Secure Auditing for SSL Transactions

Eric Rescorla, Kevin Dick
Claymore Systems, Inc.
{ekr,kevin}@claymoresystems.com

Keywords: SSL, auditing, dispute resolution

Abstract

Although SSL is certainly the dominant security protocol in use for electronic transactions, it has no real provision for dispute resolution. Digital signatures, the traditional approach to this problem, have seen little deployment, largely due to the lack of of ubiquitous client-side PKI and the need to modify both client and server software to add signature capability. This paper describes an alternate approach without these drawbacks. We use a novel combination of passive session recording, secure hardware and playback to provide third-party auditing capability for SSL transactions without changes to the applications on either side.

1 Introduction

SSL [1] and it's follow-on protocol TLS [2] are by far the most widely used Internet security protocols, to a great degree because they are easy to implement and deploy. Because they are so similar we will refer to them collectively as SSL. However, like most channel security protocols, SSL's security properties are ephemeral and cannot be verified by third parties. If two parties who have engaged in an SSL transaction later dispute the contents of that transaction, neither party can demonstrate which version of the transaction is correct. This difficulty limits the usefulness of SSL for electronic commerce.

When this issue is raised, the standard response is for the counterparties to digitally sign their messages using some standard format such as [3] or [4]. While this approach appears to solve the immediate difficulty, it comes with a variety of drawbacks. The most serious difficulty is one of deployment. The current generation of transaction software is not designed to sign messages. The introduction of digital signatures necessarily requires altering a large number of existing systems. This is an undesirable prospect under any circumstances.

Second, widespread signing depends on the existence of a certificate infrastructure for both sides of the transaction. Such an infrastructure is not currently in place, especially for clients. Additionally, the possibility (though currently not reality) of certificate revocation implies that messages must somehow be cryptographically timestamped in order to establish that they were signed prior to revocation. Although the protocols [5] are in place for this service, it is not currently widely available.

These difficulties have effectively blocked the deployment of signing-based solutions. Instead, vendors and merchants have chosen to secure their transactions with SSL and either rely upon unsecured logging for dispute resolution or ignore it entirely. SSL Auditing leverages this existing infrastructure to provide dispute resolution with little or no change to existing applications and consumes effectively the same resources as would the signing approach.

2 Overview of Our Method

In this paper, we describe an alternative to digital signatures that allows us to provide dispute resolution without rearchitecting applications and protocols. Our approach comes in two pieces. First, we capture the traffic between client and server, timestamp it, and write it to a secured audit log. This provides integrity against attackers changing transactions after they have occurred.

Second, we harden the SSL implementation on the SSL server so that the administrator cannot access his private keys or session keying material. This prevents the server administrator from faking recordings of transactions in progress, thereby allowing him to establish the correctness of his version of the transaction to a third party.

SSL is a complicated system and it is not immediately obvious that this combination of recording and trusted hardware provides the necessary security properties. In fact, naive approaches to SSL recording have some subtle security flaws, especially on playback. Therefore, in the interest of clarity we will build up to our solution in (hopefully easy to understand) pieces, examining the security properties at each stage along the way to ensure that the resulting system will have the security properties that we desire.

3 The Problem

In order to understand how to provide dispute resolution for SSL, we first have to understand what security properties SSL currently provides. In an SSL

connection, the peers use public key cryptography to establish a shared key. Once the shared key is established, they use it to generate symmetric keying material. From then on, all messages are then encrypted and MACed using those symmetric keys.

Unfortunately, neither party can prove the contents of the transaction or in general that the transaction occurred at all. There are two exceptions here: if ephemeral keying is used then the client can prove that some transaction occurred and if certificate-based client authentication is used then the server can prove that some transaction occurred.

The problem, then, is to establish three facts in the general case:

1. That an SSL transaction occurred.
2. The parties to the transaction.
3. The contents of the messages in the transaction.

It's sufficiently difficult to see how to do this with SSL that it's useful to first consider a simple abstract system. Alice and Bob share four symmetric keys, one key for transmission in each direction and one for authentication/integrity in each direction. We're only concerned with the integrity keys which we'll label $K_{a->b}$ (for transmissions from Alice to Bob) and $K_{b->a}$ (for transmissions from Bob to Alice). Alice and Bob exchange messages MACed with these keys. There's agreement on which keys are to be used for which purpose and the parties have some way of proving that to a third party. The problem, then, is to establish which party transmitted a given message and what the contents of that message were.

4 Dispute Resolution for Symmetric Cryptography

From a functional perspective the important difference between public-key digital signatures and symmetric MACs is that digital signatures are third-party verifiable whereas MACs are not. Note that we've deliberately avoided using the term *non-repudiation* here because that term seems to have acquired a substantial amount of operational and policy baggage. The relevant issue here is purely technical, namely that either communication peer can generate any number of valid MACs.

It's long been known that the combination of trusted[1] hardware and symmetric cryptography can be made to emulate asymmetric cryptography and its easy to make the transformation in this instance. The requirements are simple:

1. Each recording peer has one transmitting key and one receiving key.
2. The keys are stored in a tamperproof message security module (MSM) that never exposes them.
3. The transmitting key can be used to generate or verify MACs and the receiving key can only be used to verify them.

Such a system is essentially isomorphic to public key cryptography. Any message MACed with Alice's key must have been generated by Alice. A third party can easily verify such a MAC-signature by interrogating either Alice's or Bob's MSM. This isn't quite as convenient as a digital signature because it requires the cooperation of one of the peers but that cooperation isn't a problem in any dispute between the two parties, because the party telling the truth has an incentive to cooperate, and the other party can't cheat, even if they decide to cooperate.

4.1 Trusted Hardware on One Side

Obviously, this system isn't maximally convenient because it requires the installation of trusted hardware at both Alice and Bob's sites. Let's explore what happens if only one side has trusted hardware. Consider the case where Bob has an MSM but Alice's implementation is in software. This system has the following security properties.

Bob can demonstrate that Alice sent any message that she actually sent. He does this by presenting a copy of the message and the proof that it was MACed with Alice's key. Because Bob does not have direct access to the keying material, he could not have generated the message in question.

Alice cannot demonstrate that Bob generated any message regardless of whether he did or not. Because Alice has access to the keying material, she could easily have generated any such message.

As expected, then, such a system with trusted hardware on only one side provides only half protection. Note, however, which party has been protected—the party who deployed the trusted hardware. Bob can no longer be accused of having forged messages from Alice. Disputes over such messages are easily resolvable: any message with a valid MAC must have been transmitted by Alice at one time or another. The basic principle here is as follows: *A party can protect himself against charges of fraud by making it technically impossible for him to commit fraud.*

However, we have no such assurances with regard

1. We're using trusted here in the sense of trusted by third parties, not trusted by the owner of the device. In fact, the entire purpose of the device is to prevent the owner of the device from doing things he might otherwise wish to do.

to messages ostensibly transmitted by Bob, because both Alice and Bob can generate such messages. Thus, if Alice claims that Bob sent a given message and Bob claims otherwise, we have no way of resolving the dispute. Next, we will consider how to solve that problem.

