Hold Your Sessions: An Attack on Java Session-Id Generation

Zvi Gutterman and Dahlia Malkhi

School of Engineering and Computer Science,
The Hebrew University of Jerusalem,
Jerusalem 91904, Israel
{zvikag,dahlia}@cs.huji.ac.il

Abstract. HTTP session-id's take an important role in almost any web site today. This paper presents a cryptanalysis of Java Servlet 128-bit session-id's and an efficient practical prediction algorithm. Using this attack an adversary may impersonate a legitimate client. Through the analysis we also present a novel, general space-time tradeoff for secure pseudo random number generator attacks.

Keywords: pseudo random number generators, space-time tradeoff, HTTP, web security

1 Introduction

At the root of many security protocols, one finds a secret seed which is supposedly generated at random. Unfortunately, truly random bits are hard to come by, and as a consequence, often security hinges on shaky, low entropy sources. In this paper, we reveal such a weakness in an important e-commerce building block, the Java Servlets engine.

Servlets generate a session-id token which consists of 128 hashed bits and must be unpredictable. Nevertheless, this paper demonstrates that this is not the case, and in fact it is feasible to hijack client sessions, using a few legitimately-obtained session-id's and moderate computing resources.

Beyond the practical implication to the thousands [16] of servers using Servlets, this paper has an important role in describing an attack on a pseudo-random-number-generator (PRNG) based security algorithm and in demonstrating a nontrivial reverse engineering procedure. Both can be used beyond the Servlets attack described henceforth.

Web server communication with clients (browsers) often requires state. This enables a server to "remember" the client's already visited pages, language preferences, "shopping basket" and any other session or multi-session parameters. As HTTP [9] is stateless, these sites need a way to maintain state over a stateless protocol. Section 2 describes various alternatives for implementing state over HTTP. However, the common ground of all these schemes is a token traversing between the server and the client, the session-id.

The session-id is supported by all server-side frameworks, be it ASP, ASP.net, PHP, Delphi, Java or old CGI programming. Session-id's are essentially a random value, whose security hinges solely on the difficulty of predicting valid session id's. HTTP session hijacking is the act where an adversary is able to conduct a session with the web server and pretend to be the session originator. In most cases, the session-id's are the only means of recognizing a subscribing client returning to a site. Therefore, guessing the unique session-id of a client suffices to act on its behalf.

Driven by this single point of security, we set out to investigate the security of session-id's deployments, and as our first target, we have analyzed the generation of session-id's by Apache Tomcat. Apache [2] is an open-source software projects community. The Apache web server is the foundation's main project. According to Netcraft [16] web study of more than $48,000,000$ web servers, the Apache web server is used by more than 67% of the servers and hence the most popular web server for almost a decade.

At the time of this writing (April 2004), sites such as `www.nationalcar.com`, `www.reuters.com`, `www.kodak.com` and `ieeexplore.ieee.org` are using Java Servlets on their production web sites.

In many of these sites, the procedure for an actual credit-card purchase requires a secure TLS [8] sessions, separated from the "browsing and selection" session. However, this is not always the case. For example, Amazon's patented [10] "one-click" checkout option permits subscribing customers to perform a purchase transaction within their normal browsing session. In this case, the server uses a client's credit-card details already stored at the server, and debits it based solely on their session-id identification.

In either of these scenarios, an attacker that can guess a valid client id can easily hijack the client's session. At the very least, it can obtain client profile data such as personal preferences. In the case of a subscriber to a sensitive service such as Amazon's "one-click", it can order merchandize on behalf of a hijacked client.

Briefly, our study of the generation of Java Servlets' session-id's reveals the following procedure. A session-id is obtained by taking an *MD5* hash over 128-bits generated using one of Java's pseudo-random number generators (PRNG). Therefore, two attacks can be ruled out right away. First, a brute force search of valid session-id's on a space of 2^{128} is clearly infeasible. Second, various attacks on PRNGs, e.g., Boyar's [6] attack on linear congruential generators, fail because PRNG values are hidden from an observer by the *MD5* hashing.

