Design of a Secure Audit Web Service

Abstract. For web based applications, it is important that attempted accesses and transaction related information are kept in secure log files for misuse detection and system audit purposes. These log files may be an attractive target to attackers either because of the large amounts of potentially sensitive information contained in them, or in order to cover their tracks by deleting the last few records. Consequently, we would like to guarantee that in the event an attacker does gain access to the log files, we are able to detect any compromises. Optionally we also may want to ensure that an attacker can gain little or no information from the log files. Finally, we may wish to coordinate logging from multiple web services in one log file, thus we propose a secure audit web service (SAWS) which can provide an audit trail service for multiple clients. In this paper we describe a computationally simple method for generating a secure audit trail, which does not rely on a central trusted server, and can be stored on any untrusted machine. We present the design of the audit log file, that is impossible to be modified, tampered with or destroyed without detection, and its integrity can be validated by any client. Optionally, it can also be encrypted, making it impossible for unauthorised parties to read its contents. We also present some initial performance results that vindicate our design.

1. Introduction

With the widespread use of the Internet, more and more resources and services are available on the web. Web services are the new paradigm used for building Internet applications and services. To better help resource web sites improve their system security and reliability, a history log or audit trail is usually necessary to record all the accesses to the system, so that these can be later inspected either daily or periodically during system audits. In a computer system there normally exist several types of audit trails that are produced at different levels and by different applications, each devoted to a particular type of event or activity. An audit trail is especially important and necessary for access control services as it can form a significant part of the front-line defence for detecting system misuse or intrusion attempts. For a resource web site, the date, time, IP address, credentials or other activities related to a user should be recorded in an audit trail and may help to determine if a logged-in user was tying to usurp their authority, or was actually trying to masquerade as someone else, or if there was any resource misuse or attempted misuse. An audit trail is an important tool for administrators to detect, understand, and assess the damages from an attack. It is also an important tool in verifying that third parties can actually be trusted to do what you thought they would do. Web sites need to track what the users do to ensure accountability, so access control and other transactional information for all users should be kept in log files for security analysis and/or system audit purposes.

Unfortunately, since a large amount of sensitive information e.g. related to access control decisions, may be contained in the audit trail, and the audit trail may be stored on or moved to an untrusted machine, this may attract attackers to try to read or alter the log records, e.g. remove traces of their actions from it. We would like to guarantee, in the event that an attacker does gain access to this machine, that although we may be unable to stop him altering or removing the log records or the whole audit trail, nevertheless we are able to detect the compromise afterwards. Optionally and in addition, we can ensure that he will gain little or no information from the log files by encrypting them prior to storage. This secure audit trail capability is crucial for many applications in order to prevent audit trails from being tampered with undetectably. For a virtual organization or distributed application which spans multiple servers, multiple application components will produce log records that contribute to the whole system security analysis, in which case a centralised secure audit trail system may be required. In this paper, a secure audit web service (SAWS) is proposed for recording audit information from distributed applications in virtual organizations.

There were 4 main motivators for us to design a secure audit web service.

1) To provide an audit trail for access control policy decision points (PDPs) such as XACML [3], PERMIS [4], Akenti [5] in order to record all the access control requests made on the target resources and all the access control decisions returned to the policy enforcement point (PEP), so as to help detect if there were any intrusion attacks or misuse of the target resources.

2) To facilitate the implementation of the retained ADI (access control decision information) concept as described in ISO 10181-3 [6]. Retained ADI is needed to enforce such access control features as dynamic separation of duty (SoD) and conditional decision making based on prior decisions (e.g. such as limiting the amount of money that can be withdrawn from an ATM in a day).

3) To provide a web services audit function for trust and contract management in virtual organizations, as required by the TrustCom project (an EC Integrated Project in which 16 institutions and corporations are participating [15]).

4) To be a general purpose secure audit web service, so any web based client can take advantage of it.

The primary functionality of a secure audit service is to provide a permanent storage for log records that provides tamper detection. Note that it is not possible for the audit service on its own to provide tamper resistance – this is a functionality of the storage system[1] – since an attacker that gains unauthorised access to the system can always delete complete log files and log records from log files. The critical functionality of the secure audit service is to reliably detect when tampering has occurred. Schneier has developed a cryptographic mechanism for securing the contents of an audit log against unauthorised reading, and providing tamper detection [1]. The mechanism relies on symmetric encryption, with a central trusted server holding the decryption keys. Reading and verification of the log records is accomplished with the help of the central trusted server. But because of this, when the central trusted machine is not available, then it's impossible for users to

[1] One way of providing tamperproof audit logs is by replicating the logs and storing them in multiple locations, both offline and online. The discussion of how many copies to store and where to store them is outside the scope of this paper.

read and verify the audit trails – this could cause an inconvenience for users, a central point of failure, a central point of attack, and potentially a bottleneck to performance.

In this paper, we modify Schneier's scheme by using public key cryptography to enable independent reading and verification of the audit logs, without the need for the central trusted machine. We also use the recent developments in Trusted Computing Bases [16] to store the private and secret keys of the audit service, so that the external central trusted server is no longer needed for this either. The remainder of this paper is structured as follows: In section 2, the requirements for our secure audit web service are described. In Section 3, the architecture of the secure audit web service is presented. Section 4 presents the performance results of the various design options and describes the security mechanisms finally adopted in SAWS. In Section 5 we present some further performance results and a summary of this paper.

