Providing Software Services on the WWW Securely

Arthur Housinger, Navneet Singh, Rena Yang

December 7, 1995

1 Introduction

Sun Microsystems' revolutionary Web programming language, Java, has taken the World Wide Web (WWW) beyond a mere repository for text, images and the like. It is now possible to include executables in a Web site. Because of an enthusiastic reception of Java, we anticipate that a new industry, that of providing Software Services on the Web, is upcoming. However, such a task has numerous security issues associated with it. This project attempts to explore them and propose solutions for some of them.

Consider the different types of potential Web software providers - those who provide this service as a means of generating revenue, firms who have established clients and want to provide such a service as a customer satisfaction gesture, or organizations that provide it free as a means of advertisement. They all face different kinds of security problems related to the service they provide. Some of these are as follows:

- **The Free-Rider problem** Some people might find free access to services that are normally not free.
- **Secrecy of client's data** Some of the input/output of the software might be extremely confidential. Not only might there be a problem of data interception (although most Web executables run locally on the client), but also there's a possibility that some firm's motive behind providing such a service is to obtain access to somebody's data.
- **Authenticity of the service** Is the software that the client's using same

as the software that the firm intends the client to use?

• **Web Commerce problems** The usual problems of doing business over the WWW, e.g. means of payment.

• **Determining amount of usage** Some customers might attempt to get 2-for-the-price-of-1 deals.

• **Secrecy of the underlying algorithm of the program** Some software might incorporate a firm's confidential proprietary models.

2 Status Quo of Web Security

In this section we discuss the existing means of providing security in WWW browsers among other things. We look at the security facilities in the popular WWW browser Netscape - which has traditionally been a text-and-image browser but has recently introduced Java support - and HotJava - which was the first Java browser introduced by Sun Microsystems.

2.1 Netscape

Netscape provides three major types of security to its users - Server authentication (thwarting impostors), Privacy using encryption (thwarting eavesdroppers), and Data integrity (thwarting vandals). It uses the SSL protocol which is layered between the TCP/IP connection protocol and the application protocols like HTTP, Telnet, FTP, Gopher, etc. The SSL protocol uses the RSA encryption algorithm.

2.2 HotJava

When you start up the HotJava browser, it concurrently starts up a dialog box providing you with several security options. The first thing it allows you to do is define a firewall for the set of URL's that any particular applet can access. This set can be empty, contain simply the applet host, contain a specified set of hosts, or can be unrestricted. Similarly, the initial security dialog box in HotJava allows you to choose the set of applets that can be loaded. The options here are analogous to the options in the case of URL's. The user is also able to specify which domain, e.g. NIS or DNS, is being used.

Web security is currently restricted to primarily the above features provided by Netscape and HotJava. As we can see, these fail to address a number of the common contemporary problems, especially the ones introduced by Java. For example, they do little for the Web Commerce problems, the Free-Rider problem, and the Secrecy of the underlying algorithm of the applet. In the following sections, we discuss the security issues dealt with as well as neglected by the popular WWW browsers, and provide solutions where possible.

3 Problems and Analysis

There is a wide range of issues that arise with the inclusion of applets. In this section we try to group most of the concerns under five major headings: applet visibility, applet secrecy, applet usage, applet authentication, and user's data. Each issue is presented along with an analysis of possible solutions.

3.1 Applet Visibility

A major deficiency in most current security protection schemes lies in the protection of applet authors. One such issue is that of restricting applet visibility.

Applet visibility is similar, in many ways, to the current issue of Web page access restrictions. Because Java allows for software to be transferred across an insecure network to a client's machine, where it could possibly be tampered with or copied, these access restrictions must be even more strict than normal.

Some instances where visibility restrictions might be useful are the following: An applet developer may want to maintain a trade secret while allowing users to access his applet, or he may wish to collect a fee for use of the applet.

3.1.1 Solutions

Applets can be encrypted during transit using the authentication/encryption of the Secure Sockets Layer (SSL) discussed in section 3.4. This will restrict exposure of the applet during communication. Further restriction is necessary, however, because there are several ways in which a client destination may be altered or tampered with.

