Covert Distributed Computing Using Java Through Web Spoofing

Jeffrey Horton and Jennifer Seberry

Centre for Computer Security Research
School of Information Technology and Computer Science
University of Wollongong
Northfields Avenue, Wollongong
{jeffh, j.seberry}@cs.uow.edu.au

Abstract. We use the Web Spoofing attack reported by Cohen and also the Secure Internet Programming Group at Princeton University to give a new method of achieving covert distributed computing with Java. We show how Java applets that perform a distributed computation can be inserted into vulnerable Web pages. This has the added feature that users can rejoin a computation at some later date through bookmarks made while the pages previously viewed were spoofed. Few signs of anything unusual can be observed. Users need not *knowingly* revisit a particular Web page to be victims.
We also propose a simple countermeasure against such a spoofing attack, which would be useful to help users detect the presence of Web Spoofing. Finally, we introduce the idea of browser users, as clients of Web-based services provided by third parties, "paying" for these services by running a distributed computation applet for a short period of time.

1 Introduction

There are many problems in computer science which may be solved most easily through the application of brute force. An example of such a problem is determination of the key used to encrypt a block of data with an algorithm such as DES (Data Encryption Standard). The computer time required could be obtained with the full knowledge and cooperation of the individuals controlling the resources, or covertly without their knowledge by some means. A past suggestion for the covert accomplishment of tasks such as this involved the use of computer viruses to perform distributed computations [1].

Java is a general purpose object-oriented programming language introduced in 1995 by Sun Microsystems. It is similar in many ways to C and C++. Programs written in Java may be compiled to a platform-independent bytecode which can be executed on any platform to which the Java runtime system has been ported; the Java bytecodes are commonly simply interpreted, however speed of execution of Java programs can be improved by using a runtime system which translates the bytecodes into native machine instructions at execution time. Such systems, incorporating these Just-in-time (JIT) compilers, are becoming more common. The Java system includes support for easy use of multiple threads of execution, and network communication at a low level using sockets, or a high level using URL objects [2].

One of the major uses seen so far for Java is the creation of applets to provide executable content for HTML pages on the World Wide Web. Common Web browsers such as Netscape Navigator and Microsoft Internet Explorer include support for downloading and executing Java applets. There are various security restrictions imposed upon applets that are intended to make it safer for users to execute applets from unknown sources on their computers. One such restriction is that applets are usually only allowed to open a network connection to the host from which the applet was downloaded. A number of problems with Java security have been discovered by various researchers [5] [7].

Java could also be applied to performing a distributed computation. Java's straightforward support for networking and multiple threads of execution make construction of an applet to perform the computing tasks simple. The possibility of using Java applets to covertly or otherwise perform a distributed computation is discussed by several researchers [4] [5] [7, pp. 112–114] [8].

There have been no suggestions, however, as to how this might be accomplished without requiring browser users to knowingly visit a particular page or Web server at the beginning or sometime during the course of each session with their Web browser, so that the applet responsible for performing the computation can be loaded. This paper describes how the Web spoofing idea described by the Secure Internet Programming Group at Princeton University can be used to pass a Java applet to perform a distributed computation to a client. The advantage is that clients do not have to *knowingly* (re)visit a particular site each time, but may rejoin the computation through bookmarks made during a previous session.

There will be some indications visible in the browser when rejoining a computation through a bookmark that the user has not reached the site they may have been expecting; however, these signs are small, and the authors believe, mostly correctable using the same techniques as employed in a vanilla Web Spoofing attack.

2 About Web Spoofing

Web Spoofing was first described briefly by Cohen [3]. The Web Spoofing attack was later discussed in greater detail and elaborated upon by the Secure Internet Programming Group at Princeton University [6]. Among other contributions, the Princeton group introduced the use of JavaScript for the purposes of concealing the operation of the Web Spoofing attack and preventing the browser user from escaping from the spoofed context. JavaScript is a scripting language that is supported by some common Web browsers. JavaScript programs may be embedded in an HTML page, and may be executed when the HTML page is loaded by the browser, or when certain events occur, such as the browser user holding the mouse pointer over a hyperlink on the page.

The main application of Web Spoofing is seen as being surveillance, or perhaps tampering: the attacking server will be able to observe and/or modify the Web traffic between a user being spoofed and some Web server, including any form data entered by the user and the responses of the server; it is pointed out that "secure" connections will not help — the user's browser has a secure connection with the attacking server, which in turn has a secure connection with the server being spoofed [6].