4.2 Preventing Message Substitution

There are actually three kinds of disputes about messages sent by Bob.

1. Alice and Bob disagree about the contents of a given message: Alice claims that Bob said one thing and Bob says it was another.
2. Alice claims Bob sent a given message and Bob says he didn't.
3. Bob claims that he transmitted a message and Alice says he didn't.

In order to resolve disputes of class 1, we need to be able to distinguish the following situations: (a) Bob generated two alternative messages and sent one to Alice and kept the other or (b) Alice generated a message alternative herself and has presented it as the message generated by Bob. Applying our basic method here, we prevent this attack by removing the ability for Bob to generate message alternatives.

This is can be done by including a sequence number in each message. This sequence number must be maintained by the MSM, which guarantees that it increases by one with each message sent. Naturally, once this countermeasure is deployed, Alice can merely forge a message with a different sequence number and claim that the dispute is of type (2). Bob prevents this attack by storing a copy of every message that he sends. He can then provide proof that an alternative exists for every message that Alice might potentially claim he had generated. (This is why we require that the sequence numbers are incremented by one—so that Bob can ensure that there are no unused sequence numbers for Alice to exploit). Because each message is MACed by the MSM, there is no need for them to be kept in a tamperproof store. Instead, messages can be archived in ordinary insecure storage and then verified with the MSM when a dispute occurs.

Note that we have done nothing to protect against the third class of disputes. However, this class of attack is not prevented by simple message signing either. Instead, it's provided by having the applications generate receiver provide a signed message acknowledging receipt of the data. Because we are substituting MACs for signatures, Alice can simply generate a MACed receipt, which will have the same security properties because of the tamperproofing of Bob's MSM.

4.3 Proving Arrival Time

Earlier we stated that Bob could prove that Alice sent a message but not at what time it was sent. However, in many cases (stock transactions, for instance) arrival time is important. Again, we can solve this problem by enhancing Bob's MSM. The naive approach is simply use timestamps. Whenever a message M is received, Bob's MSM generates a timestamp record, as follows:

$$Timestamp = MAC(K_{timestamp}, Message||Time)$$

$K_{timestamp}$ is simply a secret key generated and kept inside Bob's MSM. The MSM is therefore able to verify any timestamp that it has generated. These timestamp records are then stored along with their corresponding messages, allowing Bob to prove that messages were received at a given time.

However, this procedure still allows an attack—Bob can generate multiple timestamps for the same message, thus making the message appear to have arrived arbitrarily late. We can prevent this attack by including sequence numbers in the timestamps, as shown:

$$Timestamp = MAC(K_{timestamp}, Message||Time||Seq_{timestamp})$$

Bob can prove that he has not generated duplicate timestamps by showing all timestamps generated within the relevant time and demonstrating that the sequence numbers are consecutive and that there are no duplicate messages.

Note that Bob can delay timestamping a message by merely reading it but not timestamping it. However, if messages are encrypted, the MSM can be constructed to prevent this attack by timestamping the message at decryption time and before delivering the plaintext to Bob. Of course, Bob can still delay the timestamp by delaying reading the message. This can't be prevented as long as Bob controls the network.

4.4 Proving Generation Time

It's actually desirable for Bob to timestamp his own messages as well. First, this provides direct evidence of when a message was sent as opposed to indirect evidence based on sequence numbers. Second, the more messages are timestamped, the easier it is to determine when an event occurred, even in the face of clock slippage, by triangulating between events that occurred at known times.

Note that it's still possible for Bob to generate the message and then delay its transmission until some later date. This attack is difficult to prevent, even with public key systems.

4.5 Split Construction

There's no need for the timestamping engine to be located in the same unit as the secure message processing component. Instead, it can be configured as a separate timestamping device (TSD) in the communications path between Alice and Bob, as shown in Figure 1. This approach has the advantage that a single timestamping engine can provide security for other users at Bob's site. This is an enormous operational advantage because the engineering requirements for the secure message processing component are very simple compared to those for a timestamp engine where clocks and permanent storage are required.

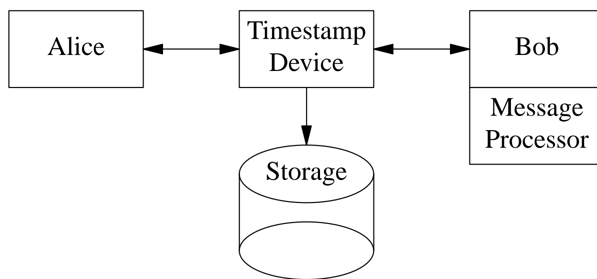


Figure 1 A split architecture

4.6 The Uses of Public Key Cryptography

So far, we have taken care to design a system which uses only symmetric cryptography in order to avoid confusion about the claim that we're replacing public key. In practice, however, it's much more convenient for the TSD to use public key cryptography to generate timestamps, because they can then be independently verified without the assistance of the TSD. Remember, the objective here isn't primarily to avoid public key but merely to avoid individual message signing by the endpoints because it's disruptive to add it there.

5 Auditing Against Reactive Fraud

There are essentially two kinds of fraud which can lead to a dispute, *proactive and reactive*. In a proactive fraud, one party sets out to cheat the other party at the outset of a transaction. An example of proactive fraud would be an attacker who desired to fraudulently order some good in the name of someone else.

In a reactive fraud, the counterparties perform the transaction in good faith and only later does one of them desire to repudiate it. For instance, consider a customer who purchases a large number of shares of a certain stock just prior to a substantial drop in the price. Such a customer might be tempted to repudiate the purchase. This would be a reactive fraud.

In addition, disputes can arise due to good faith misunderstandings. Neither humans nor computer systems are infallible and therefore its normal for customers and vendors to disagree about the price, type, or quantity of an item ordered. In many such cases, the customer's records say one thing and the vendor's records another. Because an attacker wishing to mount a reactive fraud would naturally change as many of his records as possible, resolving such disputes is essentially the same as resolving cases of reactive fraud.

Obviously, it's desirable in any dispute resolution system to be able to resolve all three kinds of disputes. However, it is possible to dispense with a substantial amount of of the infrastructure described in Section 4 if we concern ourselves only with reactive fraud.

5.1 Timestamping Only

The solution we've been exploring so far assumes secure hardware. Not everyone's messaging implementation is in hardware. Imagine that both Alice and Bob both have software-only messaging systems without any hardware enforcement. They wish to deploy some form of dispute resolution but neither is willing to displace their existing messaging infrastructure. It is nevertheless possible to provide a substantial amount of additional security.

The enabling insight is that the secure timestamping device described in Section 4.5 provides substantial security value even in the absence of a trusted messaging component. Because the TSD can be installed without changing other components of the system, whereas the message processing component is inherently disruptive, a timestamping-only system is worth exploring. Thus, we get the system shown in Figure 2.