2. Security requirements for the secure audit web service

User access requests and activities should be collected and sent to SAWS, so that administrators can at a later point in time know who did what, including who was given access to which resources, and who was denied access, for example when wanting to track down an attacker. Since several applications may share the same SAWS, and it should be able to detect tampering, this leads to several basic security requirements:

- Append mode of access: Only append mode of access should be allowed, so that users or applications cannot rewind the audit file and delete or modify information that has already been stored there
- Authorised writing: Only authorised parties should be able to append log records to the audit trail. Though unauthorised applications or attackers may gain access to the audit trail and try to append fake log records to the audit trail, or modify or remove the audit trail, this should be detected by the tamper detection mechanism.
- Timestamps: Every record in the audit trail should be timestamped by SAWS to provide a trusted record of when the audit data was received. We note that if SAWS is trusted to record the audit data without tampering with it, then it should also be trusted to append the correct time to the data. Therefore we do not propose to use a secure time stamping service. If this is insufficient for some applications, then because the format of the recorded data in each log record is application-specific and is determined by the client applications themselves, they may add their own timestamp for cross checking purposes. However the use of the latter is application dependent.
- Secure communication: The communications between a SAWS client and the SAWS server should be authorised and ensure tamper resistance and data integrity.
- Secure storage on untrusted media: Since an audit trail may be stored on untrusted machines, the SAWS security mechanism should ensure persistent and resilient storage of the audit trail, and ensure detection of tampering of the audit trail – modification, deletion, insertion, truncation, or replacement. If tampering is detected, SAWS should be able to notify the security administrator.

- Support multiple simultaneous clients. SAWS should be easily and conveniently accessible via a web service interface, and it should be able to serve multiple client applications simultaneously.
- Logging efficiency: The performance of SAWS should be as efficient as possible, both in the storage size and the throughput. Since our initial target client is the XXX authorisation system, and this can make 500 access control decisions per second on a PC [19], we made 500 records per second the minimum performance requirement for SAWS on the same platform.
- Contents transparency: SAWS should be able to record any digital content coming from any SAWS client.
- Authorised reading: Since the audit trail may contain sensitive information, then the secure audit mechanism should optionally be able to ensure that only authorised applications or people have the privilege to read the audit trail.

Based on the above requirements, we propose the following architecture and security mechanisms for SAWS.

3. Architecture of the Secure Audit Web Service

There are three types of application in the SAWS system – the SAWS server, the audit trail viewing tool (VT), and the SAWS client. The core component in the SAWS server is the Java Secure Audit Trail Service (J-SATS). J-SATS is responsible for receiving log messages from SAWS clients, for securing them (as described in Section 4) and then writing the secured audit records into one or more permanent audit log files on untrusted machines. When the SAWS server and the SAWS client are communicating via the J-SATS API, they are assumed to trust each other and do not require authentication, although Java code signing can be used if needed. When they are communicating across an untrusted network, then mutual authentication is needed between them. We have decided to use digital signatures/public key cryptography for the authentication mechanism. Consequently both client and server need to be issued and configured with a public certificate/private key pair, and their certificate issuing authorities need to be recognised and trusted by each party in the scheme. This provides the strong authentication that is necessary. On top of this, the SAWS server also requires an encryption/decryption public/private key pair. We have decided to use a dual key pair rather that a single key pair for reasons that will be explained later. Optionally, the SAWS clients and the VT may also be issued with their own encryption/decryption public/private key pairs, or they may use their signing key pairs for this purpose as well. The encryption/decryption public/private key pairs are only used for optional confidential storage of the audit information. The structure of SAWS is shown in Fig. 1 (The VT is not shown in this figure).

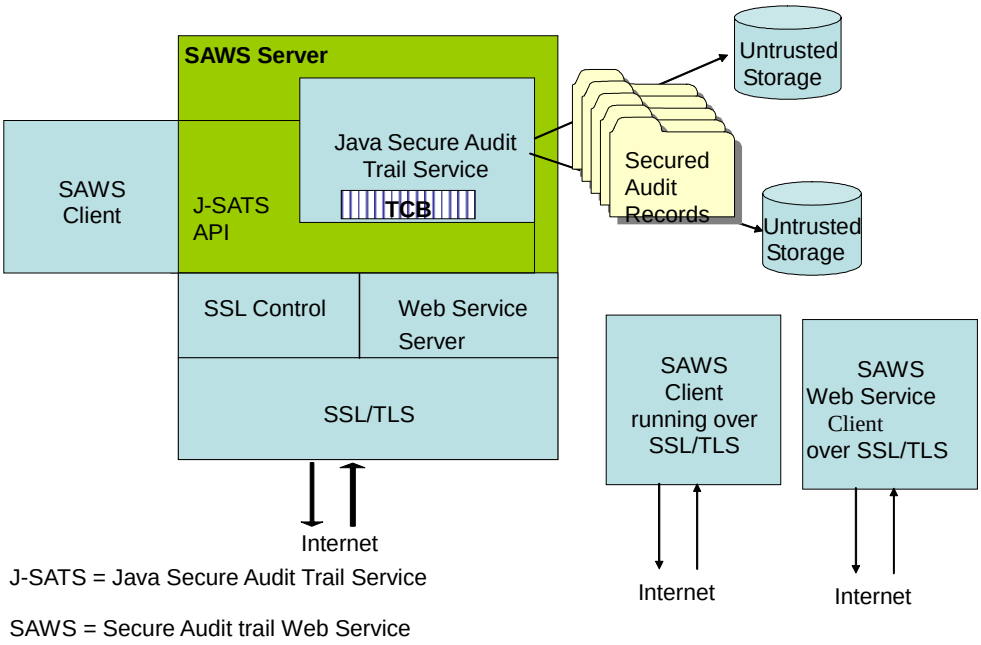

Fig. 1. Architecture of the Secure Audit Web Service

Three different client interfaces are provided for SAWS to facilitate different application scenarios:

- a Java programmable interface – the J-SATS API,
- a direct SSL-TLS socket communication interface for TCP based clients and
- a SOAP server over SSL interface for web service based clients.

We did initially intend to provide a WS-Security [22] interface as well, but as will be shown later, the performance of this only allowed a throughput of 2 records per second which was two orders of magnitude worse than our performance target.

The J-SATS API is designed for a client that is co-located on the same machine as SAWS, so that the client can call the J-SATS API directly thus reducing the communications costs and increasing the overall performance of SAWS.

The SOAP over SSL interface is provided for web services based clients that can support the SOAP protocol. With this protocol, SAWS can publish its service using the web service description language (WSDL) so that any client may connect to it through the standard SOAP over SSL interface. In this way, SAWS can support multiple clients simultaneously. Whilst web services are seen to be the service model of the latest distributed applications, and are continually increasing in popularity, nevertheless they currently suffer from a performance penalty which may be too great for some applications' audit requirements. For this reason we also provide a raw SSL-TLS socket communication interface for TCP based clients that can manage SSL-TLS communications by themselves. This interface provides higher performance than the SOAP interface (see later) but lower performance than the Java interface.

No matter which interface is used to create the audit file, they can all later be viewed and analysed by using the same audit trail viewing tool (VT).

4. Security mechanisms of SAWS

4.1 Secure communications over SSL

When the Java API is used to call the SAWS server, we assume that no authentication between the SAWS client and server is required, neither is protection of the communication channel, since the SAWS client and server are one and the same program. However, this is not the case when the client is remote from the SAWS server and is accessing it via an untrusted network such as the Internet. For this reason we need to secure the communications between the SAWS server and its remote clients. To ensure secure communications between SAWS clients and the SAWS server we need mutual authentication and message integrity. Message confidentiality may also optionally be required.

For SOAP messages there are two candidate security mechanisms: SSL[17]/TLS[18] and WS-Security [22]. We built and tested both. In our testing environment[2], for SOAP messages over SSL (using the Sun JSSE package) to transmit each 1k data block from a SOAP client to a SOAP server takes around 40 to 50ms (average 44ms), whilst to transmit each 1k data block using WS-Security (by using the Sun xws-Security package) takes around 480 to 520ms. Whilst these test results are by no means authoritative measurements of the performance of WS-Security, they are similar to previous measurements for XML security [20, 21], and roughly show that there is an order of magnitude performance difference between the two mechanisms at the moment. So for performance reasons we decided to adopt SOAP over SSL rather than WS-Security to provide secure communications between the SAWS server and clients. We appreciate that SSL only provides connection level security rather than message level security, but given that SAWS will be the only application running over SSL, the loss of security is deemed to be minimal. It does however mean that we will need an additional key pair for the SSL implementation on the SAWS server.

We also compared the performance of SOAP over Http over SSL compared to raw messages over SSL. The results are somewhat surprising: whilst raw TCP/IP outperformed SOAP over Http significantly (as expected) for each message, raw SSL underperformed SOAP over Http over SSL significantly (which was not expected) for each message. We suspected that this was something to do with the disconnection/reconnection attempts and the (none) reuse of master secrets for each SSL messages, so we repeated the raw SSL tests ensuring that the SSL link was kept open for the duration of a certain number (100 in our test) of message exchanges. In this case raw SSL

[2] IBM R40 Laptop PC, Intel Pentium CPU 1.29 GHz, 256MB of RAM; Windows XP, Service Pack2. The web service server (SOAP server) is in Tomcat 5.0; Java: Sun JDK 1.4. The messages were sent internally using TCP/IP sockets.

outperformed SOAP over SSL (as expected), and showed similar performance to raw TCP that established a new connection each time.

Table 1 Performance measurements of SOAP and socket communications with and without SSL

SOAP without SSL	100 http SOAP messages, 1k data each message	100 http SOAP messages, 5k data each message	100 http SOAP messages, 10k data each message
Average per message (ms)	33.546	33.87	36.076
time for 1k (ms)	33.546	6.774	3.6076
SOAP with SSL	100 http SOAP messages, 1k data each message	100 http SOAP messages, 5k data each message	100 http SOAP messages, 10k data each message
Average per message (ms)	43.944	47.546	50.554
time for 1k (ms)	43.944	9.5092	5.0554
Socket without SSL	100 socket (TCP/IP) messages, 1k data each message	100 socket (TCP/IP) messages, 5k data each message	100 socket (TCP/IP) messages, 10k data each message
Average per message (ms)	17.304	17.926	19.726
time for 1k (ms)	17.304	3.5852	1.9726
Socket with SSL	100 SSL socket (TCP/IP) messages, 1k data each message	100 SSL socket (TCP/IP) messages, 5k data each message	100 SSL socket (TCP/IP) messages, 10k data each message
Average per message (ms)	72.662	75.466	79.112
time for 1k (ms)	72.662	15.0932	7.9112
Socket with SSL	100 messages sent in one SSL socket (TCP/IP) session, 1k data each message	100 messages sent in one SSL socket (TCP/IP) session, 5k data each message	100 messages sent in one SSL socket (TCP/IP) session, 10k data each message
Average per message (ms)	16.906	18.646	20.33
time for 1k (ms)	16.906	3.7292	2.033