One attack of this type is a Routing Information Protocol attack. The Routing Information Protocol (RIP) is used to propagate routing information on local networks. Typically, the information received is left unchecked. An intruder may use this weakness to feed incorrect routing information to a router and to divert packets destined for a host machine to an intruder specified destination.

A similar attack is through the Internet Control Message Protocol, where Redirects are used by gateways to advise hosts of better routes. If these Redirects are left invalidated, an intruder may again set up a false route to his machine.

Because of these and several other attacks, it would seem prudent to authenticate not only authenticate the servers, but the clients as well, ensuring that the expected client is actually receiving the software/data.

One possible solution to this threat is the use of a network-external device for authentication of the user. This might be in the form of a Smart Card or some other hardware device that the User must obtain. The User uses this device to initiate communications with the server and to request an applet. The server would then check this by looking this value up in a table. Less secure, perhaps, is the use of a password for authentication of the user. If we assume untrusted devices, this may not be a useful option.

These two options are subject to man-in-the-middle attacks, however, if identity is checked only once at the beginning of the transaction, for a man-in-the-middle may wait for a User to initiate communications and to request an applet, then take over the communication line, capturing the use of an applet.

A much more complex, but more robust, solution would be to use the previously mentioned network-external hardware device or password as a decryption device. The server would still hold a lookup table for all clients, but now, it would now hold Public keys. All transactions with the user would be encrypted with the client's Public Key. The User device would then serve as a private key used to decrypt all information sent by the server. This way, even if information is sent to an intruder's host, no service will be given.

3.2 Applet Secrecy

A number of firms have proprietary models that serve as the underlying algorithms for software used internally within the firm, or provided as a service to their clients. For example, most major Investment Banks and Securities Trading Firms have proprietary pricing models that they use in order to price derivative securities, such as Options and Futures. If they use Java in order to provide this software service to clients over the WWW, then a major security issue immediately comes to mind - that of the secrecy of the proprietary model or algorithm. The firm might have done a great of deal research in coming up with this model and would hate to have it compromised.

The first thing to notice is that if the applet does all its computation where it was downloaded, then there is no perfect solution to this problem - we would have to be content with simply good measures that have a possibility of being broken. On the other hand, if some of the computation takes place on the provider's side, secrecy of the client's data cannot be guaranteed.

3.2.1 Solutions

The first solution is one that doesn't even need to be implemented. Simply the fact that the executable is a binary file which must be reverse compiled in order to figure out the algorithm provides security for the algorithm. This is so because reverse engineering compiled code is extremely tedious and sometimes difficult. Another solution for the Applet Secrecy problem is to have a part of the computation done on a provider's machine. Although we mentioned above that doing so would be a potential threat to the secrecy of the client's data, we can strike a desirable balance if we implement the scheme cleverly. The provider of a particular software service knows from past experience which subset of the input to the program (client's data) can potentially be confidential. On the other hand, the provider also knows which of the subroutines of the algorithms are more secret than the others. The goal of the provider should be to determine an optimal distribution of the algorithm that minimizes the threat to the secrecy to either the client's data or the algorithm itself.

There are also circumstances in which the provider is not too concerned about the client figuring out the algorithm but does not want third parties doing so. In this case, the provider might want cleverly encrypt the applet and share the key with desired clients. The reason it would have to be a clever encryption is that most executables are very large and it would be extremely inefficient to encrypt/decrypt them entirely using a common encryption algorithm. Consequently, one thing the provider might try is leaving most of the executable as plaintext (although it's hardly plaintext) and encrypt certain key parts of the binary file. The browser that the client is using in order to run the applets would have to be made aware of this protocol and the necessary encryption keys.

3.3 Applet Usage

As the World Wide Web becomes more and more popular, companies may wish to sell their services per hour or per computation. However, since an applet is downloaded onto the computer of the User, the ways to monitor usage are limited. This section looks at the different ways that could be used to attempt to monitor activity.

3.3.1 Solutions

The first solution that presents itself is to check the local clock. If a certain amount of time has passed since the applet was downloaded, then the applet should refuse to run. However, this is easily compromised. All a user needs to do is change his local clock.