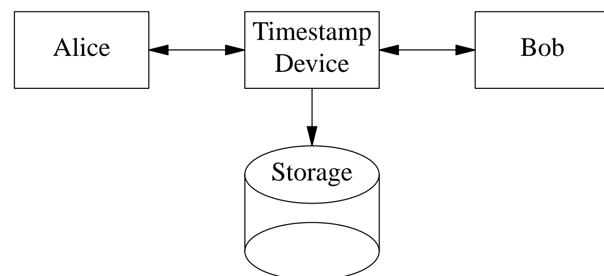


Figure 2 Seamlessly inserting a timestamping component

As before, the TSD captures every message transmitted between Alice and Bob, timestamps them as described in Section 4.3, and writes them to permanent storage. There is no need to change either end node to install such a system, one simply reconfigures the underlying message transport to pass the data through the TSD en route to the ultimate receiver. (In fact, as we'll see in Section 6.4, the TSD can be made completely passive).

5.2 Security Properties of a Simple Timestamping System

A TSD provides excellent protection against reactive attacks. Because every message passes through the TSD, it's easy for either side to demonstrate the contents of any message simply by fetching it from long term storage and verifying the timestamp. Even if the TSD is under the control of one party, they can only inject new data with new timestamps, not remove timestamped data or change its contents.

However, such a system is extremely vulnerable to proactive attacks. Any party which can control data to and from the TSD can forge any timestamp trail they wish. Consider the situation shown in Figure 3, in which Bob controls the inputs and outputs to the TSD. Bob generates two messages, one of which (M) is acceptable to Alice and one of which (M') is what he intends to later claim that he sent. He sends M directly to Alice and M' to the TSD and arranges for the output of the TSD (M') to be discarded. Alice believes M, but the audit trail says differently.

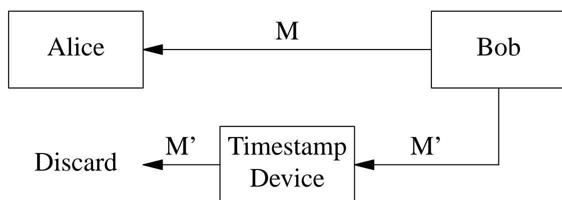


Figure 3 Forging a timestamp entry

As long as the TSD is under the control of Bob, proactive fraud is possible for Bob. In addition, this possibility allows Alice to commit fraud and blame it on Bob. On the other hand, if a third party controls the TSD and the communications channel then this problem can be avoided. For instance, if both parties negotiate a secure connection to the TSD and then pass all of their communications through the TSD, then proactive attack can be prevented; because neither party can bypass the TSD, a receiver can know that the message they received was the one that was timestamped.

6 Secure SSL Recording

So far, we've seen how to add dispute resolution to a model system which uses only symmetric cryptography. Although the previous two sections have contained a lot of detail, there are really two basic ideas:

- Secure timestamping can be used to provide security against reactive attacks.
- Trusted hardware + secure timestamping can be used to provide security against proactive as well as reactive attacks.

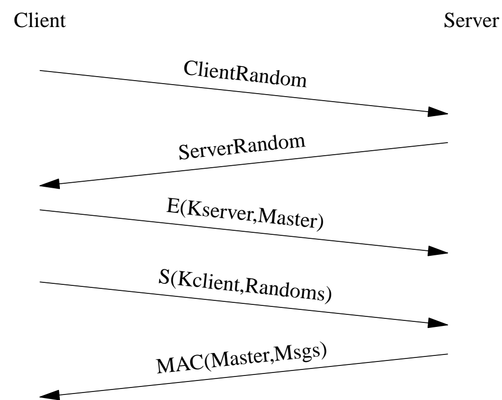
The rest of this paper describes how to apply these tools to the real-world system of an SSL client and server. SSL connections have two important differences from our model system.

1. The symmetric keys are exchanged via public key cryptography (typically RSA) rather than pre-arranged.
2. The client's identity is not always cryptographically authenticated via the protocol.

These differences complicate the problem some because in order to establish the contents of communication, we must first establish the the authenticity of the keys. There are two potential cases we need to be concerned with: mutual public-key authentication and one-way public key authentication.

6.1 Mutual Public-Key Authentication

In SSL, public key authentication is provided differently by the client and server: the server authenticates by decryption and the client by digital signature. Figure 4 shows the relevant parts of the process.



Because the client generates the Master secret, it does not need to know the server's private key in order to generate any messages. It is therefore possible for a

client to forge a complete handshake. By contrast, the server cannot forge such a handshake because it cannot generate a signature by the client without possessing the client's private key. Thus, even with no hardware support, a server can provide proof that a transaction occurred, though not of the contents of the transaction because the server owner might have captured the handshake but then forged the message contents. As before, a recording of the transaction with timestamps can be used to ensure that the server owner doesn't cheat reactively, but not proactively.

Now consider what happens if we harden the server's SSL implementation so that the server administrator/operator (who we'll call Sam) cannot access his private key or the session keying material (this is good security practice in any case). In this case it is impossible for Sam to forge a message, because that would require access to the master secret.

The situation is similar if we impose trusted hardware on the client side. Because the client user (who we'll call Charlie) cannot sign the CertificateVerify without access to his private key, the use of which is constrained by hardware, he cannot generate a valid handshake or messages without cooperation from the server. Thus, it's apparent that if we add trusted hardware to either side, we can obtain equivalent security guarantees to the purely symmetric system described in Section 4.

6.2 Server-Only Authentication

A more common case is that in which the server authenticates via a certificate but the client authenticates via some application-level mechanism such as a password or application-level challenge/response. In such a system, it's easy to prevent reactive fraud using the secure timestamping technique described in Section 5.1. Preventing proactive fraud is more difficult.

Client Side

If the client has no keying material of its own, there's essentially no way for it to remove its ability to commit fraud. The basic problem here is that the client user can always write a piece of software which communicates to the server and generates any SSL transaction he wants. So, no recording made by the client can be trusted.

Server Side

There are actually two kinds of proactive fraud we need to be concerned with (1) Sam somehow obtains valid authentication data (e.g., a password) from the client and then generates a totally false transaction and (2) Sam hijacks an existing connection and generates false data after the authentication phase. The second attack is prevented by simply hardening the server's SSL implementation, as described in Section 6.7. Because Sam has no access to the keying material, he cannot participate in an SSL session. The first attack is extremely difficult to prevent because Sam can generate any transaction Charlie could have. We'll discuss some countermeasures in Section 8.2.

	Location of hardware SSL impl. and TSD	
Authentication	Client	Server
Mutual	Proactive/Reactive	Proactive/Reactive
Server-Only	Reactive Only	Proactive/Reactive

Figure 4 Summary of SSL dispute properties

6.3 Reference Topology

Figure 5 shows a simple model network on which one might wish to deploy SSL Auditing. The client and server are both connected to the Internet and communicate via a single SSL connection. The dashed line represents the boundary between the premises of the server operator and the ISP's premises.

