The following paper was originally published in the
Proceedings of the Second USENIX Workshop on Electronic Commerce
Oakland, California, November 1996

WWW Electronic Commerce and Java Trojan Horses

J.D. Tygar and Alma Whitten
Carnegie Mellon University

WWW Electronic Commerce and Java Trojan Horses

J. D. Tygar tygar@cs.cmu.edu
Alma Whitten alma@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

World Wide Web electronic commerce applications often require consumers to enter private information (such as credit card numbers) into forms in the browser window. If third parties can insert trojan horse applications onto a consumer's machine, they can monitor keyboard strokes and steal private information.

This paper outlines a simple way to accomplish this using Java or similar remote execution facilities. We implemented a simple version of this attack. We give a general method, window personalization, *that can thwart or prevent this attack.*

1 Introduction

Computer security experts have long recognized the threat of trojan horse programs [6, 15, 13]: programs that appear to perform one function while actually performing a second, unwanted function. A particular concern arises from the presentation of an interface to a user or consumer. Since human users identify applications by their interface, a human user may be unable to distinguish a legitimate program from a rogue program's mimicking of the first program's interface. These concerns are further exacerbated when the consumer is using electronic commerce protocols on the World Wide Web. The consumer may be required to enter security crucial information (such as credit card numbers, bank routing and checking account information, billing account information, personal demographic data, etc) into the local client. If a trojan horse can grab this information or fool the consumer into submitting this information to a third party rather than to a valid electronic commerce server, then the consumer can unintentionally release confidential information to third parties. This paper gives an example of how remote execution systems such as Java can easily host such a trojan horse attack. We then give a general method of *window personalization* to address this problem.

This work was supported in part by Defense Advanced Research Project Agency (ARPA contract F33615-93-1-1330), the National Science Foundation (NSF cooperative agreement IR-9411299), the US Postal Service, and Visa International. This work is the opinion of the authors and does not necessarily represent the view of their employers, funding sponsors, or the US Government.

1.1 Trojan horses

Consumers face two types of trojan horse challenges when they engage in electronic commerce on the World Wide Web.

1.1.1 Bogus remote pages

A consumer may be looking at a forms page other than the one she believes she is communicating with. For example, although the consumer may believe that she is communicating with Firm X, she may in fact be communicating with another party. URLs are not easy to check, and there have been a number of instances of prank URLs parodying or imitating WWW sites. (For example, `http://www.dole96.com` is Bob Dole's official 1996 presidential campaign site. On the other hand, `http://www.dole96.org` is a humorous parody of the Dole election. Reportedly, many users have been fooled.) In the credit card industry, this is a serious concern — today, fraudulent use of credit cards *by merchants* is already recognized to be a serious problem (this is sometimes called the "Lyndon LaRouche problem" after the fringe presidential candidate who has been accused of credit card fraud.) To address this problem, Visa and Mastercard, in their Secure Electronic Transaction (SET) specification [11], indicated that merchants (both legitimate and bogus) should not receive credit card numbers and similar confidential billing information about consumers. Rather, credit card information entered by consumers should be encrypted before it leaves the local client, so that only an acquiring bank could read the information. Thus, SET addresses the bogus page attack, but leaves open the possibility of local trojan horses.

1.1.2 Local trojan horses

A consumer may have a local trojan horse running on her machine. This type of local trojan horse can:

- imitate remote pages, or
- grab keystrokes (such as SET entry keystrokes) from the local machine. (The possibility of this type of attack has been known in the computer science folklore for many years; recently Nathaniel Borenstein gave a dramatic demonstration of trojan horse keystroke grabbing [2].)[1]

In the past, consumers have been protected by the relative difficulty of loading trojan horse applications onto arbitrary workstations. If the consumer took moderate care to protect herself from loading untrusted software (including viruses and worms), she could reduce the risk of attack to a manageable level.

