API Keys to the Kingdom

Stephen Farrell • *Trinity College Dublin*

Many Web 2.0 services offer Web 2.0 APIs for developers to use. In this article, I review one of the security mechanisms that's often included in such Web 2.0 APIs — the use of API keys — and some of the deployment issues associated with their use.

Web 2.0 APIs

From a security viewpoint, Web 2.0 APIs are different from more traditional APIs that programmers use when calling a library from a running process. In the case of traditional APIs, significant trust boundaries aren't often crossed due to the API call because both the calling and called code generally have access to the same memory and so are effectively equivalent in terms of security.

However, with Web 2.0 APIs, we're really dealing with application *protocol* interfaces because they involve accessing some application functionality over HTTP. This generally involves crossing various trust boundaries. For example, the calling code could be JavaScript running in a browser environment, the called code might be running on some Web server machine in a demilitarized zone (DMZ), and the HTTP requests and responses used to implement the API might pass through various HTTP proxies between the calling and the called code.

The typical Web 2.0 API involves a client calling a service with parameters passed as a mixture of query string parameters — that is, as part of the URI in the HTTP request — or as XML or even tag-value pairs inside `POST` data. In most cases, responses consist of XML-structured data passed in the body of the HTTP response, with a variety of MIME content types.

Many Web 2.0 services provide APIs in various "flavors," which offer equivalent functionality via HTTP or SOAP. With SOAP APIs, some security standards (see www.w3.org/TR/SOAP-dsig) are available to cover authentication and authorization. However, in many cases,[1] developers apparently find SOAP-variant APIs to be too resource intensive, so the HTTP or Representational State Transfer (RESTful) variants tend to be more commonly used. Variants of these APIs exist that use JavaScript Object Notation (JSON) for one or both request or response parameters, rather than pass parameters as XML-structured data. In neither the HTTP case nor the JSON case are there standards for how to use cryptography to provide security services specifically for the API calls.

Of course, many security issues associated with the use of Web 2.0 style development and APIs exist. However, in this article, I only consider the use of API keys. So, for example, this article doesn't consider issues related to cross-site scripting attacks or other implementation vulnerabilities. As a matter of terminology, I refer to the service provider as the entity that supplies a service via the API and the API consumer as the entity that calls the API to use the service.

API Key Mechanism

With many Web 2.0 APIs, service providers might wish to constrain or control access to the service by issuing what they call API keys to approved partners. Sometimes, service providers require the API key to use the service. In other cases, it's optional because the service provider simply uses it for logging and to contact the API consumer in the event of failures. Other cases arise in which the API key acts like a password that grants access to the service, and in yet other situations, the service provider uses a secret associated with the API key as a cryptographic key to both authenticate the API consumer and to partially protect API requests.

There is often some loose use of terminology associated with API keys (as you can see from the last paragraph). Sometimes the API key is essentially a hopefully hard-to-guess combined

1089-7801/09/$26.00 © 2009 IEEE

account identifier and password; at other times, it's used as a secret key identifier, in which the secret-key is either distributed out-of-band or in-band via some other API call.

When a service provider uses an API key and secret to authenticate and integrity-protect an API request, many API specifications talk of using the secret to "sign" the request or talk about generating a "signature." Cryptographically, these are in fact not signatures (which require the use of asymmetric algorithms like RSA) but rather are message-authentication codes (MACs), which, in most APIs, only partially protect the API request.

One common method is to catentate the API key (acting in this case like a key identifier) with some request parameters, append the secret, and then input that string to the MD5 hash algorithm. For example, the last.fm API (see www.last.fm/api/webauth) uses this, but it's commonly used elsewhere. This particular way of generating MACs is cryptographically weak, especially because the MD5 algorithm is essentially broken for collisions so that an attacker could probably, with some work, generate different API requests that would differ but use the same MAC value.

Although such a weak MAC scheme is cryptographically undesirable, developers have justified their use in the past by claiming that they should be simple so that they can easily implement them in the broad range of environments that API consumers require — which include PHP scripts and other less capable development environments. However, at a minimum, APIs should use a better MAC algorithm — for example, HMAC-SHA1 (http://tools.ietf.org/html/rfc2104/).

It's also worth noting that service providers generally use these schemes only to partially authenticate requests, and responses are totally unprotected or else protected only via Transport Layer Security (TLS). The result is that requests are vulnerable to manipulation, and responses are vulnerable to spoofing. The use of partial protection for requests also raises an issue of understanding — developers using, defining, or modifying an API definition might not understand which fields are actually protected and which aren't, and this can lead to vulnerabilities. This is especially true because API consumer developers aren't security experts, nor should we expect them to be.

API Keys and SSL/TLS

When developers use API keys in conjunction with the Secure Sockets Layer (SSL) or TLS protocols, a couple of issues arise. Many Web deployments don't actually terminate the SSL or TLS session at the API server. Instead, they terminate in load-balancing equipment an IP hop (or so) upstream from the API server. The point is that the API server that handles sensitive information is then trusting that no other route exists for HTTP requests to arrive at that API server — other than via the load-balancing device. It also trusts the load-balancing configuration to enforce the requirement that SSL or TLS is enabled and properly used.

The consequence is that many API designers, in fact, simply assume that requests and responses are already sufficiently protected for integrity and confidentiality. Hence, they take no steps to directly protect sensitive information passed via the API. Although this approach is probably justifiable, given many Web 2.0 services' current infrastructure and relative lack of sensitivity, it might not be appropriate in the future as we access more sensitive data via Web 2.0 APIs or use the services provided on larger and larger scales.

