An Application Layer Covert Channel: Information Hiding With Chaffing

Nick Estrada moander@mit.edu
Nick Feamster feamster@mit.edu
Michael Freedman mfreed@mit.edu

December 9, 1999

1 Introduction

The purpose of our project was the implementation of an application-layer covert channel, with guaranteed confidentiality via chaffing. A covert channel is a means of passing information between two parties in such a manner that the existence of the communication channel itself is not obvious to the casual observer.

Additionally, the implementation of a covert channel is enhanced through encryptionless security – through clever uses of steganography in the covert channel itself, even if an observer is aware that a covert channel exists, he will not be able to recover the information which is being sent across the channel. The proposed covert channel achieves privacy through chaffing and winnowing, an alternative to encryption which instead pads useful ("wheat") bits with garbage ("chaff") bits, such that only the intended receiver will be able to determine which bits are wheat and which are chaff.

The goal of this project was to create a practical covert channel, such that even an active, capable administrator or observer would find it very difficult or be very unlikely to detect the channel. Even after detecting the channel and possibly recovering the data, the administrator should still be unable to understand the transmitted message.

A covert channel can be described as a type of communication channel operating such that it can be utilized to somehow violate system security policy. A covert channel is not part of the actual computer system design; rather, the channel serves as a means of transmitting a stream of bits from the sender to receiver in such a way that the very existence of a communication channel cannot be verified without full knowledge of the channel details. [7]

Covert channels have a number of applications. Information can be transmitted secretly in a number of ways. Common ideas include through process table information, over networks (local and wide), through files.

Covert channels are often implemented at the transport layer. In the case of TCP/IP, it is fairly trivial to develop applications such as those described below. [11]

- Bypassing packet filters, network sniffers, and "dirty word" search engines.
- Encapsulating encrypted or non-encrypted information within otherwise normal packets of information for secret transmission through networks that prohibit such activity ("TCP/IP Steganography").
- Concealing locations of transmitted data by "bouncing" forged packets with encapsulated information off innocuous internet sites.

Despite the fact that there are many possible ways to implement covert channels, the program described in this paper implements a covert channel at the application layer, one level above the TCP layer. This design choice was due to the fact that the Apache server code lent itself most easily to application-layer modifications.

This implementation manipulates the HTTP header information to initiate the covert channel and HTML tags to transmit bits across the channel itself. Data can be hidden just about anywhere; the most important point is that the actions or modifications which represent the transmission of bits should not seem unusual to the casual observer. Other means of implementing covert channels often involve ICMP packets, routing control information, and UDP datagrams. These topics will not be covered in this paper, although the methods contained herein can be easily adopted to exploit these areas.

2 System Description

The covert channel is designed to operate across a network using a TCP connection and the HTTP protocol; the design employs a client-server model to establish the covert channel across which bits are sent.

2.1 Channel Client

The client requests chaffed bits from the server in such a manner that this request looks like nothing more than a normal HTTP request. However, the server's knowledge of the previously agreed upon protocol allows it to realize that the client is requesting chaffed packets. Given the fact that HTTP GET requests exhibit a fair amount of variance from request to request, the client should conceivably be able to make slight modifications to its request such that the server would know that the request was "special", but that casual observation of the client's request would not raise suspicion.

Specifically, the proposed design calls for modifications to the `User-Agent` header field in the HTTP request header. By modifying the `User-Agent` slightly to a special user agent which appears to be a normal user agent, for example "Mozilla/4.61C-CC**L**-MCD" instead of "Mozilla/4.61C-CC**K**-MCD", the server can detect that the client is in fact not a Netscape client, but rather a hacked client which is to become the receiving end of the covert channel. The modification to the HTTP header is significant enough so that the server can detect the chaffing client, but subtle enough so that a system administrator or other curious party would not become suspicious.

2.2 Channel Server

The server for the covert channel is the sender of the hidden message, which is to be embedded within the HTML file itself in an unobtrusive fashion. Specifically, the proposed design hides bits in the most unobtrusive aspect of a HTML file: whitespace. Based on whether the server wishes to transmit a stream of ones or zeroes, it can find the next occurrence of a particular tag (i.e., `<P>` or `<HR>`, and pad the file with whitespaces between the tag ending character, ">", and the penultimate character.

Specifically, our channel implements spaces after "P" as representing a a string of ones, and spaces after an "R" in a tag as representing a string of zeros, although this could conceivably be expanded to more general cases, such as padding different sets of tags for each string of bits, or even perhaps runtime determination of which tags should be used based on the prominence of various tags in the HTML file itself.