Nevertheless, we are able to mount two concrete attacks. We first show a general space-time attack on any PRNG whose internal state is reasonably small, e.g., 2^{64}–2^{80}. Our attack is resilient to any further transformation of the PRNG values, such as the above *MD5* hashing. Using this attack, we are able to guess session-id's of those Servlets that use the java. util .Random package, whose internal PRNG state is 64-bits. Beyond that, our generic PRNG attack is the first to use space-time tradeoffs, and may be of independent interest.

Our second attack is on the seed-generation algorithm of Java Servlets. Using intricate reverse engineering, we show a feasible bound for the seed's entropy. Consequently, we are able to guess valid session-id's even when Servlets are using the java.security.SecureRandom secure PRNG (whose internal state is 160 bits).

The paper is organized as follows. In Section 2 we describe the HTTP state mechanisms. In Section 3 we describe and analyze the Tomcat session-id generation algorithm. Java hashCode() study is presented in Section 4. In section 5 we present our attacks on the session-id. We conclude in Section 6.

2 Stateful Web Browsing

HTTP is a client/server protocol designed for a light-weight and quick delivery of content from servers to clients. HTTP is stateless, in that a server responds to a client's request with a hypertext page and then breaks down the connection. Any additional request from the same client requires the client to build a new, seemingly unrelated connection with the server. Statelessness is part of what makes HTTP efficient and fast to implement.

However, a typical client/server interaction entails repeated interaction. For example, often a web page contains links to images and multi-media objects. Obtaining each one of these is done in a separate TCP/IP connection to the server, but they appear to be part of a single prolonged interaction. The new HTTP standard [9] (HTTP 1.1) is already in place, allowing multiple retrievals instead of a single one. Nevertheless, it is not meant to keep connections up through an involved client/server interaction, which could span multiple screens and forms. And it does not address clients returning to the same site after days have passed.

Cookies [13] change this situation. Introduced originally by Netscape and thereafter adopted widely and as part of HTTP 1.1, cookies were designed with the intention of solving the vexing problem of keeping long-lived relationships between web servers and their clients. Cookies extend the HTTP protocol by allowing a server to hand a client certain information to keep. The client's browser automatically hands the server this information, the cookie, the next time it connects to the same site. Cookies are used by servers to store a variety of information, from client membership identification to complete shopping basket contents. They greatly enhance the web browsing experience, allowing a client to be recognized by a server, accumulate shopping selections, and so on.

An analog mechanism to cookies is URL rewriting. In this framework, instead of sending a fixed web page to the client the web server encodes the session information as part of the page, e.g., within embedded URL links. URL rewriting requires less from the client side, but as far as this paper is concerned is the same session mechanism and our attack is equally applicable to it.

From a privacy point of view, it should be noted that the cookies mechanism and likewise, URL re-writing were designed to prevent leakage of information between sites, in that a cookie is returned only to the site that originally sent it. In this way, a server may only obtain information that it already had about a user. Unfortunately, there are examples of cookie-abuse, e.g., the infamous `doubleclick.com` site, that collects client clicking-profile through its advertisements on partner sites.

This work, however, is concerned with a different weakness of cookies, and more generally, with stateful web browsing. True, recognizing a returning client through cookies alleviates the need to tediously re-type a user name and a password upon each connection establishment to a site. Unfortunately, it also poses a web-identity theft potential: If one can guess a valid cookie, one can impersonate another client. As simple as that.

There is hardly a limit to what an attacker may obtain through such identity theft: She may be able to learn private user data, such as names and addresses. She could collect clients' profile information, such as preferences and shopping history. She could penetrate access protected sites. In a particularly vicious attack, using Amazon's "one-click" option, she might be able to order merchandize on behalf of impersonated customers. Essentially, there are limitless hazards.

3 Tomcat Session-Id Generation Algorithm

In this section, we describe our study of Tomcat 5 [1], the Apache Java implementation for Servlet 2.4 [19] and JSP 2.0 [14] specifications. We study version 5.0.18, which was released on January 2004. Our full study involves additional, and more challenging reverse-engineering of relevant modules of the JVM which are written in native-code. This part is deferred to the next section.

The remainder of this section describes the Tomcat session-id generation scheme, which includes two parts. One is a session-id allocation used during the set up of each new session. The second is an initialization phase that is executed once when the server comes up. We hint about potential weaknesses as we go along. The description omits unimportant implementation details such as irrelevant Java **class** names.