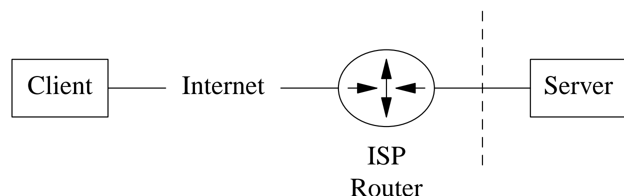


Figure 5 Reference topology

6.4 Recording Traffic

In order to have the minimal impact possible on deployed networks, we use the split architecture described in Section 4.5. In our system, the timestamping device is coupled to the long-term storage for message traffic and is called the Recorder. In the simplest case the Recorder is placed on the same network as the server, as shown in Figure 6. Rather than placing the recorder in the communications path, we simply attach it to the same wire as the server and sniff SSL traffic off the wire, thus enabling the Recorder to be deployed

more easily. The Recorder captures selected packets (those believed to be associated with relevant transactions off the network) and writes them to permanent storage with an associated timestamp attesting to both their contents and their arrival time. The timestamp is digitally signed by the recorder in order to allow the use of cheap bulk external storage.

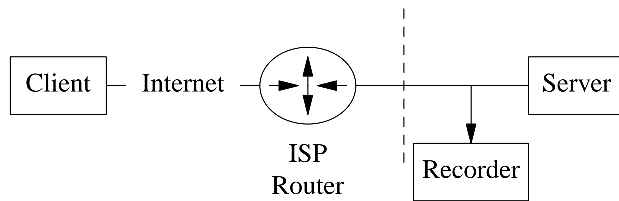


Figure 6 A network with a Recorder

6.5 Recording transactions

For the reasons mentioned in Section 4.3, it must not be possible for an attacker to inject traffic in such a way that it generates an audit entry that appears to be in the past. Because the Recorder is situated within the premises of one of the parties and that party might wish to tamper with the audit trail, the Recorder must be physically tamper resistant (or at least tamper evident).

Conceptually, audit entries consist of five pieces of data:

- The recorded packet
- A connection sequence number
- A packet sequence number
- The time at which the audit entry was recorded.
- A signature over the entire entry by the Recorder key.

As previously described in Section 4.3, the sequence numbers prevent insertion and/or deletion of timestamp entries, while the timestamp prevents backdating, even if the Recorder is otherwise idle. The signature over the entire entry prevents substituting data from one connection with data from another connection. Recovering the contents (ciphertext) of a given connection is a simple matter of finding the appropriate audit log entries, verifying their contents and concatenating them together.

A Better Data Structure

Although this scheme is secure, it is highly inefficient, for two reasons. First, it requires the Recorder to perform a digital signature for each packet that it receives. Even with hardware cryptographic acceleration, the number of signatures required is prohibitive on a high-speed connection. Second, recovering a connection requires examining every packet received by the network during a relevant period, filtering out the vast majority which belong to other connections.

The dominant cost in signing is the public key operation, not the size of the data (which is digested before signing). Thus, an efficient way to reduce cryptographic signing load is to amortize the signing by signing more than one packet at once. Before signing, we collect the timestamp data for a run of consecutive packets into a single audit entry. This entry is then signed, as shown in Figure 7. Note that each packet timestamp contains its arrival time, so no time resolution is lost.

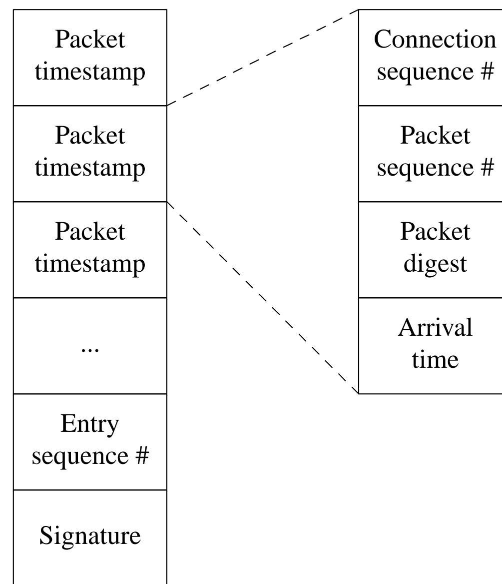


Figure 7 An audit entry

We can reduce the cost of replaying a given connection by storing the packet data and the timestamps separately. This is easily done by storing the data for each TCP connection in a separate file. Instead of storing the timestamp with the file we attach a pointer to the relevant timestamp with each packet. The resulting data structure is shown in Figure 8. When we desire to replay the connection we simply read each packet for a given connection and then look up and verify the appropriate timestamp against the current packet. An alternative solution would be to store all the packets in the same store and then have an index that said what packets were in each connection.

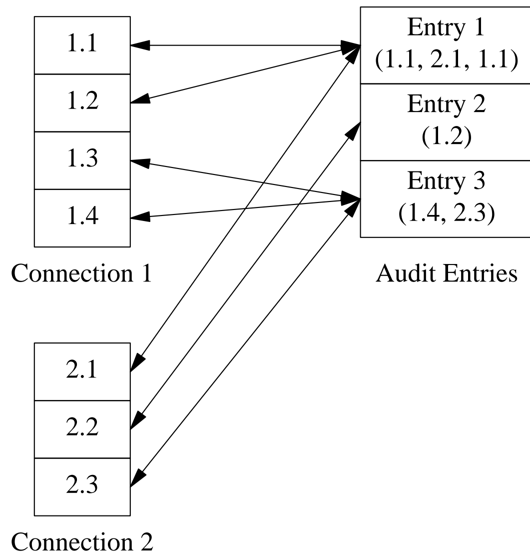


Figure 8 Audit log structure

6.6 Protecting the Recorder

When considering the system in the abstract, it was convenient to treat the timestamping engine as a single monolithic unit. However, from a practical perspective, this is not the optimal design. Conventionally available hardware security modules (HSMs) are much slower than commodity computers. Because we wish to capture a very high traffic load, the HSM can easily become a bottleneck. It's therefore desirable to move as much processing as possible out of the tamperproof module and into a general purpose computer, which can be much faster.

The network sniffing portion of the Recorder does not need to be tamperproofed. Because the network which the Recorder is sniffing is completely under the control of the entity which physically controls the Recorder, any attacker who wants to control what data gets timestamped can do it by controlling network traffic directly instead of by tampering with the recorder. The only part of the Recorder that really needs to be protected from tampering is the timestamping engine itself, namely:

- The timestamping key
- The clock
- The timestamp sequence numbers

The result of this separation is the system shown in Figure 9. There are two components, the Recorder, which sniffs the network and stores the data, and the Master, which is embedded in the HSM. The Recorder receives data from the network and buffers them in the "to-be-signed" queue. It then generates a series of timestamp requests (*ts_reqs*), which it places in the queue to be signed by the Master. The Master infinite loops, reading sequences of ts_reqs and signing them as a single audit entry.

