Exploiting Trusted Applet/Server Communication

Jonathon Giffin, Hong Lin

Department of Computer Science

University of Wisconsin−−Madison

1210 West Dayton Street

Madison, WI 53706

{giffin, honglin}@cs.wisc.edu

Abstract

We bring light to a security risk that a server deploying applets runs. In a joint client/server development environment, programmers may not define a server trust model that excludes the remote code. A server communicating with applets may then extend its trust boundary to include the applets. A client impostor process can exploit this trust by sending invalid and malicious data to the server. We developed an interactive tool to better enable attacks against remote machines. Tests conducted against four live Internet servers reveal trust vulnerabilities that may be exploited. Preliminary observations indicate this same faulty trust model exists in other common Internet content delivery technologies. The first step towards securing a server is the explicit definition of a trust model and the development of safeguards to ensure this model is accurately reflected in code.

1 Introduction

An unsafe trust model exists in Java applet and server programming. Authors of applet servers may incorrectly and unconsciously view remote applets as trusted components of the application and rely on the applets to send only valid data to the server. We examined exploits of this faulty trust model, attacking server processes that were unintentionally authored to place trust in remote executing code.

The security of a machine downloading and running a Java applet has been widely accepted, however, the safety of a web server deploying applets cannot be guaranteed. Applets execute in a sandbox that prohibits local file system access and other operations potentially damaging to the executing host. To workaround these restrictions, Sun Microsystems suggests the following applet execution model [19]: An applet creates a connection to its server and sends all offending connections––including file system access and connection establishment to third parties––back to the server for execution. The applet is physically running on the client machine but is building an execution environment on the server. If the server implicitly trusts all communication from the applet, it may be exposed to attack.

It is not difficult to subvert the communication protocols used between servers and applets. Free bytecode decompilers are available to any user on the World Wide Web, so applet source code is not secure. Using the source code as an aid in understanding the communications between a server and an applet, an attacker can later impersonate the applet, transmitting invalid and potentially damaging data to the server.

We examined extensions of trust in live applet servers running on the Internet. Based on decompiled applet code, we developed *random, alphabet,* and *scripted* attacks against the server process accepting communication from the applet. We designed a tool that poses as the applet, connecting with the server and sending and receiving data. This tool helps users construct data streams never sent by the original applet but that may be accepted as valid by the server.

These attacks have been successful against applets using Java sockets to communicate with a server process. Any server behavior that can not be generated by legitimate use of the applet shows the existence of a trust vulnerability. We targeted four live Internet servers, and all showed adverse effects.

The process of collecting web pages to find applets also revealed dynamic content delivery technologies frequently used in the World Wide Web. We examined several of these technologies and found that the trust model common in applet servers exists widely across the Internet.

Section 2 explains our view of the applet/server trust model. We describe our methods and the tool we constructed in Section 3. Section 4 presents results and a discussion of several attacks targeted against live servers on the Internet. Section 5 includes data illustrating the environments in which applets are used. Section 6 describes similar environments on the World Wide Web with trust model exploits. We list our thoughts on potential future work in Section 7. Section 8 examines related work. Section 9 presents our conclusions.

Throughout this paper, unless context indicates otherwise, the word "server" refers specifically to the process that communicates with remote applets and not to the physical machine this process runs on.

2 Applet/Server Trust Model

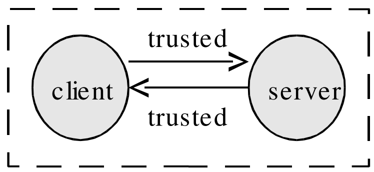

Figure 1: The server's trust boundary includes the client applet. Communi– cation is trusted in both directions. This trust model is faulty, leading to possible exploits.

Java applets and a server process communicating with the applets together comprise a distributed application. The applets are deployed on a web site and executed on an unknown client host. An important measure of the security of the server is the extension of its trust boundary.

An applet and server are most likely written by the same author or group of authors. Applets, by their nature, are individual and unique programs, implying that servers must be unique to each applet. Consequently, the applet and server will be jointly developed. Server security and robustness may be assumed by testing it against the applet rather than against all possible inputs. In these conditions, the server's trust boundary is implicitly extended to include the applet and communications link [Figure 1]. Such servers may be vulnerable to trust exploits. In contrast, a more robust trust model will assume applets may fail or be subverted.