For the direct SSL-TLS socket communication interface or the SOAP over SSL interface, the SAWS server and client should both be issued and configured with a public certificate/private key pair to facilitate SSL mutual authentication, and the certificate issuing authorities should be recognised and trusted by each party.

The SSL mechanism ensures secure communications with SAWS by providing:

- Mutual authentication: Based on the public key certificates and private keys of the SSL client and server, the SAWS client and server can authenticate each other through the SSL handshake protocol;
- Data integrity: The data integrity in the communication can be ensured through the SSL record protocol based on the MAC mechanism;
- Data privacy: The data can be protected by the SSL record protocol based on symmetric encryption, so data privacy can be ensured.

4.2 Secure storage of the audit trail

To ensure secure storage of the audit trail on an untrusted machine, the following measures are adopted to store log records in the audit trail.

1) When starting a new audit record file, the SAWS writer generates a new random secret key *RN*, which is then stored securely by SAWS so that only it can recover it after a crash. *RN* is stored in the TCB and is also encrypted with the public encryption key of the SAWS server and stored as the first record of the new audit file. *RN* will be used in the calculation of the secure hash which is appended to the end of each log record, in order to detect modification of a record's contents before the audit file is finally digitally signed and closed i.e. in the case when SAWS prematurely crashes.

2) The SAWS server places the file name and digital signature of the previous audit file as the second record in the new audit file. This chains the log files together and stops an attacker from completely deleting an entire audit file without detection.

3) Each record is given a sequence number which prevents records from being inserted or deleted in the middle of the audit trail. To detect truncation of the file from the end after a crash, the current sequence number of the log record is stored in the TCB. This allows detection of a truncation attack.

4) SAWS adds the authenticated name/ID of each client to each record they submit before writing it to the audit trail. This is to stop one client masquerading as another in the data that it submits. SAWS uses its own ID for each system record that it writes.

5) SAWS keeps a plain accumulated hash of the entire audit file as the log file is built up. This is stored in the TCB (along with the current sequence number) in order to detect a replacement attack after a crash (i.e. an attacker uses an old version of the audit trail to replace the latest version of the audit trail after his intrusion into the storage system).

6) When the audit file is complete, the plain accumulated hash is written to the end of the file and the plain accumulated hash is digitally signed with the signing private key of the SAWS server. The signature and public key certificates of SAWS are also written to the file. Now anyone can verify the completeness of the file with the verifying public key of the SAWS server.

7) Optionally the audit file can be stored in several different computers in different locations to defend against truncation and replacement attacks. This will also allow recovery in cases where one or more (but not all) copies are tampered with.

8) If a confidential audit trail is required, optionally SAWS can generate a random symmetric encryption key which is encrypted using the SAWS encryption public key and the viewing tool public key and both copies are stored in the audit file. Optionally the encryption key can also be encrypted to the public keys of all the SAWS clients, so that they can each independently verify and view the audit log by using their decryption private keys. Each log record will subsequently be encrypted with this symmetric encryption key prior to storing in the audit trail.

9) After every 1 second in the idle state, the SAWS server writes a heartbeat record to the audit file, so if a system crash happens at any time we can know the time of the crash to the nearest second.

In this design, a trusted store is needed by the SAWS server to store the SAWS server decryption private key, the SAWS server signing private key, the random number *RN*, the current values of the plain accumulated hash and the record sequence number. Ideally if the SAWS server is running on a trusted computing base (TCB) then the keys, accumulated hash and record sequence number can all be encrypted and stored by the trusted platform module (TPM) in the TCB. If a TPM is not available and the SAWS server is running on an ordinary computer, then the trusted store can be a tamper-resistant piece of hardware such as a Java iButton [2] or a smart card, or a specially protected system file encrypted with a password of the security administrator. The size of the trusted store is very small (<5Kbytes). Compared with Schneier's method [1], the basis of trust is shifted from the central trusted machine in Schneier's method to SAWS's own trusted store and optionally, an external Certification Authority. This simplifies and reduces the security requirements for the trust basis, thus it can bring more convenience and flexibility to SAWS applications and SAWS administrators. Actually TPMs are increasingly becoming standard equipment for new PCs [16], so it's reasonable to assume that our SAWS server can be based on a TCB and that a trusted store of small size is available for our use.

4.3 Hashing of the audit trail records

For integrity and authenticity purposes, a secure hash and a plain accumulated hash are needed when generating the log records for the secure audit trail. The secret random number *RN* generated by the SAWS server will be used in the calculation of the secure hash.