A similar solution would be to check the time at the server.[1] This would add a little load to the server, but the main bulk of the computation would be done on the user's computer. In order to get around this solution, the user would have to send fake server time-responses to the applet. Although possible to do, it is harder than just changing the clock. However, we do need to be careful about the secrecy of the user's data when doing this. (See

[1] This works equally well if you check the time at another server set aside just for time verification purposes.

section 3.5)

A different tactic would be to put a static counter within the applet. After a certain number of accesses to the applet, the counter would go over its maximum and access would be denied. This solution can be sidestepped with a little work. Depending on features of future Web browsers, the user might just be able to click on a button to see and save the applet code. If not, he can certainly create his own Web browser to do this. Once saved, the user can run copies of this applet instead of the original, therefore keeping the counter at its initial value.

The above suggested solutions can also all be circumvented through reverse engineering the executable code. As before, the applet author might not want to worry about this threat.

The most secure method to achieve applet security, which is also a way around reverse engineering, would be to pass some information to the server, have a computation done on the server, and pass back the result to the user. However, this poses a severe threat to the secrecy of the user's data. (See section 3.5) If all threats to the user's data have been removed, then the call back to the server can be compared to a call asking for the time. As mentioned before, the response can be faked by the user. Therefore, for the most security on the server side, the user would have to have the least security.

Before leaving this topic, it should also be noted that a user may also wish to copy the applet in order to share it with others. Although the copying cannot be stopped, there are ways to help prevent applet copying. Primarily, these are the same ways used to prevent software copying, such as extra hardware and fingerprinting applet copies. In addition, you can have the applet ping the server every time it attempts to be used. As before, a fake server ping-response can be sent to the applet to solve this problem. Finally, the discussion on doing computations on the server also apply to applet copying.

3.4 Applet Authentication

In an environment where clients must trust executables to run on their machines, perhaps having access to their confidential data and files, authentication is a key element to safe and trustworthy functionality. When a User downloads an applet, he wants to know that the applet is coming from the applet server he specified and not from some malicious interceptor. Likewise, the applet Author would like to insure that another party is not impersonating his server to users, perhaps slandering his name.

3.4.1 Solutions

Currently, the Netscape Commerce Server uses the SSL protocol, which has following key features:

- Server authentication
- Privacy using encryption
- Data integrity

This means that applet authors cannot be impersonated, that applets cannot be altered during transit to hosts, and that applet scripts will reach the host in their original form, in their entirety. Encryption during communication also solves the problem of eavesdropping communication lines to collect restricted applet code. (See section 3.1)

To use the SSL protocol, Commerce Server administrators must acquire a certificate issued by a certification authority approved by Netscape. The certification authority verifies the authenticity of the request, and if it can validate the authenticity of the request, will issue a digitally signed certificate for use by the server. By viewing the information of the certificate in the Netscape Navigator, clients can check the Applet provider's identity. In this manner, clients are assured of the identity of applet providers, while providers know that it would be very difficult for others to impersonate their server to clients.

3.5 User Data

Even with all other concerns settled, the biggest concern still remains: the secrecy of the user's data. Any information on a user's computer may be considered secret and should not be known by the server or anyone. Other information may be less secret. A system needs to be in place so that the user can be certain that levels of security are maintained.

3.5.1 Solutions

The first solution that presents itself is the one already available within HotJava. A user can specify which directories an applet can and cannot access. These access rights are followed for every applet from a particular server for all time until settings are changed.[3] Unfortunately, if a user wants to be completely secure he will have to prevent the applet from accessing any directory. This greatly limits the tasks that the applet can perform.

For a little more power, the user may be given the opportunity to specify different access powers to different directories. For example, an applet can read all files in directory 'A', but it cannot write to any file in this directory. All writing must take place in directory 'B'. This adds complexity to the Web browser. Also, the user and applet writer must agree on which directories are to be used. On the positive side, with this feature the applet cannot destroy any of the user's files.

The secrecy of the user's files is still a question. HotJava is currently set up so that all information sent by an applet must go to the server from where the applet originated[1]. Still, this limits the number of applets that a user would want to execute. For the rest of the solutions in this section I will refer to two applets. The first is an applet that checks for a new virus. The second is an applet that does the protocol for getting a blindly signed coin from a bank.