3.1 Allocation

We begin by examining the algorithm for generating new sessions-id's during the set up of new sessions. Session-id's are allocated within method generateSessionId(), and consists of 16 bytes, or equivalently, 128 bits.

Inside generateSessionId(), the allocation consists of the following steps:

1. Method getRandomBytes fills a sixteen bytes array. If /dev/urandom exists the bytes are read from it. If not, a Java pseudo-random number generator (PRNG) is invoked. Method getRandom() is invoked to obtain a handle either to Java.Security.SecureRandom or Java.Util.Random. Figures 1,2 presents these functions.
2. The 16 bytes obtained from getRandomBytes are mixed using a digest function which is *MD5* [18] by default.
3. The result is the 128-bit session-id. For convenience, it is converted into 32 ASCII characters, where each 4 bits are mapped to a matching character between '0' ... 'F'.

$$x_n := \begin{cases} \text{initial seed} & n = 0 \\ (25,214,903,917 \times x_{n-1} + 11) \bmod (2^{48} - 1) & n > 0 \end{cases}$$

Fig. 1. java.util.Random. x_n holds the PRNG next output.

$$x_n := SHA1(s_n) \qquad n = 0$$

$$s_n := \begin{cases} \text{initial seed} & n = 0 \\ (x_{n-1} + s_{n-1} + 1) \bmod 2^{160} & n > 1 \end{cases}$$

Fig. 2. java.security.SecureRandom. x_n holds the PRNG next output and s_n is the internal state.

3.2 Initialization

Given that generateSessionId() employs a Java PRNG for allocating session-id's, the next thing to investigate is how it is initialized inside the Tomcat package. We had initially hoped to find a simple weakness, e.g., initialization by a hard-wired constant, which would render session-id's easily predictable. Such weakness were found frequently in the past, e.g, [12].

That is not the case here, and the seeding of the PRNG within Tomcat is an intricate, thoughtful process, consisting of the following steps.

1. Set C = System.currentTimeMillis(). This is 64 bit field measuring the time since January 1, 1970 in milliseconds.
2. Set *Entropy* = toString(org.apache.catalina.session.ManagerBase.java). The value of *Entropy* is equal to the Java String org.apache.catalina.session. ManagerBase.java@X. The prefix of the term is always the same, and the part following the @ sign is variable. Section 4 describes our study of the X value and how we can predict it.
3. Set *Seed* = $f(C, \mathit{Entropy})$. The function f is depicted in Figure 3. It takes the *Entropy* and spreads it byte by byte (letter by letter), with 8 bytes per row (or 64 bits per row). It computes a xor of all the rows, xor'ed also with C, yielding a 64-bit value.
4. *Seed* is used for initializing the PRNG.

Despite the intricate seeding process above, this is the important part where our attack will take place. As we show below, we can indeed predict with reasonable effort the *Seed* value. As all other steps are deterministic and known from the server code, once we find the *Seed* we can predict each session-id value. This will be later presented in Section 5.

4 Java Object.toString() Algorithm

The Java Object.toString() function is used by the initialization algorithm presented in Section 3 for generating the PRNG seed. In this section, we take a

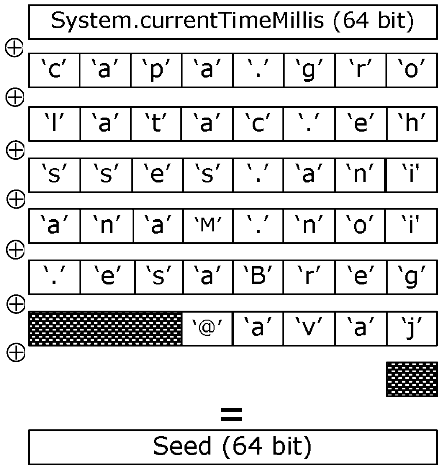
System.currentTimeMillis (64 bit)
⊕
'c' 'a' 'p' 'a' '.' 'g' 'r' 'o'
⊕
'l' 'a' 't' 'a' 'c' '.' 'e' 'h'
⊕
's' 's' 'e' 's' '.' 'a' 'n' 'i'
⊕
'a' 'n' 'a' 'M' '.' 'n' 'o' 'i'
⊕
'.' 'e' 's' 'a' 'B' 'r' 'e' 'g'
⊕
'@' 'a' 'v' 'a' 'j'
⊕
=
Seed (64 bit)

Fig. 3. The function $f()$ employed to generate a seed out of C, the time in milliseconds, and *Entropy*, a string containing org.apache.catalina.session.ManagerBase.java@X, where X is the hashCode 32 bit field marked as scattered area.

close look at Object.toString(), and show that this value is actually a very low entropy source.

