ABLS - An Attribute Based Logging System for the Cloud

Christopher A. Wood
Department of Computer Science
caw4567@rit.edu

ABSTRACT

User-based non-repudiation is an increasingly important property of cloud-based applica-tions. It provides irrefutable evidence that ties system behavior to specific users, thus enabling strict enforcement of organizational security policies. System logs are typically used as the basis for this property. Thus, the effectiveness of system audits based on log files reduces to the problem of maintaining the integrity and confidentiality of log files. In this project, we study the problem of building secure log files. We investigate the benefits of ciphertext-policy attribute-based encryption (CP-ABE) to solve a variety of log design issues. In addition, we also present the architecture and a preliminary analysis for a proof-of-concept system that fulfills the confidentiality and integrity requirements for a secure log.

1. INTRODUCTION

User-based non-repudiation is a system security property that provides indisputable evidence that links specific actions to individual users (or entities) that trigger such actions. Cryptographically speaking, non-repudiation requires that the integrity and origin of all data should be provable. In essence, this enables system audits to be conducted that can identify data misuse (and thus, potential security policy violations) by comparing the sources of system events with all entities authorized to invoke these events. Therefore, treating non-repudiation as a required system quality attribute in the architecture is likely to become a common trend in the commercial, government, and even more specifically, the health-care domain.

System audits typically use log files to determine the cause and effect of events that took place during the system's lifetime. In order to provide accurate information for non-repudiation purposes, it is often necessary to place some amount user-sensitive data in these log files that can be used to trace data back to its origin. As such, logs of events generated by a client that is being served must maintain data confidentiality and integrity should the system be compromised. These goals are commonly achieved using a combination of encryption and signature techniques [2]. However, traditional approaches to encryption and signature generation and verification are becoming less effective in the context of cloud applications. Furthermore, naive approaches to log security that are based on tamper-resistant hardware and maintaining continuous secure communication channels between a log aggregator and end user are no longer useful in the context of cloud-based applications [3].

Symmetric-key and public-key encryption of log entries are very common confidentiality techniques proposed in the literature. However, in cloud-based applications, these schemes are becoming less useful. There is a need for a robust access control mechanism that enables dynamic user addition and revocation with minimal overhead (i.e. re-encrypting a subset of the log database should be avoided). Both symmetric- and public-key cryptosystems lack in that access policies must be tied directly to keys used for encryption and decryption. If the access policy for a set of log messages needs to be changed, then both the keys used to encrypt and decrypt such log entries will need to be regenerated and distributed, and the entries must also be re-encrypted. Both of these tasks can be very expensive.

In addition, symmetric-key cryptosystems require keys to be shared among users who need access to the same set of logs, which requires a secure and comprehensive key management and distribution policy. From a storage perspective, public-key cryptosystems (e.g. RSA and ElGamal) suffer from the extra data transfer and storage requirements for large cryptographic keys and certificates. There may be insufficient resources to maintain a public-key infrastructure (PKI) for managing keys and digital certificates for all users.

In terms of log file integrity, aggregate signature schemes that support forward secrecy through the use of symmetric- and public-key cryptosystems are also becoming outdated [4]. Symmetric-key schemes may promote high computational efficiency for signature generation, but they do not directly support public verifiability for administrators and auditors. This means that robust key distribution schemes or the introduction of a trusted third party (TTP) are needed to ensure all required parties can access the necessary log information. Such schemes also suffer from high storage requirements and communication overhead. Public-key schemes have similar issues, as the increased key size leads to even larger storage requirements and less computational efficiency.

Collectively, we see that a balance between encryption and signature generation and verification performance is needed to support the unique scalability and resource usage requirements for cloud-based applications. Attribute-based encryption (ABE), a new cryptographic scheme that uses user attributes (or roles, in certain circumstances) to maintain the confidentiality of user-sensitive data, has an appealing application to logging systems maintained in the cloud and is capable of satisfying the aforementioned confidentiality requirements. In addition, authenticated hash-chains have been shown to be effective at enforcing log file integrity in numerous logging schemes [3].

