Secure Mashup-Providing Platforms - Implementing Encrypted Wiring

Matthias Herbert, Tobias Thieme, Jan Zibuschka, and Heiko Roßnagel

Fraunhofer IAO,
Nobelstrasse 12,
70569 Stuttgart, Germany
{matthias.herbert,tobias.thieme,jan.zibuschka,heiko.rossnagel}
@iao.fraunhofer.de

Abstract. Mashups were not designed with security in mind. Their main selling point is the flexible and easy to use development approach. The fact that mashups enable users to compose services to create a piece of software with new functionalities, integrating inputs from various sources, implies a security risk. However, in many scenarios where mashups add business value, e.g. enterprise mashups, security and privacy are important requirements. A secure environment for the handling of potentially sensitive end user information is needed, unless the user fully trusts the mashup-providing-platform (MPP), which is unlikely for hosted enterprise mashups. In this paper we present a proof-of-concept implementation which enables the secure usage of a mashup-providing platform and protects sensitive data against malicious widgets and platform operators.

Keywords: Secure mashups, security, mashup, enterprise mashups.

1 Introduction

Mashup technology has a clear benefit for both users and providers. The good usability and flexibility of the mashup development process enables users to build new services without requiring professional programming skills. They just have to choose and wire up different services and widgets with the help of the mashup platform. To fully leverage this advantage, especially small and medium enterprises (SMEs) need to have their platforms hosted by a platform provider to avoid additional system administration upkeep. However, both a hosted mashup platform (Mashup-providing platform, MPP) and widgets within such a platform could intercept sensitive user data. This lack of confidentiality is a barrier to the entrance of the mashup technology in the enterprise environment.

We implemented a proof-of-concept demonstrator of a secure mashup platform based on the approach described by Zibuschka et al. [1]. The authors propose an architecture which secures the mashup's users against a malicious platform or widgets. The main benefit of this approach is that it requires neither a trusted third party, nor substantial modifications of the mashup composition process. It also suggests a way to have a client-side key generation in JavaScript without the steep performance penalties associated with approaches such as [2].

The rest of this paper is structured as follows. We first present related work on mashup-security in section 2. Based on this work we present our approach in section 3 and provide the implementation details in section 4. We discuss the benefits and applications of our solution scenarios in section 5 and discuss our results in section 6 before we conclude our findings.

2 Related Work

To make our work more readable and readily comparable, we use the definitions from Hoyer and Fischer [3], who define mashup as: a web-based resource that combines existing resources be it content, data or application functionality, from more than one resource by empowering the actual end-users to create and adapt individual information centric and situational applications. They also define widgets as the visual representations of aforementioned Web-based resources that are combined into a mashup. Wiring is defined as communication between widgets within the platform, and piping as the transfer of external resources into the platform via backend service.

Security in mashups has been investigated by several previous works. Hasan et al [4] present a component controlling data flow within a mashup, using a 'Permit Grant Service' that acts on behalf of the mashup platform. Crites et al [5] introduce a modified browser security model addressing some of the security shortcomings of specifically mashups, while Jackson and Wang [6] describe a solution using the existing browser origin policies. However, none of the systems offer protection versus the platform itself. In the same vein, Keukelaere et al [7] propose a secure channel communication model. We extend on their work, adding encryption to protect confidentiality of the users' information versus the MPP. Zarandioon et al [2] [8] offer the most comprehensive approach to date, offering both a client-side identity management providing data flow control as well as a single sign-on (SSO) solution. However, it requires that each user administers a specific identity-providing server outside the MPP, and understands a set of (non-trivial) mediating components. It also requires that inter-widget (which are referred to as 'mashlets') communication is executed using specialized widgets, which is a counterintuitive modification of mashup programming practices, threatening mashups' main selling point, namely their ease of modification by non-experts [1]. HTML 5 will offer cross-document messaging, which potentially addresses integrity and confidentiality of communication between components [7]. However, this requires browser support, and will not be realistic in the next years [7], especially in enterprise scenarios, where older browsers, especially Internet Explorer 6, are still widely used [9].