Suppose LR_0 ,…,LR_j are the log records in the audit trail, then at any given time the secure hash for the log record LR_j will be $hash(LR_j, RN)$. Here *hash( )* is the one-way hash, using an algorithm such as SHA-1 [12]. Each secure hash is appended at the end of each log record, to form a secured log record. The secured log records are SR_0 ,…,SR_j where $SR_i = (LR_j, | hash(LR_j, RN))$. The plain accumulated hash of the whole log file will be $hash(SR_0 ,…,SR_j)$.

The current log record sequence number and accumulated hash are always stored and updated in the trusted location after writing each log record into the log file. The accumulated hash is used on recovery to check if the audit file is complete. Together with the sequence number it is also used to defend against truncation or replacement attacks. Note that when SAWS crashes it is possible that the sequence number and accumulated hash in the trusted location may be out of synchronisation with the audit file, since they are not written simultaneously as a transaction. Thus the sequence

number could be greater than, equal to, or less than the value in the audit file. If they are the same, the accumulated hash can then be used to check that the file contents have not been tampered with. If they are not the same, this does not necessarily mean that the file has been tampered with, so the secure hash per record will need to be checked in order to validate each log record separately. Only the SAWS server can verify the authenticity of the log records remaining in the log file after a crash, as only the SAWS server knows the secure random number. If the computed secure hashes are the same as those of the log records, then the SAWS server can trust that the log records remaining in the log file have not been tampered with. If the audit file contains one more record than that recorded in the sequence number in the secure store, then the file can be assumed to be intact. If it contains one less, then there is the possibility that the last audit record was deleted by an attacker. Security administrators will need to manually check the likelihood of this, for example, by comparing the time stamp in the last record with the time of crash and time of recovery.

When closing the current log file, the SAWS server saves the plain accumulated hash as a SAWS audit trail hash record. It then digitally signs the plain accumulated hash and saves the signature and certificate as a signature record at the end of the log file. Now in the future anyone can check the integrity and authenticity of this log file by checking the plain accumulated hash and the signature along with the stored certificate.

4.4 Functions of the secure hashes, the accumulated hash and the signature

In our design, the secure random number is encrypted by the SAWS encryption public key and stored in the log file, so only the SAWS server, not the auditor, is able to read the secure random number, and only the SAWS server is able to check the authenticity of each log record by checking the secure hash per record. This is only needed for recovery after a crash or abnormal termination of the SAWS server. Once a log file is complete the accumulated hash is digitally signed by the SAWS server. This ensures that the auditor (or any client) is able to verify the completeness and authenticity of the whole log file. But since the secure hashes, accumulated hash and signature of the log file all are a kind of positive proof that the log records or the whole log file are complete and have not been tampered with, once an attacker does gain control of a log file, he can easily remove or destroy any positive proofs on the log file. He may even be able to simply remove a whole log file; so after a log file is signed and closed by the SAWS server it should be backed up offline by the SAWS administrator (preferably multiple times, with copies stored in different locations and on irrevocable media such as DVDs). In the case of damage to a backed up log file, there is little point in the auditor invoking the SAWS server to check the secure hash per record to try to recover parts of the log records from the file damage, as the damage to the log file could be in any form – intentional or accidental – and in the worst case the proof itself (*RN*) could be damaged. To be able to recover from damages to backed up offline log files, one of the undamaged copies should be used. An undamaged copy can always be validated by checking its SAWS signature.

In order to have redundancy when SAWS crashes or terminates prematurely, the SAWS administrator could deploy multiple SAWS servers on multiple computers and make the SAWS clients send log messages to the multiple SAWS servers so as to get multiple copies of the (partial)

log files. Alternatively only one SAWS server could be used to receive all the log messages and this could then forward the log messages to other SAWS servers or write multiple copies in parallel. However, in our current implementation each SAWS server only creates one copy of a log file.

4.5 The use of dual key pairs

SAWS has a signing key pair and an encryption key pair. The signing key should never be available to any entity outside of SAWS, so that only it can create audit logs, whilst the encryption key pair needs to be backed up and recoverable in case of loss. The signing key is considered to be disposable, and if lost, a new key pair will be generated by SAWS on start up. Both the private keys are written to and protected by the TCB if this is available. In addition, the encryption/decryption key pair is exported in PKCS#12 format for backup by the administrator. The private signing key is never exported, but if a TCB is not available, then it is stored in (and recovered from) an encrypted file, using an administrator supplied seed password and an internal key generation algorithm. If an external certificate authority is available, SAWS will attempt to get both public keys certified using either CMC-PKCS#10 [9] or CMP [13]; otherwise self-signed certificates are created.

4.6 Log record format

While there are some log standards in place for storing log records in clear text format, such as the Extended Log File Format [23], Common Log Format [14] or Combined Log Format [14], etc., they are not appropriate for secure audit trails. These standard log formats don't contain sequence numbers or secure hashes, so they can't defend against tampering during or after the writing. Furthermore they are designed for serving specific applications for human readable purposes, so the log formats don't contain an index to the log records and don't support free format fields, so they are not able to record digital contents of any format. Since our SAWS audit trail is for application clients that may have different log contents and formats, and the audit trail is for computer processing purposes, then we normally would not expect a human to read the audit file with his naked eyes. Consequently we need to define our own log format for SAWS in order to support all digital content and high processing efficiency, and we also need to provide viewing and analysis tools to support the system audit process.