Our experiments attempted to exploit servers with this faulty trust model. Successful experiments indicated the existence of these models and illustrated the potential for exposed dangers.

3 Methods

The development of tests proceeded in three stages. First, we collected applets from the Internet for study. Second, we decompiled the applets to enable analysis of their communication methods. Last, we targeted several selected applet servers for attack using an interactive tool we developed.

3.1 Applet Collection

We gathered applets using two methods. First, we authored a web spider that collected a substantial number of applets by logging the current page in a database, parsing the page's HTML code, saving any embedded applets, and queuing all the links on the page. The spider ignored any links pointing to pages previously visited. Its path could be controlled somewhat by the choice of start page, however, the pages hit are mostly determined by the construction of links in the World Wide Web. Although we can not claim our results are representative of the distribution of applets on the Internet, we did attempt to hit a cross–section of the available web pages by restarting the spider in different locations several times.

Second, we intentionally searched for Java chat rooms, game rooms, and counters because we believed them to be the most popular types of applets. These manual applet searches also collected applets found behind login pages that are unreachable from hyperlinks and consequently not found by the spider.

3.2 Applet Code Analysis

We reverse engineered all collected applets to source code level. Bytecode decompilation is straightforward and quite effective [18], and decompilers are widely and freely available. After testing several decompilers [2,10,20,22], we found the lightweight, cross–platform program Jad suitable to our needs [13]. Jad correctly decompiled class files for which we possessed the source code. While we could not, of course, compare the decompiled code of an applet with the author's original, the generated code appeared to accurately represent the observed behavior of each applet. Although not all code would recompile, perfect decompilation was not required for an analysis of communication protocols.

Decompiled applets fell into several groups of increasing interest to this project:

All data self–contained: These applets do not connect with the applet server. Specific data may be stored in `PARAM` tags on the host web page and loaded by the applet using the Java method `getParam`. Some text scrollers operate in this manner.

Receive input only: These applets typically have `URL` objects and use the Java methods `getImage` or `showDocument` to initiate the transfer of data on the remote server to the client machine. All transfers occur via HTTP on port 80. For example, The applet `fphover`, included with Microsoft FrontPage, changes an image on a mouse rollover and must load all needed images from the server.

Communicate with HTTP server: A number of applets use the Java class `URLConnection` to create input streams from and output streams to the remote web server. These applets typically receive user input, perform some modification of that input, and construct and send an HTTP request to the server. Although they permute user specified data, these applets still connect only to web servers.

Communicate with applet server: A few applets use the Java `Socket` class to build a TCP connection with a server process other than an HTTP server. Such server processes are typically written specifically for use with this applet. The communication protocol is usually structured, but it is defined by the authors of the applet and server process. A server may act as a trusted third party enabling communication between applets running on different hosts. Chat applets, for instance, communicate with a specific chat server to send and receive messages from others.

Our interest rested in applets of this last type. Just as the server acts as a trusted party to the running applets, we hoped to find applets that were trusted by the server. Perverting the communication sent from the applet to the server could have adverse effects on the server.

We further analyzed applets that communicated via sockets to their server. We reconstructed the communication protocol as nearly as possible with only client code available. The process was entirely manual: observe an input from the server and trace the execution path of the applet to identify the appropriate response. Logging the packets sent between a running applet and its server would help this process; unfortunately, we did not have access to facilities enabling this benefit.

3.3 Attack Procedure

Once we understood a protocol, we began attacks against the server. We feel laboratory testing is invalid: applet servers are unique and not freely available, and, more importantly, a server set up in a lab will not match a server in the wild. Consequently, our attacks were conducted against live applet servers visible to the public. These attacks were of four types.

Manual probes: As a tool of discovery, user–specified strings or byte streams are sent to the server. These strings often aid in the reconstruction of an applet's communication protocol but are also useful in characterizing server response to specific bytes of data.

Constructed alphabet streams: Servers may be prone to the use of special characters. This attack allows a user to define an *alphabet*, or a set of characters, words, or phrases that appear to have special meaning for the communications protocol, the server operating system, or popular programming languages. Each element of the alphabet is called a *token*. Manual probes can help a user determine what tokens are useful to add to an alphabet. Streams of random assemblages of tokens are sent to the server in the hope that these meaningful symbols will elicit some response.

