Covert Channels in SSL Session Negotiation Headers

Justin Merrill
Rochester Institute of Technology
justinmerrill978@gmail.com
Daryl Johnson
Rochester Institute of Technology
daryl.johnson@rit.edu

***Abstract*—The Handshake headers of the SSL/TLS protocol contain several multi-byte random data fields used in the generation of the encryption keys used during the session. This random data can be replaced with covert messages that can be intercepted on the wire using packet capture techniques. By encoding data into these fields, a modified SSL client can send messages to a legitimate destination, with legitimate application payload data and still leak covert messages to a receiver listening on the wire.**

Keywords: SSL, TLS, Random Field, Covert Channel

I. INTRODUCTION

The Secure Socket Layer (SSL), and its successor protocol known as Transport Layer Security (TLS), are an important and ubiquitous part of the current Internet landscape. It provides the underlying security that makes services such as ecommerce and secure online correspondence possible. The TLS protocol does this by acting as an intermediate layer between the transport protocol (TCP) and whatever application protocol is being used [1]. When the client connects to the server, a secure session is negotiated between them. After this takes place all application traffic is encrypted before transmission, preventing eavesdropping.

The TLS protocol performs two primary functions [1]. The first is ensuring the application data is protected from eavesdropping and manipulation while in transit. This is done using a combination of a cipher algorithm and hashing algorithm negotiated by the client and server during the initial session setup. The second primary function is allowing the client to authenticate the server's identity before connecting with it. This is done via X.509 certificates associated with each unique server [6]. Since these certificates must be signed and checked against a limited number of well known and auditied Root Certificate Authorities who verify the request is legitimate before issuing a certificate. This makes it much harder to maliciously impersonate a well known secure service.

While TLS vastly improves security, it can also make protecting a network more difficult. Since TLS sessions encrypt all of the application data traveling over them, they make it difficult to inspect packets for hidden or malicious payloads. Wrapping covert or malicious traffic will prevent the transmitted data itself from being monitored but still leaves a few key vulnerabilities open to detection. The destination IP address and TCP port are still visible. Such attempts can be monitored and thwarted by targeting suspicious destinations and ports. So, while encrypting covert traffic can help hide it from detection, it is itself not a covert channel. A covert channel is, as Butler Lampson states in one of the earliest papers on the subject, ”not intended for information transfer at all” [3].

TLS has several potential covert channel opportunities during the initial connection process. During this time, a subset of the TLS protocol known as the TLS Handshake Protocol is used to negotiate the parameters for the session [1]. This paper will specifically focus on several fields of fixed length random bytes used in the process of generating the encryption keys used during the session.

II. AVAILABLE CHANNELS IN HANDSHAKE

When an SSL client first connects to a server, it goes through a process of negotiating the encryption suite to be used and verification of certificates. The process goes through the following steps [5]:

ClientHello
: Client opens a connection and informs the server of TLS functionality and cipher suites supported. Included is a random field that will be used to formulate the final key and prevent replaying of data.

ServerHello
: Server chooses the cipher it will use for the session, passes along its own info to the client. Server includes its own random value.

ServerCertificate
: Server passes along its public certificate so that the client may verify it.

ServerHelloDone
: Server announces to the client that it is done sending initial negotiation messages.

ClientKeyExchange
: Client exchanges pre-master secret, a value both the client and the server will use to generate the final symetric encryption keys.

ClientCipherSpec
: Client switches into secure communications mode.

Finished
: Client announces end of session negotiation.

ChangeCipherSpec
Server switches into secure mode.
Finished
Server also announces end of session negotiations. The pair is now ready to exchange application data.

Within this negotiation process there are three fields of random bytes that are used. Random fields make an excelent candidate for a covert channel. Since the data contained in the field is by definition supposed to be random, any value can be inserted into the field without interfering with the normal operation of the protocol. If steps are taken to ensure the covert messages are encoded or encrypted before being sent it will be very difficult to differentiate covert messages from random data.