However, local trojan horses are particularly dangerous for electronic commerce protocols that depend on the local client interfaces to obtain and securely handle confidential information. Examples of systems that use these features include SET (discussed above) and NetBill[16, 3] For example, if a trojan horse emulation of the SET interface is very well done, it may be difficult or impossible for the consumer to determine whether she is dealing with the true program or not. Such a rogue program could transmit the consumer's credit card number directly to the adversary; alternatively, it could store it for later retrieval.[2]

Java applets[10] have changed the balance of power — increasing the possibility of untrusted software and local trojan horses being transferred to a workstation. Java applets are usually loaded onto a workstation without requiring explicit applet-by-applet consent by the consumer. If a Java applet created by an adversary is loaded onto the consumer's workstation, the applet can put arbitrary images on the screen within the browser window and can communicate keyboard information back to a host. A consumer may not be able to distinguish the trojan horse applet display from a valid information request. Thus, Java applets can easily serve as trojan horses that mimic electronic commerce applications. Java is not the only culprit. Other *remote execution* mechanisms such as Omniware [1], Telescript [12], and Dyad [18] provide ample opportunities to download programs to a consumer's computer that can display arbitrary graphical interfaces and transmit information to an adversary. In section 3 below, we discuss an example trojan horse Java applet that we developed which performs its own emulation of a standard Netscape dialogue box. Normally, consumers expect Java applets to display windows and dialogue boxes only with a warning bar underneath them. By writing our own graphics routines, we were able to make our applet bypass the standard Java window library and write directly to the screen. This makes our Java applet output bit-for-bit indistinguishable from the standard Netscape I/O display boxes. As we discuss in section 2 below, standard Java security[3] mechanisms are useless for protection against this type of attack.

1.2 Paper outline

This paper outlines a simple way to accomplish a local trojan horse attack using Java or similar remote execution facilities. We show an implementation in Java of a simple example of this type of trojan horse.

We propose a new mechanism of *window personalization* that can thwart this type of attack. Window personalization allows a consumer to select a pattern for window display that will be unknowable (or very difficult to determine) by rogue applets and other

[1] In his First Virtual work [14, 8], Borenstein and his colleagues argue for a system where a consumer would enter a First Virtual account number, rather than directly entering credit card information. Consumers would have the opportunity to review and decline charges that were improperly made on their account. While this certainly protects the credit card account, there is still an issue of providing maximum protection of the First Virtual account number from trojan horse attack.

[2] Strictly speaking, this type of attack is not limited to electronic commerce protocols. Indeed, there appears to be no limit to the possible mischief that can be caused by trojan horse programs that infiltrate a consumer's machine. A trojan horse might emulate a word processor, causing (potentially sensitive) text to be transmitted to an adversary. It might infect an e-mail system allowing messages to be read by third parties. It might infect a compiler allowing trojan horses to be inserted in locally compiled programs [17]. Ultimately, any application program might be emulated. However, the case of electronic commerce is particularly compelling since there is a well-defined target of attack — the consumer's credit card numbers or bank account information.

[3] Recently, Dean, Felton, and Wallach [5] and others have argued that substantial flaws exist in the Java API library and in the methods used to check type safety. These flaws lead to direct attacks that can be implemented in Java applets. Our work is orthogonal to these flaws: even a hypothetical "perfectly secure" version of Java would permit these types of attacks. We discuss this further in section 2.

transmission media for trojan horses. Through window personalization, a consumer can catch almost all trojan horse emulations of human interfaces; the rogue program can not determine the correct way to format the interface, and so it is likely to appear wrong to the consumer. In section 5 we show how this method can be extended to other applications such as point-of-sale transactions and automatic teller machine transactions.

2 Java

Java is a programming language and associated set of development tools developed by Sun Microsystems. Similar to C++ in syntax, Java is intended for writing programs that will be runnable on a variety of dissimilar computer architectures. This is accomplished by compiling Java programs into an machine-independent byte code format, which can then be run on any computer equipped with a Java interpreter.

Java can be used to write stand-alone applications, but it has gained its widespread popularity by its use to create applets, which are executable objects written in Java and embedded in World Wide Web pages. When a web page containing a Java applet is loaded into a Java-enabled browser, the applet's byte code is also loaded, interpreted and executed within the browser. Applets can be used to create web pages containing animation loops, arcade games, and other highly interactive applications which would have been infeasible if executed from the server due to latency and server load.

