Exchanging Database Writes with Modern Crypto

Andreas Happe, Thomas Loruenser
Department Safety & Security
AIT Austrian Institute of Technology GmbH
Vienna, Austria
email: {andreas.happe|thomas.loruenser}@ait.ac.at

***Abstract*—Modern cryptography provides for new ways of solving old problems. This paper details how Keyed-Hash Message Authentication Codes (HMACs) or Authenticated Encryption with Associated Data (AEAD) can be employed as an alternative to a traditional server-side temporal session store. This cryptography-based approach reduces the server-side need for state. When applied to database-based user-management systems it removes all database alteration statements needed for confirmed user sign-up and greatly removes database alteration statements for typical "forgot password" use-cases. As there is no temporary data stored within the server database system, there is no possibility of creating orphaned or abandoned data records. However, this new approach is not generic and can only be applied if implemented use-cases fulfill requirements. This requirements and implications are also detailed within this paper.**

***Index Terms*—Internet, Network security, Web services**

I. INTRODUCTION AND PROBLEM STATEMENT

A common web-application user-interaction pattern is *Request-Verification-Execution*. An example can be seen in Fig. 1: the user requests an server-side operation and transmits needed data to the server. The server validates the information and stores the user-submitted data temporally on the server. To verify the user request a challenge is transmitted through a separate communication channel. After the user fulfills the challenge the operation is executed and finalized on the server. An example of this pattern is a user registration (we are basing all examples upon Ruby on Rails' *Devise* framework): after a potential new user entered her data on a website, she is presented with an confirmation email and the user account is only activated after the user has confirmed her identity through a confirmation link within this email. Similar examples are typical "user registration", "password reset" or "delete account" functions.

Problems arise if the user fails to perform the verification step or automated tools request the initial operation thousands or millions of times without ever performing the corresponding confirmation step. During the initial request temporary data is stored within the server: this data is "dead" and must be eventually removed from the database. Furthermore, additional server-side data is needed for the implementation of the verification process. For example, commonly used implementations (depicted in Fig. 2) generate and store a random token within a server-side database. This token is included within the confirmation email and later matched against the database record.

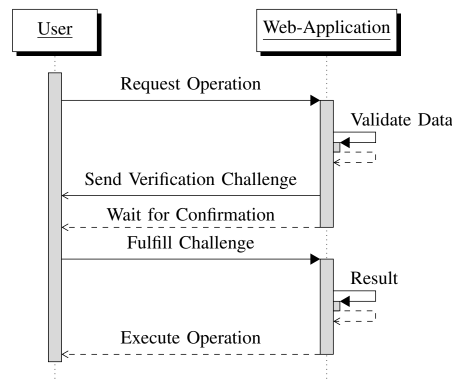


Fig. 1. Generic "Request-Verification-Execution" Work-Flow

Server-side temporary data is commonly stored within databases and therefore leads to I/O overhead. In this paper, we propose an cryptographic alternative utilizing HMACs [1] or AEAD that inherently prevents stale database entries and removes the need for both data alteration operations and additional database fields.

The tow approaches are detailed in Section II and Section III. This work is compared with existing solutions in section IV. Section V showcases advantages and disadvantages of this approach while section VI concludes this paper.

