Leaky or Guessable Session Identifiers

Stephen Farrell • *Trinity College Dublin*

Many Internet and Web applications use session identifiers. Too often, developers of those applications make the bad assumption that all is well because session identifiers are only known to authorized users. However, in many cases, session identifiers can leak out or be guessed, sometimes trivially. If presenting an identifier is the only authorization an application requires, it can represent an easily exploited vulnerability. Although these vulnerabilities are old and well-known, some recent examples of problems arising from them show that developers must remain on guard against them.

Some Real Examples

Internet and Web applications employ identifiers for users, hosts or services, and sessions. I'm concerned here with their use as part of an authorization solution, rather than for user or service authentication. The most common implementation of session identifiers today is probably as values stored in "cookies" for Web applications or their equivalents. One recent bad example of handling session identifiers relates to a small office/home office broadband router that uses session identifiers with insufficient randomness in a Web-based administrative interface. Basically, attackers can guess these identifiers because they depend on only a sequential counter and what looks like a time stamp with a 1-second granularity. The YGN Ethical Hacker Group reported this vulnerability in early August 2010 (http://yehg.net/lab/pr0js/advisories/2wire/[2wire]_session_hijacking_vulnerability), and the vendor incorporated a fix into its release later that month. However, such products are rarely, if ever, patched, so the vulnerability likely remains exploitable for quite some time.

Although major websites might be unlikely to use such simple session identifiers, other hosts are on the path between users and those websites, and the developers of those hosts might make such mistakes, even though the problems of simply-constructed session identifiers have been known for many years. The proliferation of devices such as smartphones and home gateways and the lack of an automated software update mechanism for many of them will unfortunately likely ensure that such vulnerabilities are with us essentially forever.

This example is simple, but there are less-obvious cases in which identifiers can be guessed. For example, in March 2010, Andreas Bogk reported a vulnerability in how PHP generates session identifiers (http://seclists.org/fulldisclosure/2010/Mar/519). In this case, the attacker's job is fairly complex but still within the capabilities of any significant sized enterprise. Luckily, there's a well-defined method of avoiding the vulnerability by configuring a separate source of randomness (www.phparch.com/2010/04/09/possible-vulnerabilities-found-in-php-session-ids). Nonetheless, such issues continue to arise, even with well-tested systems like PHP.

As an example of a leaked session identifier, iSecPartners, a consiulting firm, reported in April 2010 that Twitter's Web session identifiers are sometimes sent via unsecured HTTP (rather than TLS) so that Web proxies can copy them and use them to log in (www.isecpartners.com/advisories/2010-001-twitter.txt). The report also claimed that these session identifiers have the same lifetime as a user's password, so leaking them remains dangerous until a user changes his or her password. To make matters worse, the DNS entry for Twitter was reportedly hijacked temporarily in December 2009. During that time, Twitter's clients would have sent their session identifiers, in the clear, to whatever address the hijackers chose.

There are two issues here. One is that the session identifier's lifetime is very long. The second is that, because those values are sent in the clear, the end user can't validate the identifier's destination. However, a service like Twitter's, which is available on so many different platforms (including mobile devices that can't use TLS) and has such a huge user population, might reasonably continue handling session identifiers this way, given the costs of making changes. Additionally, few users check whether they're connecting to the right site because user interfaces have trained them to follow the bad practice of just clicking through certificate warnings. Nonetheless, it seems fair to criticize such a system. So far, to my knowledge, Twitter hasn't publicly responded to this report, although it might, of course, have put in place countermeasures.

This case shows how what we call session identifiers sometimes can be valid for much longer than what you might consider to be a session's lifetime. This is probably driven by the real user requirement to avoid constantly reauthenticating. However, plans for meeting that requirement should use short-lived identifiers with some secure mechanisms for automated renewal.

More generally, any system that passes such session identifiers as HTTP GET parameters via nonsecured HTTP (that is, not over TLS) is vulnerable because websites could store logs or malicious proxies that are en route to the intended destination in those values (http://cwe.mitre.org/data/definitions/598.html). So, it's unlikely that Twitter is alone in suffering from such problems.

Randomness, Entropy, and Identifier Length

Before considering the obvious question of how a developer can do session identifiers right, I'd like to clarify a little about randomness and entropy. To start, think about a fairly random string of bits. If I compress the string using some kind of perfect compression algorithm, the compressed string's length is the entropy from the original string. When considering values that attackers can guess, it's the entropy and not the original string's length that's important to consider.

For a session identifier to be hard to guess, it must appear random, have reasonably good entropy, and have a sufficiently large set of possible values from which it's selected. If the value isn't sufficiently random, then analysis of the values used (which is usually easy to do) can show us a way to trivially guess a good value. If the set of possible values isn't sufficiently large, attackers can simply try each possible value in turn until they find an acceptable one.

Developers often select bad sources for randomness, in particular for seeding pseudorandom number generators. Simply using a process identifier and the local time in seconds isn't sufficient. RFC 4086 (http://tools.ietf.org/html/rfc4086) is a good source of guidance here, and most cryptographic toolkits (for example, openssl; www.openssl.org) also offer good sources for randomness, if used correctly.

It's also important to note that attackers seldom care much about which value they guess. They can probably win the game with any authorized session identifier, and, if one session identifier is guessable, usually many are. So the number of currently valid session identifiers also comes into play — another reason for preferring short-lived session identifiers. The Open Web Application Security Project website (www.owasp.org/index.php/Insufficient_Session-ID_Length) includes a reasonable description of why session identifiers should be around 128 bits long. Specifically, with a 64-bit identifier that includes only 32 bits of entropy (say, because of encoding or other issues), a reasonably large botnet or enterprise could find a valid session identifier within a few minutes. Although that level of attack is perhaps unlikely for many websites, high-profile or financial websites could be subject to such attacks. As a developer, it's better not to assume that your code will never be used in such a context.