Executing applets in a secure manner must necessarily be the responsibility of the client's web browser. Java-enabled web browsers are expected to make use of Java's well defined type system in order to verify that the byte code they receive corresponds to a valid Java program. The Java language also includes an extensive application program interface (API) specification, which is the only way that applets can perform any input/output or other system functions. The web browser must include an implementation of the Java API class libraries, with appropriate restrictions on what applets, as untrusted code, are allowed to do.

In this discussion, we will consider Netscape Navigator as our example of such a Java-enabled web browser. Navigator incorporates three significant security restrictions in its version of the Java API:

1. No access to the local filesystem is allowed.
2. Socket connections are allowed only to the server on which the web page containing the applet resides.
3. Windows and dialogue boxes opened by applets contain vivid banners labeling them as "Untrusted Java Applet Window".

Recent work by Dean, Felten and Wallach [5] has focused on security flaws in Netscape Navigator Java which are due to implementation errors, such as significant weakness in the Java class loader. These flaws may be exploited by a sophisticated attacker in order to bypass any or all of the above API security restrictions. In our work, by contrast, we have chosen to assume a correct implementation of Java in which the intended security restrictions are perfectly enforced, and to show that there exists a particular type of attack on security which is not prevented or even significantly hindered by these restrictions.

We begin by pointing out that Java applets are allowed complete control over the screen pixels that fall within their allotted area of the browser window, and that it is unrealistic to expect that any significant limits could be placed on this control while still allowing applets to perform animation. Similarly, applets have full access to information about mouse and keyboard events that occur within that area, and that access cannot be limited much without severely hampering the interactivity that applets are intended to provide. The combination of these two abilities allows applets to bypass the third security restriction with relative ease, since it is a simple matter for an applet to draw and manage its own dialogue box within the confines of the browser window. Not only would such a dialogue box have no warning banner, it could have any appearance that the applet's designers desire.

If a rogue applet is successful in stealing information from its host environment, whether by tricking it out of the consumer or through some other method, the second security restriction does little to prevent the applet from transmitting its stolen information over the network. The applet may, with perfect legitimacy, connect back to its server via socket and report what it has stolen; alternatively it may simply make an http request to any machine on the network and include its stolen information as part of the request string.

In summary, even given a hypothetical perfect implementation of Java and the listed Navigator security restrictions, a rogue applet is still free to trick information out of the consumer through clever control of its area of the screen display, and can easily communicate such stolen information over the network to any accomplice it desires.

Figure 1: Authentication dialogue box displayed by WindowsNT Netscape browser

2.1 Signed Applets

A great deal of attention has recently been paid to the strategy of increasing Java security by having applets certified and digitally signed by some trusted authority. Current proposals for this argue for certification only as a method for establishing the identity of the code provider; it is expected but in no way guaranteed that code providers with reputations to protect will take appropriate care to verify the safety of the code they sign. Given that Microsoft, a company that has announced interest in Java code signing, has in the past released software containing hostile code [7], we believe that such certification is best used in conjunction with additional security strategies, such as the window personalization technique described in this paper, which is complementary to (and independent of) applet signing.

It may also be the case that we will see the further development of the code signing strategy beyond simple certification of provider identity to encompass at least limited verification of code safety and other attributes. In the event that this occurs, we once again note that even the most competent and conscientious certification authority is likely to occasionally err and certify a malicious applet as safe. Given the extremely sensitive nature of some of the information that such applets might attempt to steal, it clearly remains advisable to minimize that risk through additional strategies where possible.

Finally, we point out that there will be substantial pressure on many web users to allow some uncertified applets to run; the broad variety and large number of applets present on the Web today argues that applets are fulfilling a need — and requiring and checking digitally signed certification for every applet is likely to be viewed as being a logistically complex problem.

3 Example Attack

We here describe a very simple example of an attack using a trojan horse Java applet. The reader with imagination will have no trouble constructing a more elaborate example that would be sensitive to the specific browser and client platform used, and could do a broader range of sophisticated attacks.