2.3 Protocol Specifics

Given the basic interactions between the client and server previously described, it is important that the design itself specifies an agreement, or protocol, between the client and the server. This implementation assumes the existence of the following agreement:

1. The HTML files which the server sends to the client will *ordinarily* contain no spaces between the tag ending character and the penultimate character. As such, the client will be guaranteed that when spaces do exist in this location, these spaces represent the covert transmission of bits rather than simply an anomaly in the HTML file itself.

2. The client will indicate to the server that it wishes to receive bits of the covert message by changing the `User-Agent` field to "Mozilla/4.61C-CC*L*-MCD", a variant on an actual user agent, but specific to this chaffing client.

3. The bits will always be transmitted *in order* in the file itself. That is, a string of bits will be represented as whitespace padding *after* the previously padded tag in the HTML file, regardless of whether this message was 1's or 0's.

4. If the server is not able to return the entire covert message in a particular HTML file (an extremely likely occurrence), the server will send the following bits of the file upon the next request for client chaffing.

5. The server will designate whether a given packet is wheat (valid) or chaff (invalid) by digesting a buffer which contains data from the HTML file with either a valid or invalid HMAC, respectively. Only the receiving client

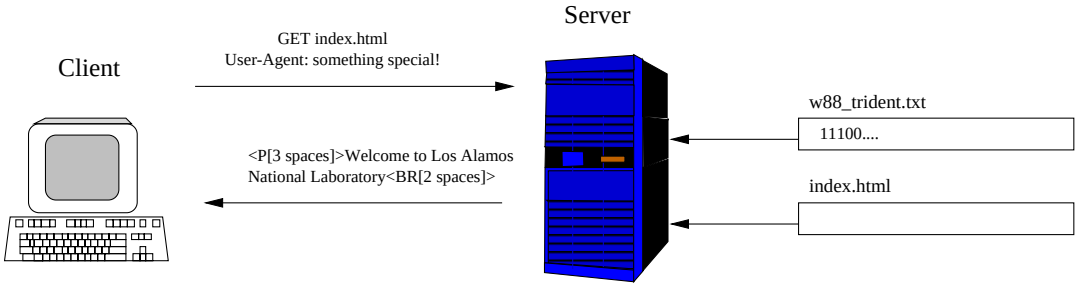

Figure 1: Client-Server Covert Channel Protocol

has the key which the server used to sign the file; thus, only the client will be able to distinguish wheat from chaff.

Given such an agreement, the server is now capable of sending covert messages to the client through HTML pages. The following steps employ the agreements described above to transmit bits of a message covertly:

1. Client sends server a forged HTTP request header, "Mozilla/4.61C-CC*L*-MCD", signifying a **message request**.
2. The server determines whether the HTML file to be sent should be filled with wheat bits or chaff bits. This is done using a threshold and a uniformly distributed random variable.
3. If the server is to send chaff, the predetermined tags representing strings of ones and zeros are padded with a random number of spaces. Additionally, whether to pad with ones or zeroes first is random.
4. If the server is to send wheat, the file which is to be chaffed is opened and is converted into a string of ones and zeroes. The HTML file is padded accordingly with whitespace, as described above, and returned to the client.

System design calls for the client to be able to request a number of different files to be sent through the covert channel, although the present implementation has not yet added this feature.

Figure 1 provides an example of the protocol described above. In this case, the client first sends a message to the server with a `GET index.html` request, including the all-important modification of the `User-Agent` field. Given this, the server knows that it is to send data covertly in the HTML page which it returns to this client. (In the case of multiple clients, it might also be necessary to have a client identifier tag. In this project, we deal solely with a single covert client.) In the figure, the server is serving `index.html` padded with wheat bits. As such, it looks at `w88_trident.txt`, and first sees a string of three 1's. Thus, it looks for the first occurrence of `P>` (or `p>`) which it has not already encountered, and inserts three whitespace characters in between these characters. The server next sees two 1's, and pads the next occurrence of `R>` with two spaces. Similar behavior ensues throughout communication of the hidden message, until the file is completely transmitted.

3 Design Choices

The most important criteria for a covert channel is that the channel not be detectable to a casual observer. The extent to which a channel is "covert" can vary: a channel might be undetectable by naive perusal, or might be more robust and extremely difficult to detect without prior knowledge of its existence.

3.1 Use of `httpd` Daemon

The communications between the sender and receiver generate network traffic. To lessen suspicion of these communications, the system utilizes a high-traffic network service, such that the existence of a connection between the sender and the receiver will not raise suspicion of the presence of foul play.