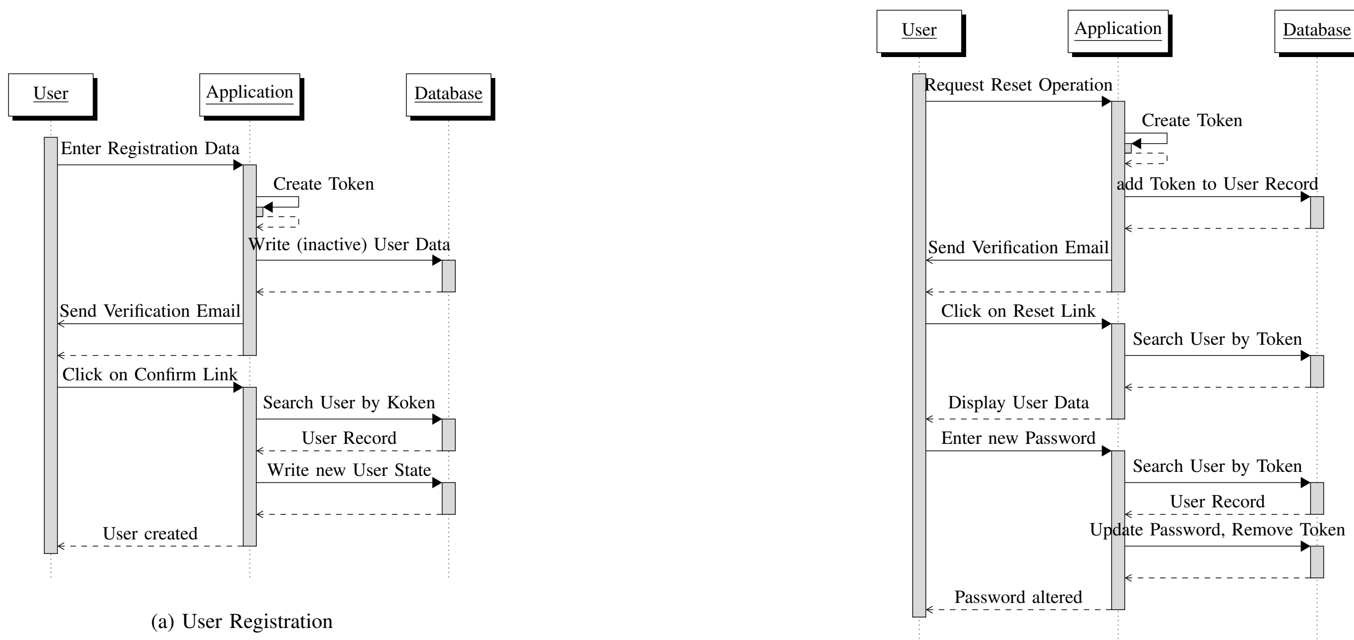
II. DATABASE-CENTRIC APPROACH

The database-centric approach stores all temporary data within the server-side database as can be seen through the multiple database calls in Fig. 2.

The web-application must track confirmed and unconfirmed operations. For example, during user registration the application must separate confirmed from unconfirmed users as the latter must not be able to login. To differentiate between those either an explicit or implicit state is used. The former can be implemented through an additional *state* field. The latter can be implemented through additional metadata, e.g., a *confirmed_at* field. Those fields must be added to each user record but are unused most of the time.

Typically, an additional database field *created_at* is used to store the timestamp of the original request. This allows detection of orphaned operation that never have been confirmed. Subsequently, the user might be sent an reminder-email, but

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-512-8



(a) User Registration

(b) Resend Password Functionality

Fig. 2. Sequence Diagrams for database-centric work flows.

eventually the stale user record has to be removed if the user does not interact.

The "user registration" use-case shows three distinct database interactions. 1) During user registration all mandatory user data is captured and stored within the database. Additional fields (e.g., *confirmation_token* and *confirmed_at*) are needed to perform the registration and are thus added to the database. If the new user never confirms his registration, the whole data set is "dead" data. 2) when the user clicks on the confirmation link contained in the mail, the user record has to be retrieved from the database and 3) after confirmation the user record's state is set and meta-information is cleaned up.

The "password reset" use-case does not depend upon record creation but alters the existing user record multiple times. To perform the action two new fields are added to the user record: *reset_password_sent_at* and *reset_password_token*. Typically, four phases of database access can be seen: 1) a new reset token is generated and stored within the user record, 2) when the user clicks on the verification link, the database is queried for it's validity and then a password entry formula is shown 3) when the user submits a new password the token is again verified and 4) the password is finally updated within the database and meta-information is cleaned up. Even if the data is retrieved directly from a in-memory data store, the session verification commonly entails database access.