The Netscape Navigator browser uses a standardized dialogue box to request username/password pairs when a web server responds to an http request with a demand for authentication. Figure 1 shows the appearance of the authentication dialogue box used by WindowsNT Netscape. To demonstrate the use of Java for trojan horse attacks, we have written an applet which fakes the appearance and behavior of this dialogue box; for the interested reader, this applet can be found at `http://blind.trust.cs.cmu.edu/spoof.html`.

The applet attempts to trick users into believing that they are authenticating themselves as usual to a particular web site. This is done by mimicking the appearance and behavior of the real site within the browser window. If the user clicks on a link that would cause the real site to request authentication, the applet displays its fake authentication dialogue box; clicks on other links are handled by loading the correct pages from the real web site. If the fake dialogue box fools the user into entering their username and password, the applet sends the stolen information back to a waiting process on `blind.trust.cs.cmu.edu` via socket connection, tells the user that the web site server is not responding, and loads the real web page from the real web site, so that the user's next attempt will succeed.

Implementing this applet required fewer than 200 lines of Java code[4]. As shown in Figure 2, our ap-

[4] Java source code for our applet is available at `http://blind.trust.cs.cmu.edu/spoof.class`.

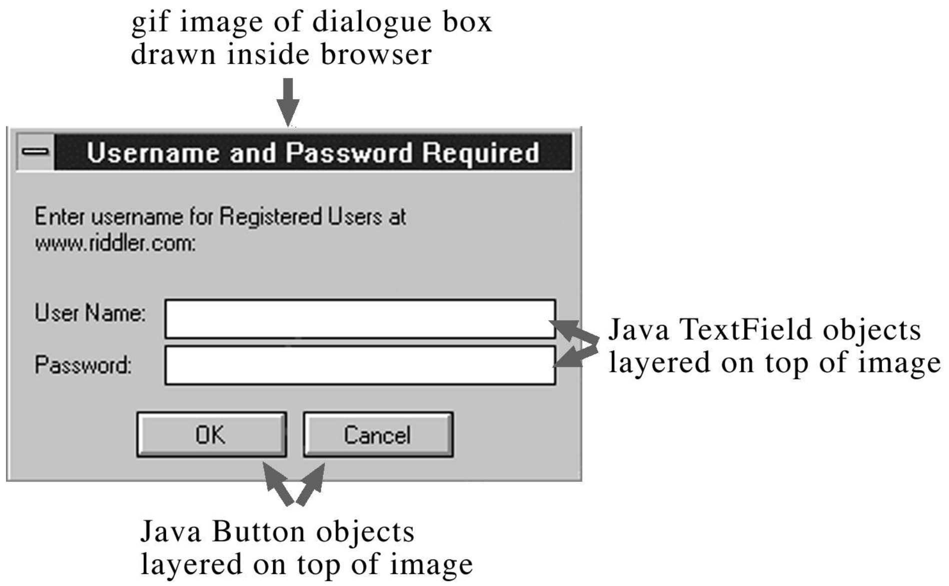

Figure 2: Our trojan horse version of the authentication dialogue box

plet uses a simple image of an actual authentication dialogue box, acquired by screen capture and edited in Adobe Photoshop, to mimic the appearance of the dialogue box within the browser window. Actual Java `TextField` objects are drawn on top of the image to provide the necessary pair of editable text fields. We overloaded the applet event handling routines in order to make the simulated buttons on the dialogue box behave appropriately in response to mouse events, including performing the necessary socket communication and http request when either button is actually pressed.

This applet was created in order to demonstrate a point, and as such, it does not currently present a seamless attack. For example, the alert user may notice that the dialogue box is not draggable, or that the Location URL displayed by Netscape is not what the user expected (however, this is located in a position where many users might ignore the information displayed by Netscape). Some of the current limitations could be fixed by putting more work into the applet, while others, like the Location URL, are effectively enforced by Netscape.

