Building Web Applications on Top of Encrypted Data Using Mylar

RALUCA ADA POPA, EMILY STARK, JONAS HELFER, STEVEN VALDEZ, NICKOLAI ZELDOVICH, M. FRANS KAASHOEK, AND HARI BALAKRISHNAN

Raluca Ada Popa is a fourth-year PhD student at MIT working in security, systems, and applied cryptography. She is the recipient of a Google PhD fellowship for secure cloud computing and a CRA Outstanding Undergraduate Award. ralucap@mit.edu

Emily Stark is a core developer at Meteor Development Group. She holds an MS degree from MIT and a BS from Stanford University, both in computer science. emily@meteor.com

Jonas Helfer is a PhD student at MIT's Computer Science and Artificial Intelligence Lab. He holds a master's degree in computer science from EPFL (Switzerland). His many research interests include systems security, Web security and software project management. helfer@mit.edu

Using a Web application for confidential data requires the user to trust the server to protect the data from unauthorized disclosures. This trust is often misplaced, however, because there are many ways in which confidential data could leak from a server. For example, attackers could exploit a vulnerability in the server software to break in [9], a curious administrator could peek at the data on the server [1, 2], or the server operator may be compelled to disclose data by law [3]. How can one build Web applications that protect data confidentiality against attackers with *full access* to servers?

We developed Mylar for this purpose. Mylar is a new platform for building Web applications that stores sensitive data *encrypted* on the server. The keys that can decrypt the data are stored in some users' Web browsers, and the data only gets decrypted in these browsers. Even if an attacker fully compromises the server, the attacker gets access to only encrypted data and does not have the necessary decryption keys. Mylar achieves this organization through a new data sharing mechanism, practical ways of computing on encrypted data at the server, and a mechanism for verifying that the application code was not tampered with by a compromised server.

Crucially, Mylar enables many classes of applications to protect confidential data from compromised servers in a practical way. It leverages the recent shift in Web application frameworks towards implementing logic in client-side JavaScript code, and sending data, rather than HTML, over the network [5]; such a framework provides a clean foundation for security.

Mylar is open source and can be found at http://css.csail.mit.edu/mylar/. This article covers how Mylar works at a high level and how to use Mylar on an example application, a chat application. For more details on the research behind Mylar (e.g., detailed decryption of each component, detailed evaluation, etc.), we refer the reader to our NSDI '14 paper [7].

Mylar's Techniques

To understand Mylar's techniques, it is helpful to consider a simple attempt to solve the problem and to observe why this attempt does not suffice. A simple idea is to give each user her own encryption key, encrypt a user's data with that user's key in the Web browser, and store only encrypted data on the server. This model ensures that an adversary would not be able to read any confidential information on the server, because he would lack the necessary decryption keys. In fact, this model has been already adopted by some privacy-conscious Web applications [4, 8].

Unfortunately, this approach suffers from three significant security, functionality, and efficiency shortcomings. First, a compromised server could provide malicious client-side code to the browser and extract the user's key and data. Ensuring that the server did not tamper with the application code is difficult because a Web application consists of many files, such

Steven Valdez is a graduate student pursuing a dual bachelor's/master's degree in computer science at MIT, focusing on systems and security research. dvorak42@mit.edu

Nickolai Zeldovich is an associate professor at MIT. His research interests are in building practical secure systems. nickolai@csail.mit.edu

M. Frans Kaashoek is a professor at MIT, where he co-leads the Parallel and Distributed Operating Systems Group (http://www.pdos.csail.mit.edu/). Frans is a member of the National Academy of Engineering and the American Academy of Arts and Sciences and is the recipient of the ACM SIGOPS Mark Weiser award and the 2010 ACM-Infosys Foundation award. He co-founded Sightpath, Inc. and Mazu Networks, Inc. kaashoek@mit.edu

Hari Balakrishnan's research interests are in networked computer systems. He is a professor of computer science at MIT. hari@csail.mit.edu

as HTML pages, JavaScript code, and CSS style sheets, and the HTML pages are often dynamically generated.

Second, this approach does not provide data sharing between users, a crucial function of Web applications. To address this problem, one might consider encrypting shared documents with separate keys and distributing each key to all users sharing a document via the server. However, distributing keys via the server is challenging because a compromised server can supply arbitrary keys to users and thus trick a user into using incorrect keys.