The Java Object method toString returns the value getClass().getName()+"@"+Integer.toHexString(hashCode()); Hence, the returned string has a fixed prefix, which is the class name, followed by the @ sign and a 32 bit field which is the result of the method hashCode.

The function java.lang.Object.hashCode() is a native one, which requires each Java virtual machine implementor to bring its own implementation.

According to the Java documentation the hashCode method must have the following properties.

1. Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must return the same integer (32 bit).
2. If two objects are equal they return the same hashCode
3. it is not required that two object which are not equal return distinct values of hashCode.
4. As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects (this is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java programming language).

It is important to note here that reading the Java documentation may lead the reader (and maybe also the Tomcat implementor) to think that the hashCode is hard to predict.

However, this is not always the case. In particular, the Microsoft Windows platform [15] does not follow the recommendation to use the pointer address space in generating the hashCode. Instead the JVM uses a linear congruential generator (LCG) to get the different hash codes. Using the IDA-Pro [7] disassembler we get

$$hashCode(object) := \begin{cases} (a \times x_n + b) \bmod m & \text{first object access} \\ \text{hash is given from a history table} & \text{otherwise} \end{cases} \tag{1}$$

We can now predict the hashCode value using the LCG values. What we need to know is the server boot sequence where our object will be called. This information should usually be available for an attacker, which in most cases can deploy the same server and verify the class loading sequence. Even when this procedure is hard to perform an adversary can narrow the valid range into 256 possible values with only few trials. This brings the Java hashCode into 8 entropy bits or less, which is far lower entropy than the presumed 32 bits and will take part in our general attack scheme.

5 Attacks

We remind the reader that the goal of an attacker in our settings is to predict legitimate session-id's that are allocated to clients, and impersonate these clients over HTTP connections with servers.

We describe two attacks. The first one is a generic attack on any PRNG whose internal state is feasibly small, e.g., 2^{64}–2^{80}. The second is an attack on the seeding procedure of Java Servlets.

5.1 Space-Time Tradeoffs for PRNG Attacks

A space-time tradeoff attack is the notion of using a large space of pre-computed values in order to reduce the time of an online attack. Ours is the first general space-time tradeoff on secure PRNG based protocols. In the following, we first present the general attack and then tailor it for the session-id's case. Our attack is a direct adaptation of a space-time tradeoff attack on stream-ciphers, recently demonstrated by Biryokuv and Shamir in [4]. For completeness, we first introduce space-time tradeoffs for block and stream ciphers.

Background. A block cipher space-time attack lets an adversary tune the values of memory M and online attack time T for a given key space K of size $N = |K|$. Hellman [11] introduced this method with a $TM^2 = N^2$ tradeoff.

Hellman's space-time tradeoff block cipher attack is made of two parts. We first conduct a pre-computation stage to set the memory tables, with computation cost $P = N$. The second stage includes the online attack. Given a cipher-text y the online stage returns the key $k \in K$ such that $y = E_k(p)$, where E is the encryption function and p is a pre-chosen plain text.

The pre-computation includes building several tables of chains as follows. For the first element in the chain, we first randomly select a key $k^0 \in K$. The second chain element is $k^1 = R(E_{k^0}(p))$, where $R(y) \in K$ is an arbitrary reduction function which maps a cipher text block to a valid key value. The reduction function can be simple truncation, or a selection of $|k|$ bits from the cipher text y, but as explained below, it is important that R is uniformly distributed over K.

A chain of length t contains repeated invocations of $R(E_{k^i}(p))$. We mark SP and EP as the start and end points. The resulting length t chain, with reduction function $R()$ is as follows:

$$SP := k^0 \longrightarrow k^1 := R(E_{k^0}(p)) \longrightarrow \ldots \longrightarrow EP := k^{t-1} := R(E_{k^{t-2}}(p)) \qquad (2)$$

The goal is to cover K with the different chains and with low or no collisions at all. Each chain starts with a different SP, and we assume that the application of $R(E_k())$ over the initial random starting points is like a random selection of elements from K.