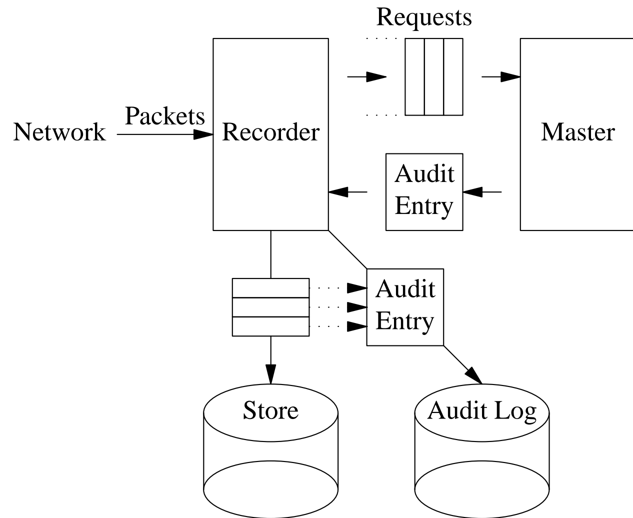


Figure 9 Packet timestamping with Recorder and Master

As we said in Section 6.5, each timestamp actually covers multiple packets. Each time the master is ready to read, it reads as much of the ts_req queue as is available (possibly up to some set maximum), concatenates them into a single data object and signs the entire message. This mechanism allows timestamp coverage to be self-clocking—under high load the Master compensates by incorporating the data for multiple packets in a single audit entry.

Because packet timestamps contain digests of packets, not the packets themselves, the required performance from the Master is quite reasonable. For instance, if we assume an average packet size of 1000 bytes, a 100 Mbit network will be able to carry roughly 10,000 packets/sec. Because packet timestamps are only 40 bytes long, this means that the Master needs to be able to to digest roughly 400 k/sec (remember that the digital signature can be over an arbitrary number of packets and therefore the amortized cost of the signature is negligible).

Once a series of packets has been signed, the Master returns the audit entry to the Recorder. The Recorder then removes the corresponding packets from the to-be-signed queue and writes them to permanent storage along with a pointer to this audit entry. Finally, the Recorder writes the audit entry to the permanent audit log.

Note that both the audit log and the connection store are located outside the Master. This is not a security problem. Because the audit log entries are signed, the audit log does not have to be stored securely. Because the connection data is encrypted with SSL, it can be stored without further encryption on disk.

6.7 Hardening the SSL Server

In general, e-commerce systems are implemented as custom or semi-custom applications on standard Web servers. Thus, it's not convenient to replace the entire server in order to harden the SSL portion. This leaves us with two basic approaches to hardening the SSL server:

1. Add a trusted module that implements SSL.
2. Move the SSL function to a hardened proxy.

It's technically possible to provide a trusted SSL module in hardware, but no such modules currently exist. Most SSL hardware modules are designed for acceleration and therefore give the server software access to the keying material. It would be a substantial engineering effort to develop such a module, though once one existed it would be straightforward to integrate it into existing systems by imitating one of the common SSL implementations such as OpenSSL. In general applications have no need to access the SSL keying material and so this replacement can be done compatibly.

The preferred solution is to use a hardened version of an existing SSL proxy, such as those manufactured by Intel[6] or SonicWall [7]. Such proxies are in wide use to accelerate SSL transactions by offloading the SSL processing. It would be relatively straightforward to harden such a proxy by simple software modifications to protect the private key and then tamper-sealing the case so that the key could not be extracted without leaving evidence.

6.8 Sequential Sequence Number Generation

Because the server and the Recorder are not tightly coupled and Sam controls the network, it is possible for Sam to mount a proactive attack on any given connection by temporarily disconnecting the Recorder. Sam can then deny that the transaction ever occurred. This attack is easily blocked by slightly modifying the generation value for the SSL ServerRandom value to include a transaction sequence number. Because the ServerRandom value is 32 bytes with 28 bytes of randomness (the first four bytes are time of day) a four or even eight byte sequence number still leaves plenty of entropy. It then becomes simple to ensure that all transactions have been recorded by checking that all recorded sequence numbers are consecutive.

It's critical to note at this point that the modifications we're making to the server are simple and confined solely to the SSL implementation. Most e-commerce systems are built as custom components on top of an application server containing a generic SSL implementation, which usually uses one of a few SSL toolkits. Small changes to the SSL implementation can therefore be made by replacing the toolkit with a custom version. This is somewhat disruptive, but much less so than modifying the applications—which generally requires custom programming—or the wire protocols used by the applications, both of which are required to deploy message signatures.

7 Secure SSL Replay

Conceptually, replay is a three stage process:

1. Recover the connection data from the recorder and verify its timestamps.
2. Decrypt the connection.
3. Decode the application layer traffic.

Because this is a security paper, we will not discuss the third stage, except to mention that each application layer protocol generally requires its own custom decoder. We have developed such decoders for HTTP [8] and SOAP [9].

7.1 Recovering and Verifying the Connection Data

Recovering the connection data is straightforward. We simply read it out of the Recorder's long term data store. Then we find the audit entries pointed to by each packet record and verify that:

- The signature on the audit entry is valid.
- The digest over the packet matches the digest in the audit entry.
- The sequence numbers in the stored packets are consecutive and match those in the packet timestamp entries.

The first step ensures that the audit entry itself is valid. The second step ensures that the data in a given packet has not been tampered with. The third step ensures that no packets in the connection have been removed or reordered. Once all three steps have been completed, we

know that the entire connection is valid. We then perform TCP reassembly and pass the data stream on to the SSL decoder.

7.2 Decrypting the SSL Data

Decrypting SSL traffic is simple if we have access to the private key, provided that the SSL cipher suite uses static RSA (which nearly all connections now do). The Player simply does the same computations as an SSL server would. We simply identify the ClientKeyExchange message and decrypt the EncryptedPreMasterSecret field to recover the master secret. From there, the Player generates the traffic keys and can decrypt and verify data flowing in either direction. The publicly available program ssldump [10] already does this job.

7.3 Key Revelation

The procedure just described assumes that the Player has access to the SSL server's private key. In general, this is unsatisfactory. Recall that in order to protect against proactive fraud, the server's key must be kept in trusted hardware. If the Player had access to the server's private key, it could engage in proactive fraud. While we could potentially place the Player in a trusted device along with the Master, it's much more convenient if it can be a simple piece of software. Even if we were willing to have a hardware-only Player, it's bad security practice for the server to give up its private key unless absolutely necessary.

The simplest approach is to use the server as an oracle to decrypt the connection for the Player. The Player sends the EncryptedPreMasterSecret and some authentication information to the "Key Server", which is installed on the server. The Key Server returns the PreMasterSecret. This is the only information that the Player needs to decrypt the connection.

This approach has three major flaws:

1. It requires the Key Server to perform access control checks.
2. If the Player recovers a key for a connection which is currently active, it can forge traffic on that connection.
3. It allows the Player to generate fake new connections by replaying the handshake and then inserting chosen application data (because it already knows the traffic keys).