Third, this approach requires all of the application logic to run in a user's Web browser, because it can decrypt the user's encrypted data. But this is often impractical: For instance, doing a keyword search would require downloading all the documents to the browser.

Mylar overcomes the challenges mentioned above with a combination of systems techniques and novel cryptographic primitives, as follows:

1. Data sharing. To enable sharing, each sensitive data item is encrypted with a key available to users who share the item. To prevent the server from cheating during key distribution, Mylar provides a mechanism for establishing the correctness of keys obtained from the server: Mylar forms certificate paths to attest to public keys and allows the application to specify which certificate paths can be trusted in each use context. In combination with a user interface that displays the appropriate certificate components to the user, this technique ensures that even a compromised server cannot trick the application into using the wrong key.
2. Computing over encrypted data. Keyword search is a common operation in Web applications, but it is often impractical to run on the client because it would require downloading large amounts of data to the user's machine. Although practical cryptographic schemes exist for keyword search, they require that data be encrypted with a single key. This restriction makes it difficult to apply these schemes to Web applications that have many users and hence have data encrypted with many different keys.

   Mylar provides the first cryptographic scheme that can perform keyword search efficiently over data encrypted with *different* keys. The client provides an encrypted word to the server, and the server can return all documents that contain this word without learning the word or the contents of the documents.
3. Verifying application code. With Mylar, code running in a Web browser has access to the user's decrypted data and keys, but the code itself comes from the untrusted server. To ensure that this code has not been tampered with, Mylar checks that the code is properly signed by the Web site owner. This checking is possible because application code and data are separate in Mylar, so the code is static. Mylar uses two origins to simplify code verification for a Web application. The primary origin hosts only the top-level HTML page of the application, whose signature is verified using a public key found in the server's X.509 certificate. All other files come from a secondary origin, so that if they are loaded as a top-level page, they do not have access to the primary origin. Mylar verifies the hash of these files against an expected hash contained in the top-level page.

Mylar's Architecture
There are three different parties in Mylar: the users, the Web site owner, and the server operator. Mylar's goal is to help the site owner protect the confidential data of users in the face of a malicious or compromised server operator.

System Overview
Mylar embraces the trend towards client-side Web applications; Mylar's design is suitable for platforms that:

1. Enable client-side computation on data received from the server.
2. Allow the client to intercept data going to the server and data coming from the server.
3. Separate application code from data, so that the HTML pages supplied by the server are static.

AJAX Web applications with a unified interface for sending data over the network, such as Meteor [5], fit this model. Such frameworks provide a clean foundation for security, because they send data separately from the HTML page that presents the data. In contrast, traditional server-side frameworks incorporate dynamic data into the application's HTML page in arbitrary ways, making it difficult to encrypt and decrypt the dynamic data on each page while checking that the fixed parts of the page have not been tampered with.

Mylar's Components

The architecture of Mylar is shown in Figure 1. Mylar consists of the four following components:

Browser extension. It is responsible for verifying that the client-side code of a Web application that is loaded from the server has not been tampered with.

Client-side library. It intercepts data sent to and from the server and encrypts or decrypts that data. Each user has a private-public key pair. The client-side library stores the private key of the user at the server, encrypted with the user's password. (The private key of a user can also be stored at a trusted third-party server, to better protect it from offline password guessing attacks and to recover from forgotten passwords without regenerating keys.) When the user logs in, the client-side library fetches and decrypts the user's private key. For shared data, Mylar's client creates separate keys that are also stored at the server in encrypted form.

Server-side library. It performs computation over encrypted data at the server. Specifically, Mylar supports keyword search over encrypted data, because we have found that many applications use keyword search.

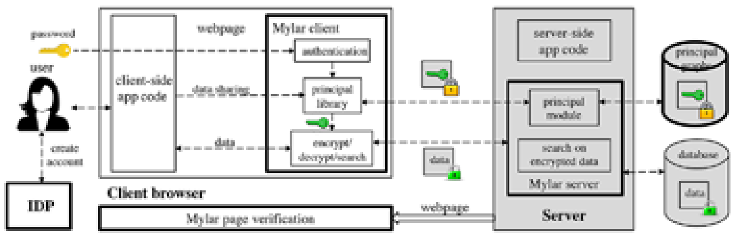

Figure 1: System overview. Shaded components have access only to encrypted data. Thick borders indicate components introduced by Mylar.