Random byte streams: All observed protocols expect a human readable stream of data. As a generalized form of the alphabet attack, the random attack sends a stream of bytes that includes both printable and non–printable characters. As in the alphabet attack, certain printable characters have significance to the server and applet. Further, some bytes––such as the null character––have special meaning in programming languages and may affect server behavior. The random attack may be viewed as a network version of previous fuzz test experiments [16].

Neither random streams nor alphabet streams limit the length of the stream they transmit, so they both exploit the possibility of a buffer overrun and memory reference fault at the server. The random stream can be defined to exclude the transmission of `\0` and `\n`, both of which often signify the end of a sequence of data. All such symbols are excluded by default in an alphabet stream and will be included only if they exist as part of a token.

Scripted attacks: If an observable weakness in a server is found, an attack will be designed to exploit the problem. These attacks are highly repeatable. Further, a scripted attack developed for one server may be useful in attacking other servers with similar implementations.

Any attack is considered successful if it results in observable behavior that cannot be generated by use of the original applet's interface. However, we do not consider a server–initiated close of the connection as a successful attack, for this is often a desired action in the face of invalid data.

3.4 Interactive Tool Development

To help facilitate discovery of both appropriate attack strategies and the communication protocol used, we developed an interactive tool allowing a user to specify a form of data to be output to the server and allowing easy observation of responses. The tool was developed after preliminary analysis of applets collected early in the spider process. Given our discovery that only human–readable messages are passed, the output to and the input from the server are displayed as text. Users interact with the program with a windowed interface.

The tool yields many benefits. First, users can choose from the four attack types and change the type over the course of one connection. Alphabet and random attacks can be run automatically, with the tool continuously generating a stream of data, or manually, requiring the user to verify each packet before

1. No Communication	96	22.50%
2. Receive Input Only	265	62.20%
3. Communicate with HTTP Server	60	14.10%
4. Communicate with Applet Server	5	1.20%

Figure 2: Of the 426 applets collected, most receive input only from a remote server. Five applets were of most interest to us, and four of the applets in this group were analyzed fully.

it is sent. These manual attacks allow a user to insert other forms of data within an alphabet or random stream. Second, servers that use timeout values will be more easily subverted with a tool that rapidly provides responses to challenges. Third, scripting helps an attacker incrementally establish the communication protocol between the applet and the server by executing message exchanges up to a known point and allowing the attacker to try new messages at that point. As correct messages are found, the known point is pushed further in the message sequence. Fourth, all message exchanges are logged to allow easy replication of an event. Logging is particularly useful when a randomly generated data stream produces an interesting result. The exact data stream may then be repeated, enabling reproducibility of results.

This tool is useful beyond attacks of Java applet servers. Many network server processes would benefit not only from random fuzz tests, but from alphabet attacks. Authors could load the alphabet with tokens that hold special meaning in their programs to evaluate the program's robustness.

4 Results

We collected and analyzed 426 applets. See Figure 2 for a breakdown of applets collected by type of connection to a server machine. Five applets fell in the fourth group of communication use, connecting via sockets to a server process other than an HTTP server. Given available time, we analyzed the communication protocol in four of these applets and subverted the server that communicated with each applet. We have replicated all attacks successfully.

All protocols passed human readable ASCII strings to and from servers. Applet code simply compared an incoming string to a set of expected strings written into the application. When a match was found, a certain predefined action was taken. As we expect servers to process data similarly, the only possible attacks involve permutations of text strings.

We present detailed analysis of the communication protocol and successful attack against each of the four applets: AnnoyingChat, OnLineCounter, Counter, and SpaceBattle.

4.1 AnnoyingChat

AnnoyingChat is an applet that provides simple chatting capabilities to a client [1]. Any user can log in with any user name and then send and receive messages to and from other individuals in the chat room. Server connection information is stored in the HTML code as parameters to the applet. The server appears to have the ability to maintain a number of different chat rooms. We observed no other users of the chat room during our time testing.

Other than initiation messages, the communication protocol is made of strings containing three parts separated by pipe characters. Enumeration is given for reference only:

1. `UserID|General Announcement|Text Here`
2. `UserID|UserName|Message Here`

The UserID is an integer provided by the server when the applet opens the connection. If a message of type 1 is sent, no user name appears with the message. (This is used only when a user enters or exits the chat room, and the message is generated by the applet, not the user.) Messages of type 2 contain the message sent by a particular user. The newline character marks the end of a message.