ABLS, an attribute-based logging system that supports ciphertext-policy attribute-based encryption (CP-ABE) [1] and authenticated hash-chain constructions for log file confidentiality and integrity, respectively, was recently designed and implemented by the primary author. The preliminary ABLS architecture and design was weak with regards to the scalability of encryption operations under heavy traffic loads, the relational database schema to store log-data and other sensitive information, and the interactions with database servers. To address these issues, we propose a set of extensions that modify ABLS in the following ways:

1. The CP-ABE encryption scheme will be replaced with a hybrid cryptosystem in which the Advanced Encryption Standard (AES), the standardized symmetric-key encryption algorithm, is used to encrypt user session data by a key that is protected with CP-ABE encryption. In the event that a user generates log messages with varying sensitivity levels during a single session, multiple symmetric keys will be produced to maintain the confidentiality of information in different security classes.

2. The relational database storage scheme will either be changed to include a masking column in the appropriate log table to hide user identities or replaced entirely with a document- or object-based database. Both of these modern database systems are similar in semantics (if documents are conceptually treated as serialized objects), so a product and literature survey will be conducted prior to selecting an adequate replacement that satisfies the necessary security and performance requirements. Though, based on preferences and architectural experience, a document-based database such as MongoDB will be the likely candidate. Also, since the ABLS architecture is highly structured around a relational schema to store data, this change will require modifications made in the logging, attribute authority, and auditing modules in order to maintain functional correctness.

3. Database security mechanisms, including fine-grained access policies for protected databases and encryption of data-in-transit, will be implemented. This is particularly important for the databases that store cryptographic keys. Also, since preference is given to document-based databases such as MongoDB, third-party services such as zNcrypt (provided by Gazzang) that are specifically tailored to this DBMS will likely be used to enforce access control to sensitive databases.

4. The auditing module will be extended to include automated audits of database operations, including changes to the database structure and both successful and unsuccessful client connections with the database. Currently, the auditing module is only designed (and partially implemented) to support strategy-based audits on database contents. However, in this type of application, database performance and security are just as critical. Thus, with this change, there will be two audit techniques that can roughly be equated to data and policy inspections, both of which will only be accessible to users with the appropriate privileges. This access control will be likely be enforced using username and password credentials.

Altogether, the security features of database authentication, encryption, and auditing will be further expanded upon in the existing ABLS architecture. With these changes, we will then re-evaluate the ABLS at an architectural, security, and performance perspective to determine its usefulness in cloud-based settings.

2. LOG DESIGN

In this section we present our log generation and verification schemes. They are influenced by past work done by Schneier et al [3], Bellare et al. [**?**], Ma et al. [2], and Yavuz et al. [4].

2.1 Log Construction

Log integrity is achieved through hash chains and message-authentication codes. Each log entry is a five-tuple element that contains the generating source information, the encrypted payload of the entry, a hash digest that provides a link between the current and previous hash chain entries, and an authentication tag for this digest. In the proof-of-concept system implementation, Keccak is used as the standalone hash function H and the $HMAC$ function is built using SHA-512. Formally, each log entry L_i is built using the following protocol (as depicted in Figure 1):

$$\begin{aligned} X_i =& H(X_{i-1}, E_{SK}(D_i)) \\ Y_i =& HMAC_{EpK_j}(X_i, Ep_j) \\ L_i =& (U_{ID}, S_{ID}, E_{SK}(D_i), X_i, Y_i) \end{aligned}$$

In this scheme the X_i elements are used to link together consecutive entries in the hash chain. Similarly, the Y_i elements are used to provide authentication for the X_i element using an authentication tag that is computed from X_i and the previous epoch digest Ep_{j-1}.

In this context, an epoch Ep_j simply corresponds to a fixed-size set of log entries that are being processed (i.e. an epoch window). For example, if the epoch size is n entries, then the log generation scheme will cycle after n log entries have been constructed and begin working on a new set of log entries. After a cycle is completed, a context block for the most recent epoch is created and inserted into an epoch chain (similar to the log chain). These log generation cycles create frames (or windows) in the entire log chain in which the scheme generates log entries using a single epoch block and key Ep_j and EpK_j, respectively. More specifically, Y_i is the authentication tag that is built using only the entries within the current epoch window.