Identity provider (IDP). For some applications, Mylar needs a trusted identity provider service (IDP) to verify that a given public key belongs to a particular username. An application needs the IDP if the application has no trusted way of verifying the users who create accounts, and the application allows users to choose whom to share data with. For example, if Alice wants to share a sensitive document with Bob, Mylar's client needs the public key of Bob to encrypt the document. A compromised server could provide the public key of an attacker, so Mylar needs a way to verify the public key. The IDP helps Mylar perform this verification by signing the user's public key and username. An application does *not need* the IDP if the site owner wants to protect only against attackers that do not actively change a server's behavior (namely, attackers that only read the data at the server, and do not install software at the server), or if the application has a limited sharing pattern for which it can use a static root of trust (as described in our full paper [7]).

An IDP can be shared by many applications, similar to an OpenID provider [6]. The IDP does not store per-application state, and Mylar contacts the IDP only when a user first creates an account in an application; afterwards, the application server stores the certificate from the IDP.

Threat Model

Threats

Both the application and the database servers can be *fully* controlled by an adversary: The adversary may obtain all data from the server, cause the server to send arbitrary responses to Web browsers, etc. This model subsumes a wide range of real-world security problems, from bugs in server software to insider attacks.

Mylar also allows some user machines to be controlled by the adversary and to collude with the server. This may be either because the adversary is a user of the application or because the adversary broke into a user's machine.

Guarantees

Mylar protects a data item's confidentiality in the face of arbitrary server compromises, as long as none of the users with access to that data item use a compromised machine. Mylar does not hide data access patterns or communication and timing patterns in an application. Mylar provides data authentication guarantees but does not guarantee the freshness or correctness of results from the computation at the server.

Assumptions

To provide the above guarantees, Mylar assumes that the Web application as written by the developer will not send user data or keys to untrustworthy recipients and cannot be tricked into doing so by exploiting bugs (e.g., cross-site scripting). Our

Function	Semantics
idp_config(*url, pubkey*)	Declares the *url* and *pubkey* of the IDP and returns the principal corresponding to the IDP.
create_user(*uname, password, auth_princ*)	Creates an account for user *uname*, which is certified by principal *auth_princ*.
login(*uname, password*)	Logs in user *uname*.
logout()	Logs out the currently logged-in user.
collection.**encrypted**({*field*: *princ_field*}, ...)	Specify that *field* in *collection* should be encrypted for the principal in *princ_field*.
collection.**auth_set**([*princ_field, fields*], ...)	Authenticate the set of *fields* with principal in *princ_field*.
collection.**searchable**(*field*)	Mark *field* in *collection* as searchable.
collection.**search**(*word, field, princ, filter, proj*)	Search for *word* in *field* of *collection*, filter results by *filter*, and project only the fields in *proj* from the results. Use *princ*'s key to generate the search token.
princ_create(*name, creator_princ*)	Create principal named *name*, sign the principal with *creator_princ*, and give *creator_princ* access to it.
princ_create_static(*name, password*)	Create a static principal called *name*, hardcode it in the application, and wrap its secret keys with *password*.
princ_static(*name, password*)	Return the static principal *name*; if a correct password is specified, also load the secret keys for this principal.
princ_current()	Return the principal of currently logged in user.
princ_lookup(*$name_1$, ..., $name_k$, root*)	Look up principal named *$name_1$* as certified by a chain of principals named *$name_1$* rooted in *root* (e.g., the IDP).
granter.**add_access**(*grantee*)	Give the *grantee* principal access to the *granter* principal.
grantee.**allow_search**(*granter*)	Allow matching keywords from *grantee* on *granter*'s data.

Figure 2: Mylar API for application developers split in three sections: authentication, encryption/integrity annotations, and access control. All of the functions except `princ_create_static` and `searchable` run in the client browser. This API assumes a MongoDB storage model where data is organized as collections of documents, and each document consists of fieldname-and-value pairs. Mylar also preserves the generic functionality for unencrypted data of the underlying Web framework.

prototype of Mylar is built on top of Meteor, a framework that helps programmers avoid many common classes of bugs in practice.