We can repeat the chain building procedure and make m such chains. In order to complete our attack these chains must cover K. However, some collisions will occur, i.e., a chain will occasionally reach a key that already appears in a previous chain. Once such a collision occurs, the remainder of the chain, which is computed in a deterministic way, will repeat the same, already computed chain. Furthermore, when existing chains cover as little as N/t out of the N elements, the probability for collision in the next t elements is a non negligible constant.

Hellman suggested to solve the collision problem using r different reduction functions $R_1, \ldots, R_r$. Each reduction function is chosen as a different selection of $|k|$ bits out of the cipher text y. For each reduction function we build a table with m chains, each of length t, such that $mt = N/t$ (the point beyond which producing additional chains is wasteful). The different reduction functions ensure that even when an element occurs in two different tables, the next element in the chains will be different in the two tables, hence the total number of collisions is low.

The assignment of m, t and r such that $mtr \geq N$ solves both our collision and coverage concerns. In the rare occasion that during our pre-computation two chains end with the same EP we select the longer chain.

An additional important technique which can improve the table lookup performance is due to Rivest. Instead of stopping after t steps we can stop at a *Distinguished Point* which is a point with some easy to verify property, e.g., all its $\log_2 t$ first bits are zero. As $R_i(E_k(p))$ is distributed uniformly, the average chain length will be t. In this way, instead of looking up each key value in the pre-computed endpoints, we will only need to look for values which are *Distinguished Points.*

Here, care should be taken to avoid loops. When building a chain, there is a small probability of a loop, in which case we may never reach a distinguished point. In this rare event we just keep any such loop chain. The additional computational and storage complexities are negligible.

The second part of the space-time attack is the online attack. At this stage we assume that r tables with m different chains, each of length t were computed and stored. Each such chain is stored as a pair of SP and EP.

Given a cipher text y' we can now find a key $k^{'}$ such that $E_{k'}(p) = y'$ as follows. The idea is to find the chain in which y' appears, and then find y''s predecessor in the chain, which is k'. We locate the chain by setting $k^0 = R_i(y)$, and then repeatedly applying $R_i(E_{k^j}(p))$ with the r different reduction functions. Once getting to a distinguished point we look it up in the i-th table. If matched, we found the chain represented as SP,EP. We can now repeat the $R_i(E_{k^j}(p))$ invocation starting from SP, until we find k' such that $y' = E_{k'}(p)$.

Neglecting logarithmic factors, we can conclude Hellman's space-time attack for block ciphers with online cost $T = tr$ (though only r expensive table lookups), space $M = mr$, pre-computation $P = trm = N$. Together, these yield $TM^2 = N^2$.

Hellman's attack can be quite practical. In fact, Oechslin demonstrates in [17] a very feasible implementation of Hellman's space-time attack for breaking Windows passwords. That work is based on the fact that the key space is rather small, 2^{37}, and on the fact that Windows password encryption uses the password to encrypt a fixed known plain text.

That said, Hellman's method has two main drawbacks. The first is the pre-computation cost, which is equal to the entire key space size N. The second is that it is a chosen plain-text attack. All the table values were computed using a chosen plain text and are relevant only for attacking that plain text cipher.

Recently, Biryokuv and Shamir [4] extended space-time attacks for stream ciphers. A stream cipher works as a state machine that is initialized with a secret key and outputs a keystream sequence that contains bits from the internal state of the machine. Encryption consists of xor-ing the keystream bits with the plain text. Once we find a correct state of the stream cipher machine, not necessarily the initial key or the first state but **any** state, the remainder of the stream cipher output is predictable. Hence, the search space K is no longer the initial key space but rather the internal stream cipher state. That is, given a state s of the stream cipher, the next keystream $k(s)$ (of some pre-determined length) produced by the stream cipher is determined. Now, given a known plain-text p' and its cipher-text c', we can determine whether $k(s)$ is the key producing the cipher and conclude that the stream-cipher's internal state is s. This may be done for any known plain-text, not a specifically chosen plain-text p as before.