Some Recommendations for Developers

So, how should a developer properly handle session identifiers? The details will depend on the application, but it's generally good practice to ensure that session identifiers are as short-lived as possible. That way, if they do leak out, the impact is minimized. If possible, it's preferable to automatically and frequently regenerate session identifiers, particularly if some significant part of the session state has changed (for example, if users want to modify their stored settings or move from the website's shopping page to its checkout). Wherever possible, session identifiers should be limited to secure TLS (or HTTPS) transport so that they're less vulnerable to theft. Indeed, there's really no good reason these days to manage session states at all via unsecured connections because everything required for securing sessions with TLS is readily available.

Session identifiers should also have sufficient entropy to make guessing impractical in all circumstances that the developer can envisage, not just for current use cases. Code to generate and check identifiers should also be part of a generic library that gets reused. You don't want to redo this kind of development for each new application, and you do want to make sure that whoever writes and tests this code knows something about the relevant threats.

If I were developing a generic session-identifier-handling library, I would probably consider encrypting all the identifier content and including a message-authentication code (MAC) within the plaintext before encryption. Because the same application servers usually both encrypt and decrypt, using a strong symmetric key and the Advanced Encryption Standard (AES) algorithm is probably sufficient. Load-balancing and other types of redundancy add a bit of work to this, as does the requirement to enable encryption key changes without disturbing the service, but there are well-known ways to handle those issues. If possible, use some kind of hardware support to store keys. Storing them in files or a database makes it all too easy for an administrator to accidentally or deliberately leak keys. Finally, as I've mentioned previously in this column, use different keys for testing and the live service.

In many cases, a trade-off exists between including data within a session identifier and including just handles or pointers to a database or other store. Doing the former reduces the work on the server when it receives a session identifier but also makes the identifiers bigger (usually undesirable), and means that you can't easily change or revoke data within the identifier while the identifier is still valid. The right balance here will be application specific, so the generic session-identifier library will probably have an interface that lets different applications include different pieces of optional data within the identifier.

Session identifiers can also contain values that attempt to enforce session continuity (for example, a source IP address). The goal here is to prevent session-identifier theft. However, many sessions will appear to come from the same IP address (usually an outbound HTTP proxy), so this technique has limited value. Furthermore, it can cause problems if a user roams or has a multihomed host, which is becoming more common these days with devices supporting both 3G and Wi-Fi. And finally, unless the session identifier is properly protected — that is, encrypted and containing a MAC — attackers can fake any such values.

It's important to note that attackers seldom care much about which value they guess.

However session identifiers are handled, developers should carry out some form of review before deploying the code. Typical issues that arise include key storage (if identifiers are encrypted) and key life-cycle management. If different contexts require different identifier encoding and interfacing to applications, a generic session-identifier library might not offer all the right interfaces that applications require. The main thing to consider is, that even though the developer probably designed a generic session identifier for just one or two applications, people will likely use it again in other contexts.

If the applications that use session identifiers handle more sensitive data, such as financial information, then they'll quite likely undergo external security review — for example, for conformance to the Payment Card Industry Data Security Standard (PCI DSS; www.pcisecuritystandards.org/security_standards/pci_dss.shtml) or other security standards. In such cases, developers should review any relevant security standards before starting work and not (as is common) attempt to retrofit conformance, which is often problematic. PCI DSS annual reviews have also been known to show up false positives (http://kb2.adobe.com/cps/404/kb404762.html) related to handling session identifiers. And, although a developer might consider modifying code in such cases to be a gratuitous change, in the long run, it's better to conform to avoid the cost of an annual discussion with external security reviewers.

What about Users?

How can users tell whether a random-looking session identifier is actually simply a base64-encoded version of the one-and-only password they use for everything? In general, unfortunately, they can't and so must simply decide whether to trust the system that's generating the session identifiers. This is usually not going to be a very well-informed decision, so the best they can do is to use a different name and password for each service, though that's often more work than they're willing to do.

The PCI DSS standard attempts to provide more assurance to users in such cases. Merchants that fail a PCI DSS review face fines for noncompliance, so they're generally motivated to comply (www.rbsworldpay.com/pcidss/index.php?page=penalties&l=x). However, outside the financial community, no such standards are enforced. Of course, compliance with PCI DSS only means that the merchant handles credit-card information properly. It doesn't mean that all other aspects of the merchant's system are similarly robust.

Perhaps a more interesting case is a partnership in which one organization is using, or depending on, the session identifiers the other generates. In this case, you might expect a better review process, but the complexity of linking various systems can mean that even what's retrospectively a really obvious flaw can still occur. Perhaps the best publicized recent case was the AT&T/Apple iPad flaw (www.guardian.co.uk/technology/2010/jun/10/apple-ipad-security-leak) in which guessing the identifier on a subscriber-identity-module card was sufficient to get a user's email address. The actual information leaked in that case was less damaging than it could have been, but the scale of the leak (more than 100,000 email addresses) generated enough bad publicity to serve as a warning to carefully review your partners' security measures, as well as your own, when linking complex Web services.

Despite fairly simple and well-known ways to properly handle session identifiers, sloppy development practices continue to result in the use of insecure session identifiers on the Internet. Some kind of standard secure session-identifier format that many tools can use might make such mistakes less likely to occur in the future. However, I'm aware of no such activity currently under way, in spite of how many systems handle session identifiers. Perhaps a start in that direction would be for developers of popular Web applications and toolkits to publish specifications for how they handle session identifiers. From this, a de facto standard might begin to emerge.

Stephen Farrell is a research fellow at Trinity College Dublin and cofounder of Tolerant Networks Limited. 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.