The `httpd` process serves web pages, typically HTML files which often contain tags which reference other objects, such as images and links to be served. HTTP is a high traffic protocol by nature; as such, connection requests

often repeat as users surf through web pages, and normal requests come from a wide variety of hosts. Also, if an `httpd` daemon is running on a system behind a firewall, the served pages could potentially be transmitted across the firewall.

Therefore, we have chosen to use HTTP as a means for creating a covert channel for the following reasons:

- **High Traffic:** The `httpd` daemon is designed to handle a large amount of traffic; the effectiveness of the proposed covert channel relies on the existence of high traffic because it assumes that the slight anomalies in the client request header and padded HTML files will go unnoticed by system administrators observing the operation of the server.
  - *Repeated Requests:* HTTP requests commonly occur in bursts. Often when a client is surfing a site, it will hit the same page repeatedly in a relatively short amount of time. High traffic (essentially, noise) masks the request patterns of the client; thus, it will be very difficult for an observer to detect unusual activity on the part of the client.
  - *Diversity of Hosts:* A typical web server receives hits from all around the world; because the model for the world wide web is inherently a public model, it could be considered a perfectly normal operation for a Russian citizen to be browsing a site in the United States. If such activity isn't inherently suspicious (and it shouldn't be), then the possibility definitely exists in this instance for the creation of a covert channel.
- **Firewall Portal:** While a user may not be able to access data behind a firewall, he may be able to reach the proxy, which serves as a portal in the firewall. Since these servers have access to information on both sides of the firewall, it is conceivable that installing a modified `httpd` daemon could create a potential path for information through the firewall.
- **Application Protocols:** HTTP is one of the few application level protocols which often has access to both sides of a firewall. Clever implementation of a covert channel can allow the client to simulate telnet (or other) access for the client in the covert channel through a clever bit-passing scheme.

3.2 Use of Chaffing and Winnowing

Techniques of chaffing and winnowing have been used to achieve confidentiality across the covert channel. System administrators or network sniffers are unlikely to detect a channel due to the high volume of network traffic involved with web servers. To ensure that a sniffer will not be able to recover the covertly transmitted bits, even with prior knowledge of their existence, we have incorporated plain view techniques of data hiding.

3.2.1 Basic Chaffing and Winnowing

Chaffing and winnowing are novel techniques to maintain privacy and nondisclosure of information without encryption. The technique was proposed by Ronald Rivest as an alternative to encryption that would not be limited by export regulations and fall into the same debate of key recovery for governmental and law enforcement agencies. [8]

A system based on winnowing begins by separating a message into discrete packets. The sender authenticates each packet with his secret key. A message authentication code (HMAC), based on a cryptographic hash function, is appended to each information packet. [5] Finally, the sender ensures that a receiver can recombine packet contents in a correct order by adding a sequence number to each packet. These valid packets are referred to as "wheat". This technique achieves confidentiality by adding invalid packets, or "chaff", marked with the same sequence numbers as wheat packets. These invalid packets include a message to be discarded and an invalid HMAC signature.

The sender and the receiver share a secret key, agreed upon through some previous key establishment protocol. Upon receiving the packets, the receiver checks each packet's HMAC and "winnows" the packets. Winnowing is the process of separating wheat (signal) from chaff (noise). For this protocol, winnowing refers to ignoring chaff and keeping only the wheat. The message is reconstructed by linking together the wheat packets according to their sequence numbers.

To explain how the system works, let us consider the set of packets shown in Figure 2. If we assume that the HMAC of the second packet for each sequence number is valid, the winnowed message reads: *Hi Roger, This final project was very fun. -Nick.*

(1,Professor Rivest-,532105)	(1,0,351216)
(1,Hi Roger,,465231)	(1,1,895634)
(2,The last problem set,782290)	(2,0,452412)
(2,This final project,793122)	(2,1,534981)
(3,was very time consuming.,891231)	(3,0,639723)
(3,was very fun.,344287)	(3,1,905344)
(4,-Mike,553419)	(4,0,321329)
(4,-Nick,312265)	(4,1,978823)

Figure 2: Chaffing and Winnowing

Chaffing and winnowing provide confidentiality of information, even though the information itself remains in plain view. Cryptanalysts may reconstruct messages by exhaustively searching the combinations of message fragments. Given 20 sequences of 4 packets per sequence (1 wheat, 3 chaff), there exist 4^{20} possible combinations of the packets, which is on the order of 10^{12}. However, if each packet contains several words, the difficulty in cracking the transmitted message greatly decreases. Therefore, it is desirable to increase the granularity of the message to prevent an eavesdropper from figuring out a message from context.