3 Concept

Our paper is based on Zibuschka et al. [1], who propose reversed identity based encryption within a public-key-infrastructure to realize a secure MPP. Specifically, to avoid the need of a Trusted-Third-Party inherent in identity based encryption, the authors propose to let the users themselves act as the private key generators for communication between widgets. The originally proposed system is based on WebIBC [10], with a security model based on nested iFrames [7]. Widgets use the

generated private keys to communicate, sending encrypted messages through the MPP. This idea reduces the amount of parties the user has to trust, satisfying both the business requirements stated in the introduction of this paper, as well as strong security requirements in the vein of privacy-enhancing identity management [11]. However, Zibuschka et al. [1] only describe the theoretic concept in detail, and do not give details of a proof-of-concept implementation. Our contribution aims to fill this gap.

In contrast to their approach, we decentralize key generation by having each widget in a mashup create its own set of keys within the user’s browser. This offers several advantages over their solution. First, a key distribution from key generator to the widgets, which in general offers a point of attack, is not necessary, as everybody generates their own keys. Second, the individual keys are protected against unauthorized access within each widget by the Same Origin Policy (SOP). The nested iFrame model assures that each widget is loaded within its own domain, yet is still able to communicate with the MPP and other widgets. This solution allows key generation without any browser plug-ins [2] [3]. However, this approach requires a high-performance JavaScript-implementation for the key generation.

We also use hybrid encryption instead of Combined-Public-Key-cryptography, as we were unable to find a working implementation of CPK in JavaScript. We use RSA [12] encryption to exchange the symmetric AES [13] session keys. A key exchange takes place for each wiring the user defines in the wiring tool. Wiring is unidirectional, meaning that if bidirectional communication between two widgets is desired, two wirings have to be established. This enables the user to define services that are only able to receive messages and not to send them.

Fig. 1. Architecture of the mashup providing platform (MPP)

Figure 1 shows the aspects of our architecture relevant to security. Alice and Bob each generate their own set of keys. The public RSA keys are published to the MPP. When Alice wants to send Bob a message, she uses his public RSA-key to encrypt her

AES-key. Only Bob can decrypt Alice’s AES key, which will be used for the actual communication.

We want the user to be able to use the mashup in a very intuitive way. Because of this requirement we fully automated key generation within the widgets and bound the key distribution to the wiring paths. During the initiation process of the mashup every participating widget generates its own keys, with a public and a private RSA key for the key exchange of symmetric AES session keys. The widgets send their public keys to the MPP, which forwards them to every wired widget. All unwired widgets are not able to read or manipulate the communication because of the SOP.

4 Implementation

For the proof-of-concept-implementation of our secure mashup platform we rely on existing open source software and libraries. We use the Dreamface [14] mashup platform as our MPP, which provides most common mashup functionalities, including a wiring tool, user management and a drag and drop interface for the arrangement of widgets.

Our approach requires reliable implementations of both asymmetric and symmetric ciphers in JavaScript. A quick survey showed that there are a number of libraries that implement symmetric algorithms, but only very few that implement asymmetric ones. The complicated key generation procedures of ciphers such as RSA make asymmetric schemes difficult to implement efficiently in an interpreted script language like JavaScript.

The implementation of CPK presented in [10] turned out to be incomplete. PidCrypt [15] provides a strong implementation of RSA in JavaScript, but also requires a browser plug-in, which we want to avoid. The developers of Clipperzlib [16] are working on an implementation of a public key cryptography system based on elliptic curve cryptography, but describe their implementation as “still slow and incomplete”. Therefore, we decided to use the JavaScript Cryptography Toolkit [17], which provides several well-known algorithms. Among them is an implementation of RSA that will not freeze the user’s browser during the time-consuming key generation. The key generation is performed in a time frame ranging from a couple dozen milliseconds to a few seconds, depending on the browser.

We use the standardized and efficient AES algorithm to encrypt the actual messaging within the mashup. We chose the implementation of AES provided by the Crypto-JS library [19] as it offers a compact interface, has a low performance overhead, and only requires a single JavaScript file to be included.