We observed two interesting behaviors. First, the UserID field never appears to be checked. In fact, all inserted messages with any non–empty string replacing the true UserID were accepted as valid. Second, the server appears to simply accept a text string from one client and forward that same text string, unchanged, to all other clients in the same chat room. It is nothing more than a form of echo server. In contrast, many other chat servers parse incoming data and send only messages that correctly fit the protocol.

This applet throws an exception when receiving any message starting with a pipe character. Since the server does not check the validity of messages, an inserted message with a leading pipe will be accepted by the server and broadcast to all client applets. All applets then throw an exception and lose the ability to display any further messages. Moreover, Netscape users must clear their disk and memory cache before opening a web page with *any* applet. This appears to be a bug in Netscape, as other browsers handled the exception without further difficulty. In any event, this is a denial–of–service attack of greater or lesser severity against all current users of the chat room.

Two points of failure exist: the hostile incoming message is accepted as valid at both the server and the applet. Removing any incoming message with a leading pipe would close this hole.

This was run as a scripted attack.

4.2 OnLineCounter

The OnLineCounter applet updates the page hit–count in real time as other individuals browse to the page [14]. The client applet has an extended connection with the server, and they periodically exchange handshaking messages. If the hit count has changed since the last handshake, the server sends an additional message containing the new value. Server location information is included in the applet parameter tags in the HTML page.

There are four types of messages:

1. `Call0 0` *`PageURL`* `No 1 -1`
2. `Call0` *`HandshakeID HitCount`*
3. `Call` *`HandshakeID`* `Kontrola`
4. `Answer` *`HandshakeID`*

The type 1 message initiates the communication by identifying the location of the page containing the counter. We have been unable to determine the significance of the individual zero or of the data that follows the PageURL.

The type 3 and 4 messages define the handshaking protocol. The applet periodically receives a Call message with a random integer identifier and must respond with an Answer message and the same identifier before a timer at the server expires. If a type 2 message with the last HandshakeID is received, the hit count has changed.

This applet, both in theory and in construction, is bizarre. First, we see no good explanation for the Call–Answer handshaking. Failure to respond or a response containing an incorrect HandshakeID results in a server close of the connection, so correct responses are enforced. However, there are more efficient means of detecting a closed connection and it certainly doesn't provide any form of security, so the motivation for this handshaking is unknown.

Second, the applet code appears to be the *same* code that is running at the server. Significant consternation was encountered when trying to analyze the applet code until we realized we had both the client code and the server code in one Java class. In fact, sending Call messages to the server results in the server responding with the appropriate Answer message. At this time, we have not discovered an exploitable weakness in the server code, but this does not preclude the existence of security hole.

While the possiblities of developing a meaningful attack are increased because an attacker holds the server code, simpler attacks are possible. The server fails on random byte streams. During a random stream attack, the server closed the connection suddenly. Further attempts to connect to the counter server were unsuccessful. All pages using the OnLineCounter applet displayed only a blank box where the counter was previously displayed. The machine hosting the counter server still responded to ping, so it appeared only the counter server process itself had failed. The server was down for the better part of a day, appearing to have been restarted on the day after our attack.

4.3 Counter

Counter is a straightforward applet delivering a page hit count once at the load of the page [9]. Two types of messages are sent to the server. Non−whitespace text is emphasized only to separate it from special characters.

1. `**e_peer**\n\t**user**\t***PageURL***\n\n**a**\n`
2. `**counter**\n\t**url**\t***PageURL***\n\n**a**\n`

PageURL is the web page containing the counter. The e_peer message serves no discernible purpose and the communication is successful whether or not it is transmitted. The server response to the counter message includes the hit count value. The two characters `a` and `\n` that close each message may also be omitted with no observable change in behavior.

The counter server has been robust against both random and alphabet attacks. However, a possible exploit is available in the PageURL field. The server accepts all printable characters as a valid URL. If the URL is stored in a database of entries and an exploit of the database exists, this server is a delivery tool enabling execution of the exploit. Section 6.3 shows an example of a database attack.

