A Brief Analysis of ASP.NET Session Identifiers

Timothy D. Morgan
<tim-sessions {α} sentinelchicken.org>

December 18, 2008

Abstract

(ASP).NET[1] is a widely used web application development environment. In addition to many features considered standard for a web application platform, it provides built-in session management. Session identifiers are automatically generated and are typically provided to users (web browsers) as HTTP cookies. This paper provides a basic outline of how these session identifiers are generated which helps in better understanding their security.

1 Introduction

Web application session identifiers are commonly relied on to track user sessions and to authenticate users with each request. Because the HTTP protocol is inherently stateless from one request to the next, session identifiers are a critical component of flexible authentication systems. This also means that if an attacker were to guess or discover a victim's session identifier, she would typically be able to hijack the session and all application privileges associated with it.

The motivation for this research came about through empirical analysis of `ASP.NET_SessionId` cookies. These cookies are commonly the default session identifier in ASP.NET applications. The following identifiers are typical of the cookie values assigned:

```
w2wk4155d3gonl4533gl1w55
rss3yp45qz4g0b55kgpwjxi2
qaqgbjrz23didt45err5is55
0sufqo55sbnr2rjujaabch55
tnp20555gvssup45movzc555
p2qjfb45wu1oop551ljqf045
z5pptp3ga0u1bg45c2b05smc
1yurinjug5uugqu0fym3x055
```

Immediately, visual inspection of these values unveils some suspicious patterns in the data. Alphanumeric digits 7, 8, 15, 16, 23, and 24 seem to have unusually frequent occurrences of "4" and "5". This is confirmed with more rigorous statistical analysis using stompy [5], a session identifier analysis tool. The results from stompy indicate that these suspicious columns are indeed less than perfectly random and that the overall session identifiers contain at most around 117.5 bits of entropy. By current standards, this amount of entropy would be more than sufficient for long term security (let alone temporary access). However, a Microsoft article [2] indicates that these identifiers should have about 120 bits of entropy. This discrepancy is small, but taken together with the non-uniformity of the cookie digits, we felt it worth while to investigate further the method by which .NET generates these tokens.

2 Analysis

A basic ASP.NET application was created (on Windows 2003 / .NET 2.0.50727) with a single page which caused a session cookie to be set. A debugger, OllyDbg [4], was then used to attach to the `w3wp.exe` process.

[1] All trademarks are properties of their respective owners.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p

16	17	18	19	20	21	22	23	24	25	26	27	27	29	30	31
q	r	s	t	u	v	w	x	y	z	0	1	2	3	4	5

Table 1: Character Encoding Table

Breakpoints were set at commonly used cryptography functions, which revealed that `CryptGenRandom` [3] was called exactly once with each sessionless request. The output of this function (a binary buffer) was followed until the cookie encoding function was located and analyzed.

As is advertised by Microsoft, we found that 15 bytes, or 120 bits, of entropy were returned by `CryptGenRandom`. However, we also found that due to a bug in the cookie encoding algorithm, the effective entropy can be somewhat smaller. The encoding algorithm treats the 120 bit buffer as three blocks of 40 bits each. For each block the following C-like pseudocode roughly outlines the algorithm:

```
Let BLOCK be the 40-bit block of entropy to encode
Let BUFF be a 32-bit register initialized to 0
Let ENCODE be a character array
Let OUTPUT be a string
BUFF = BLOCK[0]
BUFF = BUFF | (BLOCK[1] << 8)
BUFF = BUFF | (BLOCK[2] << 16)
BUFF = BUFF | (BLOCK[3] << 24)
Append ENCODE[BUFF & 0x1F] to OUTPUT
BUFF = BUFF >> 5
Append ENCODE[BUFF & 0x1F] to OUTPUT
BUFF = BUFF >> 5
Append ENCODE[BUFF & 0x1F] to OUTPUT
BUFF = BUFF >> 5
Append ENCODE[BUFF & 0x1F] to OUTPUT
BUFF = BUFF >> 5
Append ENCODE[BUFF & 0x1F] to OUTPUT
BUFF = BUFF >> 5
Append ENCODE[BUFF & 0x1F] to OUTPUT
BUFF = (BLOCK[4] << 2) | BUFF
Append ENCODE[BUFF & 0x1F] to OUTPUT
BUFF = BUFF >> 5
Append ENCODE[BUFF & 0x1F] to OUTPUT
return OUTPUT
```