For the implementation of our prototype we chose an incremental approach. We began with the functions necessary to read from and write to the fragment identifier of an iFrame and implemented the structure of three nested iFrames that forms the basis for our secure mashup. Next, we integrated this structure into the Dreamface framework. As Dreamface itself uses no frames that could be used as a point of reference during communication, we had to assure that each message sent by a widget was properly identified by the mashup. Communication from any widget to the

mashup platform is handled by a tunnel-page provided by the mashup platform. The Widgets load this page on demand in an iFrame and include their message in the fragment identifier of the page's URL. The tunnel iFrame passes the message on to the mashup platform. The mashup platform has no way of knowing which widget sent the message, as all widgets use the same tunnel iFrame. Dreamface assures that all widgets within a mashup have a unique name, even if they are instances of the same widget class. Thus, we included the name of the widget with the message from the widget to the mashup platform so the mashup platform knows which widget sent the message and can handle it properly. This technique allows an impersonation attack if a widget supplies the name of another widget in the mashup instead of its own. However this attack can be thwarted by also including a random seed with each message that only the mashup platform and the original widget know. An overview of the mechanics of message transmission is given in figure 2. Upon clicking the "Send message" link in the widget-iFrame, a test message is sent to the tunnel-iFrame and displayed in an alert window. After the window is closed, the test message is forwarded to the parent frame (representing the MPP) and displayed again.

Fig. 2. Message transmission in test UI

As vital elements of our architecture are located at the widgets, we implemented an example widget class called "Alice" that was used to realize an "Alice & Bob"-type communication scenario within a mashup. A widget consists of two parts: The actual widget that is loaded within an iFrame and is located on the widget server, and a few JavaScript functions that need to be implemented within the mashup platform to display the widget (in our case this function loads the iFrame containing the actual widget) and handle the messaging. The latter part is a concession we had to make due to the architecture of Dreamface that assumes widgets are realized in JavaScript and provided by the mashup platform. It is not part of our architecture and can be neglected when a similar architecture is implemented within another mashup platform.

Messages in plaintext can now be exchanged between two instances of our "Alice"-widget class. In the next step, we integrated the cryptographic components

into our prototype, beginning with RSA to properly exchange the symmetric AES key. As the key exchange within the mashup is done using the same messaging mechanism also used for regular messages, we defined a protocol for these messages, stating message type, message source and target, whether the message is encrypted and of course the message itself. Figure 3 shows a screenshot of the simple communication scenario. On top are the widgets “Alice” and “Bob” and below is a test widget that visualizes the messages transmitted over the MPP is displayed.

Fig. 3. Test Widgets within Dreamface

The implementation of AES provided by the Crypto-JS library offers a compact, easy to use interface. AES accepts any string as key, thus time-consuming key-generation procedures aren’t necessary. In our prototype, we use the name of the widget the key belongs to as identifier. An implementation using entirely random generated keys will be included in a future version of our prototype. Usage of a password-based key derivation function like PBKDF#2 is also a possibility.

We made several changes to the mashup platform to automate the key exchange. All communication is encrypted with the AES key of the sending widget, which needs to be distributed to all widgets acting as receiver. Thus, this key needs to be encrypted with the public RSA key of any receiving widget during key distribution. This requires a reversal of message flow, as usually messages are only sent from sender to receiver, but not the other way round – if bilateral communication between two widgets is required, two communication channels need to be defined; in both channels the AES key created by the widget defined as sender will be used for encryption. We implemented several functions to achieve reversal of message flow during key exchange. Whenever a widget publishes its public RSA key, these functions iterate over the communication channels the user defined via the wiring tool. If the widget is

defined as the receiver in at least one of those relations, the mashup platform forwards the key to the widget defined as sender. This widget uses the key to encrypt its own AES key and sends it to the widget that sent the RSA key. The key handler functions forward each AES key only to the widget that created the RSA key, that is used to encrypt the AES key. As a widget can be defined as the sending party in several communication channels, this relation can't be evaluated from the wiring. Correct delivery of all keys is guaranteed as each message contains the name of the widget sending it. When sending an encrypted AES key, the message just has to be forwarded to the widget that sent the RSA key used.