We did discover an interesting flaw in the server, although we speculate it is intentional. If the second message is sent with the word `user` replacing `url`, the server will always respond with a hit count value of 666. We consider this to be incorrect behavior as it is not treated as an error code in the applet. Given its significance in American Christian culture, we believe this response was a programmer's idea of a joke.

4.4 SpaceBattle

Multiplayer Internet games are a popular use of sockets in Java. SpaceBattle is a strategic two player game similar to the board game Battleship playable on a commercial web site [17]. Players also have the ability to chat with each other. The messages passed in this program are conceptually similar to remote procedure calls:

`#`*`FUNCTION`*`#[`*`argument`*`[,`*`argument`*`,`*`argument`*`,]]`

All function names are coded into the applet. Server responses are of the form

`#`*`RESPONSE`*`#`

with expected responses also included in the applet code. Any other word sent between pound signs is treated as a text string.

The server has been robust against random and alphabet attacks, although the format of the chat protocol has been found to be flawed. Once a chat session is established, chat messages are sent with no function name at all. Consequently, the applet user may manually type in certain function names complete with pound signs and the text will be treated as a command to the server. In particular, entering `#QUIT#` as a chat message will close the TCP connection at the server. The applet code, however, did not supply the termination command and treats the disconnection as an error. This problem could be fixed simply by sending chat messages as an argument to a function name.

4.5 Trust Models

The successful attacks demonstrate that servers extend trust to applets. Servers expect that communication with remote clients will match the coded behavior of the applet. Server authors do not consider the prospect of a hijack or an impersonation of the applet. The veracity of incoming data is blindly accepted. When this data differs from the designed protocol, the server and applets respond unpredictably.

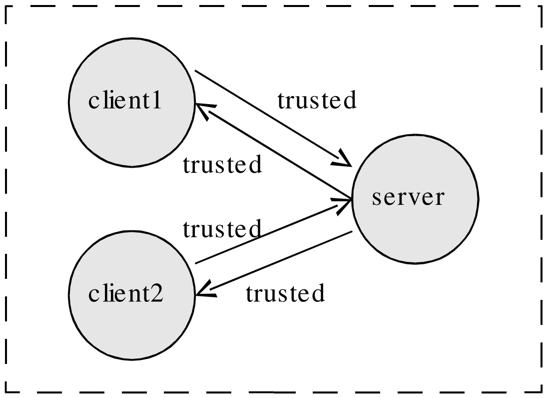

Figure 3: As trusted communications from one applet are passed by the server to others, trust develops among all the applets currently running. A client's trust of the server has grown to include all applets on other unknown hosts.

The four examined applets match the trust model presented in section 2. Each individual applet trusts data from the server and the server trusts data from each applet. However, AnnoyingChat extends this model further. Given the nature of a server that echoes data from one client to all other clients, trust implicity forms among all clients. Each applet's trust boundary extends to include all other applets executing on other hosts [Figure 3].

A paradigm shift is required. As authors of distributed applications, applet and server programmers must explicity determine the extent to which trust is granted. All incoming network data must be treated as insecure and hostile and accepted only after verifying that it matches an expected model of communication [Figure 4].

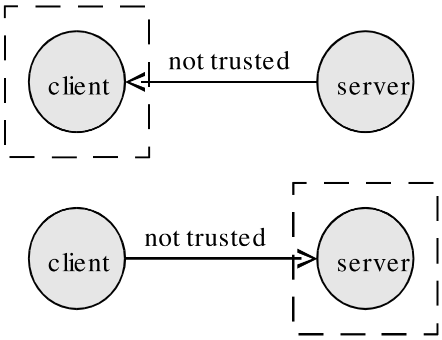

Figure 4: To avoid trust vulnerabilities, applets and servers must explicitly treat all communication as hostile. Only after verification of the validity of incoming data should the data enter the trust boundary of the program.

Category	*# of pages*	*% of total*	*# of pages with an applet*	*% of pages of this catetory*
Organization	9121	2.24%	166	1.82%
Government	2729	0.67%	36	1.31%
Network	10061	2.48%	109	1.08%
Commercial	211543	52.07%	1914	0.90%
Educational	101211	24.91%	743	0.73%
Other	71636	17.63%	1096	1.53%
Total	**406301**	**100.00%**	**4064**	**1.00%**

Figure 5: Statistics on the types of web sites visited by the spider. Many pages in the "other" category include pages from foreign nations whose classification into a known category was not possible. The number of applets contained in sites of each type is also listed. While most applets came from commercial sites, a web site for a non−profit organization is most likely to contain an applet.