Here, as in the C programming language, the bitwise operation `<<` denotes left shift, `>>` is bitwise right shift, `|` is bitwise OR, and `&` is bitwise AND. The `ENCODE` table is a simple translation table detailed in Table 1.

The bug here arises due to an oversight in the behavior of the right shift operator. The operation used fills any left over high bits with the value of the highest bit when the shift was started. Therefore, when the first four bytes are loaded into the 32-bit register, each successive right shift causes all left over high bits to either be set to 1, or to 0, based on what the highest bit of `BLOCK[3]` was. (For instance, a 32-bit value of `0x80000000` right shifted 30 bits would be `0xFFFFFFFC`.) Once the 32-bit register is reduced to 2 remaining usable bits, it is ORed with the remaining byte from `BLOCK`. If the high bits of the register were 1's, then this additional byte is completely lost, since 1 ORed with anything is 1. However, if the high bit of `BLOCK[3]` was 0, then the entropy is preserved for this block.

Now it becomes clear why the final two digits of each 40-bit block are commonly 4 or 5. Fifty percent of the time, the last digit in each group can only be 5 (all 1's) and the second to last digit in that case can only

Case	Frequency	Maximum Entropy
All `BUFF` high bits set	1/8	96
Two `BUFF` high bits set	3/8	104
One `BUFF` high bit set	3/8	112
Zero `BUFF` high bits set	1/8	120
	Weighted Mean	108

Table 2: Mean Identifier Entropy

be 4 or 5 (two bits, but one of them is the culprit high-bit, which we know is 1). The other fifty percent of the time, these two digits can be random, all directly derived from the `CryptGenRandom` call. This issue can be trivially avoided by applying a simple mask to `BUFF` prior to ORing in the final `BLOCK[4]` byte.

As to the impact on the session identifiers' security, we see that in the worst case, each of the three 40-bit block groups could lose 8 bits of entropy. This would leave us with 120 - 24 = 96 bits of data directly derived from `CryptGenRandom`. While this function's algorithm has received some criticism [1], it appears to generate reasonbly safe output and 96 bits of entropy is considered safe by today's standards, especially for online attacks. However, in the typical scenario, the expected entropy would depend on the specific possible cases and their frequency, as outlined in Table 2.

3 Conclusion

This brief analysis shows that while .NET's base session identifier algorithm exhibits some flaws, it should withstand current known attack methods. The specifics of the algorithm, as included here, will hopefully help the public understand the impact of any future flaws found within `CryptGenRandom` or other parts of the .NET framework. If nothing else, other curious cryptanalysts will no longer need to wonder about the statistical anomalies exhibited by these identifiers or waste time trying to attack them through blind statistical analysis.

References

[1] Benny Pinkas Leo Dorrendorf, Zvi Gutterman. *Cryptanalysis of the Random Number Generator of the Windows Operating System.* 2007. Available at: http://eprint.iacr.org/2007/419.pdf.

[2] Microsoft. *How and why session IDs are reused in ASP.NET.* 2006. Available at: http://support.microsoft.com/kb/899918.

[3] Microsoft. *CryptGenRandom Function.* 2008. Available at: http://msdn.microsoft.com/en-us/library/aa379942(VS.85).aspx.

[4] Oleh Yuschuk. *OllyDbg.* Available at: http://www.ollydbg.de/.

[5] Michal Zalewski. *Stompy the Session Stomper.* 2007. Available at: http://freshmeat.net/projects/stompy/.