Web spoofing works as follows: when an attacking server **www.attacker.org** receives a request for an HTML document, before supplying the document every URL in the document is rewritten to refer to the attacking server instead, but including the original URL in some way — so that the attacking server is able to fetch the document that is actually desired by the browser user. For example,

```
http://www.altavista.digital.com/
```

might become something like:

```
http://www.attacker.org/f.acgi?f=http://www.altavista.digital.com/
```

There are other ways in which the spoofed URL may be constructed. The Princeton group gives an example [6].

The first part of the URL (before the '?') specifies a program that will be executed by the server. This is a CGI (Common Gateway Interface) program. For those not familiar with CGI programs, the part of the URL following the '?' represents an argument or set of arguments that is passed to the CGI program specified by the first part of the URL. More than one argument can be passed in this way — arguments are separated by ampersand ('&') characters.

So, when the user of the browser clicks on a spoofed URL, the attacking server is contacted. It fetches the HTML document the user wishes to view, using the URL encoded in the spoofed URL, rewrites the URLs in the document, and supplies the modified document to the user. Not all URLs need be rewritten to point at the attacking server, only those which are likely to specify a document containing HTML, which is likely to contain URLs that need to be rewritten. In particular, images do not generally need to be spoofed. However, as many images would be specified using only a partial URL (relative to the URL of the HTML page containing the image's URL), the URLs would need to be rewritten in full, to point at the appropriate image on the server being spoofed.

There will be some evidence that spoofing is taking place, however. For example, the browser's status line and location line will display rewritten URLs, which an alert user would notice. The Princeton group claim to have succeeded at concealing such evidence from the user through the use of JavaScript. Using JavaScript to display the proper (non-spoofed) URLs on the status line when the user holds the mouse pointer above a hyperlink on the Web page is straightforward in most cases; using JavaScript to conceal the other evidence of spoofing is less obvious.

In the course of implementing a program to perform spoofing, we have observed that some pages using JavaScript do not seem amenable to spoofing, especially if the JavaScript itself directs the browser to load particular Web pages.

It seems that the JavaScript has difficulty constructing appropriate URLs if the current document is being spoofed, due to the unusual form of the spoofed URLs.

We have implemented only the spoofing component of the attack, as well as the simplest use of JavaScript for concealment purposes, that of displaying non-spoofed URLs on the browser status line where necessary, for the purposes of demonstration; the aspects of the attack that provide more sophisticated forms of concealment were not implemented. We believe that the work on concealment of the Web Spoofing attack done by the Princeton group can profitably be applied to concealing the additional evidence when using Web spoofing to perform a distributed computation.

2.1 Some HTML Tags to Modify

Any HTML tag which can include an attribute specifying a URL may potentially require modification for spoofing to take place. Common tags that require modification include:

– Hyperlinks, which are generated by the `HREF` attribute of the `<A>` tag. A new HTML document is fetched and displayed when one of these is selected by the user.
– Images displayed on an HTML page are specified using the `<IMG>` tag. Its attributes `SRC` and `LOWSRC`, which indicate from where the browser is to fetch the image data, may require adjustment if present.
– Forms into which the user may enter data are specified using the `<FORM>` tag. Its `ACTION` attribute can contain the URL of a CGI program that will process the data entered by the user when the user indicates that they wish to submit the form.
– Java applets are included in an HTML page using the `<APPLET>` tag. Since applets are capable of communicating through a socket connected to an arbitrary port on the applet's server of origin, the applet should be downloaded directly from that server. The default otherwise is to obtain the applet's code from the same server as supplied the HTML page containing the applet. An applet's code can be obtained from an arbitrary host on the Internet, specified using the `CODEBASE` attribute. For spoofing to function properly in the presence of Java applets, the `CODEBASE` attribute must be added if not present. Otherwise, it must be ensured that the `CODEBASE` attribute is an absolute URL.

These are just a few of the more common tags with attributes that require modification to undertake a spoofing attack.

3 Application of Web Spoofing to Distributed Computing

The Secure Internet Programming Group suggest that Web Spoofing allows tampering with the pages returned to the user, by inserting "misleading or offensive material" [6]. We observe that the opportunity to tamper with the pages allows a Java applet to perform part of a distributed computation to be inserted into the page. Tampering with the spoofed pages in this manner and for this purpose has not been previously suggested.

As Web pages being spoofed must have their URLs rewritten to point at the attacking server, it is a simple matter to insert the HTML code to include a Java applet into each page in the course of performing the other modifications to the page. The Java applet can be set to have only a small "window" on the page, which makes it difficult for users to detect its presence on the Web pages that they view.