5 Spider Survey Results

In the process of gathering applets, the web spider collected 406,301 distinct web pages from 86 different countries. (For comparison, Google indexes almost 1.35 billion pages [8]). We attempted to get a cross−section of popular web sites, focusing on commercial and educational sites. Samples include sites rated most popular by a World Wide Web measurement company [12]. We conducted basic statistical analysis of this data to provide perspective on the types of applets collected by the spider [Figure 5]. More importantly, this analysis reveals dynamic web delivery technologies currently used.

Most web sites use server−side technologies to present clients with customized content and client−side technologies to beautify this content and enhance the page's presentation effect. Server−side technologies take user input to generate page content. A server receives a file request with attached user data. The file contains a script or executable program, run by the server using the data provided by the user and producing HTML code as output. The generated HTML is returned to the client browser. The most popular server−side technologies include CGI (Common Gateway Interface), ASP (Active Server Pages), JSP (JavaServer Pages), PHP3, and ColdFusion [Figure 6]. Some websites use less powerful technologies such as JHTML (ancestor of JSP) and SHTML (Server−side HTML). JavaScript is a client−side technology used by some sites to verify user input. Applets, of course, provide a client with functionality and interactivity not present in other technologies.

6 Further Trust Model Exploits

Although Java applets are not the dominant dynamic web page delivery method, the same trust model exists in these other technologies. When client and server code are authored together, trust is often unintentionally extended from the server to the client. We have found vulnerabilities arising from this blind trust in currently popular technologies.

6.1 Bypassing Client–Side Validation

Some online forms are validated on the client machine by JavaScript so that the client need not wait for the round trip time to the server. If the JavaScript checks are bypassed and the server does not repeat these checks, incorrect data may be accepted by the server. It is simple to bypass JavaScript checking: first download the web page containing forms with JavaScript validating functions, then remove these functions from the execution flow. Load the modified page into the browser and fill out the form with invalid data. If this data is accepted by the server, trust has been extended to the client–side JavaScript code.

We randomly checked 25 popular commercial web sites, among which 4 use JavaScript to validate user input. We found the following two web sites vulnerable:

- Esprit.com [6]

  Invalid dates such as `dd/mm/0000` were successfully saved in their database.

- Citibank.com [4]

  The first few questions in the account application form can be skipped, apparently against the original design of the site.

Total Pages Visited: 406301		
Technology	***Number of Pages***	***Percentage of total pages***
CGI	40751	10.03%
ASP	34218	8.42%
SHTML	10085	2.48%
ColdFusion	5430	1.34%
PHP3	3755	0.92%
JSP & Servlet	1986	0.49%
JHTML	1651	0.41%

Figure 6: Of the 406301 pages visited by the spider, 20% used measurable content delivery technologies. Fully 10% of the visited pages were generated by a server–side CGI program.

6.2 Cross–Site Scripting Attack

Most web browsers have the capability to interpret scripts embedded in a web page and are usually installed with this capability turned on by default. Some web pages, such as guest books, allow unknown users to transmit text for display to other users viewing the page. A web server that does not carefully verify these inputs may generate a page containing hostile embedded code [3]. We verified this with a simple test: we browsed a dozen public forums, message boards and guest books. At each site, we posted the following message:

```
Hello<script>alert('sesame.html')</script>!
```

Among the web sites we tested, most of them correctly handle the embedded tag <script> by either striking the character < or replacing it with other characters. However, two web sites accepted the message without any modifications. A message box saying `sesame.html` then popped up anytime someone loads the page. Should the `alert` function be replaced with a snippet of malicious code, other users' browsers would automatically execute the code and harm their own systems. Vulnerable sites include:

- `http://www.castles.org/guestbook/guestbook.cgi`
- `http://thewaytotruth.org/guest/guestbook.html`

6.3 Arbitrary Command Execution

Embedded scripting language in the input may coax the server–side program to execute arbitrary commands [7]. Allaire presents the following ColdFusion attack: If a hostile user changed a URL to look like:

```
http://myserver/page.cfm?x='|shell("cmd /c 1 > c:\temp\foo.txt")|'
```

and the template, `page.cfm`, contained a query like:

```
SELECT * FROM USERS WHERE lname = '#URL.x#'
```

The URL would cause Microsoft Access to create a file at `c:\temp\foo.txt` on the server.

