HACKING WEB APPLICATIONS USING COOKIE POISONING

AMIT KLEIN, FORMER DIRECTOR OF SECURITY
AND RESEARCH, SANCTUM

A whitepaper from Watchfire

TABLE OF CONTENTS

Except as expressly agreed by Watchfire in writing, Watchfire makes no representation about the suitability and/or accuracy of the information published in this whitepaper. In no event shall Watchfire be liable for any direct, indirect, incidental, special or consequential damages, or damages for loss of profits, revenue, data or use, incurred by you or any third party, arising from your access to, or use of, the information published in this whitepaper, for a particular purpose.

www.watchfire.com

SUMMARY

Cookie poisoning is a known technique mainly for achieving impersonation and breach of privacy through manipulation of session cookies, which maintain the identity of the client. By forging these cookies, an attacker can impersonate a valid client, and thus gain information and perform actions on behalf of the victim. The ability to forge such session cookies (or more generally, session tokens) stems from the fact that the tokens are not generated in a secure way.

In this paper, we explain why session management (and session management security) is a complex task (which is why it is usually left for commercial products). We describe how the tokens are generated for two commercial application engines. We then analyze the strength of each mechanism, explain its weakness, and demonstrate how such weakness can be exploited to execute an impersonation/privacy breach attack. We discuss the feasibility of the attack. Finally, we recommend an approach to session management which separates the security from the functionality – the latter is carried out by application engines, while the former should be provided by a dedicated application security product.

THE SISYPHEAN IN-HOUSE SESSION MAINTENANCE

In web application programming, Session Management is complex and awkward. The programmer needs to worry about many aspects of session management which can defocus him/her from the main goal – implementing the business logic that makes the site unique and profitable.

Specific issues are:

- Session creation and identification – how to ensure that when a new session is needed, it is indeed created? The programmer must identify that a client has a need for a session, create the session and assign the client a session.
- Concurrency issues – when two clients access the site simultaneously, each requiring a new session, it is necessary to make sure that the session creation process will still function correctly.
- Session termination and timeout – what triggers a session termination? How are the resources of the terminated session recycled? What happens if the client tries to access the site when the termination process is taking place? What happens when a client tries to access a site with a stale session?
- Session data storage, multiple servers, fail-over – where is the session data stored (on disk? in RAM?)? What is the performance penalty? What happens in a multi-server site if a client accesses a first server (and establishes a session with it) and then is directed (by a load balancer) to a second server? What happens to the client session data in case the original server crashes?

Security-wise, the following considerations must be made:

- It should never be possible for one client to be able to predict the token another client received, or is in the process of receiving, or will receive. This is obviously a 'must have' in order to prevent impersonation attacks and consequently breach of privacy.
- Furthermore, it is desirable that a client will not be able to predict the next token he/she will get when accessing the site. This is useful in minimizing the damage of stealing the token while it travels (in the clear) to and fro, and while it is stored on disk at the client.
- Any token should have a reasonable expiration period – again, to minimize the damage of it being stolen.

As can be seen, it is not very easy to fulfill all these requirements, especially if the session mechanism is developed ad-hoc. The more intricate security requirements are definitely something developers, especially ones not versed in security, may easily miss.

There are many examples of insufficient security in tokens, which were demonstrated in the work of MIT Laboratory for Computer Science ("Dos and Don'ts of Client Authentication on the Web" by Kevin Fu, Emil Sit, Kendra Smith and Nick Feamster) http://cookies.lcs.mit.edu/pubs/webauth:tr.pdf.

So we see that it is difficult to come out with a good session management solution, let alone a *secure* session management solution. This is one of the reasons why application servers are so popular.

APPLICATION SERVERS/ENGINES – A SOLUTION AND A PROBLEM

An Application Server (or Application Engine) is a software program designed to make the life of the application developer easy. It usually offers the programmer the ease of writing HTML pages with directives for the server embedded in them, instructing the server to perform various tasks. Most application servers provide the programmer an environment that takes care of the session automatically, relieving the programmer from all the worries mentioned in the above section.