This attack can easily be generalized to mimic any user interface element that has a standardized appearance and requests sensitive information from the user; some other obvious examples of such information are credit card numbers for electronic commerce applications, account numbers for banking, or, in the case of hospitals, a wide variety of potentially sensitive medical information.

4 Window Personalization

Trojan horse attacks such as we have described rely on imitating the visual appearance of some program that the consumer already trusts with sensitive information. Standardized user interface appearance is a great help to the designers of such attacks, since it removes the problem of determining what appearance the consumer expects.

We propose that programs which require the consumer's trust should employ window appearances which are easily recognizable to the consumer yet difficult to predict by an attacker. The obvious mechanism for accomplishing this is to require the consumer to personalize the appearance of the software at the time the trust relationship is formed, which may be installation or account initialization depending on the application involved. Figure 3 shows a hypothetical user's personalization of the Netscape dialogue box with a Batman logo. Our trojan horse attack would not be able to predict this personalization, and its ability to fake the appearance of the real dialogue box would be substantially weakened.

As another example, consider a SET dialogue box. If the dialogue box has a standardized, predictable appearance, it will be an easy target for a similar trojan horse interface-emulator applet. On the other hand, suppose that the consumer is able to select a background display for the dialogue box (out of a range of thousands of potential displays). As long as the Java applet is not able to determine what

Figure 3: Authentication dialogue box protected by personalization

the background display expected by that particular consumer is, the applet will not be able to credibly simulate the user interface of the program under attack.

In order for this strategy to be most effective, the trusted application should not only require the consumer to participate in a personalization process, but should, within that process, strongly convey to the consumer the importance of not trusting any window that does not display characteristics in conformance with the personalization. The process should also be designed to maximize the unpredictability of the consumer's personalization choices; for example, when offering the consumer a choice of window backgrounds, it would be best to offer more than just two or three backgrounds to choose from, and to present the possible backgrounds in a randomized ordering or layout to compensate for the fact that most consumers may simply choose the first option they are given.

Window personalization is most effective against attackers who either do not have an opportunity to find out the potential victim's personalization choices, or who are not interested in creating a trojan horse designed to fool a specific individual.

5 Extensions

In the more general case, window personalization may be applicable to any situation in which users need to confirm that an interface is being presented by an entity with which they have a prior trust relationship, and not by an imposter.

As one example, there have been a number of publicized cases of fake automatic teller machines (ATMs), set up in public places and used to steal card and PIN information from unsuspecting customers. If window personalization were employed, the ATM could be expected to display the user's personalized window style at the time it requests the entry of the PIN number.[5] The absence of the correct window style would be a signal that the machine did not have access to the bank database of window styles corresponding to cards, and therefore should be regarded with suspicion.

A similar problem arises in the case of point-of-sale (POS) transactions, in which the user must communicate with a trusted entity, the bank or perhaps the stored value card, via untrusted POS equipment belonging to the merchant. A corrupt merchant might have modified the POS equipment to display a false charge amount, in an attempt to trick the customer into entering a confirmation for a charge which is actually larger than that which appears on the POS display. If, however, a personalized display style is a shared secret between the user and the trusted entity, then the user can take the display of the charge amount in the correct style as confirmation that the amount displayed is the actual charge according to the trusted entity, even though it is displayed on untrusted equipment.

The careful reader will note that these examples are not fully satisfactory. He will wonder: why can't a trojan horse POS system perform a "man in the middle" attack — actually connecting to the POS network, observing the message transmitted in both directions, and recording the information for later pickup? That attack is certainly possible, although it requires substantial preparation; the attacker would need to connect to the POS network

[5] Note: This differs substantially from the way that ATM machines operate. Typical ATM authentication today relies only on local encrypted information stored on a single ATM card. No central database query is done until after authentication is complete [4]. The solution discussed here would require an initial lookup by a centralized machine of the user's particular window personalization style, making the ATM protocol less efficient.