When the browser encounters a Java applet for the first time during a session, it usually starts Java, displaying a message to this effect in the status line of the browser. It may be possible to conceal this using JavaScript if necessary (although this has not been tested); however, if Java applets become increasingly common and therefore unremarkable, concealment may be deemed unnecessary.

Users may bookmark spoofed pages during the course of their session. If this occurs, the user will rejoin the computation when next that bookmark is accessed. Rather than knowingly (re)visiting a particular site to acquire a copy of an applet, the user unknowingly contacts an attacking server which incorporates the applet into each page supplied to the user.

A site that employed distributed computing with Java applets and Web spoofing could potentially be running applets not only on machines of browser users who visit the site directly, but also on the machines of users who visit a bookmark made after having directly visited the site on some previous occasion. Thus, the "pool" of users who could be contributing to a computation is not limited to those that directly visit the site, as it is with other approaches to covert distributed computing with Java. For example, consider a Web server that receives on average 10,000 hits/day. If the operators of the Web server elect to incorporate an applet to perform distributed computation in each page downloaded from the server, they will steal some CPU time from an average of 10,000 computers each day. However, by using Web spoofing as well, on the second day after starting the spoofing attack and supplying the applet, CPU time is being stolen from the (on average) 10,000 users who knowingly (re)visit the site, and also from users who have visited the site on the previous day and made bookmarks to other sites subsequently visited.

The level of load on the attacking server can be controlled by redirecting if necessary some requests directly to the actual server containing the resource, foregoing the opportunity to perform Web Spoofing, and of stealing computation time from some unsuspecting browser user, but keeping the load on the server at reasonable levels.

An attacker might decide to increase the likelihood of bookmarks referring to spoofed pages by modifying a Web search engine to return answers to queries that incorporate spoofed links, but not require the search engine itself to participate in the spoofing (of course, if the pages of an unmodified search engine are being spoofed when a search is performed, rewriting of the URLs in the response will take place automatically).

The user's Web browser will display a URL in its location line which exhibits the presence of spoofing when revisiting a bookmark made of a spoofed page; however, the authors believe that this may be concealed after the page has commenced loading using JavaScript, although again this has not been implemented.

3.1 An Implementation

For reasons of simplicity, the Web spoofing attack for the purposes of demonstration was implemented using a CGI program. An off-the-shelf Web server was used to handle HTTP requests.

The applet to demonstrate the performance of a simple key cracking task was of course implemented in Java. The program which kept track of which subproblems had been completed (without finding a solution) and that distributed new subproblems to client applets was also written in Java (the "problem server").

Client applets use threads, one for performing a computation, another for communicating with the server to periodically report the status of their particular computation to the problem server. Periodic reporting to the server guards against the loss of an entire computation should the client applet be terminated before completion of the entire computation, for example, by the user exiting the Web browser in which the applet is being run.

The thread performing the computation sleeps periodically, to avoid using excessive resources and so unintentionally revealing its presence.

The problem server keeps records using the IP numbers of the computers on which a client applet is running, and will allow only one instance of an applet to run on each computer, to avoid degrading performance too noticeably. A new client applet will be permitted to commence operations if some amount of time has elapsed without a report from the original client applet.

4 Countermeasures?

4.1 Client-side Precautions

Obviously, disabling Java is an excellent way for a user to ensure that he or she does not participate unwillingly in such an attack, as the Java applet to perform the computation will be unable to run. The disadvantage of this approach is that applets performing services of potential utility to the user will also not be able to run.

The merits of disabling JavaScript are briefly discussed by the Princeton group [6]. This prevents the Web spoofing from being concealed from the clients. How many clients would take notice of the signs is an interesting question, especially given that in the future clients may have become accustomed to the use of strange URLs such as those produced by a Web spoofing program, as there are several sites providing legitimate services with a Web spoofing-style program[1].

[1] "The Anonymizer": see `http://www.anonymizer.com/` [6]; the "Zippy Filter": see `http://www.metahtml.com/apps/zippy/welcome.mhtml` [6]; the "Fool's Tools" have been used to "reshape" HTML: `http://las.alfred.edu/~pav/fooltools.html`.

4.2 Server-side Precautions

During the preparation of a demonstration of the approach to distributed computing with Java described here, it was observed that there were some sites whose pages included counters of the number of times that a site had been visited, and links to other CGI programs, all of which failed to produce the expected results when the page containing the counter, or link to CGI program, was being spoofed. Note that a page visit counter is commonly implemented by using a CGI program through an `<IMG>` HTML tag.