Examples of application servers:

Microsoft ASP.NET – runs on top of IIS
Macromedia ColdFusion
Apache Tomcat
Apache JServ
PHP
BEA WebLogic
IBM WebSphere

Some frequency analysis can be found here (https://secure1.securityspace.com/s_survey/data/man.200203/cookieReport.html), through associating the cookie names with the server that issues them. This is of course biased, since some servers and sites use tokens in form parameters rather than in cookies.

The upside of application engines is the fact that they completely relieve the programmer from worrying about session management. All functionality aspects of session management are taken care of, usually much better than an in house programmer could have achieved.

The downside of application engines is the fact that they seem to relieve the programmer from worrying about the security of the token, yet we can show that the harsh reality is far from that. In fact, some very popular application engines do not provide secure tokens. As a result, the programmer obtains a false sense of security.

We examined the tokens generated by two popular application servers. In both cases, we were able to demonstrate that the token is not as random as it seems, and that it is possible (in one case, with ease), to predict the values of the token for the next sessions (of a different client).

Example 1 – beating a time based token

The target of this attack is a very popular commercial application engine. The product uses two cookies to identify a session. The pair formed by the two cookies identifies the session. The first cookie is merely a counter, incremented once per new session. It probably ensures that no two pairs are ever identical. The second cookie is the token cookie, apparently intended to secure the pair by being "unpredictable". Since it is very easy to predict the first cookie, we focus on the second cookie, which we'll denote as "TOKEN."

At first glance, TOKEN seems to be a sequence of random 8 decimal digits. The entropy (amount of randomness) here is $10^8 = 2^{26.57}$ which may be considered sufficient, considering that it's quite unfeasible to try such amounts of requests (100 million) against a site without triggering some kind of alarm and human attention.

But, a closer look reveals that in fact, TOKEN obeys the following equation:

Let us denote by $\boldsymbol{t}$ the GMT time, in seconds, since 01/01/1970 00:00, as set on the application server.

Let us denote by $\boldsymbol{m}$ the milliseconds portion of the tick counter on the application server.

Then:

$$\text{TOKEN} = (\ 31415821 * (\boldsymbol{t} + \boldsymbol{m}) + 1\ ) \bmod 100000000$$

It is interesting to note that $\boldsymbol{t}$ can be extracted from the HTTP Date header the server sends back to the client together with the first time the cookies are set.

This means that the TOKEN cookie is quite predictable. In fact, if one knows a range of time $T \le t < T+\Delta T$ (in seconds) in which a cookie was generated, one can infer that TOKEN has one of ΔT+1000 values, which is a rather short list of values. Testing a bit more than a thousand values against the server may take few minutes, in which the victim session is likely to remain active.

The outline of an attack algorithm is as following:

```
Obtain a first pair ($id_1$, $TOKEN_1$). Record $t_1$ – the server time (from the Date HTTP header)

Wait $\Delta T$ seconds.

Obtain a second pair ($id_2$, $TOKEN_2$). Record $t_2$ – the server time (from the Date HTTP header)

if ($id_2 > id_1 + 1$)
begin
    // we have a victim session interjected here.
    for ($x = t_1$ ; $x < t_2 + 1000$ ; x++) // which is $\Delta T$+1000 iterations
    begin
        Try the pair ($id_1$ +1, ( 31415821 * x + 1 ) mod 100000000)
    end
end
```

In fact, it is possible to improve this algorithm in some cases by using the fact that on some operating systems, the tick counter does not have millisecond granularity, but rather a coarser granularity of around 10msec. This can be used to reduce the search space even further.

