Improving OpenSSL to Process Out of Order Data

By Sam Gensburg
May 2, 2011

Yale University
Computer Science Senior Project
Advised by Brian Ford

Abstract

The SSL protocol and its successor, TLS, represent the foundation of modern internet security, and are indispensable to those wishing to keep their information secure as they communicate over the internet. However, these protocols were designed under the assumption that they would only be used to read information in the order it was sent. While, in the past, this was not a huge deficiency, it has become increasingly disadvantageous as real time communication over the internet has become more commonplace. Previous work has been done to alter SSL and TCP to allow applications to read data out of order, thereby receiving up to date information as soon as possible, but the existing SSL cryptosuites are not designed to allow data to be read in such a fashion. In my project, I worked to create an improved cryptographic suite to allow data to be successfully read out of order.

Introduction

With the increased availability of broadband internet, real-time communication via audio or video has become increasingly commonplace. Such communication methods are used both for personal use, using programs such as Skype for video, or Google Talk for VoIP, and in commercial settings, where it is use for long distance video conferencing. These systems are only useful so long as they provide near real-time communication. However, these programs must function as any other program, and also generally require some sort of encryption to keep messages private. In order to satisfy this need for encryption, many such programs use Transport Layer Security (formerly known as Secure Sockets Layer).

In its available forms, however, TLS does not support out of order processing. Given the high frequency of packet loss, this often results in short periods of interruption of the video or audio stream. Professor Bryan Ford and Fitz Nowlan have already created a version of the existing OpenSSL protocol that successfully allows out of order processing, but the implementation uses much of the existing cryptographic protocol, which is not optimized for out of order processing. In particular, the existing version had difficulty identifying record boundaries and easily identifying the sequence number used in creating a given record's message authentication code (MAC).

In addition to implementing solutions to these two issues, the version I created allows for a client and server to negotiate as to whether they will use a traditional version of the SSL protocol, or use the changes I have implemented. I also forced my cryptosuite to use counter mode, rather than cypher-block chaining. Although this seems to be better suited to reading data out of order, it actually does not make is easier to read data out of order, as an entire record must still be present before decryption can take place.

This paper will discuss the design decisions made, along with some advantages and disadvantages thereof. I will also discuss some of the shortcomings of my implementation, and areas for future improvements..

Design

Although many of the basic parameters of my implementation were chosen before I began, there were many smaller choices I had to make along the way. In making these choices, I tried to keep my additions as isolated as possible from the rest of the OpenSSL source code. OpenSSL is an incredibly powerful tool with a vast array of options, versions, and tools, all of which has been extensively tested and examined for security vulnerabilities. To infringe upon this would be dangerous. To this end, almost all of the changes I am about to describe are found in a single file.

Identifying Record Boundaries

The most invasive addition I made to OpenSSL was the use of COBS encoding to help delineate record boundaries. This allows my version to quickly and unambiguously identify where records begin and end, and also unambiguously determine if a complete record has been received. The COBS is encoding is done at the outermost layer of encoding, after the data has been compressed and encrypted, and a MAC as been added to the end of the record. Thus, the encoding is the very last step before transmitting a record, and the first step upon receiving a full packet.

I built the record identification method from the already existing data storage structure by adding in a single simplifying assumption. Specifically, data blocks are separated along each zero byte such that each block either begins with a zero byte or represents an incomplete record. With only this small additional assumption, it becomes simple to search through the linked list and find complete records. Furthermore, this assumption doesn't hurt compatibility with other methods of identifying records.

The disadvantage of this method is that it renders communication via the altered protocol almost immediately distinguishable from regular SSL based communication. While not necessarily a security vulnerability, there is a possibility that this could lead to traffic being blocked by some routers. It also allows a passive viewer to determine the number and length (within a few bytes) of the records being transmitted. Once again, this isn't necessarily a security vulnerability, but it could potentially reveal information about the type of communication taking place between two parties if they are not careful to ensure this information is not significant.

I created my own COBS encode and decode functions for this aspect of the project. They allow any byte, not just zero, to be used as a record delineator rather than just zero. I had originally intended this as an answer to the inefficiency of encoding a text document using zero as a delineator. However, the reality is that SSL records are encrypted and therefore appear to be pseudo-random, and there is thus no advantage to using any particular delineator, no matter what content may be transmitted. However, this implementation allows the delineator to be changed by simply changing a single statically defined constant, and that constant could even be rendered dynamic if that were desirable.

MAC Adjustments

In order to prevent an attacker from replacing a later record with one that has already been sent, SSL requires that an additional argument, beyond the plaintext and encryption key, to be given when calculating the MAC. This key should be one that is not sent over the wire, but is instead calculated by the sender and receiver identically but separately. Normally, this key is the record number of the record to be authenticated, which can be kept as a simple counter incremented with each record written or returned. However, identifying this key is impractical when processing records out of order. Thus, my implementation instead uses the byte offset at the beginning of the record, specifically of the zero byte preceding the record. This can be calculated from the tcp offset returned by the kernel with each packet. In specific, I use one plus the byte offset over five. This may seem random, but five is the minimum length of a record (specifically, the header length), and helps offset the fact that the offset in the write stream is also used as part of the counter for actual encryption. Although the details of such a vulnerability are unclear, it seems unwise to use the same number for two simultaneous cryptographic purposes.