The confidentiality of information afforded by this technique is based on the difficulty of determining wheat packets from chaff packets. As shown in Figure 2, single bits of information can be hidden in each packet, as opposed to words or sentence fragments, to increase the difficulty of plain text attack. Therefore, less information per packet improves the effectiveness of chaffing, yet decreases the throughput of the channel.

3.2.2 More Efficient Chaffing: All-or-Nothing

One extension to chaffing is an application known as the "all-or-nothing" transform, also known as a "package transform". [10] Such a scheme would make it impossible for Bob to see the entire message which Alice is sending until he receives all of the smaller components from Alice. Until Bob has received enough of these packets, the wheat packets which he receives from Alice look like random noise. Once Bob has received all of the packets from Alice, however, he is able to recover the message.

3.2.3 Chaffing over Networks

Chaffing can theoretically be implemented on any layer of the protocol stack, although sending HMACs across the physical or link layer could potentially be very tedious. The most feasible alternatives are sending chaff through the network, transport, or application layer.

End-to-End Argument One particularly important disadvantage of chaffing at the network layer is that this layer does not guarantee a reliable transmission of packets. Thus, the possibility exists for an important wheat packet to be dropped at the network layer; this is an undesirable characteristic. The transport layer guarantees transmission of all packets, but does not address out-of-order packet reception. Since packets come out of order, sequence numbers are required for each segment of data, such that the valid data can be assembled in order. Fortunately, this reordering is taken care of at the application layer. As such, it could be considered most practical in many cases to implement chaffing at the application layer, because of the end-to-end argument.

4 Implementation

The following sections describe the implementation of the covert channel application. Specifically, the modified function calls within Apache are discussed, as well as the implementation of independent functions which enable chaffing and the establishment of the covert channel itself.

4.1 Server Implementation

The covert channel described in Section 2 was implemented through adaptation of open source software. Specifically, the server for the covert channel is a derivative of Apache 1.3.9, which was hacked according to the previously described specifications. [2] The implementation of the server does not inherently require Apache source code. A trivial server could have been, and in fact was, developed to return HTML files with padded bits.

Apache was chosen as the base for the covert channel server so as to provide the general appearance of a fully functional web server. By tailoring the Apache source code as needed to perform covert channel actions, it was possible to create a server which functioned exactly like the standard distribution of Apache, except under special circumstances (i.e., specific `User-Agent` field values).

4.1.1 Apache Modifications

Most of the modifications to the Apache source code itself occurred in one particular file, `http_protocol.c`, where functions which send the buffered HTML file back to the client. Specifically, our server used the provided function `ap_table_get()` to get the string associated with the user agent of the client. If the string of the user agent matched that of the "special" covert client user agent, the server invoked a modified version of `ap_send_mmap()`, which made the function calls to the underlying functions responsible for actually modifying the buffer in the specified manner of adding whitespace and appending an HMAC to the header.

Specifically, the function `hide_covert_message()` is called on the buffer `mm`, which contains the data corresponding to the requested HTML file. `hide_covert_message()` and its underlying support functions were implemented entirely independently of Apache, and are described in Section 4.1.2.

4.1.2 Independent Support Functions

The `hide_covert_message()` is the high-level function which performs both the insertion of whitespace and the addition of either a valid or invalid HMAC. Essentially, this function performs the following operations:

1. Copies the buffer containing the HTML buffer to a new scratch buffer.
2. Determines whether a packet is to be wheat or chaff by calling `is_wheat_packet()` (based on `srand()`).
3. Adds an appropriate HMAC to the header, depending on whether the file will contain wheat or chaff.
4. Calls either `hide_chaff_data()` or `hide_file_data()`, depending on the whether the file will contain wheat or chaff. These functions call lower level functions which actually add the appropriate number of spaces to the HTML file in the correct locations.
5. Returns the modified (note, also larger) buffer back to the Apache code and reassigns the modified buffer to the address of the old buffer.

This sequence of events essentially describes how the operation of adding whitespace to a particular HTML file is performed. The bulk of this functionality is contained in `htmlconv.c`.

4.2 Client Implementation

The base for the client was a much simpler block of code because it did not have to appear and behave like a legitimate mainstream client like Netscape (although such a feature could be a future development; see Section 5). Other than basic network functionality provided by the socket library, the only other operations which the client was required to support was the decoding of the chaffed HTML data.