(or subvert an existing POS), properly authenticate the bogus POS machine to the bank, and then interpret formatting messages coming from the bank and properly adjust the display presented to the user. While this might perhaps be possible, it certainly would require a much higher level of skill to successfully pull off this attack.[6]

6 Conclusions

The popularity of Java applets means that a vastly increased number of users will run untrusted software on a regular basis. Current security strategies that focus on limiting applet access to dangerous system calls are useful but insufficient; the ability of applets to freely manipulate even a portion of the screen display, combined with the ability to send information back to their source machines via the network, allows applets to mount trojan horse attacks by imitating the user interface elements of trusted software. It is infeasible to increase security by further limiting those applet abilities, because doing so would greatly limit the ability of applets to provide the interactivity and animation on which so much of their appeal is based.

Window personalization is a supplementary security strategy which is independent both of the strategies described above and of code signing. If users are strongly encouraged to personalize the appearance of the user interface elements of their trusted software, in ways that are highly recognizable to the user yet very difficult to predict by others, then the designers of rogue applets will not be able to mimic those user interface elements convincingly because the personalized aspects of the appearance will be unknown to them.

The window personalization strategy can be extended to any situation in which a user needs the ability to verify that a user interface is being presented by a trusted entity and not by an imposter. Two examples in which this need for verification may arise are automatic teller machines and point of sale transactions.

[6] Even smartcards can not completely solve these problems; for a discussion of the difficulties with the smartcard approach to solving the POS integrity problem, see [9].

References

[1] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the 1996 ACM SIGPLAN Symposium on Programming Language Design and Implementation*. ACM Press, May 1996.

[2] Nathaniel Borenstein. Vulnerability of software based credit card encryption. At `http://fv.com/ccdanger/index.html`; see also *San Jose Mercury News*, 29 January 1996, "Program shows ease of stealing credit information" by Simpson L. Garfinkel.

[3] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. NetBill security and transaction protocol. In *Proceedings of the First USENIX Workshop in Electronic Commerce*, pages 77–88, July 1995.

[4] Donald Watts Davies and W. L. Price. *Security for Computer Networks: an Introduction to Data Security in Teleprocessing and Electronic Funds Transfer, 2nd Edition.* Wiley, 1989.

[5] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to Netscape and beyond. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[6] Dorothy Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

[7] Peter J. Denning. *Computers Under Attack: Intruders, Worms, and Viruses.* ACM Press, New York, N.Y., 1990.

[8] N. Borenstein *et al.* Perils and pitfalls of practical cybercommerce: The lessons of First Virtual's first year. In *Proceedings of Frontiers in Electronic Commerce*, October 1995.

[9] Howard Gobioff, Sean Smith, J. D. Tygar, and Bennet Yee. Smartcards in hostile environments. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, November 1996.

[10] James Gosling and Henry McGilton. The Java language environment: A white paper. Technical report, Sun Microsystems, May 1996. See also `http://www.javasoft.com/doc/language_environment/`.

[11] Mastercard International and Visa International. *Secure Electronic Transaction (SET) Specification*, June 1996. See `http://www.visa.com` or `http://www.mastercard.com`.

[12] General Magic. Telescript technology: The foundation for the electronic marketplace.

Technical report, General Magic, 1996. See also `http://www.genmagic.com/Telescript/Whitepapers/wp1/whitepaper-1.html`.

[13] Peter G. Neumann. *Computer Related Risks*. ACM Press and Addison-Wesley, 1995. Also see Risks Digests at `ftp://ftp.sri.com/risks`.

[14] Darren New. Internet information commerce: The First Virtual approach. In *Proceedings of the First USENIX Workshop in Electronic Commerce*, pages 33–68, July 1995.

[15] Adrian R. D. Norman. *Computer Insecurity*. Chapman and Hall, London, 1983.

[16] Marvin Sirbu and J. D. Tygar. NetBill: an internet commerce system optimized for network delivered services. *IEEE Personal Communications*, pages 34–39, August 1995.

[17] Ken Thompson. Reflections on trusting trust. In Robert L. Ashenhurst and Susan Graham, editors, *ACM Turing Award Lectures, The First Twenty Years, 1966 – 1985*, pages 163–170. Addison-Wesley, 1987.

[18] Bennet Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 155–170, July 1995.