browsing procedures. The authors of [FBD96] present a procedural approach that users can use to detect that they are browsing a pseudo page. They suggest that "when the mouse is held over a Web link [on the page], the status line displays the URL the link points to. Thus, the victim might notice that a URL has been rewritten." However, in that same paper they also mention that this function can be blocked. In fact, the second experiment reported in this paper demonstrates this. That is, the spy applet can display anything in the status line, including the name of the desired link in place of the false link that actually exists. This example demonstrates a procedural approach that is not sufficient. In addition, the property that placing the mouse over an applet always displays the link for that applet, which was overridden in the experiment, is an example of a property that should always be preserved in a secure browser.

8 References

DFW96 Drew Dean, Edward W. Felten, and Dan S. Wallach, Java Security: From HotJava to Netscape and Beyond, Proc. of 1996 IEEE Symposium on Security and Privacy.

(http://www.cs.princeton.edu/sip/pub/secure96.html)

Dig97 Digicrime, Online Security Experiments, Digicrime, Inc.

(http://www.digicrime.com)

FBD96 Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach, Web Spoofing: An Internet Con Game, Technical Report 540-96, Dept. of Computer Science, Princeton University, Dec 1996.

(http://www.cs.princeton.edu/sip/pub/spoofing.html)

GJS96 James Gosling, Bill Joy, and Guy Steele, The Java Language Specification, ISBN 0-201-63451-1, The Java Series, Addison Wesley, 1996.

(http://www.nge.com/home/java/spec10/index.html)

LY96 Tim Lindholm and Frank Yellin, The Java Virtual Machine Specification, ISBN 0-201-63452-X, The Java Series, Addison Wesley, 1996.

MRR97 D. Martin, S. Rajagopalan and A. Rubin, "Blocking Java Applets at the Firewall", in Proceedings of the InternetSociety Symposium on Network and Distributed System Security, February 10-11, 1997.

Ros96 Jim Roskind, Navigator 2.02 Security-Related FAQ, May 1996.

(http://home.netscape.com/newsref/std/java_security_faq.html)

Ste97 Lincoln D. Stein, WWW Security Faq, version 1.3.7, March 30, 1997.

(http://www.genome.wi.mit.edu/WWW/faqs/www-security-faq.html)