6.4 Parallels

These attacks are equivalent to the applet attacks presented previously, exploiting the same naïve trust model. Just as in the Counter applet, failing to revalidate data checked by a client script allows incorrect data to be stored in a server database. Arbitrary command execution exploits this same trust, with even more striking results. The cross–site scripting attack exploits a trust model equivalent to that of the AnnoyingChat applet. While a user may trust a single web server to provide safe content, when she loads a page containing contributions from other unknown users she unconsciously includes these users in her trust boundary.

7 Future Work

We believe that interesting attacks can be launched by exploiting the same trust model in other newer technologies. Besides the examples explored earlier, we have identified cookies and hidden fields in forms as possible client data that holds server trust. A cookie is a text file containing client information generated by a web server and stored on a client machine. All future connections will transmit this stored information back to the server. Similarly, hidden fields on forms often store data required by the server but not meant for modification by the user. Modification of these containers exploits the applet/server trust model.

Although the interactive tool has been useful, several notable improvements would significantly boost its power. The current scripting capabilities are little more than reading data from a file and spewing the bytes to a socket. We envision a strong scripting language allowing users to specify expected incoming messages as well as data to send. Further, input messages would have the ability to store data to variables that may be included in output messages. With the addition of control flow, applets could be mocked with great accuracy and ease.

The text–based messaging diminishes the problem of understanding server data, but we envision attempting to analyze and insert data into applications that do not communicate with human–readable characters. For example, had we found an applet using remote method invocation, understanding data streams and generating meaningful data in response would have been impossible with this tool. To help see data in a more meaningful way, we would like to see the tool act as a proxy, connecting both to a remote server and to a local application such as a web browser. Data received from the server may be passed to the application for a meaningful representation, and the application can be used to generate outgoing data. Notably, a proxy tool also has uses in text–based messaging. Allowing an applet to generate data would decrease the time needed to initially understand the communication between the applet and server.

8 Related Work

Previous applet security works have addressed only client security. The "attack applets" they expose exploit a particular deficiency in either the SecurityManager or the virtual machine upon which the applet executes. The damage potential is against the client machine running the applet. Admirably, much has been done to correct the dozen or so flaws found to date and the assurances of Java applet security may be generally accepted [15].

The Condor high−throughput computing system [5] suffers from the same poor trust model as applet servers. Condor dispatches processes to remote idle workstations for execution. To work around the distributed ownership problem, remote processes send certain system calls back to the submitting host for execution. Condor trusts that these system calls are genuine. Remote processes have been hijacked, enabling malicious system calls to be executed on the submitting machine [23].

Recent work has examined the security implications of deploying mobile agents to unknown hosts [11,21]. Mobile agents offer an alternative to client−server programming, running code on a machine in the control of an unknown party. The agent may migrate to other unknown hosts, returning data to the originator at completion. This model of computation suffers heavily from attacks exploiting the trust between the source machine and the remote agent. For example, data and code stored in the agent can be read or mangled to return incorrect results to the source. As in our attacks, reverse engineering of the agent's code may lead to sophisticated attacks against the originator.

9 Conclusions

We believe a dangerous trust model exists in the use of Java applets that communicate with a server process. The applet and server are developed together, and preconceptions about the form and content of incoming data shape the verification process. In particular, a server may trust an applet to ensure only correct data is transmitted. The server extends the boundary of its trust to all running applets.

A malicious user, however, may pose as the applet and send invalid data to the server. Since trust has been extended to the remote host, these hostile messages will be accepted by the server.

To make such impersonation possible, we developed a tool that connects to a server process and sends data intended to exploit this trust relationship. Using our techniques, we have crashed one server, executed a unique denial–of–service attack using another, and revealed interesting implementation characteristics in two others. Server trust of client applets has been found to exist.

This model of trust on the Internet is not limited to Java applets. Many examples illustrate what seems to be a common misconception among Internet authors that a client process will execute as intended. Subversion or outright removal of these processes invalidates this model of execution.