III. Alternative: Usage of HMAC/AEAD

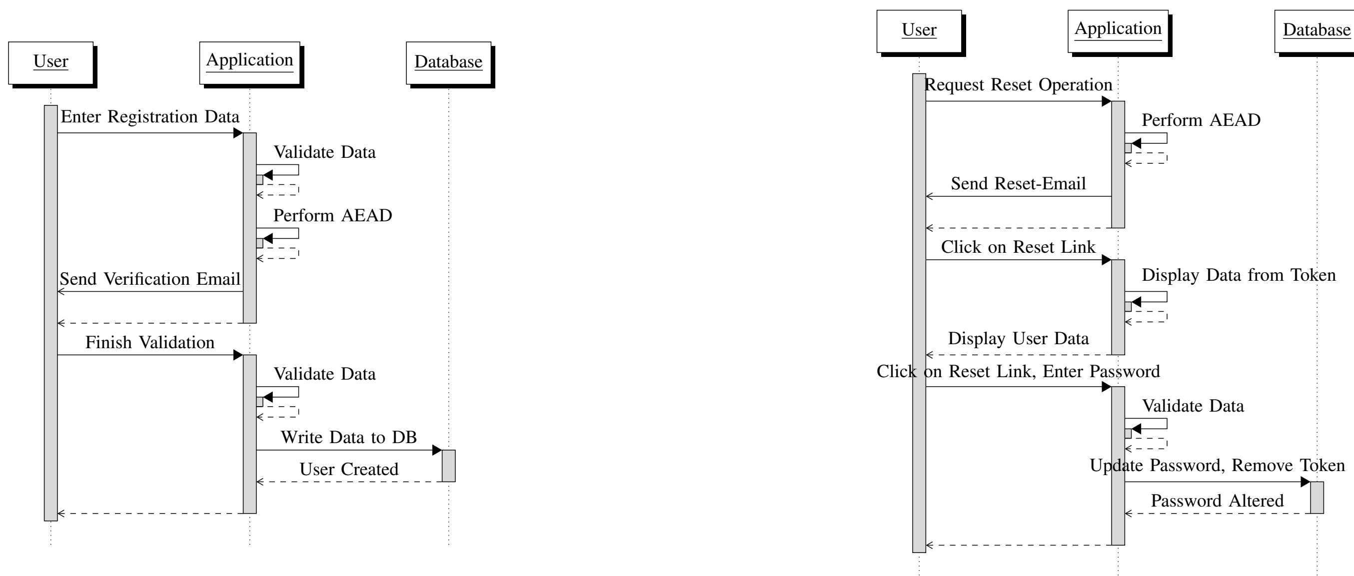
Our proposed alternative approach does not depend upon server-side storage of temporary state but uses cryptography to authenticate data presented by the user during the verification step. Cryptography is used to ensure that neither users nor third-parties are able to tamper data passed on by the server.

HMACs[1] offer a solution to this problem. A MAC combines the plain-text data with a secret-key and produces a tag that verifies the plain-text data's integrity and authenticity. In our case, the server creates the tag and transmits the plain-text data and the tag within the verification challenge. When the user fulfills the challenge, the tag verifies that the data has not been tampered with. The secret-key never leaves the server. The transmitted data has to be careful chosen to prevent replay attacks as will be detailed later on.

HMACs provide authentication and integrity but not data confidentiality as all input data is transmitted in plain sight. AEAD in contrast provides confidentiality, integrity and authenticity. An AEAD algorithm uses the plain-text and a secret key as input, and produces a cipher text and authentication tag as output. The overall protocol is similar to the HMAC-based protocol but replaces the plain-text data with the cipher text to additionally provide data confidentiality. Also similar to the HMAC-based algorithm, the encoded data must be sufficient to prevent replay attacks.