During the ClientHello and ServerHello phases of the session setup, both sides generate a 28 byte random value and send it to the opposing side [1]. This value is combined with a 32-bit datetime stamp and sent over the wire in unencrypted plaintext. This technique has already been partially explored as a means to leak and intercept the encryption keys and perform an attack on the encrypted session [2]. Its usefulness is not limited to leaking keys, as a full covert message payload can be transmitted using the same channel.

One of the things that makes this channel particularly useful is that the data is sent in plaintext between the two endpoints. This means the party wishing to receive the covert message does not need to operate or hijack the sever endpoint in order to receive the message. As long as they are able to capture the TLS negotiation packets somewhere along the path between client and server, the message can be recovered. Another benefit is that, if both the client and server endpoints are compromised by the covert message sending party, a bi-directional data stream can be used as both client and server hello messages contain random fields.

Use of this covert channel presents a slightly more difficult vector to the receiver. The client encrypts the premaster secret before putting it on the wire. This requires the receiver to either control the sever being used as an endpoint or have copied the servers private key needed to decrypt the message. However, the encrypted premaster secret can also work in the covert message sender's favor, as it makes it very difficult to examine the contents and discover the covert message.

III. PROCEDURE

A. Design

After deciding on the random fields as potential covert channels, several obstacles needed to be overcome to make the channels usable. The first was to find a way of generating SSL packets with their ClientHello random value field filled with our desired covert message. This task is more difficult than it would first seem. The packet can not be modified outside of the original sending applications SSL library. If one was to modify the random value in transit, the client and server would generate different master keys and the session setup would fail. Therefore it was necessary to modify the SSL library itself to inject the data into the ClientHello message.

Modifying the SSL code is complicated by the fact that almost all applications using SSL do not implement the protocol themselves but rely on a shared library. The most common of these are OpenSSL with open source and Unix software and Schannel in the Microsoft DotNet framework. These were quickly ruled out for the first round of testing as they are shared across the system and would be difficult to modify. Microsofts Schannel is not possible to use as it is closed source and can not be modified.

What was needed was a self-contained SSL implementation with source code available that could be easily implemented. Enter the Legion of the Bouncy Castle. Bouncy Castle is an implementation of several cryptographic functions, including SSL/TLS support, in operating system independent libraries. It was created to produce crypto libraries, in C and Java, that would be free of backdoors and government meddling [4]. The rest of the projects code was planned to be written in C (because of its ease of use and rich framework) and the Bouncy Castle API was well documented so it was chosen as the SSL library for the tests.

B. Testing

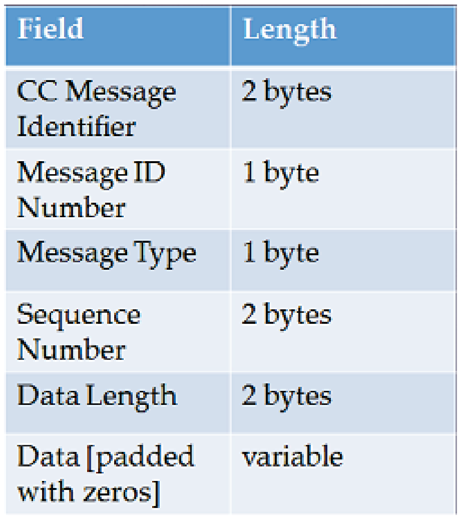
The first test was to see if both random fields could be exploited to send data without affecting the ability to create a valid SSL session. After reading over the code, the spots where the random fields were formulated were located. They were each one simple line of code which invoked a random byte generating function to get the required number of bytes and insert them into the packet structure. A static variable containing a preformatted message was added to each class and the random generator functions were replaced with a simple array copy that inserted the pre-created message into the field. Some additional code was written to open up an SSL session to a webserver, running on the local LAN, on HTTPS port 443.

With the test code written, the Wireshark packet capture tool was started and a display filter was added to show only SSL traffic. When the test code was executed, it successfully opened a connection to the server, negotiated a complete SSL session, than disconnected. When the packet capture was examined, both fields had successfully carried their messages without disrupting the proper operation of the protocol.