The fist applet needs to be able to read all of the user's files in order to do its check. This puts the secrecy of user's data in peril. Therefore, the user wants no data flow to go out. So a suggested rule might be that no data may go out once any file has been read. However, there may be times when you want to send out files. Therefore, we allow directories to have either "read" or "read and send" access rights. Once a file in a "read" directory has been read, no more information may flow to the server. Once again, this limits the capabilities of an applet, but it does so at the security of the user's data.

Now, let's take a look at the second applet for blind signatures. Let us say that the applet wants to read all 100 blinded coins, send them to the bank, get the bank 99 requests, send the 99 packets with unblinding information, and get the one signed coin back. As of now, there is nothing stopping the applet from actually sending 100 unblinding packets back. There are two solutions: ask permission from the user for each file sent or have the user do some work requested by the applet.

The first solution has two problems. First, it is not fun for the user to grant permission 99 times and make sure that he does not permit it a 100^{th} time. Secondly, although granting permission for file requests seems valid in this example, other examples may hide information in what files it returns. The second solution, like the first, is not fully automated. In this case, we might have the user put 99 of the r's from a "read" a "read and send" directory.

In short, there is no way to have full automation of all applets and also full secrecy of files. Some compromise needs to occur. Where this compromise lies depends on what is more important to the public.

4 Recommendations

We believe that the highest concern should be placed on user data. With the speed that the Web is growing, there will be many opportunities for applet authors to attempt to compromise user data. However, relatively few applet authors will use proprietary algorithms.

We do believe that the task of reverse engineering the executable will be enough of a security measure for most applet source code. We do suggest, however, that transfer of secret applet executables use encryption options currently available within Netscape. Introducing more security, such as personalized private keys, adds complexity which many users do not want to face. Besides, if someone is willing to go to the trouble of re-routing packets and decrypting, there is a good possibility that they can get around many deterrents that we can present. Therefore, we suggest that simplicity of user interface be weighed more heavily than security of applet executable.

In order to protect the user's data, we suggest that a Web browser be modified to give directories different permissions for different directories. The permissions should be allowed to be any subset of read, read and share, write, create, list, and delete. With these permissions, a user may give himself the amount of protection that he desires.

Finally, we recommend that the Web browser not allow any information to flow back to the server once a file in a "read" directory has been accessed. This ensures that information about a user's files are not hidden in data being sent back to the server.

5 Conclusion

Most innovative schemes face a number of roadblocks on their way to successful implementation. Technological advancements, in particular, are often looked at with cynicism, and have to fight the stigma of being threats to individuals' privacy. For example, Web commerce will likely not be practiced widely for a number of years to come because of the reputation it has received of being an insecure means of doing business.

However, we think that the idea of providing software services over the WWW shows a great deal of promise. We also think that the primary bottleneck will be the bandwidth of the communication channels and not the security issues that we have discussed in this report. We have demonstrated how to virtually guarantee the integrity of the Applet User's data and the Authenticity of the Service, and provided good solutions for the Free-Rider problem and achieving Secrecy of the Underlying Algorithm among other things. Although we have not provided perfect security solutions, and are aware of the fact that perfect solutions for all of the issues discussed might not even exit, we are convinced that our proposals pass the threshold necessary for security not to be a roadblock.

A Terminology

App, Applet An autonomous section of Java code.

Author Author of an HTML file (Web page).

client Workstation that is running a Web browser to view URLs.

HotJava A Web browser that reads and interprets Java references in Web pages.

HTML HyperText Markup Language

Java A new language that allows for applications to be used in Web pages.

server Process that is sending out information about URLs.

URL Uniform Resource Locator. It is an address of a Web page.

User Someone on the client machine.

Web Browser Application that reads HTML code and interprets it.

Web page HTML file.

WWW, Web World Wide Web

References

[1] Sun Microsystems, Inc.
http://java.sun.com/people/mrm/reply.html, May 1995.

[2] Purdue University. *CMAD III Report*,
http://www.cs.purdue.edu/homes/swlodin/cmad_report.html#spoofing, May 1995.

[3] Sun Microsystems, Inc. *Configuring HotJava Security*,
/afs/sipb.mit.edu/project/java/alpha3/doc/misc, May 1995.

[4] Netscape, Inc. *On Security*,
http://home.netscape.com/info/security-doc.html, 1995.