That is, given an HTML file with whitespace-padded tags and an HMAC in the header, the client must determine:

1. if the hidden data which the server returned is valid, and
2. the characters to which the hidden data correspond.

The network functionality is based largely on a TCP-based client example by David Mazieres, and includes additional functionality to perform whitespace parsing and HMAC comparison. [6] Whitespace parsing is performed in `htmlrev.c`. Additionally, the implemented design provides functionality for conversion to and from ASCII files.

Table 1: Source Code Location

Directory	Description
/mit/mfreed/S6.857/code	code for performing basic MD5 functionality [9]
/mit/mfreed/S6.857/apache-1.3.9/src	root Apache directory
/mit/mfreed/S6.857/apache-1.3.9/src/main	location of modified files
/mit/feamster/S6.857/project/code	root of independent source code
/mit/feamster/S6.857/project/code/io	HTML padding, unpadding, ascii to bit functions, etc. (the meat)
/mit/feamster/S6.857/project/code/chaff	location of the covert channel client source
/mit/feamster/S6.857/project/code/network/client	support code for the covert client
/mit/feamster/S6.857/project/code/network/server	simple TCP server
/mit/feamster/S6.857/project/code/examples	some code from which our code is derived

4.3 `gzip` Compression and `uuencode`

Because the bandwidth of the existing covert covert channel is fairly low as it stands, it is a good idea to first compress the file which consists of the hidden message. Once the client reconstructs the message which the server sends, it must the reconstruct the original message and subsequently perform decompression of the compressed data, thus allowing for recovery of the initial hidden message. By compressing the file before sending, the channel's bandwidth has effectively increased. Of course, the message's resilience to error is decreased as a result of the compression, but this effect is of less concern since the current design does not have an error correction functionality.

4.4 Source Code Availability

Source code for the modified Apache code is located at `/mit/mfreed/S6.857/apache-1.3.9/src`. Source code for the functionality independent of Apache is located at `/mit/feamster/S6.857/project/code`. HMAC-related source is located at `/mit/mfreed/S6.857/code`. Table 4.4 describes the contents of each directory and the functionality of the files contained within.

Table 4.4 shows the location of all of the source code used to develop the covert channel client/server system. Some of the network code was inspired by a number of sources, particularly `synk4.c` and a paper by David Mazieres. [6]

5 Future Development

Our client-server model of an application-layer covert channel is a proof of concept implementation. The modified server sends information embedded in web pages to a client which requests the chaffed data, such that only a client who knows the HMAC key will be able to perform winnowing. The following areas remain for further development.

5.1 Randomness

The system relies on randomness for several aspects of its functionality. The decision to send a wheat or chaff packet the number of number of whitespaces in each tag in chaff packets, and whether to chaff with one or zero first, are all determined at random.

The actual randomness of these processes depends on the randomness of the UNIX `srand()` function. The `rand` function (seeded by `srand`) is only pseudo-random. Since system time is often used as a seed value, and served web pages are already timestamped, an attempt to recreate the sequence of random numbers produced and thus sequence of wheat versus chaff packets can be made and will likely be successful.

Randomness can be improved by adding functionality that is more unrelated to the packet and its timestamps. System performance, hardware thermal noise, and user input such as keystroke speed in a manner similar to PGP key generation provide a greater degree of randomness. [4]

5.2 Covering Tracks

The Apache web server keeps a log of pages sent, stored in `access_log`, the specified access log file. Normal logs reflect the extra whitespace padding of sent wheat and chaff pages. For example, given a default Apache *index.html* file size of 1622, the following logged entries might appear very suspicious to network administrators. Notice the last field for each entry of the log file, the size of the buffer which was sent.

```
127.0.0.1 - - [07/Dec/1999:03:38:40 -0500] "GET /index.html HTTP/1.0" 200 1644
127.0.0.1 - - [07/Dec/1999:03:45:27 -0500] "GET /index.html HTTP/1.0" 200 1652
127.0.0.1 - - [07/Dec/1999:05:34:53 -0500] "GET /index.html HTTP/1.0" 200 1647
127.0.0.1 - - [07/Dec/1999:07:28:07 -0500] "GET /index.html HTTP/1.0" 200 1643
127.0.0.1 - - [07/Dec/1999:08:50:40 -0500] "GET /index.html HTTP/1.0" 200 1657
```

In order to cover the covert channel's tracks, the size written to logs should correspond to the actual HTML file on the server, as opposed to the length of the padded buffer transmitted. The vast majority of HTTP requests come from normal clients receiving normal HTML files. Such transmissions obviously require no modifications of access logs.