For the second test, the process was automated more on the sending side to allow a larger user defined message to be sent. The sending program was modified to take a target webserver hostname on the command line and ask the user to enter a message. Due to the way the Bouncy Castle library was implemented and the impossibility of getting the private key for a major website, the second test focused only on the ClientHello random field channel. To ensure proper message transmission and make the channel more useful, the message data is encoded into a simple packet format before being inserted into the header field. A library was written to do the process of encoding the data and returning an array of packets. The program then quickly opens and tares down sessions until all packets have been sent. The packets were retrieved the same way, using Wireshark packet captures. This test was also successful and the message was decoded properly by hand on the receiving end.

C. Implementation

Although it needed to be simple, a few fields were needed to comprehensively create the covert message transmission format. The specific byte layout is shown below in Figure 1. The first field is the message identifier field, to allow this to be identified as a covert packet. The value chosen for the second test was cc. That is followed by a message ID number, to help the receiver distinguish between multiple messages from the same client. Since this test code only sends one message at once, it is always set to zero. Next is the packet type field. It is either set to one, for a normal data packet, or two for the final packet carrying an MD5 hash of the complete message. This is followed by a sequence number, incremented once for each packet in the message and a length field for how many of the remaining bytes are data and how many are zero padding.



Field	Length
CC Message Identifier	2 bytes
Message ID Number	1 byte
Message Type	1 byte
Sequence Number	2 bytes
Data Length	2 bytes
Data [padded with zeros]	variable

Fig. 1: Covert packet format

IV. BANDWIDTH

The throughput of this covert channel is highly variable. After leaving space for the overhead bytes, the Hello message channel can carry 20 bytes a session and the premaster secret can carry 38 bytes a session. The number of sessions that can be opened is limited more by the client then any physical limitation of the system. It would not be unusual to have many sessions opening and closing at once as modern websites often load a lot of dynamic content and update items on the page automatically every few seconds. With that in mind, opening and closing connections as fast as possible for extended periods would look highly suspicious. It is a balance between thoughput and stealth.

V. FUTURE WORK

The ground work that has been laid in this paper proves these two channels are usable as a method of covert communications. There is still a lot of work that would need to be done to turn the concept into a usable implementation however. The biggest weakness currently is the necessity of using Wireshark and manual packet reassembly to extract the message at the other end. A program would need to be written to log all SSL Handshake sessions which are visible to the packet sniffer then examine them for covert messages. Once the messages are extracted, it is a simple process to reassemble and they can be verified as complete, due to the included MD5 hash.

Another problem is creating a stealthy transmission client. The current one is simple because it was built for experimental use. If this channel was actually to be used in the field, the covert channel should be worked into a system library like OpenSSL or a legitimate application like Firefox. If the pre-master secret channel is to be used, a decoy server the message sending party controls needs to be set up or the private keys of a legitimate site would need to be exfiltrated. Without that private key, the channel is not visible to the observing packet capture program.

VI. CONCLUSION

TLS, the current successor protocol to SSL, is a ubiquitous part of the modern Internet landscape. While using SSL to hide covert messages in the Application data is fairly obvious, there are a few novel ways of sending covert messages using the pre-session negotiation handshake packets. Specifically, three random data fields that can be used to carry covert messages. Using a specially written program, otherwise normal looking TLS packets can be copied and the hidden messages decoded. This could present a useful vector for exfiltrating data from a network without arousing suspicion.

REFERENCES

[1] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[2] Eu-Jin Goh, Dan Boneh, Benny Pinkas, and Philippe Golle. The design and implementation of protocol-based hidden key recovery. In Colin Boyd and Wenbo Mao, editors, *Information Security*, volume 2851 of *Lecture Notes in Computer Science*, pages 165–179. Springer Berlin Heidelberg, 2003.

[3] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973.

[4] Legion of the Bouncy Castle Inc. Bouncy castle. https://www.bouncycastle.org/", 2013.

[5] Robertckl. "how does ssl work? what is an ssl handshake?". http://www.symantec.com/connect/blogs/how-does-ssl-work-what-ssl-handshake, 2014.

[6] Carlos Scott. Network covert channels: Review of current state and analysis of viability of the use of x.509 certficiates for covert communications. Technical Report 1, University of London, Egham, Surrey TW20 0EX, England, 1 2008.