All components of a distributed application such as an applet and server cannot be assumed to operate as written. Care must be taken to ensure trust does not extend beyond those machines and processes within one's own control. A preliminary step towards securing a server process is the explicit definition of a trust model and the development of safeguards to ensure this model is accurately reflected in code. Communication from unknown hosts must be assumed malicious until validation proves otherwise.

10 References

[1] *Annoying Chat*, 7 May 2001, <http://www.chat.axxs.net/fict–1.0.3/annoyingchat.html>.

[2] Fabrice Bellard, Gilles Muller, Barbara Moura, Charles Consel and Ulrik Pagh Schultz, *Welcome to the Harissa home–page,* 7 May 2001, <http://www.irisa.fr/compose/harissa/>.

[3] CERT Coordination Center, DoD–CERT, the DoD Joint Task Force for Computer Network Defense, the Federal Computer Incident Response Capability, and the National Infrastructure Protection Center, *CERT® Advisory CA–2000–02 Malicious HTML Tags Embedded in Client Web Requests*, 7 May 2001, <http://www.cert.org/advisories/CA–2000–02.html>.

[4] Citicorp, *Creating an Account*, 7 May 2001, <https://web.da–us.citibank.com/cgi–bin/citifi/scripts/ao_online/create_your_account.jsp>.

[5] Condor Team, University of Wisconsin––Madison, *Condor High Throughput Computing*, 15 May 2001, <http://www.cs.wisc.edu/condor/>.

[6] Esprit, *Esprit – Register Club Esprit*, 7 May 2001, <http://www.esprit.com/shop/clubesprit/register.ehtml>.

[7] Malcolm Gin, *Top Five ColdFusion Security Issues*, 7 May 2001, <http://www.allaire.com/handlers/index.cfm?ID=19491>.

[8] Google, *Google,* 7 May 2001, <http://www.google.com/>.

[9] Cameron Gregory, *Free Java Hit Web Counter Applet*, 7 May 2001, <http://www1.counter.bloke.com/>.

[10] Jochen Hoenicke, *Java Optimize and Decompile Environment (JODE),* 7 May 2001, <http://jode.sourceforge.net>.

[11] Fritz Hohl, "A Model of Attacks of Malicious Hosts Against Mobile Agents," *Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations*, INRIA, Frace, 1998, pp. 105−120.

[12] Infospace, Inc, *100hot Web Rankings*, 7 May 2001, <http://www.100hot.com/>.

[13] Pavel Kouznetsov, *Jad − the fast JAva Decompiler,* 7 May 2001, <http://www.geocities.com/SiliconValley/Bridge/8617/jad.html>.

[14] Miodrag Krunic, *OnLineCounter by Miodrag Krunic*, 7 May 2001, <http://www.oaza.co.yu/counters>.

[15] Gary McGraw and Ed Felton, *Java Security: Hostile Applets, Holes and Antidotes,* John Wily and Sons, Inc, 1996.

[16] B. P. Miller, L. Fredrikson, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM* 33(12), December 1990, pp. 32−44.

[17] Pimpernel Online Entertainment, *SpaceBattle,* 7 May 2001, <http://www.pimpernel.com/spacebattle/space.html>.

[18] Todd A. Proebsting and Scott A. Watterson, "Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?)," *Proceedings of the 3rd USENIX Conference on Object−Oriented Technologies and Systems*, June 1997, pp. 185−197.

[19] Sun Microsystems, *Security Restrictions,* 7 May 2001, <http://java.sun.com/docs/books/tutorial/applet/practical/security.html>.

[20] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, Scott A. Waterson, *Toba: Java For Applications: A Way Ahead of Time (WAT) Compiler,* 7 May 2001, <http://www.cs.arizona.edu/sumatra/toba/>.

[21] J. Vitek and G. Castagna, "Mobile Computations and Hostile Hosts*," Journ'ees Francophones des Langages Applicatifs,* February 1999, pp. 113−132.

[22] Hanpeter van Vliet, *Mocha, the Java Decompiler,* 7 May 2001, <http://www.brouhaha.com/~eric/computers/mocha.html>.

[23] Victor C. Zandy, Barton P. Miller, and Miron Livny, "Process Hijacking," *The Eighth IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, California, August 1999, pp. 177−184.