Analysis of “Real-World” Cryptographic Systems

Dissertation submitted for the degree of

“Doctor of Philosophy”

by

Zvi Guterman

Submitted to the Senate of the Hebrew University in Jerusalem

May 2008
This work was carried out under the supervision of

Prof. Noam Nissan and Dr. Benny Pinkas.
# Contents

1 Introduction  
1.1 Properties required of pseudo-random number generators. 2  
1.2 Existing PRNG implementations. 3  
1.2.1 Analysis of PRNGs. 4  

2 Hold Your Sessions: An Attack on Java Session-id Generation 5  
2.1 Introduction 5  
2.2 Stateful Web Browsing 8  
2.3 Tomcat Session-id Generation Algorithm 9  
2.3.1 Allocation 10  
2.3.2 Initialization 11  
2.4 Java Object.toString() Algorithm 12  
2.5 Attacks 14  
2.5.1 Space-Time Tradeoffs for PRNG Attacks 14  
2.5.2 The Seed Attack 21  
2.6 Conclusions 23  

3 Analysis of the Linux Random Number Generator 25  
3.1 Introduction 25  
3.1.1 The Linux Pseudo-Random Number Generator (LRNG) 26
4.5 Comparison to the Linux PRNG ............................................. 79
4.6 Conclusions ................................................................. 80
  4.6.1 Conclusions ......................................................... 80
  4.6.2 Recommendations ................................................ 81
  4.6.3 Open Problems ..................................................... 82

5 Summary ........................................................................... 85

A LRNG Entropy Collection ............................................ 95

B LRNG Probability Calculations .................................... 97

C RC4 ........................................................................... 99
Chapter 1

Introduction

This dissertation studies the relationship between theory and practice through the investigation of the security of real-world cryptosystems. There are several “environmental factors” that differentiate this type of research from pure cryptanalysis. First, even in 2008, many designers believe in security by obscurity. Hence, understanding the algorithm behind a security system often requires a challenging reverse engineering process, which is a major obstacle to the analysis. Second, when implementing cryptographic algorithms in a real system new issues arise, presenting new problems and additional attacks which cannot be identified if the cryptographic algorithm is investigated in isolation. On the other hand, some attacks against cryptographic algorithms may not be easily applied to systems which use them. Rescorla [48] presented an example where recent attacks on hash functions (e.g., [53]) did not affect higher order algorithms, which use these functions, such as TLS.

Our results summarize the study of three security systems. The first (Chapter 2 and [26]) analyzes the security of the Apache Java web server. Our cryptanalysis proves that it is feasible to break the separation between different users communicating with the server. Hence, using our attack, an adversary is able to break into other users’ interaction with the server. Our second result (Chapter 3 and [27]) presents an analysis of the most commonly
used open source PRNG — the Linux PRNG. The complex structure of the Linux PRNG hides flaws in the design as well as vulnerability to a cryptanalysis under certain conditions. Our latter result (Chapter 4 and [16]) presents an analysis of the Windows random number generator. Our study reveals the exact implementation as well as a cryptanalysis attack and some design flaws.

1.1 Properties required of pseudo-random number generators.

A pseudo-random number generator must be secure against external and internal attacks. The attacker is assumed to know the code of the generator, and might have partial knowledge of the entropy used for refreshing the generator’s state. We list here the most basic security requirements, using common terminology (e.g., of [6]). A more detailed list of potential vulnerabilities appears in [35].

- **Pseudorandomness.** The generator’s output looks random to an outside observer.

- **Forward security.** An adversary which learns the internal state of the generator at a specific time cannot learn anything about previous outputs of the generator.

- **Break-in recovery / backward security.** An adversary which learns the state of the generator at a specific time does not learn anything about future outputs of the generator, provided that sufficient entropy is used to refresh the generator’s state.

The Pseudorandomness requirement is sufficient in attack scenarios where the adversary does not have knowledge of the generator’s state. In many scenarios, however, an adversary might be able to learn the state of the generator; for example, by bypassing access restrictions of the operating system, or by reading the state from memory or from the hard disk if it is
stored there. The *forward security* requirement ensures that an adversary which learns the generator’s state at a certain time cannot gain information about the output of the generator at previous times. The *break-in recovery* requirement ensures that learning the state at a particular time does not compromise all future outputs of the generator.

1.2 Existing PRNG implementations.

In the past, PRNGs were either a separate program or a standard library within a programming language. The evolution of software engineering and operating systems introduced PRNGs which are part of the operating system. From a cryptographic point of view, this architecture has three main advantages: (1) the ability to introduce more complex algorithms which are implemented using unique kernel optimization, (2) the ability to use kernel based entropy events as input to the generator, and (3) the fact that in a multi-user, multi-threaded environment, many consumers can read random bits, and therefore an adversary might be prevented from reading a long stream of consecutive bits of the PRNG output.