The format of a SAWS log record is shown in Fig.2. All log records in the log file use the same format. The *j*th log record LR_j is defined as:

$LR_j = Sn_j \,||UID_j \,||\, ST_j \,||\, T_j \,||\, L_{j-1} \,||\, L_j \,||Encrypt_j \,||\, M_j \,||\, H_j$

Sn_j	UID_j	ST_j	T_j	L_{j-1}	L_j	$Encrypt_j$	M_j	H_j

Fig.2 Format of a log record in the secure audit trail

(1) Sn_j is the sequence number (4 bytes).

(2) UID_j is the User ID of this record; it indicates the identity of the client that provided this log record. Every SAWS client is assigned a unique user ID after authentication, and this mapping is held in the **SysClientID** record (see later). The UID_j = 0x00 is reserved, and indicates that the log record is written by the SAWS server itself (1 byte).
(3) ST_j is the log record type (1 byte).
(4) T_j is the timestamp of the *j*th log record in the format of a long integer (8 bytes).
(5) L_{j-1} is the length of the last log record LR_{j-1} (4 bytes).
(6) L_j is the length of the current log record LR_j (4 bytes).
(7) $Encrypt_j$ is the encryption indicator for the encryption method used for this log message. It could be **SymmetricEncryption** (0x01), **AsymmetricEncryptionForSAWS** (0x02), **AsymmetricEncryptionForVT** (0x03), **AsymmetricEncryptionForClient** (0x04), or **NoEncryption** (0x00) (1 byte).
(8) M_j is the *j*th log message received from the SAWS client or from the SAWS server itself (indefinite bytes).
(9) H_j is the secure hash value for this record (20 bytes).
(10) || represents the concatenation operation.
Generally there are the following types of log records in the audit trail:

- **ClientLogData**

This record type is for holding the clients' log messages, and the UID_j field holds the user ID of the actual client who sent this log message to the SAWS server. For those applications that need a confidential audit log, they can either send encrypted data to SAWS, or SAWS can be configured to encrypt client log messages with the symmetric encryption key K_s generated by the SAWS server (or both).

- **SAWSRandomNumber**

Every time SAWS starts up, it generates a secret random number, and this secret random number is used in the secure hashing calculation for each log record. This random number is encrypted by the SAWS encryption public key, so the SAWS server can read this log record at a later time. When ST_j = **SAWSRandomNumber**, $Encrypt_j$ = **AsymmetricEncryptionForSAWS**, $M_j = PKE_{PKs}(RN)$, where *PKs* is the SAWS encryption public key, $PKE_{PKs}( )$ is the public-key encryption, under the encryption public key *PKs*, using an algorithm such as RSA [7].

- **SymmetricEncryptionKey**

A symmetric encryption key may optionally be generated by the SAWS server when the SAWS server starts up, if record level encryption is required as directed by the configuration parameter "RecordSymmetricEncryption". This symmetric key is encrypted by the audit trail viewing tool's encryption public key and the SAWS encryption public key respectively so that both VT and SAWS are able to retrieve this symmetric key at a later time. Optionally some clients may also be empowered to retrieve this symmetric key, as directed by the configuration file. The encrypted key is thus stored in two or more log records. Assuming that K_s is the symmetric encryption key, then when ST_j = **SymmetricEncryptionKey** and $Encrypt_j$ = **AsymmetricEncryptionForVT**, this record holds the encrypted key $M_j = PKE_{PKvt}(K_s)$ for VT, and when ST_j = **SymmetricEncryptionKey** and $Encrypt_j$ = **AsymmetricEncryptionForSAWS,** this record holds

the encrypted key $M_j = PKE_{PKs}(K_s)$ for SAWS, where *PKvt* is the VT encryption public key, and *PKs* is the SAWS encryption public key. When ST_j = **SymmetricEncryptionKey** and $Encrypt_j$ = **AsymmetricEncryptionForClient,** this record holds the encrypted key for a client, along with the client's corresponding public key certificate.

- **SAWSLastFile**

This record is for chaining log files together. It contains the file name, the plain accumulated hash and signature of the previous log file. Every time a new log file is generated, this log record will be generated and stored in the new log file to ensure the consecutive links between log files. It is also protected by the SAWS encryption public key.

- **SAWSAccHash**

This is the SAWS plain accumulated hash for validating the whole log file. Before SAWS closes the current audit trail it saves the plain accumulated hash of it in this record of the log file. The **SAWSAccHash** record is always the penultimate record of a SAWS audit file.

- **SignatureRecord**

This record is for holding the digital signature of the SAWS server on the plain accumulated hash. The SAWS server signs the plain accumulated hash by using the SAWS signing private key and saves the signature as a SAWS signature record at the end of this audit trail file. When ST_j = **SignatureRecord**, $M_j = SIGN_{SKs}(AccumulatedHash)$, where *SKs* is the SAWS signing private key, and $SIGN_{SKs}( )$ is the digital signature using an algorithm such as RSA or DSA [8]. With this signature record, any application can verify the authenticity and integrity of this log file, using the accompanying SAWSCert record. The **SignatureRecord** is always the last record of a SAWS audit file.

- **SAWSCert**

This record is for holding the X.509 public key certificate of the SAWS server and is used for validating the signature on the audit file, so that the log file can be self-validated at a later time. This is usually the third record from the end of a complete audit file.