The attack described above enables the attacker to impersonate a victim, provided that such victim was assigned a cookie between the two samples the attacker made of the site cookies. Since the attacker can repeat the algorithm as many times as he/she would like, it is possible for him/her to obtain these cookies for all clients, at a price of sampling the site (say, one request every minute), and additionally some 1060 requests per any new client discovered. Again, as hinted above, it is possible to sample at closer intervals (once a second) and exploit the granularity problem of the clock ticks, in which case it is probably possible to arrive at 100 requests per new client.

It is likely that if an attempt to impersonate a client is performed while the site is loaded with traffic, then the additional hundreds/thousands of request would go unnoticed, at least momentarily.

Example 2 – When Random() isn't random
In this example, we deal with a still popular (yet a bit outdated) application engine. This engine generates a single cookie for each new session. This cookie (which we shall name ID) comprises of 3 mandatory fields (F1, F2 and F3), and one optional (server configuration dependent) field (F4, preceded by a dot), concatenated. The fields are as following:

F1 = 6 characters (A-Z0-9) – PRNG (Pseudo Random Number Generator) data, represented in base 36 with leading zeroes
F2 = 3 characters (A-Z0-9) – server time (milliseconds), divided by 2000, mod 36^3 (=46656), represented in base 36 with leading zeroes
F3 = 3 characters (A-Z0-9) – session count in this 2 second time slice, represented in base 36
F4 = constant string (per server)

As can be seen, F4 (if it exists) is constant, and hence trivially predictable. F2 is simply the server time (in seconds) divided by 2, modulo 46656, which is quite predictable, and F3 is not too obscure as well – as it is sequentially incremented in the 2 seconds time slice (always begins at one).

The only interesting field is therefore F1. Apparently, it holds enough entropy to secure the system, since it can assume 36^6 values (=$2^{31.0}$). Yet again, what seems secure at first sight appears not so secure when performing a full analysis. Explanation on how and why F1 can be predicted is provided in Appendix A, since it is too long for inclusion here. The problem we exploited with F1 is the fact that it uses a PRNG (Pseudo Random Number Generator), which in itself is predictable. So knowing several values of F1 suffices to fully predict the PRNG, and hence future (and past) values of F1.

The outline of an attack is as following:

Preparation:
Obtain three IDs, in the shortest time intervals possible.
Extract the PRNG internal state (as explained in Appendix A).

Interception Cycle
Obtain an ID, and record the server time, t. For simplicity, assume t is even.

```
Find the PRNG internal state that was used to generate this ID (as explained in Appendix A)
Wait ΔT seconds (where ΔT is even)
Obtain a new ID.
Advance the PRNG, and record all internal states between the PRNG state of the old ID and the
PRNG state that generated this ID (As explained in Appendix A). Let the list of internal   values be L
// ΔT/2 iterations:
for (T=t; T<t+ΔT; T+=2)
begin
        for each internal PRNG state L, i.
        begin
                Try an ID cookie consisting of:

                        F1=generate from sample of PRNG at state i and i+1;
                        F2=T;
                        F3=1; // first session in this 2-second time period
                        F4=F4 of any ID above; //constant per server
        end
end
```

As can be seen, it is feasible, although not trivial, to predict some ID cookies. For feasibility, it is required that the time interval (ΔT) be short (with respect to the expected usage of the server), in order to minimize the length of L (the list of possible internal PRNG states). If these intervals are indeed very short (less than two seconds), it may be possible, with correct timing, to tell whether a new session was interjected at the current 2 second time slice, which makes the attack more effective (since it requires launching the additional requests only when it is known that a new victim session was indeed created). It should also be mentioned that in order not to lose synchronization (of the PRNG internal state) with the site, it is necessary to keep requesting a new ID from time to time, in order to advance the attacker's PRNG internal state to the new value. It should be remembered that the PRNG is likely to be used for many purposes, not just the creation of sessions. This means that the site may use the PRNG intensively, thus causing a quick de-synchronization (to counter which it is necessary to re-sync at close time intervals, e.g. every few minutes). On the other hand, it may be possible to get a clearer glimpse of the internal PRNG state by inspecting other random values that may be used in the site. This may offer a shortcut saving a lot of computation power.