We can remove all of these problems by using the Master (which, recall, is embedded in the HSM) as a choke point for decryption, as shown in Figure 10. As before, the Player sends the EncryptedPreMasterSecret to the Key Server. However, instead of returning the PreMasterSecret the Key Server encrypts it under the Master's public key. The Player then transmits the ReEncryptedPreMasterSecret to the Master along with its credentials and the SSL ClientRandom and ServerRandom values for this connection. The Master checks the Player's credentials, generates an audit log entry for this revelation and returns the traffic keys for the connection.

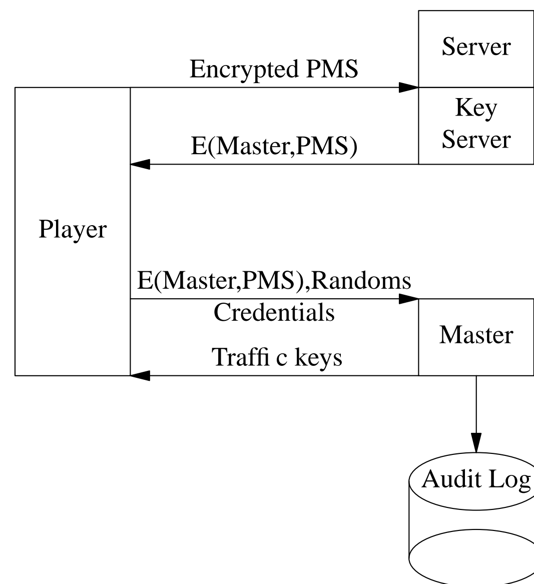


Figure 10 Revealing the keying material

Although the Player sees the ReEncryptedPreMasterSecret, it can't do anything useful with it because it is encrypted for the Master. Thus, the Key Server can avoid performing any access control checks because they can be done on the Master. The purpose of the logging the revelation of the key is to solve the second and third problems. When replaying connections, a Player simply checks that the keying material was not revealed prior to the end of the connection and thereby can be sure that some other Player user has not enabled a proactive attack. Note that because we reveal only the traffic keys and not the MasterSecret, other connections from the same SSL session are not exposed.

7.4 Session Caching

Because the cost of RSA operations is relatively high, SSL incorporates a feature which allows peers to reuse the MasterSecret established in one connection for

future connections. A connection which reuses the MasterSecret in this fashion is said to have *resumed* the session. Connections which resume a session cannot be replayed independently because the EncryptedPreMasterSecret does to appear in the handshake. In order to decrypt such connections, the Player must identify the connection which created the session and first recover the MasterSecret for that connection and use that to decrypt the resumed session. Because the session identifiers are present in the handshake in the clear, we can readily identify the relevant connection without having access to any secret information. Only the key revelation requires the cooperation of the server, and this is accomplished in essentially[2] the same way as with an ordinary connection.

7.5 Ephemeral Keying

Although the vast majority of SSL sessions use static RSA, SSL contains two ephemeral keying modes, one using DH and an export-compatible ephemeral RSA mode which is largely obsolete. These modes provide some Perfect Forward Secrecy, which is generally considered to be a feature but which is a problem for replaying.

Because essentially all clients support static keying modes, it's probably easiest to simply configure the server not to support ephemeral keying. However, if the client requires ephemeral keying this can be supported with relatively minor changes to the SSL implementation. The trick is to have predictable ephemeral keys. For instance, one might use a strong PRNG to generate the ServerRandom and then generate the ephemeral DH private key using

$$X = HMAC(Static_Secret, ServerRandom)$$

The Key Server could then recover any connection's traffic by computing the appropriate DH share from the ServerRandom.

Obviously, using this technique means that the connection no longer has PFS, which is theoretically undesirable. However, in practice, DH is used quite rarely and ephemeral RSA is generally used with a long-lived "ephemeral" key and so the reduction in security is small.

2. SSL uses a two stage key derivation process in which the PreMasterSecret is hashed with the random values to generate the MasterSecret, which is hashed again with the randoms to generate the traffic keys. Because a resumed connection uses the same MasterSecret but different random values, the Player must provide the Master with the random values from both connections in order to allow it to generate the MasterSecret (using the original randoms) and the traffic keys (using the new randoms).

7.6 Partial Data Streams

It's well known that SSL does not work properly in environments where packet loss or reordering is possible. The problem is that the encryption context (CBC state, stream cipher keystream, sequence numbers) is chained through multiple records and it is therefore problematic to decrypt a record without having seen the entire connection. This presents a real problem for replay: because the Recorder is implemented as a passive sniffer, it's possible to miss packets.

With some thought, however, packet loss turns out not to be insurmountable. Although it's true that in some cases, packet loss makes a connection unrecoverable (for instance, if the packet containing the ClientKeyExchange message is lost) in most cases it is possible to recover the bulk of the traffic.

Block Ciphers

It's easiest to recover if a block cipher is being used. Imagine that we see a run of records, then lose a packet, producing a gap of *Length* bytes. Call the first record after the gap R_x, the next one R_{x+1}, etc. Recall that SSL uses the last cipher block of record R_{x-1} as the CBC IV for record R_x. Thus, if record R_{x-1} is lost, we are not able to decrypt the first cipher block of record R_x. However, the rest of the record can be easily decrypted. Thus the records we didn't see are unavailable, but we can mostly access the records we managed to capture.

Obviously, because we don't have the first plaintext block of R_x, we can't verify the MAC. However, we *can* verify record R_{x+1}. Unfortunately, because we don't know how many records have been lost, we don't know what the sequence number for R_{x+1} is, and it's required to verify the MAC.

We can brute-force resynchronize by iterating over all possible sequence numbers until we find one where the MAC matches, at which point we must be synchronized. The question now becomes whether resynchronization can be done efficiently. The SSL sequence number space is 64 bits. Clearly, then, searching the entire space is prohibitive. Intuitively, there cannot have been more records than the number of lost bytes, so the number of sequence numbers we need search cannot be greater than the length of the missing region.

In practice, we need search far fewer than that. The maximum sequence number can be found by asking what the maximum number of records could have been contained in the missing TCP segments. Assuming that records are non-empty, the minimum plaintext size is $1 + M$ where M is the size of the MAC. Because M is either 16 (for MD5[11]) or 20 (for SHA-1[12]) and we have to pad to a block boundary, the ciphertext will be

24 bytes for DES and 3DES and 32 bytes for AES [13][14]. With the 5-byte record header, the minimum record size *MRS* is 29 bytes (37 bytes with AES).

If we assume that the gap is smaller than the maximum SSL record, then the minimum number of missing records is 1. Thus, if we have a gap of *Length* bytes, this could contain anywhere from 1 to *Length*/29 records. If the last record before the gap has sequence number N, then R_{x+1} has a sequence number in the range $[N+3, N+2+(Length/MRS)]$. (The first missing record has sequence number $N+1$ and R_x has sequence number $N+2$). If we've lost a single Ethernet frame, this means checking no more than 50 sequence numbers for DES/3DES and 40 for AES.

