Securing Restful API Services

Lance Tyler, Matt Morris

CSC 458: Cryptography

Dr. Zachary Peterson

December 1, 2014

Contents

1 Introduction

1.1 Motivation

Many student developers find it difficult to turn their simple application idea into usable code. They may either not have the time it takes to develop a new code base from the ground up, or they may not have the expertise in the numerous systems required to create a new product. Our web service helps solve this problem by providing generalized functions used by social applications, such as managing user associations, managing users, and managing user data. Instead of having to setup their own database and server, application developers now simply use our public web service as a backend when creating their application.

As an extension to our senior project, we chose to implement some common practices in securing a web service for our final research project. Developers using our service will benefit by not having to design or implement any of the security measures required for a web service. Clients using the applications developed using our product will benefit by knowing that their data is secured by people with experience in cryptography.

1.2 Securing Web Services

When securing our web service there were five main attacks we chose to defend against: unauthorized access, parameter manipulation, network eavesdropping, disclosure of configuration data, and message replay attacks. [2] The sections below summarize what these attacks are and how they apply to the web service we are protecting.

Unauthorized Access A web service should restrict access to any sensitive data that it stores. In our service these resources are the associations between two users, and data that a user owns. Only a properly authenticated user may create an association with another user, or set the visibility of their data. [2]

Parameter Manipulation In our service application developers program what parameters are passed to the web service. A malicious adversary may modify the uniform resource identifier (URI) in an attempt to gain access to restricted resources. Our URI is modeled after Restful resource naming practices:

```
/serverIP/ApplicationName/
    userID/resource
```

The resource is either associations, or data. This means that only the client who authenticates as the userID may access these resources.

Network Eavesdropping An adversary with access to the connection between the client and the server may be able to intercept and collect sensitive data. In todays world we can never be quite sure who can see our public information when using unencrypted means

of communication, so preventing passive eavesdroppers is a must.

Disclosure of Configuration Data Many RESTful frameworks are set by default to display error information, most notably uncaught exceptions, to clients who causes the error in question. This can disclose potentially sensitive information to users of the service who shouldn't see that information. At a minimum, disclosing this information will allow adversaries to learn what specific framework your service is using, which could allow them to hone in on framework-specific vulnerabilities. In a worst case scenario, disclosing certain info could lead to something like a padding oracle attack.

Message Replay Even if requests to a RESTful service are encrypted, there is still nothing in place that stops some eavesdroppper from collecting packets and simply re-sending copies of them to their destination. This would obviously be paramount in things like banking systems, but it is also important for general security for less critical applications as well.

2 Design

2.1 Confidentiality

Encryption In securing our web-service, we chose to provide confidentiality by running our service over TLS 1.2. The largest implications of this are that our service should theoretically have complete confidentiality of its communications with users of the service. The cipher suite used by the SSLContext is $tls_ecdhe_rsa_with_aes_128_cbc_sha$. Although this is not the best cipher suite due to the CBC component of it, this was unfortunately the best out of the box option availible for our deployment server.

Replay Attacks Another benefit of using TLS is that our service is protected from message replay attacks. While our messages are still encrypted, nothing is stopping from an adversary from storing our request packets and re-sending them to contrive some sort of attack, or simply to cause harm to the system. Luckily, TLS inherently protects against message replay attacks by incorporating its own HMAC primitive. By including a sequence number in each packet that monotonically increases with each packet sent, the TLS protocol can effectively discard any replayed packets, since the replayed packet will have a sequence number which would be lower than the current sequence number (meaning it has been seen before). The TLS HMAC is computed from the TLS MAC secret, the sequence number, the message length, the message contents, and two fixed character strings. [6] Factoring this implication in, our service should now provide total encryption of our communications, as well as assurance that our encrypted packets cant simply be replayed for malicious purposes.

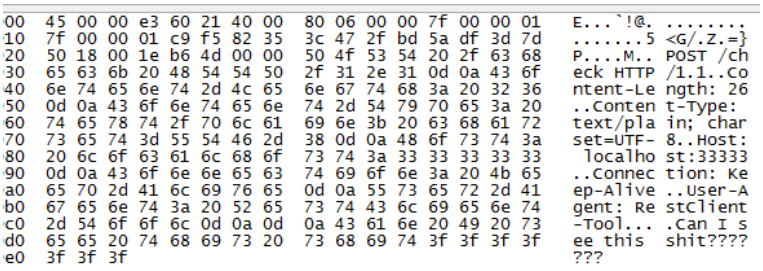

Figure 1: Wireshark screen shot of no encryption used

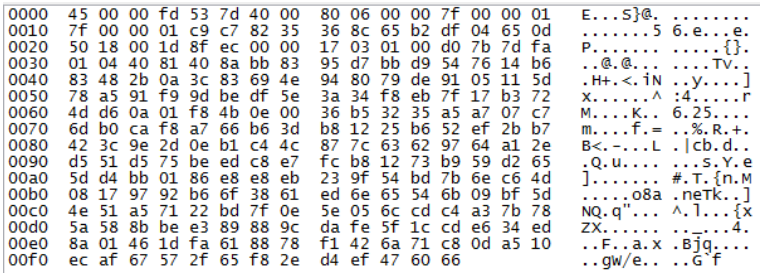

Figure 2: Wireshark screen shot of TLS encryption

Verification of TLS To ensure that TLS was turned on for our server we used Wireshark to sniff packets that were being sent to it. See figure 1 and 2. Although "scrambled" data is not a good indication of if something is enncrypted, we trust that the creators of the SSLContext class properly implemented TLS.