5.3 Runtime Options: Secrecy versus Bandwidth

Covert channels are generally low-bandwidth means of transmitting information. The greater the signal-to-noise ratio of covert data, the more difficult it becomes to hide the information. Chaffing, on the other hand, relies on a high-bandwidth channel where most information (chaff) can be winnowed from the valid wheat. Secrecy of the data in transmission can be increased by sending out more chaff, and the secrecy of the channel itself can be increased by sending less data through it. However, both of these are schemes lower bandwidth, which is undesirable. There is an inherent tradeoff between secrecy and bandwidth, where the specific amount of secrecy or bandwidth may vary depending on the file being sent, as well as its source and destination hosts.

The choice of entire wheat or chaff pages on the application layer (i.e., signing the entire web page in the HTTP transport headers), stems from this problem. HMAC digests are 16 bytes long. In order to encode HMACs in whitespace, the file would look very suspicious: 128 bits of HMAC whitespace compared to several bits of message whitespace. Additionally, encoding the HMAC itself results in greatly reduced bandwidth, because most of the bandwidth of the covert channel is consumed by the HMAC.

A desirable addition to the server would be the ability to adapt to varying secrecy requirements. This implies runtime determination of a specific client's requirements from either an explicit client list or from various levels of secrecy which could be specified in the client request header. For clients requiring more covert transmission of information, a lower-bandwidth covert channel should be employed.

The use of explicit `P>` and `R>` tags should be modifiable at runtime to utilize the distribution of tags on the served web page. Because a bitstream is essentially strings of 1's and strings of 0's, then, knowledge of whether the last stream of bits was ones or zeros indicates the content of the current stream. As a result, a client could ask the server to pad every tag in a HTML page; this requires an agreement on what the value of the first transmitted bit in each HTML file will be.

5.4 Receiver-Unaware Covert Channels

The network client used to formulate forged HTTP requests has a simple command line interface. Its purpose is to act as a recipient on a covert channel, as opposed to serving as a fully functional web browser (although it is a functional HTML client).

A mainstream client such as Netscape or Lynx could be altered to create a receiver-unaware covert channel. The modified browser could conceivably transmit requests similar to the currently-implemented client. The server would then reply with HTML pages with hidden chaff or wheat bits. While browsers ignore whitespace when displaying interpreted HTML pages, users may still view the document source. Since additional whitespace appended to HTML tags is generally rather suspect, the modified client should be able to remove these spaces before displaying the document source to the user or writing the file into the cache.

A mainstream client could then be used to establish a covert channel unbeknownst to the user of the browser. The trusted distribution of a hacked client to users is a separate problem, but this condition may be exploited so

that requests originate from the unaware receiver's IP address, increasing the variety of the hosts (possibly with his permissions or certificates). Therefore, security analysts would have a greater difficulty in determining from where covert requests originate.

6 Discussion and Application

The following sections describe various applications of the covert channel described in this paper. Because our goal was primarily to construct a simple covert channel, the opportunity exists for further elaborations. For instance, the creation of a covert telnet client, the handling of multiple clients, and the use of more elaborate steganography, such as hiding information in audio and video files, could be performed.

6.1 Covert `telnet` Access

Currently, the client requests are generic requests for a pre-specified file, which the server returns through the covert channel. By easily modifying client requests to include extra information, the client could conceivably issue system commands such as `ls`, `rm`, `cat`, and `grep` to the server. Upon receiving such a request, the server performs these actions and returns the result of such a system call through the covert channel.

A web server also provides a portal through a network firewall, allowing one to simulate command line access even when a firewall is in place; thus, even if telnet or ftp access is not permitted through a firewall, an HTTP-based covert channel could bypass other restrictions.

The `httpd` web daemon is often run with `root` access on the server host. Given such access and the ability to install the modified daemon in its place, the server can perform any system commands on the root level. Even without root access to files, web servers may perform both malicious active and information-compromising passive activities.

6.2 Classical Steganography

Steganography is the art of hiding information in a way such that an adversary cannot detect the presence or contents of the hidden information. The proof of concept implementation only establishes a covert channel by hiding information directly in the HTML pages served. However, the presence of such a channel through a web server is extendable to many other methods of information hiding. Examples of steganographic techniques are the digital watermarking of images in low-order bits or other means, or fingerprinting files with serial-numbers and extra information. The security of steganography lies in the realm of information hiding, steganography communication intelligence revolves around the interception and direction-finding of such information. [12]

The implemented client-server model suggests alternative methods to chaffing. Files included in web pages – images, sound files, applets, etc. – contain numerous avenues for the covert transmission of information. [3] The server may incorporate these classical steganographic techniques to hide more information, reaping the benefits of information transmitted in various encoded formats, thus making hidden information harder to detect than whitespace padding or similar ASCII plain-text modifications.