In practice it's rarely necessary to try more than a few sequence numbers. Most SSL implementations use relatively consistent large record sizes so the loss of a single frame probably only means losing a single record.

In many cases, the data being transmitted will be highly structured, in which case we will have a good chance of guessing a single cipher block. If we can guess the plaintext of the leading block of the first record after the gap, we can verify it by checking to see if the MACs match. Note that it's much more efficient to resynchronize on the next record and then go back and try to verify our incomplete record: if there are n possible sequence numbers and m possible first blocks, this strategy requires $m+n$ operations rather than the $m*n$ operations required to resynchronize and guess plaintext blocks at the same time.

Stream Ciphers

If the traffic is encrypted using a stream cipher, the problem becomes figuring out exactly what section of keystream to use. We know from the TCP sequence numbers how much data we have lost but because SSL record sizes are variable, we don't know exactly how many bytes of keystream have been used (recall that the record headers are not encrypted). However, we can use a technique similar to the one we used for block ciphers to recover the appropriate keystream offset.

Naively, we could simply try each potential stream cipher offset/sequence number combination until R_x verifies correctly, but this could be very expensive. Instead, we can take advantage of the fact that the keystream offset can be predicted if you know how many records you've missed. The logic is as follows: We know that *Length* bytes of TCP data are missing. Some of that data is un-encrypted headers. The rest was encrypted. The amount of keystream used is the size of the data which was encrypted, which is equal to *Length* minus the un-encrypted headers. Because SSL record headers are 5 bytes long, we get:

$$Offset = Length - (Records * 5)$$

Thus, we simply iterate over the possible values for *Records* until a MAC matches. Because each value of *Records* corresponds to one sequence number for our target record, we don't need to iterate over sequence numbers as well.

As before, we can determine the upper and lower bounds for the number of records lost. Because the records are unpadded, the minimum SSL record size when a stream cipher is used (again, assuming 1-byte long plaintext) is given by $MRS = 5 + 1 + M$. Thus, there may have been anywhere from 1 to *Length*/*MRS* records lost. For example, if we've lost 1460 bytes, there could be anywhere from 1 to 66 records (if the MAC is MD5) and 1 to 56 records (if the MAC is SHA).

Partial Record Loss

Obviously, it's possible to lose only part of a record. With a block cipher, we can recover exactly as described above, except that we cannot verify the MAC at all on the partial record. With a stream cipher, we perform the recovery procedure described above on the next whole packet and then work backwards to figure out what the keystream for the partial packet must be.

Handshake Message Loss

In general, losing handshake messages is bad, but it's possible to recover from the loss of the Finished messages, simply by treating them as data messages and proceeding as described above. If the hello messages or the ClientKeyExchange are lost, however, we will be unable to generate the keying material required to decrypt the connection.

Preventing Packet Loss

Although we have techniques for recovering from packet loss, it's obviously desirable to prevent it instead. At the very least, the Recorder needs to be engineered with enough excess capacity to ensure that it can capture all data on the wire without getting bogged down. Obviously, multiple redundant recorders can decrease the loss rate further. However, if no loss whatsoever is acceptable, one can deploy the Recorder as a transparent proxy, thereby guaranteeing that data is recorded before it is transmitted to the communicating peers.

7.7 Client-Side Recording and Replay

We have primarily focused on recorders placed at the server's site because it's simpler for servers to deploy this sort of hardware than clients. However, as stated in Section 6.1, client-side recording is secure if client authentication is used. However, without server assistance, replay becomes impossible because the client's private key cannot be used to recover the PreMasterSecret. There are two potential fixes available. The first is to use the technique described in Section 7.5 for handling ephemeral keys to generate the PreMasterSecret. The second is to use the escrowing technique described in [15]. Escrowing has the advantage that the session can still be recovered even if the client is somehow unavailable—this seems a lot more likely for clients than servers.

8 External Recorders

Recall that if the server is not installed in trusted hardware, some proactive attacks are possible. These attacks stem from the fact that Sam controls the Recorder and can therefore control what traffic it records. In Section 5.2, we proposed fixing this problem by having the peers create secure tunnels to a the Recorder and passing the protocol messages over those tunnels. Although this solution works in theory, in practice it's too burdensome, because it requires changing the network protocols, which is one of the primary things that SSL Auditing is designed to avoid.

We can gain many of the benefit of this approach by simply installing the Recorder outside Sam's domain of control, at the ISP. If the Recorder is at the ISP, it becomes practical to do without trusted hardware on the server, because the bypassing attack of Section 5.2 is effectively blocked. Although it's of course possible for Sam to generate an arbitrary amount of traffic that bypasses the Recorder, unless he has completely subverted network routing, any traffic he transmits through the Recorder will arrive at the client. Similarly, he cannot stop traffic generated by the client from being recorded at the Recorder. Obviously, the protection here is not up to the usual cryptographic standards but should provide sufficient evidence for commercial purposes.

8.1 Split Recording

3. Actually, if the Recorder is co-located with the server, only the client needs to create a secure tunnel, because Sam knows that he's talking to the Recorder.

One difficulty with deploying recorders at the ISP is that the Recorder and its associated storage is a relatively substantial device, including mass storage and, in large installations, some form of archival storage. However, we can minimize this burden by observing that the Recorder actually serves two purposes: (1) storing the SSL traffic (2) providing proof of their authenticity. These services need not be provided together.

The solution is to have two Recorders, one at the ISP and one at the server, as shown in Figure 11. The ISP Recorder is a "stub", which simply digests the traffic and writes audit entries (but not data) to permanent storage. Because audit entries don't contain the actual message data they are quite small and the associated storage costs are low and therefore a single Recorder can be used to provide auditing for a large number of customers. The Recorder at the customer site stores the actual data.

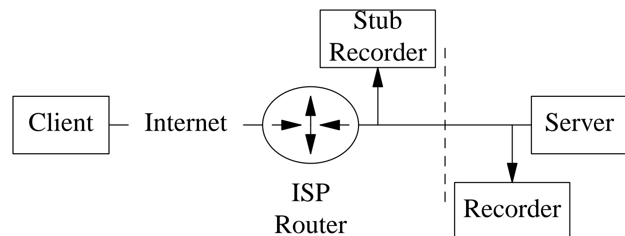


Figure 11 Split recording

In order to replay, we need to combine the two sources of data, using the timestamps from the ISP to verify the recordings made at the Server. Note that the Recorder at the customer site does not need to be tamper-hardened in this case, because the security is provided at the ISP. Only storage is being provided by the customer. Because the storage requirements are so low, an ISP could deploy a single stub recorder to protect the traffic for all its customers. Modern cryptographic hardware is easily capable of keeping up with gigabit processing speeds, though multiple recorders may be needed to ensure that all network traffic is captured.

8.2 Countermeasures for Server Fraud

As we discussed in Section 6.2, it's quite difficult to prevent proactive fraud by Sam when the client is authenticating with a symmetric credential such as a password instead of a certificate. The problem is that Sam can capture the client's credential and forge an arbitrary number of transactions with that credential.