5 Scenarios

We implemented two example scenarios to illustrate the functionality of our prototype. These are the aforementioned simple communication scenario and a micropayment scenario, which is shown in figure 4. In both scenarios users can either use encrypted or non-encrypted communication. A special widget visualizes the message flow as the mashup platform sees it.

Fig. 4. Micropayment scenario

The micropayment scenario shown in figure 4 further demonstrates the possibility for MPPs to provide altogether new services to the customers that would be impossible without secure mashup technology. With secure and encrypted communication the MPPs are able to realize new products like the 'money account - widget' shown in the second scenario. Service and widget providers could provide payable services with a lot of functionalities and use these money widgets as a central and secure way of payment. This offers the advantage that the user can use the same widget as an interface to pay for multiple different services by various providers. Neither does he need to enter his payment information with each widget separately, as

is the case with current payment solutions, nor does he have to trust forwarding mechanisms that, for example, direct him from the online shop to his credit card or micropayment provider and back again.

6 Discussion

We implemented a secure mashup-providing platform by combining existing open source software and libraries. Our implementation uses a hybrid encryption model to secure message exchange within the mashup. It offers the same usability as older, insecure platforms. Key generation, key exchange and all encryption and decryption procedures are automated and run in the background. There is no significant difference in performance to existing MPPs. The user is not impeded in his use of the mashup in any way, which is a clear advantage over other solutions. She does not need to enter a master passphrase. The wiring process is the same as in similar, insecure frameworks. Our secure MPP can be used on any current browser without having to install plug-ins or additional software. Furthermore, it does not rely on a trusted-third-party to ensure security and the users don't have to register to additional services to use the mashup.

As mentioned above we have chosen the open-source platform Dreamface as the MPP for our proof-of-concept implementation, but our approach can be generalized. In contrast to other work, our implementation is portable to other MPPs, like IBM Mashup Center [18] or JackBe [19]. Nearly every MPP is able to integrate this widget-based client-side encryption to provide a secure and privacy-friendly mashup environment. The integration is manageable without difficult changes and has potential to enable several new uses cases that require security for MPP's increasing their service spectrum and appeal.

One possible business scenario is implementing web applications offering chargeable services to users, e.g. in the context of telecommunication providers. These services can also be made available in form of widgets. The user entrusts such a mashup with potentially sensible personal information, e.g. location or payment details, the confidentiality of which needs to be ensured.

Another important use case is the enterprise environment, where companies could expand their software with mashup technology. Especially in this case the sensitive business data has to be protected. Our implementation of a secure mashup platform allows the move towards pay-per-use and enterprise mashups while maintaining security. It proves that mashup technology can be transferred to more security sensitive business fields.

Our work has several limitations. The most prominent one is the potential of a man in the middle attack that can be executed by the MPP. This is a clear disadvantage in comparison to the solution proposed in [1], which mitigates this problem using CPK. However, as to our knowledge no working JavaScript implementation of CPK exists, such a solution is not feasible at the moment, but might be possible in the near future. In the meantime several steps could be undertaken to mitigate this problem. One possible solution would be to use certification schemes to ensure the authenticity and

integrity of the exchange keys. However this would also require the presence of trusted third parties, which was one of the things we wanted to avoid to sustain the paradigm of easy service composition. An alternative approach could be to visualize the fingerprints of the exchanged public keys in each of the widgets. This would enable users to spot a man in the middle attack due to inconsistencies in the fingerprints. However, this approach does require users that are at least partially familiar with the basic functionality of asymmetric encryption. According to [18], this might be too much to expect. Even if both of these approaches are not feasible there is still some hope that such an attack would be spotted in an open source environment.