The problem was eventually traced to an improperly set `Referer:` field in the HTTP request sent by the spoofing program to fetch an HTML page from the server being spoofed. The `Referer:` field that was originally being sent included a spoofed URL.

The `Referer:` field of an HTTP request is used to specify the address of the resource, most commonly a Web page, that contained the URL reference to the resource which is the subject of the HTTP request, in cases where the URL reference was obtained from a source that may be referred to by a URL; this field would be empty if the source of the request was the keyboard, for example [9] [10].

The value of the `Referer:` field can be checked by a CGI program to determine that a request to execute a CGI program comes only from a URL embedded in a specific page, or a set of pages. This prevents easy misuse or abuse of the CGI program by others.

An implementation of distributed computing with Java in the manner described in this paper would want to keep the amount of data passing through the attacking server as small as possible, to minimise response time to client requests, and so that the number of clients actively fetching pages and performing computations could be maximized. To achieve this, only URLs in pages which are likely to point at an HTML document (whose URLs will need to be rewritten, so that the client continues to view spoofed documents) are rewritten to point at the spoofing server — all other URLs, specifically URLs in HTML `<IMG>` tags are modified only so that they fully specify the server and resource path; they are *NOT* spoofed.

It is easy enough for the attacking server to adjust the `Referer:` field so that it has the value which it would normally have were the page not being spoofed. However, this does not help with the fetching of non-spoofed resources such as images — the attacking server never sees the HTTP request for these resources. So the `Referer:` field will not be set as the spoofed server would expect.

So we propose that an effective countermeasure against a spoofing attack for the purpose of performing a distributed computation is for Web servers to check the `Referer:` field for images and other resources that are expected to be always embedded in some page being served by the Web server for consistency; that is, the `Referer:` field indicates always that the referrer of the document is a page served by the Web server. Usually it should be sufficient to verify that the address of the Web server that served the page which contained the URL for the resource currently being served is the same as the Web server asked to serve the current request. The Web server could refuse to serve a resource if its checks of the `Referer:` field were not satisfied, or display an alternative resource, perhaps attempting to explain possible causes of the problem.

While checking the `Referer:` field and taking action depending on its contents does not prevent an attack from taking place, it does mean that unless all the images contained on a page are also spoofed there will be gaps where a Web server has refused to serve an image because its checks of the `Referer:` field have failed. Given the high graphical content of many Web pages, it is unlikely that a user would wish or be able to persist in their Web travels while the pages were being spoofed. Either they would find a solution or stop using the Web. Spoofing all the images would increase the amount of data processed by the spoofing server, which as a result would greatly limit the number of clients who could be effectively spoofed at the one time.

Unfortunately, at this time there are some inconsistencies in the way in which different Web browsers handle the `Referer:` field. Common Web browsers like Netscape Navigator and Microsoft Internet Explorer appear to provide `Referer:` fields for HTTP requests for images embedded in Web pages, for example. Other less widely used browsers, such as Apple Computer's Cyberdog, do not do so.

It should be noted that this countermeasure would be most effective in protecting users if many of the Web servers in existence were to implement this sort of check. A site could, however, implement this countermeasure to help ensure that users of that particular site were likely to detect the presence of Web spoofing.

5 Computing for Sale

It is not unusual to find that a Java applet that performs a distributed computation is classed as a "malicious" applet [7, pp. 113–114] [8]. The computation is undesirable because the user is not aware that it is being performed.

On the other hand, it is not difficult to imagine a large group of users donating some of their computer time to help perform a long computation. Examples include efforts to crack instances of DES or RC5 encrypted messages[2], or finding Mersenne Primes[3]. Using Java for this sort of purpose avoids many troublesome issues of producing a client program for a variety of different computer platforms; there is, however, currently a heavy speed penalty that must be paid, as Java is not as efficient as a highly-tuned platform-dependent implementation. Improvements in JIT compilers, mentioned earlier, will help to reduce this speed penalty, but will not eliminate it entirely.

We introduce the idea of "Computing for Sale" — that sites which provide some form of service to clients could require that clients allow the running of a Java applet for some fixed period of time as the "price" for accessing the service. An excellent example of a service to which this idea could be applied is that of a Web search engine, or perhaps an online technical reference library or support

[2] See `http://www.distributed.net/`

[3] See `http://www.mersenne.org/`

service. Clients that are unable or unwilling to allow the Java applet to run so that it may perform its computation could be provided with a reduced service. For example, a client of a Web search engine who refused to run the applet could be provided with E-mail results of their query half an hour or so after query submission rather than immediately. This provides an incentive for clients to allow computation applets to perform their tasks.