Hellman's attack framework presented above is used in a similar way here with one important change. The chain step maps an internal state of the stream cipher into the appropriate keystream it generates, and from the keystrem is reduces back using a reduction function to an internal state. The rest of the parameters – N, m, t, r and the distinguished points can be used in the same way.

When working on stream ciphers, Biryukov and Shamir explain how the two main drawbacks for block ciphers are solved. Cipher stream encryption is used as a one time pad for the plain text. Therefore, given any exposed plain-text, we recover the keystream with which it is encrypted. This keystream is the same for a given internal state of the streamcipher, regardless of the plaintext it encrypts. Given an exposed cipher text, we first (trivially) find the keystream that encrypts it, and then we attempt to recover the stream-cipher's internal state that results in this keystream. Hence, this is a **known** plain text attack and not a chosen plain text attack as in the block cipher case. The distinction is huge, since we can use a one-time preparation stage for all future attacks on the stream-cipher.

We can also use this fact to reduce the search space using multiple known plain-texts. Let us denote the number of exposed cipher texts given to the adversary by D. Since every exposed cipher text (equivalently, every keystream) corresponds to some unknown internal state of the stream cipher, we can find one of the keystreams with good probability if we cover only N/D of the states space. Thus, if an adversary can expose D cipher texts, it is enough to pre-compute only N/D of the states space. We therefore set $r = t/D$ instead of $r = t$, and compute only r different tables.

The space-time tradeoff for stream ciphers can now be written as time $T = Dtr = t^2$ (as in Hellman's attack), space $M = mt/D$, where $mt^2 = N$ which is better than before, and likewise the pre-computation $P = N/D$ is lower. We get a tradeoff of $TM^2D^2 = N^2$, which is much better than the block-cipher tradeoff of $TM^2 = N^2$.

Session-Id's Space-Time Tradeoffs Attacks on pseudo random generators can be addressed in a similar way to stream ciphers, thus we attack the PRNG internal state using a space-time attack. Below, we demonstrate the attack using the specific example of the Tomcat session-id generation algorithm. However, the same principles can be applied for other uses of the bits produced by a PRNG.

We can describe a PRNG as a state machine with states $x_1, x_2, x_3, \ldots$. In any state x_n, some bits are made available as output, and then the PRNG shifts to state x_{n+1}. Consequently, there is a deterministic sequence of bits $b_1, b_2, b_3, \ldots$ produced by the PRNG from any particular state x_n onward. For example, in java. util .Random(), the bits produced by the LCG state x_n are x_n itself. We denote $f(x_n)$ the deterministic 128-bit sequence produced by the PRNG from state x_n. The Tomcat session-id is generated as follows:

$$y := session\_id := MD5(f(x_n)) \tag{3}$$

Although the *MD5* transformation (or any other transformation, for that sake) effectively masks the values of the PRNG, we do not need to break *MD5* in order to predict session-id's. The session-id generation algorithm is deterministic and has no additional entropy sources along the algorithm. In this sense, our PRNG algorithm is similar to the stream-cipher where the encryption is based on the internal state cipher. Once we break any session-id value and reverse it to its state value x_n we can generate the entire series of next values.

Assume for the sake of demonstration that states are 64 bit values. The space-time attack we employ targets the "key space" K of PRNG internal states. Thus, $N = |K| = 2^{64}$.

We denote the transformation of Equation 3 by F. Given a value y, our goal is to find x such that $x = F^{-1}(y)$. We do this with a time-space tradeoff as follows. The start-point of chains are m randomly selected values k representing states of the PRNG. The chaining step from k_i to k_{i+1} is the transformation F followed by reduction functions R_j, $j = 1..r$. We use for R_j a truncation and a simple xor in order to reduce the 128 bits F values into a 64 bits internal PRNG states: $R_i(y_{0-127}) := y_{0-63} \oplus i$ where $i \in \{1 \ldots r\}$. As before, we maintain r tables, each containing m chains, and each terminating with a distinguished end-point (e.g., whose lowest $\log_2 t$ bits are zero). For each chain, we store only the start and the end points.

Suppose we are able to obtain D distinct valid session-id's. In practice, collecting session-id's from a working web-server is easy, and even a large number of sessions requested by the same client over a short time frame may not raise suspicion. Note that, these session-id's need not be consecutive, which is important in the framework of current distributed clients accessing a web server.