2.2 Authenticity and Integrity

The method we chose to prevent unauthorized access and to prevent parameter manipulation was to use the hashed based method authentication code (HMAC). An HMAC is a combination of a MAC and a hashing algorithm. According to authors of RFC2104 the cryptographic strength of the hash, the size and quality of the key, and the size of the output, all contribute to the strength of the HMAC. [1] The HMAC can also provide authentcity of the origin of the resource request. The sections below illustrate the problems we encountered when desgining a system that uses an HMAC, and the solutions to those problems.

2.3 Generating, and Transporting the Shared Key

Registation Before the client uses the service, they must first register over the TLS channel with a user ID, and a pass-

word. It is best practices to hash and salt a password before storing it in a database. For our implementation we used the SHA-256 hashing algorithm with a 128 bit salt generated from the CSPRNG found in java.security.SecureRandom. In the event that our database were to be compromised, user passwords would still be secured. This is due to the fact that a hash function is one way, meaning the original input of the function can never be derived from the output. Salting the user passwords protects against techniques such as rainbow table, or lookup tables. This is when an adversary precomputes the hashes to passwords ahead of time, allowing them to simply do a reverse lookup of a hashed password to find the original inputted password. [5]

Authentication To get the shared key a user must first authenticate with their user ID password pair over TLS. To prevent against a brute force attack against a users password we have enforced a rate limit of 5 authentication attempts per user ID every 15 minutes. The goal of the rate limited authentication is to prevent any brute force guesses against the users password. This allows an adversary only $4(60min/15minslockedout) * 5(attempts) = 20attemptsperhour$. A password minimum length of 8 characters is also enforced. The result is the adversary does not have an efficient way to guess the entire $3.026 * 10^15$ passwords. [4]

Generating Shared Key As part of the requirement for generating an HMAC, a shared key must be established between the server and the client. During the authentication process the server generates a shared key, which it returns to the user, and stores in its database. These keys are generated using the java.crypto.SecureRandom, a FIPS compliant random number generator.[3] This key is ephemeral meaning it is only recognized as valid by the server for only 1 hour. The key the client receives may be incorporated into a session cookie by the application developer.

Generating the HMAC

$HMAC(K, m) = H((K \oplus opad)|H((K \oplus ipad)|m))$
$H = HashFunction$
$K = Key$
$m = message$
$opad = outerpadding(x5cblock)$
$ipad = innerpadding(x36block)$
[1]

An HMAC requires that the underlying hash function used to generate it does not have any targeted collisions. For generating the HMAC we chose the SHA-256 hash function which doesn't have any documented collisions. [1] To ensure a properly formated HMAC we chose to use Java code from an AWS HMAC-SHA256 function. For the client we chose the crypto-js implementation of the HMAC-SHA256 found at: https://code.google.com/p/crypto-js/. Using the shared key acquired from

authenticating with the server, the client signs each requests (HTTP verb + URI of the resource + payload).

Timing Attack Prevention Upon receiving a resource request, the server will recreate the HMAC using the same information the client used, and compare the two HMACs. To prevent any timing attacks against the server we implemented our own .equals method for comparing the two HMACs that runs at constant time.

Restricting Resource Access In addition to ensuring that the client has not tampered with the parameters of the resource request, the HMAC also allows the server to verify the identity of a client requesting access to a resource. The server uses the following logic to determine if the request is allowed to access the resource:

```
userId -> from URI
HMAC -> found in HTTP header
SharedSecret ->
  dbLookup(userID)

if SharedSecret is null:
    unauthorized()

ServerHMAC ->
  computeHMAC(URI, SharedSecret)
if HMAC = ServerHMAC:
  allowAccess()
else:
  unauthorized()
```

If the server does not find a valid shared secret for that userID, it means the client requesting access to the resource is not authenticated. If the HMAC computed by the server does not match the HMAC sent by the client, then the client does not have the correct shared secret, and is not authorized access to the resource.

3 Conclusions

For this project we learned the importance of securing a web service. The big take away was that SLL/TLS is not that difficult to implement in a server, and should be used for protecting the users of a web service. We also learned how to authenticate users, as well as other security methods for managing user credentials.

References

[1] R. C. H. Krawczyk, M. Bellare, "Hmac: Keyed-hashing for message authentication," February 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2104.txt

RFC 2104, HMAC, description

[2] Microsoft, "Building secure web services," June 2003. [Online]. Available: http://msdn.microsoft.com/en-us/library/ff648643.aspx

guide to building secure web services

[3] Oracle, "Class securerandom." [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html

API document on Java's CPRNG

[4] T. Pornin, "How many possible combinations in 8 character password." [Online]. Available: http://math.stackexchange.com/questions/739874/how-many-possible-combinations-in-8-character-password

Stackechange answer on number of passwords for 8 chars

[5] D. Security, "Salted password hashing - doing it right." [Online]. Available: https://crackstation.net/hashing-security.htm

How to salt and hash passwords correctly

[6] E. R. T. Dierks, "The transport layer security (tls) protocol," April 2006. [Online]. Available: http://tools.ietf.org/html/rfc4346#appendix-F.2

RFC 4346, TLS, description