Apart from the above log records, which are used for the log file protection purposes, there are also some system log records associated with the SAWS server itself. These records include:

- **SysSAWSStartup**: For holding the SAWS server startup time;
- **SysSAWSShutdown**: For holding the SAWS server shutdown time;
- **SysHeartbeat**: This record is written after every 1 second in the idle state, so that if a system crash happens at any time we can know the time of the crash to the nearest second.
- **SysUnauthorisedConnectionAttempt**: For recording unauthorised connection attempt to the SAWS server.
- **SysAuditorNotification**: For recording notifications sent to auditors when an event happens (such as unauthorised connection attempts).
- **SysClientID**: For recording the identities of successfully authenticated SAWS clients. This record will contain the distinguished name of the SAWS client, taken from its public key certificate, and the client ID that is used to identify its records in the log file.

With all the log records sharing the same format, all the records can be concatenated together to form a whole log file. Each record in the log file then can be retrieved one after another according to L_{j-1} and L_j in each record – the length of the previous and current log record, in the direction from head to tail or from tail to head of the log file.

4.7 Initialisation of the secure audit trail

When the SAWS server first starts up it generates the two asymmetric key pairs and both private keys are written to the TCB if this is available. In addition, the encryption/decryption key pair is exported in PKCS#12 format for backup by the administrator. If a TCB is not available, the signing key is stored in (and recovered from) an encrypted file, using an administrator supplied seed password. The public key certificates are created.

Next SAWS generates a secret random number and encrypts it using the SAWS encryption public key, and saves it to the **SAWSRandomNumber** record, so in the future, on recovery, SAWS will be able to retrieve it. If a configuration parameter requires it, SAWS will optionally generate a symmetric encryption key and encrypt it with the audit trail viewing tool's public key, SAWS' public key, and optionally the clients' public keys, and save these in the **SymmetricEncryptionKey** records. The administrator is prompted for the name of previous log file, SAWS opens this, validates it by checking its digital signature, then stores the file name, the plain accumulated hash and signature of the previous log file in the **SAWSLastFile** record.

After initialisation, the SAWS server can then receive SAWS client log messages, calculate the secure hashes and the accumulated hash, optionally encrypt the log records, and save them as **ClientLogData** records in the log file.

4.8 Restarting of the SAWS server

Every time the SAWS server restarts, it needs to first perform validation of the current log file. This entails the following operations:

- Recompute the plain accumulated hash of the whole log file and check if the signature in the log file is present and correct. If it is, this validates the entire log file.
- SAWS then confirms that the last sequence number of the log file is equal to that stored in the TCB. This can prevent a replacement attack.

If the signature validates, then SAWS will start a new log file as described above. If any error is found, or the signature is absent, this means that SAWS either terminated prematurely or the system crashed. In this case SAWS will notify the administrators of the problem and will need to recover the incomplete or invalid log file.

4.9 Validation of the log file

Validation of a complete log file by any application is straightforward. Any application can open the log file, recompute the accumulated hash of the whole log file, and check the signature on the

log file. If several log files exist, then the log files can be checked one after another individually from the last one to the previous one, and the **SAWSLastFile** records in the log files can be checked to ensure that all the log files are correctly linked together.

When VT needs to read any encrypted client log records, it can decrypt the **SymmetricEncryptionKey** record in the log file to get the symmetric encryption key by using the VT private encryption key, then use this symmetric encryption key to decrypt the client log records.

4.10 Recovery of an incomplete or corrupted log file

SAWS retrieves the secure random number from the log file, checks the secure hash on each record and checks the sequence numbers. SAWS displays an error message for each record whose secure hash does not validate correctly, and for each pair of records whose sequence numbers are not sequential. SAWS also displays the details of the last audit file that is recorded in the current log. The administrator can determine whether to subsequently validate this or not, depending upon the outcome of the current process. SAWS recomputes the accumulated hash of the log file, and compares this and the last sequence number in the log file with those stored in the TCB – if they are the same, SAWS can be sure that the log records in the current log file are authentic and complete, in which case, SAWS adds the certificate record to the audit file, writes the plain accumulated hash to the file, digitally signs it, writes the signature record at the end of the file and closes it. It then start a new log file as above. If a log file is found to have been tampered with, the administrator is informed, SAWS inserts a **SysAuditorNotification** record, and can then be directed to sign and close the file, and start a new one.

5. Performance and Conclusions

5.1 Log file performance data

By appending a secure hash to a log record, the size of each log record will increase by 20 bytes. The sequence number, timestamp, client identity, and other auxiliary fields can add up to another 22 bytes, so the log file writing performance and the size of storage could be adversely affected. The size of the storage is not a problem nowadays since large volume storage is common and cheap. To make sure that the log file writing performance is not significantly affected by adding the hashes, sequence numbers, timestamps and client identities, etc., we performed a simple test with Java in a Windows XP environment. The results showed that appending a 1k log record into the log file only cost 0.015ms, while recording a 10k log record cost 0.047ms – i.e. increasing the record size 10 times reduced performance by only about 3 or 4 times. The testing also showed that the performance did not differ noticeably when altering the number of records after which a file writing flush operation is forced, nor did it differ if rewinding operations on the file are performed or not. One can see that the time needed to write log records into a log file is approximately two orders of magnitude faster than the time needed for a SAWS client to send a log record message across the

network. We can conclude that since storage is cheap and fast, we don't need to worry too much about either the audit trail writing performance or the addition of system data to each log record.