Our attack is then mounted as follows: For each of the D known session-id's y, and for $j = 1..r$, apply $R_j(F())$ repeatedly until a distinguished point is reached, and search for it in the j'th pre-made table. If found, then go back to the start point, and reach the state x_i such that $F(x_i) = y$. From state x_i onward, the session-id's generated by this server are predictable.

Letting $r = t/D$ as in the stream cipher attack, we obtain a tradeoff of $P = N/D$ pre-computation time, space $M = mt/D$ where $mt^2 = N$, and on-line computation time $T = t^2$. This yields $TM^2D^2 = N^2$.

For concrete numbers, we assume that it is possible to obtain $D = 1000$ valid session id's without raising suspicion. We put $t = 2^{22}$. Then our space of $N = 2^{64}$ PRNG states can be broken with storage $M = 2^{64-22-10} = 2^{32}$, and an on-line computation time $T = t^2 = 2^{44}$, both very feasible today with a moderately powerful workstation.

5.2 The Seed Attack

Some installations of Java Servlets use the java. security .SecureRandom PRNG, rather than java. util .Random. As outlined in Section 4 above, SecureRandom has an internal state of 160 bits. Hence, the general PRNG attack we described so far is not feasible against it. Here, we attack the protocol using another weakness, a low-entropy seed.

According to the description in Section 3, the space of seeds for the PRNG is determined by combining the range of possible clock readings in milliseconds (counted from 1970), and a value set by the method hashCode(). A day has about 2^{26} milliseconds and a year has about 2^{35}. Hence, the entropy of this value is between 26 to 35, depending on how accurately we can estimate a server's uptime. As for the value of hashCode(), Our reverse engineering of this method constrains it to within a small set of values, typically less than 128 different ones. Thus, the effective total range size of seeds is bounded between 2^{33} and 2^{42}. Certainly this is a space that can be searched exhaustively with a moderate computation power, especially if the uptime of a server is estimated relatively accurately.

While this is a weakness of the session-id generation algorithm, in itself it does not lead to a practical attack. The difficulty is in verifying the correctness of a guessed seed. The naive way is to involve the server. That is, one can guess a seed value, generate one or several “session-id’s” originating with the seed value, and attempt to “hijack” a customer session with this session id. As this procedure involves an interaction with the server for each guessed value, even for for a space of 2^{32} values it is very time consuming. Moreover, it would be very easy to detect that such an attack is going on at the server side. The server can protect itself against repeated connection attempts from the same domain over a short period of time by slowing down its response or refusing recurring attempts, and thus thwart the entire attack.

Our strategy is therefore to mount an almost entirely off-line attack as follows:

1. Get a valid session-id by connecting to the attacked web server. Mark this valid session-id as Sid.
2. Set T as an upper limit for the server uptime, since the last reboot. The value is in milliseconds.
3. Set $hash\_min, hash\_max$ as the lower and upper limit on the JVM hashCode(). Mark $\Delta_{hash} = hash\_max - hash\_min$.
4. Set sid_min, sid_max as the minimal and maximal number of valid session-id’s assigned so far by the attacked server. Mark $\Delta_{sid} = sid\_max - sid\_min$.
5. Generate all the possible session-id’s using all the possible $T \times (hash\_max - hash\_min)$ seeds, and for each potential seed, producing $(sid\_max - sid\_min)$ session-id’s. Compare Sid against this space, until a valid seed is revealed.

The above ignores the variability that different architectures and JVM versions may have in generating hashCode() values. If that is not known by the attacker, this should incur a multiplicative factor over the range of possible hashCode() values.

In the above attack, the size of the potential sessions-id’s space is 2^E, where the exponent E is given by the following sum:

$$E = \log_2(T) + \log_2(\Delta_{hash}) + \log_2(\Delta_{sid}) \tag{4}$$

If we take fairly conservative values, a server up-time of a month, hash values range 128, and valid session-id range 32,000 we get $E = 29 + 7 + 15 = 51$. This is certainly a searchable space.

6 Conclusions

This paper presents a practical attack on one of today’s main E-commerce building blocks, the session-id. Our attack shows that the presumably secure 128 bits can be broken using 2^{64} or less computation steps. Our attack can be mounted using limited computing resources, and has the same communication fingerprint of a legitimate user accessing the attacked web server. Hence, it is difficult for a server to detect and stop such an ongoing attack.