7 Conclusion

We presented a proof-of-concept implementation enabling the secure usage of a mashup-providing platform, protecting sensitive data against malicious widgets and platform operators. We gave the design rationale, implementation details, and discussion of merits and limitations including possible application scenarios. Benchmarking of several possible approaches against each other is in process, but the first results from the proof-of-concept presented here are very encouraging.

Acknowledgments. This work was supported by the German Federal Ministry of Education and Research (BMBF) under Grant Number 01BS0824 (COCKTAIL).

References

1. Zibuschka, J., Herbert, M., Roßnagel, H.: Towards Privacy-Enhancing Identity Management in Mashup-Providing Platforms. In: Foresti, S., Jajodia, S. (eds.) *Data and Applications Security and Privacy XXIV*. LNCS, vol. 6166, pp. 273–286. Springer, Heidelberg (2010)
2. Zarandioon, S., Yao, D., Ganapathy, V.: OMOS: A Framework for Secure Communication in Mashup Applications. In: *Proceedings of the 2008 Annual Computer Security Applications Conference*, pp. 355–364. IEEE Computer Society (2008)
3. Hoyer, V., Fischer, M.: Market Overview of Enterprise Mashup Tools. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) *ICSOC 2008*. LNCS, vol. 5364, pp. 708–721. Springer, Heidelberg (2008)
4. Hasan, R., Winslett, M., Conlan, R., Slesinsky, B., Ramani, N.: Please Permit Me: Stateless Delegated Authorization in Mashups. In: *Proceedings of the 2008 Annual Computer Security Applications Conference*, pp. 173–182. IEEE Computer Society (2008)
5. Crites, S., Hsu, F., Chen, H.: OMash: enabling secure web mashups via object abstractions. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 99–108. ACM, Alexandria (2008)
6. Jackson, C., Wang, H.J.: Subspace: secure cross-domain communication for web mashups. In: *Proceedings of the 16th International Conference on World Wide Web*, pp. 611–620. ACM, Banff (2007)
7. Keukelaere, F.D., Bhola, S., Steiner, M., Chari, S., Yoshihama, S.: SMash: secure component model for cross-domain mashups on unmodified browsers. In: *Proceeding of the 17th International Conference on World Wide Web*, pp. 535–544. ACM, Beijing (2008)

8. Zarandioon, S., Yao, D., Ganapathy, V.: Privacy-aware identity management for client-side mashup applications. In: Proceedings of the 5th ACM Workshop on Digital Identity Management, pp. 21–30. ACM, Chicago (2009)
9. Leyden, J.: One in five workers still clinging to IE6 - The Register, http://www.theregister.co.uk/2010/08/19/zscaler_web_security_study/
10. Guan, Z., Cao, Z., Zhao, X., Chen, R., Chen, Z., Nan, X.: WebIBC: Identity Based Cryptography for Client Side Security in Web Applications. In: International Conference on Distributed Computing Systems, pp. 689–696. IEEE Computer Society, Los Alamitos (2008)
11. Hansen, M., Berlich, P., Camenisch, J., Clauß, S., Pfitzmann, A., Waidner, M.: Privacy-enhancing identity management. Information Security Technical Report 9, 35–44 (2004)
12. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21, 120–126 (1978)
13. Rijmen, V., Daemen, J.: The Design of Rijndael: AES. In: The Advanced Encryption Standard. Springer, Berlin (2002)
14. Dreamface: DreamFace Interactive, <http://www.dreamface-interactive.com/>
15. Versaneo GmbH: pidCrypt - pidder's JavaScript crypto library, <https://www.pidder.com/pidcrypt/>
16. Solario, G.C., Barulli, M.: Clipperzlib Javascript Crypto Library, <http://sourceforge.net/projects/clipperzlib/>
17. Oka, A.: JavaScript Cryptography Toolkit, <http://ats.oka.nu/titaniumcore/js/crypto/>
18. IBM: IBM Mashup Center, <http://www-142.ibm.com/software/products/de/de/mashupcenter/>
19. JackBe: JackBe Mashup Editor and Composer, <http://www.jackbe.com/products/composers.php>,