Fig. 3 details two possible AEAD-based workflows. Instead of storing temporary state within the server-side database, the state is encrypted and attached to the verification challenge, e.g., encoded in the verification link contained in the sent verification email. The usage of HMACs/AEAD prevents users or third-parties from maliciously altering the transmitted data. During verification, the server verifies and decrypts the received encrypted data and uses it as intermediate state that finally is committed to the database. Note that only a single

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-512-8



(a) User Registration. The encrypted data includes an operation identifier, expiry as well as all data needed for the user registration.

(b) Resend Password Functionality. The encrypted data includes an operation identifier, user, expiry, hash of the stored password hash.

Fig. 3. Sequence Diagrams for cryptographic-based work flows. Note the reduced amount of database operations.

database transaction is performed. This also removes metadata fields, as well as "verification"-states from user records.

A. Data-Field Selection

Proper selection of fields used as input for the HMAC/AEAD-function prevents replay attacks. This selection should be sufficient to uniquely identify the executed operation and its context.

An unique *operation identifier* is mandatory. Otherwise given $operation_1$ and $operation_2$, the encoded data from $operation_1$ could be used during verification of $operation_2$.

A *stable user identified* is needed to associate the encoded data with an user account.

Once the server has authenticated a data block, it is valid until the server's private key is changed. A real-world implementation would include a *creation or expiry date* to limit the lifetime of the verification token.

An *anchor element* is needed to differentiate between subsequent states. For example, a "reset password"-function should include a hash of the current password. The hash will be compared to the currently stored hash during operation execution and thus limit's the operation's validity to changing the "current" password. Otherwise, the same verification token could be maliciously used to repeatedly change the user's password.

All *user-supplied data* that would otherwise be stored as temporary data within the server-side database must be included within the data block.

In our "user registration" case, we could select the following parameters: *[operation-id: "registration", expiry: "2016-1-30", user-id: "andreas.happe@ait.ac.at", user-data: "..."]* as input for the AEAD/HMAC function. For a potential "password reset" function, we would chose *[operation-id: "reset", expiry: "2016-1-30", user-id: "andreas.happe@ait.ac.at", old-password-hash: "hash of my currently stored password"]* as verification token.

IV. Related Work

To the best of our knowledge, the first and only study—and thus inspiration for this paper— of using HMACs for *Request-Verification-Execution* workflows was published within a blog post by Mahmoud Al-Qudsi[2]. In contrast, our paper details potential replay attack vectors and introduces data confidentiality through usage of AEAD schemes.

A comparable mechanism lies at the heart of HTTP session management: here, the session is managed on either server- or client-side. With client-side management the session's data is signed on the server with a secret key, transmitted and stored at the client. On subsequent requests the cookie is transmitted to the server and validated with the server's secret key. When using a server-side scheme all session-data is stored at the server and only a session identifier is stored at the client. The overall scheme is similar to our proposed approach but it's applicability differs: HTTP cookies are limited to the client's browser and thus allow for identification of a browser-based session. We see usage for our scheme within persisted mails. As mails can be stored and forwarded between clients our approach is more akin to token passing.

As an concrete example, the Ruby on Rails' *CookieStore* stores session information within a client-side cookie and verifies the validity of the cookie through a server-side computed HMAC utilizing a secret server-side key. An alternative session-store is database-backed. The Ruby on Rails Security Guide[3] describes implications of the different session stores that are similar to the those of using AEAD/HMAC-based temporary state storage.

In August 2008, AEAD schemes were made mandatory for TLSv1.2[4], [5]. The ambiguity of encrypted HTTPS communication lead hardware vendors to the inclusion of dedicated encryption co-processors and/or inclusion of encryption-specific instructions within their microcode thus making hardware-supported high-performance AEAD functions commonly available on server-grade hardware. The situation will further improve in the future as the current TLSv1.3 draft mandates the usage of AEAD ciphers.

V. Implications and Limitations

Usage of the cryptography-centric approach has several implications:

No server-side state. As no temporal data is stored, the server has no opportunity to detect operations waiting for user confirmation. This functionality is sometimes used to implement an "reminder" email to improve user response rate.