Mylar also assumes that the IDP correctly verifies each user's identity (e.g., email address) when signing certificates. To simplify the job of building a trustworthy IDP, Mylar does not store any application state at the IDP, contacts the IDP only when a user first registers, and allows the IDP to be shared across applications.

Finally, Mylar assumes that the user checks the Web browser's security indicator (e.g., the `https` shield icon) and the URL of the Web application they are using before entering any sensitive data. This assumption is identical to what users must already do to safely interact with a trusted server. If the user falls for a phishing attack, neither Mylar nor a trusted server can prevent the user from entering confidential data into the adversary's Web application.

Security Overview
At a high level, Mylar achieves its goal as follows. First, it verifies the application code running in the browser, so that it is safe to give client-side code access to keys and plaintext data. Then, the client code encrypts the data marked sensitive before sending it to the server. Because users need to share data, Mylar provides a mechanism to securely share and look up keys among users. Finally, to perform server-side processing, Mylar introduces a new cryptographic scheme that can perform keyword search over documents encrypted with many different keys, without revealing the content of the encrypted documents or the word being searched for.

Implementation and Evaluation
To evaluate Mylar's design, we built a prototype on top of the Meteor Web application framework [5]. We ported six applications to protect confidential data using Mylar: a medical application for endometriosis patients, a Web site for managing

homework and grades, a chat application called kChat, a forum, a calendar, and a photo-sharing application. The endometriosis application is used to collect data from patients with that medical condition and was designed under the aegis of the MIT Center for Gynepathology Research by surgeons at the Newton-Wellesley hospital (affiliated with Harvard Medical School) in collaboration with biological engineers at MIT; the Mylar-secured version is currently being tested by patients and is undergoing IRB approval before deployment.

Our results show that Mylar requires little developer effort: We had to modify an average of just 36 lines of code per application. We also evaluated the performance of Mylar on three of the applications above. For example, for kChat, our results show that Mylar incurs modest overheads: a 17% throughput reduction and a 50-ms latency increase for the most common operation (sending a message). These results suggest that Mylar is a good fit for multi-user Web applications with data sharing.

Using Mylar

Mylar for Developers

The developer starts with a regular (non-encrypted) Web application implemented in Mylar's underlying Web platform (Meteor in our prototype). To secure this application with Mylar, a developer uses Mylar's API (Figure 2), which we show how to use on a chat example. First, the developer uses Mylar's authentication library for user login and account creation. If the application allows a user to choose which other users to share data with, the developer should also specify the URL and public key of a trusted IDP.

Second, the developer specifies which data in the application should be encrypted and who should have access to it. Mylar uses principals for access control; a principal corresponds to a public/private key pair and represents an application-level access control entity, such as a user, a group, or a shared document. In our prototype, all data is stored in MongoDB collections, and the developer annotates each collection with the set of fields that contain confidential data and the name of the principal that should have access to that data (i.e., whose key should be used).

Third, the developer specifies which principals in the application have access to which other principals. For example, if Alice wants to invite Bob to a confidential chat, the application must invoke the Mylar client to grant Bob's principal access to the chat room principal.

Fourth, the developer changes their server-side code to invoke the Mylar server-side library when performing keyword search. Our prototype's client-side library provides functions for common operations such as keyword search over a specific field in a MongoDB collection.

Finally, as part of installing the Web application, the site owner generates a public/private key pair and signs the application's files with the private key using Mylar's bundling tool. The Web application must be hosted using https, and the site owner's public key must be stored in the Web server's X.509 certificate. This ensures that even if the server is compromised, Mylar's browser extension will know the site owner's public key and will refuse to load client-side code if it has been tampered with.