The key challenge of using this method is ensuring that both server and client have the same zero point within the tcp stream. I guaranteed this by setting the value used to one at the beginning of the first data record. However, this may not work following renegotiation or other data reset procedures, which haven't been tested.

Counter Mode

As noted before, using CTR mode rather than the more standard CBC mode doesn't actually add to the capabilities of the cryptosuite, although it may be more secure. It may seem that CTR mode would allow the decryption of partial records, and this may actually be the case, assuming we enforced strict alignment. However, we would be unable to verify the value of the MAC before the entire record has been received, so this would result in using unauthenticated data: a huge security vulnerability. On the other hand, using CTR mode legitimizes initializing the initialization vector to the counter-like value before described below before encoding or decoding each record.

With that in mind, the counter consists of three different segments. The 16 lowest order bits are the counter for individual record. It is unclear to me whether records can be more than 65536 blocks long, but such a case is unlikely, and doesn't pose any security risks. The next 64 bits (or

32 bits for cyphers encoding 64 bit blocks) contains the tcp offset at the beginning of the record. This ensures that the same counter value is never used twice. Furthermore, this value will only overflow when transferring more than 16 exabytes, so we don't yet need to worry about rekeying. The final 48 bytes contain random bytes generated from the master key.

Negotiation

The current negotiation procedure is designed around the selection of a finite number of items. In particular, the negotiation procedure seeks to force the client and server to agree upon a version number, a pair of random values, a compression method, and an SSL_CIPHER object. While it might have been possible to user one of the flag fields within these cipher objects to indicate whether our out of order enabling features should be used, this seemed a dangerous option, as these objects are directly used by the large and complex OpenSSL Crypto library functions. Instead, I added an additional byte to the end of the client and server hellos to indicate their status with respect to the encoding features previously described. It is imaginable that, as additional options are created, this byte could be used to determine more than a binary choice, but, as it currently stands, the choice is fairly simple. There is likely a better way to integrate this negotiation, and this is certainly an area in which the possibility for improvement exists.

Areas for Future Improvements

Although my work on this project met its original goals, there are several ways in which the project could be improved, some more important. In particular, the current implementation forces the application layer to figure out what data it has received, which should instead be done at the transport level. Although the test applications work as desired, this is because they contain their own, rather expensive, order processing code.

Transmission of Sequence Numbers

The most important aspect of this is that record sequence numbers need to be transmitted at the transport level and communicated to the application layer. Applications have to know whether they are receiving sequential or non-sequential data. It is impossible or extremely difficult to piece together without knowing what order they should come in. While the current implementation solves this problem by sending sequence numbers at the application level, it would be much more practical to do this at the transport level. It seems best to simply add the information as an extra section of the header. This would be relatively simple, and since we are already changing the structure of the tcp stream to such a great extent, it wouldn't be unreasonable. This information could then be directly to the application layer via an additional argument or as the first few bytes of data returned by SSL_read().

It should be noted that this addition wouldn't allow the MAC changes previously described to be rolled back. Once the sequence number is sent over the wire, it becomes possible to fake the value, and we therefore cannot trust the security thereof. The point of using the sequence number as part of the MAC calculations is to use information that is not sent over the wire, rather than to use data that is not included in the record itself.

Application-level Ordering Options

Currently, the transport aspects of the project are only designed to be used with the custom application created to test them. While other applications may be able to work under the same assumptions, others require more complex forms of interaction with an out of order transport layer. To begin with the simplest option, it seems impossible that some applications, during at least part of their execution, would not need data to be returned in order. For example, if a message consists of an application level header followed by the message body. In this case, the application might not know what to do with data until the header has been received, and would want a way to request these bytes in order, and then take the rest of the data in whatever order received.

Adding this option would also make it easy to adapt legacy programs to the new version of SSL Without the ability to request data in this fashion, applications would either have to manually shut down this option or be updated to handle data stored in this way.

The implementation of this feature would require only line or two of code in find_record(), and would make it much easier for all types of applications to interact with and take advantage of out of order communication.

Ignoring Old Data

Out of order communication is most advantageous in applications that stop caring about data as it becomes dated. Nevertheless, the current out of order version of SSL_read() will return this data and leave it up to the application to sort out that it is no longer of use. While it would be simple if we were already returning sequence numbers to the application level to also ignore outdated data, it would be simpler if we could instead ignore out of date data at the transport level. In particular, an additional OpenSSL library function could be implemented that would allow applications to move up the currently stored cumulative accumulation point or indicate a record seek number. This would also have beneficial interactions with the ability to require in order delivery.

Conclusion

While my implementation of OpenSSL is clearly not yet ready to be used on real-world machines, it demonstrates an important step in implementing out of order features in OpenSSL and TLS. Further work is needed to make this option viable in the wild, but my implementation meets the goals it was created for. Valid transmissions never suffer from MAC failure, and the receiver never has difficulty identifying, decrypting, or authenticating a complete record once it has arrived. The transmission remains, to the extent that I can tell, secure, and represents a considerable step forward when compared with previous attempts to implement out of order communication through SSL.