We implemented the attack and tested it under distilled environment conditions. In our case, we set up a Tomcat server and obtained session-id’s from it. We staged our attack on the same machine, so any uncertainty about Java versions and platforms was completely alleviated. Given the session-id's we obtained, we were able to predict the PRNG sequence within a day of CPU time. We did not try our attack on working servers to avoid legal complications.

Beyond the attack on session-id generation, we present a general scheme with a space-time tradeoff for attacking pseudo random number generators. To the best of our knowledge, this is the first space-time tradeoff for PRNG attacks. The attack may have important ramifications on presumably secure uses of PRNGs, such as BlumBlumShub [5], and emphasizes the need for deploying these with a large internal state.

This paper proves again a common cryptographers' knowledge. The complexity of a security scheme does not make it secure; nor is it made secure by using building blocks such as one way functions and secure pseudo random number generators.

It is important to note that Tomcat bring web server administrator the option to harden the session-id generation. The simple option is to add secret entropy to the seed. Other options require either using a different random number generator or a different session-id scheme.

The Tomcat web server is an open-source project. As such, it is an easy target for analysis, through both dynamic and static reverse engineering. The equivalent "binary only" attack requires more sisyphean work, usually through the low level assembly code. In a sense, this is the Achilles' heel for the security aspects of open source code. We believe that this is true only for the short term. In the long term, an open source project can benefit from a large audience testing its security, while closed projects might wrongly be presumed secure just because their study is complex. One such example is the GSM encryption scheme, which was considered secure for long, but was recently proven not so [3].

Acknoledgments

The authors wish to thank Tzachy Reinman and Yaron Sella for reviewing early drafts of the paper.

References

1. Apache Software Foundation (ASF). Apache jakarta tomcat. `http://jakarta.apache.org/Tomcat`.
2. Apache Software Foundation (ASF). Apache web server. `http://www.apache.org`.
3. E. Barkan, E. Biham, and N. Keller. Instant ciphertext-only cryptanalysis of gsm encrypted communication. In *Proceedings of CRYPTO'2003, LNCS 2729*, pages 600–616, 2003.
4. A. Biryukov and A. Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *Lecture Notes in Computer Science 1976, proceedings of ASIACRYPT'2000*, pages 1–13, 2000.
5. L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. 15:364–383, 1986.

6. J. Boyar. Inferring sequences produced by a linear congruential generator missing low-order bits. *Journal of Cryptology*, 1(3):177–184, 1989.
7. Datarescue. Ida: The interactive disassembler. `http://www.datarescue.com/idabase/`.
8. T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force, January 1999.
9. R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. J. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.
10. Hartman. Method and system for placing a purchase order via a communications network, September 1999. U. S. patent 5,960,411.
11. M. E. Hellman. A cryptanalytic time-memory trade off. *IEEE Trans. Inform. Theory*, IT-26:401–406, 1980.
12. M. Heuse. Websphere cookie and session-id predictability, 10 2001. `http://www.securiteam.com/windowsntfocus/6Q0020K2UU.html`.
13. D. Kristol and L. Montulli. HTTP state management mechanism. RFC 2965, Internet Engineering Task Force, October 2000.
14. M. Roth. JSR 152: JavaServer PagesTM 2.0 Specification, November 2003. `http://jcp.org/aboutJava/communityprocess/final/jsr152/index.html`.
15. Sun Microsystems. The java virtual machine version 1.4.2. `http://java.sun.com/j2se/1.4.2/index.jsp`.
16. Netcraft. Market share for top servers across all domains august 1995 - march 2004. `http://news.netcraft.com/archives/web_server_survey.html`.
17. P. Oechslin. Making a faster crytanalytical time-memory trade-off. In *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, Santa Barbara, California, USA, August 2003. 23rd Annual International Cryptology Conference, Springer. ISBN 3-540-40674-3.
18. R. Rivest. The MD5 message-digest algorithm. RFC 1321, Internet Engineering Task Force, April 1992.
19. Y. Yoshida. JSR-000154 JavaTM Servlet 2.4 Specification (Final Release), November 2003. `http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html`.