7 Conclusion

In a matter of weeks, an application-layer covert channel was implemented in C based on a traditional client/server model using Apache `httpd` as the base distribution for the modified web server. Although many improvements remain with respect to the implementation, the program written successfully demonstrates a *concept*: covert channels can be successfully implemented in the context of HTTP by taking advantage of the idiosyncrasies of the HTTP client request header and by exploiting seemingly insignificant whitespace in certain positions in the retrieved file.

In particular, some of the most difficult code to develop was not that which was provided the Apache server functionality, but rather the code which manipulated the buffers which passed the HTML data about in `char*` buffers. Inexplicable errors emerged during one heap allocation: the `insert_data()` function would operate properly within the debugger and while running test suite code, but when the same function was called with the same arguments from Apache, the function ceased to function properly. When a particular buffer in the function was

changed from heap allocation to stack allocation, the problem disappeared. Needless to say, this was a hack, but a quick fix.

The use of chaffing provides a means to achieve confidentiality without encryption. Governmental export laws might not be applicable. If the technique gains widespread use, the government's position might notably change. But more importantly, chaffing is a demonstration that legal restrictions of secrecy may always be subject to loopholes.

A key lesson to take away from this paper is that covert channels can exist almost anywhere – the requirement for a covert channel is not that the data itself be hidden, but rather that the existence of the channel is not be evident to an external observer.

A HTTP Sample Client Request Queries

Valid HTTP Query by Netscape 4.61

```
Received 288 bytes, data [GET / HTTP/1.0c
Connection: Keep-Alive
User-Agent: Mozilla/4.61C-CCK-MCD  [en] (X11; U; SunOS 5.6 sun4u)
Host: localhost:6001
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

HTTP Query by Client Simulating Netscape 4.61 to initiate covert channel

```
Received 288 bytes, data [GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.61C-CCL-MCD  [en] (X11; U; SunOS 5.6 sun4u)
Host: localhost:6001
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

Valid HTTP Query by Netscape 3.01

```
Received 182 bytes, data [GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.01 (X11; U; SunOS 5.6 sun4u)
Host: localhost:6001
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Valid HTTP Query by MS Internet Explorer 5.0

```
Received 341 bytes, data [GET / HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.ms-excel,
application/msword, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)
Host: w20-575-109.mit.edu:6001
Connection: Keep-Alive
```

Valid HTTP Query by Lynx

```
Received 591 bytes, data [GET / HTTP/1.0
Host: localhost:6001
Accept: x-world/x-vrml, text/ksign, application/pdf, application/x-dvi, applicaion/postscript,
audio/*, audio/x-pn-realaudio, video/*, video/mpeg, text/html, ext/plain, text/sgml, text/
x-sgml, application/x-x509-ca-cert, application/x-x59-user-cert, application/x-wais-source,
application/html, video/mpeg, image/jpe, image/x-tiff, image/x-rgb, image/x-png,
image/x-xbitmap, image/x-xbm, image/gf, application/postscript, */* ;q=0.001
Accept-Encoding: gzip, compress
Accept-Language: en
Negotiate: trans
User-Agent: Lynx/2.7.1 libwww-FM/2.14
```

B Sample Execution of Covert Channel

B.1 Data Received by Client

Below is an excerpt from the HTML file which the client receives from the covert channel server:

```
  ...

  <P>
  If you can see this page, then the people who own this domain have just
  installed the <A HREF="http://www.apache.org/">Apache Web server</A>
  software successfully.  They now have to add content to this directory
  and replace this placeholder page, or else point the server at their real
  content.
  </P>
  <HR >
  <BLOCKQUOTE>
   If you are seeing this page instead of the site you expected, please
   <STRONG>contact the administrator of the site involved.</STRONG>
   (Try sending mail to <SAMP  >&lt;Webmaster@<EM>domain</EM>&gt;</SAMP>.)
   Although this site is
   running the Apache software it almost certainly has no other connection
   to the Apache Group, so please do not send mail about this site or its
   contents to the Apache authors.  If you do, your message will be
   <STRONG><BIG>ignored</BIG></STRONG>.
  </BLOCKQUOTE>
  <HR   >
  <P >
  The Apache
  <A
   HREF="manual/index.html"
  >documentation</A>
  has been included with this distribution.
  </P>
  <P>
  The Webmaster of this site is free to use the image below on
  an Apache-powered Web server.  Thanks for using Apache!
  </P>

  ...