We also did testing on the verification performance of the log file. The preliminary results show that to verify the accumulated hash and the signature of a log file with 10^9 records (1 billion records) of 100 bytes each, takes around 1.7 hours; to verify the accumulated hash and the signature of a log file with 10^9 records of 1000 bytes each, takes around 9.1 hours. If we write 500 log records every second of every day (our performance target), it will generate 43 million records per day, or 1.3 billion records each month. Since the verification of the log file is a very time consuming process (18 minutes for a daily log of 43 million records of 1K bytes each), we expect that before the log file is backed up to irrevocable media, checking the signature on the log file will play an important role in ensuring the authenticity and completeness of the log file. Once the log file is written to irrevocable media, organisational security measures will play a more important role in ensuring the authenticity and completeness of the log file, though verifying the signature on a log file can still be applied whenever necessary.

5.1 Conclusions

In this paper we have presented a secure audit web service (SAWS) that can provide secure audit trail services to multiple distributed applications. SAWS is able to receive and save client log messages in a secure audit trail file, which can be stored on an untrusted machine. Any party can subsequently verify its authenticity and optionally only authorised parties can read it. Any type of tampering with the secure audit file, such as modification, deletion, insertion, truncation, replacement or unauthorised appending, can be detected. In addition, all the audit trail files are chained together in order to detect the loss of a complete file. Whilst SAWS does not directly prevent intrusion or misuse of web resources, nevertheless it can aid the detection of intrusions or misuses by providing tamper resistant evidence of them after the fact. SAWS combined with a trusted computing base, or other physical tamper-resistant hardware can form the basis for highly trusted auditing capabilities. SAWS is designed to be general purpose, and any application can make use of its services for logging and audit purposes.

6. Acknowledgements

The authors would like to thank both UK JISC for partially funding this work under the DyCom project and the EC for partially funding this work under the TrustCoM project.

References

1. Bruce Schneier, John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, May 1999, vol.2, no.2, pp.159-176.

2. Cheun N Chong, Zhonghong Peng, Pieter H Hartel. Secure audit logging with tamper-resistant hardware. (http://www.ibutton.com) *18th IFIP TC11 Int. Conf. on Information Security, Security and Privacy in the Age of Uncertainty (SEC)*, D. Gritzalis and S. De Capitani di Vimercati and P. Samarati and S. K. Katsikas (eds.) , published by Kluwer Academic Publishers, Boston, held in Athens, Greece, May. , 2003, pp. 73-84.

3. "OASIS eXtensible Access Control Markup Language (XACML)" v2.0, 6 Dec 2004, available from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

4. D. W. Chadwick, A. Otenko, E. Ball. "Role-based access control with X.509 attribute certificates", *IEEE Internet Computing*, March-April 2003, pp. 62-69

5. Johnston, W., Mudumbai, S., Thompson, M. "Authorization and Attribute Certificates for Widely Distributed Access Control," IEEE 7th Int Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE), Stanford, CA. June, 1998. Page(s): 340 -345 (see also http://www-itg.lbl.gov/security/Akenti/)

6. ITU-T Recommendation X.812 (1996) | ISO/IEC 10181-3:1996, *Information technology – Open systems interconnection – Security frameworks for open systems: Access control framework*.

7. Rivest, R., Shamir, A., and Adelman, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2(Feb.), 120-126.

8. NIST, 1994. NIST FIPS PUB 186, Digital Signature Standard. U.S. Department of Commerce.

9. M. Myers, X. Liu, J. Schaad, J. Weinstein. "Certificate Management Messages over CMS". RFC 2797, April 2000.

10. Lai, X., Massey, J., and Murphy, S. 1991. Markov ciphers and differential cryptanalysis. In Advances in Cryptology (CRYPTO' 91). Springer-Verlag, New York, NY, 17-38.

11. Schneier, B. 1994. Description of a new variable-length key, 64-bit block cipher (blowfish): Fast software encryption. In Proceedings of the Cambridge Security Workshop. Springer-Verlag, New York, NY, 191-204.

12. NIST FIPS PUB 180, Secure Hash Standard. U.S. Department of Commerce. 1993

13. Adams, C., Farrell, S., "Internet X.509 Public Key Infrastructure Certificate Management Protocols," RFC 2510, March1999

14. Apache. "Log files". Available from http://httpd.apache.org/docs/logs.html

15. http://www.eu-trustcom.com

16. https://www.trustedcomputinggroup.org/

17 A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 Protocol", Netscape Communications Corp., Nov 18, 1996.

18. Dierks, T., Allen, C. "The TLS Protocol Version 1.0", RFC 2246, January 1999.

19. Reference withheld

20. Mundy, D. and Chadwick, D.W., "An XML Alternative for Performance and Security: ASN.1", IEEE IT Professional, Vol 6., No.1, Jan 2004, pp30-36

21. P. Sandoz, S. Pericas-Geertsen, K. Kawaguchi, M. Hadley, and E. Pelegri-Llopart. "Fast Web Services", Aug 2003, Available from: http://java.sun.com/developer/technicalArticles/WebServices/fastWS/

22. OASIS. "Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)". OASIS Standard 200401, March 2004

23. W3C. "Entended Log File Format". W3C Working Draft *WD-logfile-960323*. Available from http://www.w3.org/TR/WD-logfile.html