A second issue relates to how API consumers manage certificate and certificate-authority information. If the API consumer is actually running as a service itself, then using server-authenticated TLS (the norm between browsers and Web sites when using HTTPS) requires that the API consumer use a TLS library that's properly configured with information about the certification authority that issued the service provider's certificate.

Because most TLS libraries also support mutual authentication based on certificates, service providers and such API consumers should be able to use this much stronger form of authentication — no intrinsic reason exists why managing certificates for API consumers should be any harder than managing API keys and secrets. In fact, one could argue that managing API consumer certificates would be better because the service provider would no longer be vulnerable to accusations of using API consumer's secrets, thanks to the nature of the public key cryptography that mutually authenticated TLS uses. That said, there would still be the issue of terminating the TLS session at the load balancer, in which case the load balancer would have to assert the API consumer's identity to the service.

Key Life Cycle

Given the API key scheme's broad use and likely continuation for some time to come, its worthwhile to consider the life cycle of these API keys and secrets, just as we would any cryptographic key or password.

The first point is that, because API keys are intended for developers, it's likely there will be a stage in which developers use the API during development before using it with a "live" service. To reduce the exposure of API key values, service providers should have a set of development API keys and secrets for such a situation. When the API consumer has debugged their implementation using the development version of the API key, they can then switch over to use "live" API key values. In many operational environments, this sim-

ple step can significantly reduce API keys' exposure because different people will be involved in developing and operating the live service. So, the developers need not access the API keys and secrets that will be used in the live service.

Of course, we should only securely transfer API keys and secrets — for example, using encrypted mail systems such as S/MIME, which can be easily set up for the numbers involved here, or via some Web site accessed over TLS (preferably with mutual authentication). However, it's important to remember that these values are as sensitive within API consumer organizations as they are when sent between the service provider and the API consumer. In fact, API keys and secrets transmitted via internal email or version-control systems are probably at their most exposed, given the type of infrastructure that many API consumers typically use.

Second, it should be possible to change API keys and secret values without impacting the operational service. This implies that some kind of key versioning is required and that API consumers should provide tools to let consumer developers add new API keys and secrets and remove or deprecate old ones. Service providers should publish a policy on how they manage API keys and secrets in this respect and could, for example, reduce the level of access granted for old or development keys compared to the set of currently "live" ones. However, when supporting changes in these values, API consumers must also be vigilant to not introduce new phishing opportunities — for example, via user interfaces that accustom end-users to seeing changes that a phishing attack could exploit.

All these issues require service providers to establish some, albeit lightweight, infrastructure for key management, but that's an inherent part of offering a secure service, there being no free lunches.

API Keys and Open Source

The Web 2.0 services that use the API key approach have a problem with open source implementations that might have to include API key values in the code. They also have a problem with downloadable applications in which they must embed the API key values into the distributed application. Consumer developers can handle this situation in a couple of ways, depending on the type of API consumer.

For API consumers that run as a service (for example, as part of a Web application framework), it might be reasonable to include a development API key with the open source version and require that the installer request a new "live" API key from the service provider. In this case, the keys that the consumer developers distribute with the code are similar to the development keys I referred to earlier and could reasonably result in a lower level of access compared to live keys.

However, in other cases, the API consumer might be a downloadable application (or plug-in) that's intended for end users. In such cases, it's hard to simultaneously meet security and usability requirements if the downloaded application must have access to the API key values. One possibility would be for the API consumer to offer a service that authenticates the user, so that the API key values remain under the consumer's control. However, this, again, could cause deployment, performance, or usability issues. In the end, the API key scheme doesn't really handle this case well. Therefore, we should really consider such API key values to be publicly available because, for a popular application, even obfuscating the key values in the downloadable application won't keep them confidential.

Going Forward — OAuth

So, I've described a range of ways in which service providers and API consumer developers use this API key scheme in practice, with significant variability both in terms of functionality and security. So, you might wonder whether any standards-development activities are relevant here, and the answer, as usual, is "yes." Probably, the main current focus in this area relates to the OAuth specification (www.oauth.org), which various service providers are successfully using with quite a large range of API consumers. The IETF is also developing a standards-track set of RFCs for OAuth (see www.ietf.org/dyn/wg/charter/oauth-charter.html) that should significantly improve its security and interoperability properties so that service providers and API consumers can use the resulting specification to meet the goals of the various ad hoc API key schemes currently in use.

I recommend that service providers and API consumers look at the OAuth specification when developing new Web 2.0 APIs. Additionally, they should consider whether they can, in fact, use mutually authenticated TLS. In any case, service providers should provide some infrastructure for managing API key values' life cycles to reduce their systems' vulnerability.

Reference

1. C. Pautasso, O. Zimmermann, and F. Leymann, "Restful Web Services vs. Big Web Services: Making the Right Architectural Decision," *Proc. 17th Int'l Conf. World Wide Web* (WWW 08), ACM Press, 2008, pp. 805–814; http://doi.acm.org/10.1145/1367497.1367606.

Stephen Farrell is a research fellow at Trinity College Dublin and chief technologist with NewBay Software. His research interests include security and delay/disruption tolerant networking. Farrell has a PhD in computer science from Trinity College Dublin. Contact him at stephen.farrell@cs.tcd.ie.