It would be possible for a service to deny access or provide only a reduced service to a client whose computation applets consistently fail to report results for some reason, such as being terminated by the client.

In the interests of working in a wide variety of network environments, applets used for this purpose should be able to communicate with the server using methods apart from ordinary socket connections, such as, by using Java's URL access capabilities, HTTP POST or GET messages [11]. For example, a firewall might prohibit arbitrary socket connections originating behind the firewall but permit HTTP message traffic.

The service provider would be able to sell computation time in much the same way as many providers sell advertising space on their Web pages. Some clients of such services might prefer to choose between an advertising-free service which requires that client assist in performing a computation, and the usual service loaded with advertising, but not requiring the client to assist with the computation by running an applet.

6 Conclusion

There are many problems in computer science which can best be solved by the application of brute force. An example is the determination of an unknown cryptographic key, given some ciphertext and corresponding plaintext. Distributed computing offers a way of obtaining the necessary resources, by using a portion of the CPU time of many computers.

Such a project can either be conducted with full knowledge and cooperation of all participants, or covertly. There have been some suggestions that applets written in Java and running in Web browsers might perform covert distributed computations without the knowledge of browser users, but requiring browser users to knowingly visit a particular site.

We observe that Web Spoofing offers a way of not only adding Java applets to perform covert distributed computations to Web pages, but also of increasing the likelihood of past unwitting contributors contributing again when they revisit bookmarks made during a prior spoofed Web browsing session.

Some simple measures which make a successful attack more difficult and less likely have been examined. These include disabling Java. We also proposed that servers examine the `Referer:` field of HTTP requests, and refuse to serve the object, or serve some other object explaining the problem, should the `Referer:` field not be consistent with expectations.

We introduced the idea of browser users "paying" for access to services and resources on the Web through the use of their idle computer time for a short period. Service providers could then sell these CPU resources in the same way as advertising is now sold, and may feel it appropriate to offer an advertising-free service for users who "pay" using their computer's idle time.

References

1. S. R. White. Covert Distributed Processing With Computer Viruses. In *Advances in Cryptology — Crypto '89 Proceedings*, pages 616–619, Springer-Verlag, 1990.
2. Sun Microsystems. The Java^{TM} Language: An Overview. See `http://java.sun.com/docs/overviews/java/java-overview-1.html` [URL valid at 9 Feb. 1998].
3. Frederick B. Cohen. Internet holes: 50 ways to attack your web systems. *Network Security*, December 1995. See also `http://all.net/journal/netsec/9512.html` [URL valid at 20 Apr. 1998]
4. Frederick B. Cohen. A Note on Distributed Coordinated Attacks. *Computers & Security*, 15:103–121, 1996.
5. Edward W. Felten, Drew Dean and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *IEEE Symposium on Security and Privacy*, 1996. See also `http://www.cs.princeton.edu/sip/pub/secure96.html` [URL valid at 9 Feb. 1998]
6. Drew Dean, Edward W. Felten, Dirk Balfanz and Dan S. Wallach. Web spoofing: An Internet Con Game. Technical report 540-96, Department of Computer Science, Princeton University, 1997. In *20th National Information Systems Security Conference* (Baltimore, Maryland), October, 1997. See also `http://www.cs.princeton.edu/sip/pub/spoofing.html` [URL valid at 9 Feb. 1998]
7. Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes.* John Wiley & Sons, Inc., 1997.
8. M. D. LaDue. Hostile Applets on the Horizon. See `http://www.rstcorp.com/hostile-applets/HostileArticle.html` [URL valid at 12 Feb. 1998].
9. RFC 1945 "Hypertext Transfer Protocol — HTTP/1.0". See `http://www.w3.org/Protocols/rfc1945/rfc1945` [URL valid at 9 Feb. 1998].
10. RFC 2068 "Hypertext Transfer Protocol — HTTP/1.1". See `http://www.w3.org/Protocols/rfc2068/rfc2068` [URL valid at 9 Feb. 1998].
11. Sun Microsystems White Paper. Java Remote Method Invocation — Distributed Computing For Java. See `http://www.javasoft.com/marketing/collateral/javarmi.html` [URL valid at 9 Feb. 1998].