It should be noted, that once the attacker is in synch with the site, and if IDs are extracted frequently enough, it is possible to impersonate any client at the expanse of sending few (depends on the usage of the PRNG) requests.

WHAT THE INVOLVED VENDORS SAID

Vendor 1 acknowledged the weakness, and informed us that its customers should use SSL certificates for session management. While this is perhaps a good idea for some customers (but definitely not for all customers – moving to SSL and SSL certificates is definitely not trivial, and sometimes not possible), the documentation for its product leads the reader to believe that the built-in session management is secure (they name it "the client security token" in their documentation for developers). Also, the vendor does not make this suggestion public.

Vendor 2 acknowledged the weakness yet wrote us “session cookies are -NOT- a replacement for authentication tokens. A session cookie in conjunction with a random auth token or auth login validation is both reasonable mechanisms. This should be true in designing session based scripts - even where the session tokens are 'trusted' today.” – thus laying the responsibility in the hands of the developers.

The two vendors, while technically acknowledging the problem, dismissed it as a non-security issue. That is, both vendors assume their customers implement their own session security tokens, not relying on the vendor tokens. The vendors, therefore, claim that their tokens are used (or should be used) solely to better differentiate between different users, and not as a security measure. In the documentation, we did not find any warning against using the token as a secure session identifier. Furthermore, Vendor 1’s documentation uses phrases that lead one to believe that this token is secure. And in reality, of course, most sites use the tokens issued by vendors as a secure session identifier, oblivious to the fact that it is weak.

In a sense, the application developer is back to square one: he/she cannot trust the built-in session identification mechanism, and thus is forced to write his/her own such mechanism, with best effort to fulfill all the requirements mentioned above and to avoid the delicate pitfalls of cryptography.

CONCLUSION

We see session security falls between the cracks –vendors don’t do it right, don’t care for it, or delegate the responsibility for it to the developers, while in-house development is error-prone, and requires a deep understanding of security.

In this paper, we provided real life examples for both insecure tokens in commercial application engines, as well as in home grown applications.

Our solution is simple – the world of web applications should consist of *three* components:

- The application (which is developed in house, and expresses the business logic, as well as the novelty and specialty of the company/site).
- The application environment (the application engine and web server, which enable easy application development and focus on the application rather than on infrastructure).
- Web application security component, which takes care of the application security, again relieving the developers (and to some extent, the application engine developers too!) from having to worry about secure implementation of their application.

In all the above cases, a web application firewall would have fortify the tokens generated by the application engines (or by the in house developed application) transparently (the developer needn’t even be aware of this), and ensure, through using strong cryptography and security tested mechanisms, that the tokens sent to the application are indeed genuine, and not forged.

APPENDIX A: ANALYSIS OF THE PRNG USED IN EXAMPLE 2

The PRNG in example 2 is a linear congruence type PRNG. Its internal state consists of 48 bits (the variable "state"). The PRNG is seeded once (that is, an initial value for "state" is provided), and then advances in the following manner:

$$\text{state} = (\text{state} * 25214903917 + 11) \bmod 2^{48}$$
$$\text{sample} = \text{state} / 2^{16}$$

As can be seen, sample is a 32 bit number.

The ID generation mechanism concatenates two consecutive samples to form a 64 bit integer, which may be negative (if the most significant bit is 1). Then, absolute value of this number is taken, and mod 36^6 is applied to yield F1.

And now to some mathematics: we want to be able to predict the values sampled from this PRNG.

We do get a direct glimpse at the state bits. To understand why, let us consider the mathematical representation of F1. Let the two samples needed for F1's generation be denoted S1 and S2.