Context blocks for epoch windows are stored in the same way as log chains. Each epoch chain entry Ep_j is built as follows:

$$\begin{aligned} Ep_j =& HMAC_{EpK_j}(Ep_{j-1}, L_l) \\ Ep_0 =& HMAC_{EpK_0}(0) \end{aligned}$$

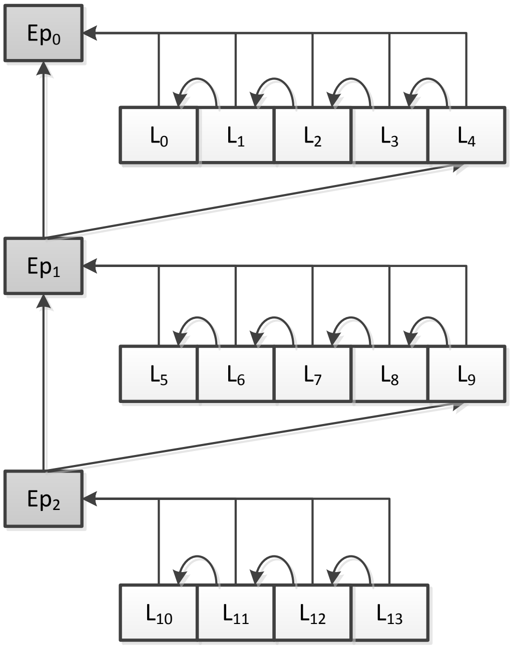

Figure 1: A visual depiction of the hash chain construction scheme. In this case, the epoch window is 5 log entries, as shown by the epoch cycle after 5 consecutive log entries.

In this context, L_l is the last log chain entry for the previous epoch Ep_{j-1}. Thus, each epoch chain entry maintains the integrity of the log chain at each epoch cycle by linking the most recent log chain entry to the previous epoch context block. The key EpK_j that is used to compute the Y_i authentication tag is based on the current epoch Ep_j, and only evolves when the epoch window cycles. This update is done with a pseudorandom function H (which, in our case, is simply the Keccak hash function), and is defined as follows:

$$EpK_{j+1} = H(EpK_j)$$

The initial epoch key EpK_0 is a secret that is initialized when a session is started. Corruption of this key can enable a determined attacker to reconstruct the log chain and epoch chain without detection. Without this information, however, such modifications are always detectable. We refer to Section **??** for a more detailed description of this issue.

Finally, as a third layer of integrity, a single digest for the entire log T_i chain is stored as the log chain is iteratively constructed. Formally, T_i is built as follows:

$$\begin{aligned} T_i =& HMAC_{T_{K_i}}(L_i, T_{i-1}) \\ T_0 =& HMAC_{T_{K_0}}(L_i, 1) \end{aligned}$$

The secret key T_{K_0} for the entire log chain is another secret that is initialized when the session is started. Similar to the epoch key, it is evolved with a pseudorandom function H as follows:

$$T_{K_{i+1}} = H(T_{K_i})$$

A visual representation of this protocol is shown in Figure 2. It is important to note that a chain of T_i elements is not maintained. Instead, only the most recent element is persisted to the database. This is critical to prevent truncation attacks.

2.2 Log Generation Rationale

The construction of each log entry satisfies the following properties:

1. Each log entry payload is encrypted and only viewable by those with the appropriate attributes.
2. The integrity of the log chain is ensured through the links generated by X_i elements, which are verifiable by the Y_i authentication tags.
3. The integrity of the entire log chain is guaranteed with the T_i element. This protects the log chain against truncation attacks.
4. The epoch-based log generation enables the logger to control the frequency of data sent to the log server, which helps with load balancing and protects against wiretapping attacks.
5. The epoch window is assumed to be a fixed (constant) size, and thus it requires $\mathcal{O}(1)$ time to verify. The log chain can be verified in $\mathcal{O}(n)$ time, where n is the length of the log chain. These results are discussed in Section **??**.
6. There are three different modes of verification that can be performed (weak, normal, and strongest). These modes are described in Section **??**.

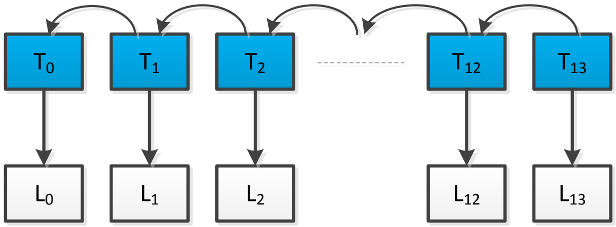

Figure 2: A visual depiction of the protocol used to build the log chain authentication tag.

7. The log is searchable by the user identity and session numbers, which leads to very fine-grained database queries.

3. IMPLEMENTATION

In order to test both the correctness and performance of ABLS, a proof-of-concept system has been developed. In this section we provide the design and rationale for the prototype architecture.

3.1 Deployment

ABLS is designed to be a centralized logging system backed by a set of distributed databases. A context diagram for the ABLS deployment scheme is shown in Figure **??**, and a more detailed depiction of the flow of data between application and database servers is shown in Figure **??**.