A *stable ID and anchor element are needed to prevent playback attacks*. For example, if user-changeable email addresses are used as stable identifiers, an attacker can create a new "user deletion token", change his email account, wait until a victim creates an account using the same email address, and then delete this account until the token expires. To solve this, an additional anchor element has to be included, in this case the initial creation time could be used. For another example of an anchor element, consider the mentioned "reset password"-function: request should include a hash of the current password. This anchors the update to the current database state and prevents the replay attack. Without an unique anchor element, race conditions can occur. For example, if two users sign-up with the same email address the first user that confirms (through his link) will be created. A similar race condition is present at the traditional scheme, but in this case the critical section is the initial data entry form.

The *wrong HTTP Verb is used.* Confirmation will commonly happen through a HTTP GET link. Data alteration operations should be performed through a HTTP PUT, POST or DELETE operation. The same discrepancy is true for traditional schemes.

The *Privacy of the secret key is paramount.* If an attacker retrieves the secret key (used for encryption or HMAC-operation) from the server he can forge any operation request. We think this attack vector to be non-critical, as an attacker with this capability can already access the database directly.

Size limit for transmitted messages. While the HTTP/1.1 protocol initially did not limit the transmission size for GET request[6], a later revision did place a limit of 8000 octets[7]. Real-world web browsers and servers enforce different limits, especially older Internet Explorer Versions (2048 bytes). Current browsers and servers should be able to cope with 8192 bytes of data while 2048 bytes should be reasonable safe.

Aesthetic Considerations. With AEAD all information is encoded within the passed parameter. This parameter can get large and leads to an unaesthetic URL – this can be a problem for usage within text-based emails while it is acceptable within HTML-based emails.

Usage before confirmation is not possible. Sometimes web-applications offer limited functionality between user-sign-up and user-verification. This is not possible with our alternative scheme as the server-side user-account is only created upon user-verification.

Also please note that the integrity and confidentiality of the transport layer needs to be provided by additional means. With a captured token the attacker has all needed means for an Man-in-the-Middle attack. This implies mandatory usage of TLS.

VI. Conclusion

This paper initially introduced common problems that arise with server-side state management. Cryptographic means allow alternative schemes that do not share those issues, but in turn, their applicability highly depends upon the surrounding use-case. We have shown how two common use-cases, "user registration" and "password reset" can be implemented using our new scheme.

As shown with those examples, we believe that our alternative solution has real-world merits and can improve existing software solutions. In addition, we hope, that this paper shines more light upon the generic technique and increases it's visibility among software engineers.

VII. Acknowledgements

The authors would like to thank Mahmoud Al-Qudsi for his detailed comments.

References

[1] H. Krawczyk, M. Bellare, and R. Canetti, "Rfc 2104: Hmac: Keyed-hashing for message authentication," *URL http://www.rfc.net/html/rfc2104*, 1997.

[2] M. Al-Qudsi. (2015) Life in a post-database world: using crypto to avoid db writes. [Online]. Available: https://neosmart.net/blog/2015/using-hmac-signatures-to-avoid-database-writes/

[3] Ruby on rails security. [Online]. Available: http://guides.rubyonrails.org/security.html#session-storage

[4] T. Dierks and E. Rescorla, "Rfc 5246: The transport layer security (tls) protocol version 1.2," *URL http://www.rfc.net/html/rfc5246*, 2008.

[5] J. Salowey, A. Choudhury, and D. McGrew, "Rfc 5288: Aes galois mode (gcm) cipher suites for tls," *URL http://www.rfc.net/html/rfc5288*, 2008.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616: Hypertext transfer protocol–http/1.1, 1999," *URL http://www.rfc.net/rfc2616.html*, 2009.

[7] R. Fielding and J. Reschke, "Rfc 7230: Hypertext transfer protocol (http/1.1): Message syntax and routing," *URL http://www.rfc.net/rfc7230.html*, 2014.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-512-8