Then:

S1= [don't care]
$S2 = \text{state} / 2^{16}$

$N = S1 * 2^{32} + S2$
if ($N \geq 2^{63}$) $N=2^{64}-N$ //make sure N is "positive" – i.e. most significant bit is 0.
$F1=N \bmod 36^6$

Since $36^6 = 2^{12} * 3^{12}$, it follows that
$F1 \bmod 2^{12} = N \bmod 2^{12}$ or $F1 \bmod 2^{12} = (-N) \bmod 2^{12}$

And since $N = S1 * 2^{32} + S2$, it follows that
$F1 \bmod 2^{12} = S2 \bmod 2^{12}$ or $F1 \bmod 2^{12} = (-S1) \bmod 2^{12}$

We see, therefore, that F1 mod 212 provides us with two options for the 12 least significant bits of S2, which, in turn are the bits 16-27 of state (denoting the least significant bit as 0 and the most significant bit as 47).

Now, we can guess the 16 least significant bits of state, and together we'll have the 28 least significant bits of state. We have 2^{17} guesses (2^{16} for the 16 least significant bits of state, and 2 for the original sign of N).

The number of guesses can be easily reduced by taking another sample, as close as possible (i.e. with as few samples of PRNG in between), and verifying against the 11 bits of information (12 bits minus the sign bit). If it is possible to achieve two IDs with less than (say) 16 advances of the PRNG in between, then with a calculation of 2^{17}*16 we can reduce the number of guesses we have to 2^10. Applying this argument twice more will show that with 4 ID's generated close enough, it is possible to come out with a single verified value for the 28 least significant bits of state (for all IDs), with no more than few million calculations.

Finally, we can also guess the 20 most significant bits, and we can easily verify them using the ID's we have, because once all the bits of state are known, it is possible to calculate F1 accurately.

In order to check all guesses at this phase, we need to perform few million calculations.

The above can be easily performed using a standard PC (Pentium-III or Pentium-4) in few minutes or less.

After this, the full state of the PRNG becomes known. This enables to accurately follow the PRNG to the future and to the past.

For example, if one has the current value of the PRNG, and an ID that was produced from the PRNG after some advances have taken place, it is possible to find the PRNG state associated with the ID, as well as all PRNG states in between (and the ID that may have been produced for them) via simply advancing the PRNG and generating the ID, until the ID generated matches the one obtained from the server. This provides both a list of possible IDs between the time the PRNG was at the known state and the time the ID was obtained, as well as the current state of the PRNG (the one matching the obtained ID).

It should be noted that an ID is obtained from sampling two consecutive states of the PRNG. But since it is impossible to know how the PRNG is used, we must check every possibility for having two consecutive pairs. So if the PRNG states are A, B, C and D we must list the IDs formed from (A,B), (B,C) and (C,D).

APPENDIX B: ADDITIONAL INFORMATION

Please Note: There are newer versions available for the two application servers mentioned in the examples in this document which solves the problems discussed.

“Hold Your Sessions: An Attack on Java Session-Id Generation” (Zvi Gutterman and Dahlia Malkhi): http://research.microsoft.com/~dalia/pubs/GM05.pdf

ABOUT WATCHFIRE

Watchfire provides Online Risk Management software and services to help ensure the security and compliance of websites. More than 500 enterprises and government agencies, including AXA Financial, SunTrust, HSBC, Vodafone, Veterans Affairs and Dell rely on Watchfire to audit and report on issues impacting their online business. Watchfire has been the recipient of several industry honors including the HP/IAPP Privacy Innovation Award, *InfoSecurity Product Guide*'s Hot Security Company 2006, *Computerworld*'s Innovative Technology Award, and "Recommended" rating by *Computer Reseller News*. Watchfire was named by IDC as the worldwide market share leader in web application vulnerability assessment software. Watchfire's partners include IBM Global Services, PricewaterhouseCoopers, TRUSTe, Microsoft, Interwoven, EMC Documentum and Mercury. Watchfire is headquartered in Waltham, MA. For more information, please visit www.watchfire.com