----- Ints -----
Filename: covert_data.txt, size [7]
0110001
----- Done -----
Received 1906 bytes total
```

B.2 Apache Error Log

Below is a section of the Apache `error_log` file corresponding to the transmission of the wheat data in the previous section. We have added additional information for debugging purposes.

```
HMAC Digest; [23R2)=]
bits sent: 7, offset: 0
Buffer:<! 23R2)=>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
 <HEAD>
  <TITLE>Test Page for Apache Installation on Web Site</TITLE>
 </HEAD>
<!-- Background white, links blue (unvisited), navy (visited), red (active) -->
 <BODY
  BGCOLOR="#FFFFFF"
  TEXT="#000000"
  LINK="#0000FF"
  VLINK="#000080"
  ALINK="#FF0000"
 >
  <H1 ALIGN="CENTER">
   It Worked!  The Apache Web Server is Installed on this Web Site!
  </H1>
  <P>
  If you can see this page, then the people who own this domain have just
  installed the <A HREF="http://www.apache.org/">Apache Web server</A>
  software successfully.  They now have to add content to this directory
  and replace this placeholder page, or else point the server at their real
  content.
  </P>
  <HR >
  <BLOCKQUOTE>
   If you are seeing this page instead of the site you expected, please
   <STRONG>contact the administrator of the site involved.</STRONG>
   (Try sending mail to <SAMP  >&lt;Webmaster@<EM>domain</EM>&gt;</SAMP>.)
   Although this site is
   running the Apache software it almost certainly has no other connection
   to the Apache Group, so please do not send mail about this site or its
   contents to the Apache authors.  If you do, your message will be
   <STRONG><BIG>ignored</BIG></STRONG>.
  </BLOCKQUOTE>
  <HR   >
  <P >
  The Apache
  <A
   HREF="manual/index.html"
  >documentation</A>
  has been included with this distribution.
  </P>
  <P>
  The Webmaster of this site is free to use the image below on
  an Apache-powered Web server.  Thanks for using Apache!
  </P>
  <DIV ALIGN="CENTER">
```

```
   <IMG SRC="apache_pb.gif" ALT="">
  </DIV>
 </BODY>
</HTML>
bappl1b~h
Buffer Size: 1660
File to Hide: /usr/local/apache/htdocs/chaff.txt
Offset (Total Bits): 7
Bits Sent: 7
Wheat: 1
[Wed Dec  8 05:28:57 1999] [error] [client 127.0.0.1] Info: After hiding: Content-Length [1622]
```

References

[1] Ross Anderson's Home Page
http://www.cl.cam.ac.uk/users/rja14/

[2] Apache web server open-source distribution.
http://www.apache.org/dist/apache.1.3.9.tar.gz

[3] W. Bender, D. Gruhl, N. Mormoto, and A. Lu. "Techniques for Data Hiding."
http://www.research.ibm.com/journal/sj/mit/sectiona/bender.html

[4] D. Eastlake, S. Crocker, and J. Schiller. "Randomness Recommendations for Security." RFC 1750.
http://www.ietf.org/rfc/rfc1750.txt

[5] H. Krawczyk, M. Bellare, and R. Canetti. "HMAC: Keyed-Hashing for Message Authentication." RFC 2104.
http://www.faqs.org/rfcs/rfc2104.html

[6] Mazieres, David. *Using TCP Through Sockets.*
ftp://cag.lcs.mit.edu/pub/dm/source/net/net.ps.gz

[7] National Computer Security Center. *A Guide to Understanding Covert Channel Analysis of Trusted Systems.* November 1993.
http://www.fas.org/irp/nsa/rainbow/tg030.htm

[8] Rivest, Ronald. "Chaffing and Winnowing: Confidentiality without Encryption."
http://theory.lcs.mit.edu/~rivest/chaffing.txt

[9] Rivest, Ronald. "The MD5 Message-Digest Algorithm." RFC 1321.
http://www.faqs.org/rfcs/rfc1321.html

[10] Rivest, Ronald. "All-or-Nothing Encryption and the Package Transform." Proceedings of the 1997 Fast Software Encryption Conference. (Springer, 1997).

[11] Rowland, Craig. "Covert Channels in the TCP/IP Protocol Suite."
http://www.firstmonday.dk/issues/issue2_5/rowland/

[12] Steganography and Digital Watermarking.
http://www.jjtc.com/Steganography/