Based on the purpose of each piece of database used in the log, it is best to physically separate databases that store data of different security classes rather than rely on MAC with polyinstantiation. Of course, access control and authentication mechanisms for all database servers is enforced at the operating system level, thus prohibiting immediate access to all unauthorized users other than the `LoggingModule` and `PolicyEngine` processes.

In the current ABLS prototype, all database servers are separated as individual SQLite database files. Once the deployment platform is selected and properly configured, these will be replaced with instances of MySQL databases running on separate servers.

3.2 Key Management

In order to increase the performance of log entry encryption during a given user session, symmetric-key encryption using AES-256 is chosen to encrypt all log entry messages. The symmetric key is then encrypted using the CP-ABE scheme. This design enhancement enables increased improvement without sacrificing the level of confidentiality granularity that is needed for each log entry.

The basic algorithm for encrypting a log entry is shown in Algorithm 1. Once encrypted, the cipher text is stored as specified in Section 2.1.

Algorithm 1 Log entry encryption

```
Require: An unencrypted log entry $L_i$ for session $S_j$ of user $U_k$
Let $P$ be the access control policy for the message of $L_i$, as determined by the PolicyManager
if The symmetric key $K$ for $(U_k, S_j)$ has not been generated for $P$ then
    Generate $K$ and encrypt it (with the CP-ABE encryption module) using $P$
    Persist $K$ to the key database
Encrypt $L_i$ with AES-256 using $K$, yielding $E(L_i, K)$
Persist $(L_i, K)$ to the log database
```

3.3 Database Design

As shown in section 3.1, there are four main databases that must be maintained by ABLS: the log, key, user, and policy database. The log database maintains the all information in the log chain for every single user and session pair. The key database stores the cryptographic keys that were used to construct such log chains. The user and policy databases store user information and policy rules for ABLS, respectively. In the current prototype of ABLS, the policy database is not used externally. All event rules are hard-coded into the `PolicyEngine`.

In order to link the entries in the log tables to their corresponding verification and encryption keys in the key database, common user and session IDs are used (though not as the primary key for the tables since they do not satisfy the uniqueness property). However, because such the storage of such user and session information in plaintext may lead to violation of user privacy, they are deterministically masked before being stored in the database.

This masking procedure works by encrypting the user and session attributes with a symmetric key generated by the logger's master key salted by the user's unique identifier. Specifically, the encrypted user and session IDs, $[U_i]$ and $[S_j]$, are generated as follows.

$$[U_i] = E(M_k || H(U_i), U_i)$$
$$[S_j] = E(M_k || H(U_i), S_j)$$

TODO: discuss how the salt changes for each table so attackers can't link things together...

4. COMPLETED WORK

For Phase 1 of the project, I focused mainly on fixing some of the design and implementation flaws in my previous prototype, redesigning the database to support the new symmetric key management scheme and data partitioning for different security classes, specifying the deployment scheme, and improving the quality test of my internal unit tests and the functionality of the test driver program. The specifics of each of these improvements is outlined in the following sections.

4.1 Symmetric Key Management

As described in Section 3.2, the process of encryption log messages was changed to use symmetric-key encryption instead of CP-ABE due to the added performance improvement. Pairing-based cryptography is computationally expensive, and under the assumption that ABLS might be subject to very heavy traffic loads at any particular time, the overhead of encrypting data to be stored in the database should be as minimal as possible. Therefore, each unique policy that is needed to encrypt a log message is associated with a symmetric key, which is in turn encrypted and serialized using CP-ABE. The Charm crypto package allows all cryptographic objects (which tend to be nested Python dictionaries and other complicated data structures) to be serialized to byte representations for database persistence.

Also, in order to improve the performance of the logger, the per-policy symmetric keys for a user session are kept in memory until the session has been closed. This avoids the need for the logger to query the database for the key when a new piece of data to be inserted into the log.

4.2 Relational Database Design

The relational database design was modified in order to support the new symmetric key management scheme and provide enhanced security for the stored data. In particular, rather than storing the cryptographic keys in the same database as the log information, these two databases are now segregated and are expected to be deployed on different database servers. By hardening both databases, a malicious attacker would need to circumvent the access control mechanisms protecting two physically disjoint databases, rather than just one.

In addition, to better support audits that use the log database, a timestamp field was added to each table as a required attribute. Not only does such information capture the exact timing of critical system events, it acts as a protection mechanism in the event that log messages are inserted into the database out of order. Also, it is important to note that each and every timestamp for a log message is generated as soon as the `ClientHandler` reads the data from its open socket. This is done to provide the most accurate timing information.