Chat Application Example

To demonstrate how a developer can build a Mylar application, we show the changes that we made to the kChat application to encrypt messages. In kChat, users can create chat rooms, and existing members of a chat room can invite new users to join. Only invited users have access to the messages from the room. A user can search over data from the rooms she has access to. Figure 3 shows the changes we made to kChat, using Mylar's API (Figure 2).

```
// On both the client and the server:
idp = idp_config(url, pubkey);
Messages.encrypted({"message": "roomprinc"});
Messages.auth_set(["roomprinc", ["id", "message",
   "room", "date"]]);
Messages.searchable("message");

// On the client:

function create_user(uname, password):
        create_user(uname, password, idp);

function create_room(roomtitle):
        princ_create(roomtitle, princ_current());

function invite_user(username):
   global room_princ;
   room_princ.add_access(princ_lookup(username, idp));

function join_room(room):
   global cur_room, room_princ;
   cur_room = room;
   room_princ = princ_lookup(room.name,
      room.creator, idp);

function send_message(msg):
   global cur_room, room_princ;
   Messages.insert({message: msg, room: cur_room.id,
      date: new Date().toString(),
      roomprinc: room_princ});

function search(word):
   return Messages.search(word, "message",
      princ_current(), all, all);
```

Figure 3: Pseudo-code for changes to the kChat application to encrypt messages. Not shown is unchanged code for managing rooms, receiving and displaying messages, and login/logout (Mylar provides wrappers for Meteor's user accounts API).

The call to *Messages*.**encrypted** specifies that data in the "message" field of that collection should be encrypted. This data will be encrypted with the public key of the principal specified in the "roomprinc" field. All future accesses to the *Messages* collection will be transparently encrypted and decrypted by Mylar from this point. The call to *Messages*.**searchable** specifies that clients will need to search over the "message" field; consequently, Mylar will store a searchable encryption of each message in addition to a standard ciphertext.

When a user creates a new room (create_room), the application in turn creates a new principal, named after the room title and signed by the creator's principal. To invite a user to a room, the application needs to give the new user access to the room principal, which it does by invoking **add_access** in invite_user.

When joining a room (join_room), the application must look up the room's public key, so that it can encrypt messages sent to that room. The application specifies both the expected room title as well as the room creator as arguments to **princ_lookup**, to distinguish between rooms with the same title.

To send a message to a chat room, kChat needs to specify a principal in the **roomprinc** field of the newly inserted document. In this case, the application keeps the current room's principal in the *room_princ* global variable. Similarly, when searching for messages containing a word, the application supplies the principal whose key should be used to generate the search token. In this case, kChat uses the current user principal, **princ_current**().

Mylar for Users

To obtain the full security guarantees of Mylar, a user must install the Mylar browser extension, which detects tampered code. However, if a site owner wants to protect against attackers who only read server data (as opposed to actively modifying data or installing software at the server), users don't have to install the extension and their browsing experience is entirely unchanged.

Conclusion

Mylar is a novel Web application framework that enables developers to protect confidential data in the face of arbitrary server compromises. Experimental results show that using Mylar requires few changes to an application, and that the performance overheads of Mylar are modest.

Acknowledgments

This research was supported by NSF award IIS-1065219, by DARPA CRASH under contracts #N66001-10-2-4088 and #N66001-10-2-4089, by Quanta, and by Google.

References

[1] D. Borelli, "The Name Edward Snowden Should Be Sending Shivers Up CEO Spines," *Forbes,* Sept. 2013: http://www.forbes.com/sites/realspin/2013/09/03/the-name-edward-snowden-should-be-sending-shivers-up-ceo-spines/.

[2] A. Chen, "GCreep: Google Engineer Stalked Teens, Spied on Chats," *Gawker,* Sept. 2010. http://gawker.com/5637234/.

[3] Google, Inc. User Data Requests—Google Transparency Report: http://www.google.com/transparencyreport/userdatarequests/, accessed Sept. 2013.

[4] MEGA: The Privacy Company: https://mega.co.nz/#privacycompany, accessed Sept. 2013.

[5] Meteor, Inc., Meteor: A Better Way to Build Apps: http://www.meteor.com, accessed Sept. 2013.

[6] OpenID Foundation, OpenID: http://openid.net, accessed Sept. 2013.

[7] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, "Building Web Applications on Top of Encrypted Data Using Mylar," *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14),* Seattle, WA, Apr. 2014.

[8] Cryptocat: https://crypto.cat/, accessed Sept. 2013.

[9] J. Tudor, "Web Application Vulnerability Statistics," June 2013: http://www.contextis.com/services/research/white-papers/web-application-vulnerability-statistics-2013/.