Having the recorder situated externally makes this attack more difficult in two ways. First, it stops Sam

from mounting a bypassing attack where he uses the bypassed connection to capture the client's connection and then forges a connection of his choice. The client can always prove that Sam had an opportunity to capture his password. Second, it provides additional forensic information, such as IP address records and timing data that could be used to detect such forgeries. For instance, if Sam forges a connection from an IP address that he controls, the client can argue that he could not have originated the connection.

9 Comparison to Digital Signatures

9.1 Security

The security of digital signatures is essentially a binary proposition. If signatures are used, some measure of non-repudiation is provided (contingent upon the availability of up-to-date revocation information). By contrast, SSL Auditing offers a variety of security levels, depending on the amount of infrastructure which is put in place. At the minimum, SSL auditing offers security against reactive attacks. This security can be provided no matter what form of client authentication is used. At the maximum, when each side has key pairs and the server's key pair is in secure hardware, SSL auditing offers security guarantees which are equivalently tight to those provided by digital signatures.

9.2 Ease of Deployment

The primary advantage of SSL Auditing over digital signatures is that it is far easier to deploy, because no changes need to be made to the application layer protocols or software. In a minimal system designed to counter only reactive attack, SSL Auditing can be deployed simply by reconfiguring the network to include the Recorder. In a maximal system designed to prevent proactive attacks, installation entails replacing one of the peer's SSL implementation with one that includes hardware tamperproofing. Note that this still leaves the difficult-to-change application layer intact.

9.3 Performance

The performance cost of SSL Auditing needs to be considered along two axes, the cost to the end-systems and the cost on the Recorder. Because the end-systems just run SSL, the performance cost is essentially equivalent to that of a digital signature system, less if SSL would be used for transport security in a digital signature system.

The primary performance obstacle on the Recorder is capturing all the network traffic on a fast network. However, commercial products [16] that do this are already available and Antonelli et al. [17] have shown that such capture devices can be built economically with commodity hardware and as shown in Section 6.6. The primary additional challenge we face is performing the packet signatures and we can minimize the load on the slow HSM by aggregation techniques as described in Section 6.6. We anticipate that a single recorder will be able to handle a 100 Mbit Ethernet. Multiple recorders in a cluster can be used to scale up to gigabit speeds.

9.4 Storage Requirements

SSL Auditing and digital signature solutions both require a fair amount of storage. With SSL Auditing, the Recorder needs to retain the contents of all transactions. With digital signatures, the recipient needs to retain copies of all received messages in order to prove their contents to a third party. In general, SSL auditing requires somewhat more storage, for two reasons. First, the packet headers and time stamps need to be stored with the SSL traffic. This only amounts to perhaps 5-10% overhead.

Second, the SSL traffic cannot be compressed. Digital signature traffic can be compressed in two cases. (1) if encryption is not used. (2) if sign-then-encrypt is used and the data is decrypted before archival. Thus, we expect that digital signatures will require perhaps a quarter to a half of the space of SSL auditing, assuming compression is used. If compression is not used, the space will be roughly commensurate. Note that all the storage is in one centralized location, the Recorder. This may be good or bad depending on the situation.

10 Current Status and Future Work

We currently have a proof-of-concept version of SSL Auditing implemented entirely in software. The software runs on FreeBSD [18] and uses BPF [19] to sniff the network, storing the data on the disk of the Recorder. Similarly, we have prototypes of the Key Server and the Player, implemented as software. Many of the fine points of recording and replay discussed here were worked out in the process of building these prototypes.

The proof-of-concept demonstrates that SSL Auditing is technically possible. The next stage is to build a system which is hardened and capable of capturing the

data on a production network. Most of the required components are already available off the shelf, including programmable HSMs (we intend to use the IBM 4758 [20, 21] card) and large disk arrays for storage of the recorded data. Nevertheless, it will be necessary to assemble all the pieces, port the software to the HSM, and do enough performance tuning to ensure that we can keep up with a production network.

11 Conclusions

This paper has presented an alternative approach to dispute resolution for electronic transactions. Instead of using digital signatures, we leverage the existing SSL infrastructure by recording SSL transactions as they occur. Recorded transactions can be replayed at a later date to resolve later disputes. SSL Auditing can be easily deployed on an existing network with no change to the existing protocols. Security against reactive attacks can be provided with the introduction of a new network component to record transactions. Security against proactive attacks can be added by hardening the server's SSL implementation.

Acknowledgments

The authors would like to thank Lisa Dusseault, Brian Korver, and Terence Spies for their review of this paper.

References

[1] Freier, A.O., Karlton, P., and Kocher, P.C., *The SSL Protocol Version 3.0* (November 1996).
http://home.netscape.com/eng/ssl3/draft302.txt

[2] Dierks, T., and Allen, C., "The TLS Protocol Version 1.0," RFC 2246 (January 1999).

[3] Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., and Repka, L., "S/MIME Version 2 Message Specification," RFC 2311 (March 1998).

[4] Eastlake, D.E., Reagle, J., and Solo, D., "(Extensible Markup Language) XML-Signatue Syntax and Processing," RFC 3275 (March, 2002).

[5] Adams, C., Cain, P., Pinkas, D., and Zuccherato, R., "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)," RFC 3161 (August, 2001).

[6] Intel, *Commerce Accelerator 1000 User Guide.*

[7] SonicWall, *High Availability Options for SonicWALL SSL Devices.*
http://www.sonicwall.com/products/↵
documentation/High_Availability_SSL.pdf

[8] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "Hypertext Transfer Protocol," RFC 2616 (June 1999).

[9] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D., *"Simple Object Access Protocol (SOAP) 1.1"* (May 2000).
http://www.w3.org/TR/SOAP

[10] Rescorla, E., *ssldump.*
http://www.rtfm.com/ssldump

[11] Rivest, R., "The MD5 Message-Digest Algorithm," RFC 1321 (April 1992).

[12] National Institute of Standards and Technology (NIST), and Secure Hash Standard, FIPS PUB 180-1, U.S. Department of Commerce (May 1994).

[13] Chown, P., "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)," RFC 3268 (June, 2002.).

[14] National Institute of Standards and Technology,, "Specification for the Advanced Encryption Standard (AES)," FIPS 197 (November 2001.).

[15] Goh, E., Boneh, D., Golle, P., and Pinkas, B., "The Design and Implementation of Protocol-Based Hidden Key Recovery," manuscript.

[16] NIKSUN, *NetVCR.*
http://www.niksun.com/

[17] Antonelli, C., Coffman, K., Fields, J., and Honeyman, P., *Cryptographic wiretapping at 100 Megabits.*

[18] FreeBSD Project.
http://www.freebsd.org/

[19] McCanne, S., and Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *(USENIX) Winter,* pp. 259-270 (1993).

[20] *"IBM Cryptographic Coprocessor".*
http://www-3.ibm.com/security/crypto-cards/html/overhardware.shtml

[21] Smith, S., and Weingart, S., *Building a High-Performance, Programmable Secure Coprocessor* (October, 1998).