4.3 Database Column Masking

The initial implementation of the database masking procedure, which is outlined in Section 3.3 was started in this phase of the project. Currently, only the ability for the logger to store the encrypted user and session IDs is supported. The verifiers have yet to be updated to use this encrypted data to perform the strong verification check. This will finished in the next phase of the project.

4.4 Bootstrapping, Test Enhancement, and Deployment

In order to streamline the test and deployment phases of development, a bootstrap script was implemented to configure new (empty) versions of the local SQLite databases. The main executable script (main.py) was also modified to support debug and production modes of operation, in which the debug mode clears the contents of every database and then proceeds to insert false user data into the users table to begin the logging process. The test driver program, TrafficProxyDriver.py, is then modified accordingly to use the default data contained within the database. With these changes, the typical process to start and interact with the ABLS system is as follows.

1. Run the bootstrap script to create new versions of the local SQLite databases. In the actual deployment of an ABLS system, this script would connect to the remote databases and re-specify their schemas accordingly. Thus, it is meant only for development purposes and should not be used on a live ABLS instance.

2. Run the main executable file (main.py) with the -c (clear) to flag to enable debug mode, in which the databases are wiped clean and replaced with a small set of fake data. If you wish to start the ABLS system at this point in time, you may also add the -s (start) flag to the script so that it starts the ABLS instance.

3. Run the test driver program (TrafficProxyDriver.py) and point it to the host and port at which the traffic proxy within the ABLS instance is listening. By default, this is "localhost" and port 9998, but this can easily be configured in the source code.

Aside for the initialization code, the test driver program now includes a more robust suite of tests to simulate varying traffic loads. The user interface of this program was also improved so as to aid the developers in interacting with the ABLS prototype at runtime. Given the difficulty of testing this distributed system at runtime, creating a more sophisticated test driver was crucial to the development process that enabled smoke tests to be run with minimal effort.

However, despite the increased complexity of the test driver, it does not, nor will it ever, support the ability to acquire diagnostic information from the ABLS runtime. This information is logged by the ABLS runtime to the appropriate log file, and the administrators for the ABLS system can check this information at their discretion.

4.5 Literature Survey

A comprehensive literature survey on related logging and non-repudiation work was also completed during this phase of the project. The corresponding articles that were read are included in this phase's submission package. The digest of this work is expected to be a part of the final report and potential publication.

5. FUTURE WORK

For the remainder of the project I plan to finish up the partially completed work introduced in the previous section and focus on the auditing aspect of ABLS. Auditing will be supported both manually and automatically through a robust log query interface, which places the ABLS runtime in the role of the reference monitor, and automated strategy-driven tasks run within the context of the ABLS runtime, respectively. The following sections expand upon all of these action items in more detail.

5.1 Complete Database Masking

Following the design outlined in Section 3.3, the database masking scheme will be completed by adding support to the database verifiers. Since the verifiers run within the context of the ABLS system, they will be granted permission to all of the database servers so that they may request the appropriate entity and epoch keys used to perform strong verification checks on the data in user session log chains. Also, this must be the first activity that is completed because the log querying and automated audit tasks depend on the database columns being properly masked.

5.2 Log Collection

As part of the proposed design
TODO: started on the log collection actors

5.3 Log Querying

- design and implement an easy API for querying information from the log (and only the log) table that checks user authentication privileges first - the logger is the reference monitor to the outside world

- variety of selection functions in which the user can specify the user ID or user and session ID, all data that comes back is verified using strong verification, contents of the log messages are filtered depending on their policy (only if the policy matches the client's attributes will the results be returned)

- outline process of querying (client is authenticated with username/password, attributes are generated, database is queried using encrypted data, results are filtered based on policy, and result is returned)

5.4 Audit Strategy Design

- auditing design and implementation (define automated worker tasks that use pre-defined strategies for auditing (search by userID) or whatnot that can be triggered by any web application set in front of ABLS - making it easy for auditors to start their own automated audits).

6. REFERENCES

[1] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 321–334, Washington, DC, USA, 2007. IEEE Computer Society.

[2] D. Ma. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 341–352, New York, NY, USA, 2008. ACM.

[3] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1999.

[4] A. A. Yavuz and P. Ning. Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 219–228, Washington, DC, USA, 2009. IEEE Computer Society.