A published overview of the use of cryptography in OpenBSD [14] includes a very informative introduction to this operating system’s usage of PRNGs, and of cryptography’s general role in this operating system. The PRNG of the FreeBSD operating system is described in [43]. It uses time stamps of events as an entropy (i.e., physical randomness) source. These are hashed using AES [44] into two pools, each of 256 bits. When output is extracted, AES encryption is used to repeatedly encrypt a 256-bit counter, using an encryption key that is taken from the entropy pools. FreeBSD implements a single non-blocking device and the authors declare their preference of performance over security.

Castejon-Amenedo et al. [32] propose a PRNG for UNIX environments. Their system is composed of an entropy daemon and a buffer manager that handles two devices—blocking and non-blocking. The buffer manager divides entropy equally between the two devices, such
that there is no entropy that is used in both. A notable advantage of this scheme is the absolute separation between blocking and non-blocking devices, which prevents launching a denial-of-service attack on the blocking device by using the non-blocking device (such an attack is possible in Linux, as we later discuss in Section 3.3.4).

1.2.1 Analysis of PRNGs.

A comprehensive discussion of the system aspects of PRNGs, as well as a guide to designing and implementing a PRNG without the use of special hardware or of access to privileged system services, is given by Gutmann [24]. Issues related to operating system entropy sources were discussed in a recent NIST workshop on random number generation [33, 25].

An extensive discussion of PRNGs, which includes an analysis of several possible attacks and their relevance to real-world PRNGs, is given by Kelsey et al. in [35]. Additional discussion of PRNGs, as well as new PRNG designs appear in [34, 18].

The recent work of Barak and Halevi [6] presents a rigorous definition and an analysis of the security of PRNGs, as well as a simple PRNG construction. This work suggests separating the entropy extraction process, which is information-theoretic in nature, from the output generation process. Their construction is based on a cryptographic pseudo-random generator $G$, which can be implemented, for example, using AES in counter mode, and which does not use any input except for its seed. The state of the PRNG is the seed of $G$. Periodically, an entropy extractor uses system events as an input from which it extracts random bits. The output of the extractor is xored into the current state of $G$. This construction is much simpler than most existing PRNG constructions, yet its security was proved in [6] assuming that the underlying building blocks are secure. We note that our analysis shows that the Linux PRNG construction, which is much more complex than that of [6], suffers from weaknesses which could have been avoided by using the latter construction.
Chapter 2

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

2.1 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 Chapter \(^1\), 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 chapter 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 [45] of servers using Servlets, this chapter 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

\(^1\)The research described in this chapter was previously published as [26]
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 [19] is stateless, these sites need a way to maintain state over a stateless protocol. Section 2.2 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 [5] is an open-source software projects community. The Apache web server is the foundation’s main project. According to Netcraft [45] 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.

In many of these sites, the procedure for an actual credit-card purchase requires a secure TLS [15] sessions, separated from the “browsing and selection” session. However, this is not always the case. For example, Amazon’s patented [28] “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 [12] 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 rest of this chapter is organized as follows. In Section 2.2 we describe the HTTP state mechanisms. In Section 2.3 we describe and analyze the Tomcat session-id generation algorithm. Java hashCode() study is presented in Section 2.4. In section 2.5 we present our attacks on the session-id. We conclude in Section 2.6.
2.2 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 [19] (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 [36] 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
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.

### 2.3 Tomcat Session-id Generation Algorithm

In this section, we describe our study of Tomcat 5 [4], the Apache Java implementation for Servlet 2.4 [54] and JSP 2.0 [37] specifications. We study version 5.0.18, which was released on January 2004. Our full study involves additional, and more challenging reverse-
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.

2.3.1 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 2.3.1,2.3.2 presents these functions.

2. The 16 bytes obtained from getRandomBytes are mixed using a digest function which is MD5 [49] 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’.
\[ x_n := \begin{cases} 
\text{initial seed} & n = 0 \\
(25, 214, 903, 917 \times x_{n-1} + 11) \mod (2^{48} - 1) & n > 0 
\end{cases} \]

Figure 2.3.1: java.util.Random. \( x_n \) holds the PRNG next output.

\[ x_n := SHA1(s_n) \quad n = 0 \]

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

Figure 2.3.2: java.security.SecureRandom. \( x_n \) holds the PRNG next output and \( s_n \) is the internal state.

### 2.3.2 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., [30].

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 \( C = \text{System.currentTimeMillis()} \). This is 64 bit field measuring the time since January 1, 1970 in milliseconds.

2. Set \( Entropy = \text{toString(org.apache.catalina.session.ManagerBase.java)} \). 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 2.4 describes our study of the \( X \) value and how we can predict it.

3. Set \( Seed = f(C, Entropy) \). The function \( f \) is depicted in Figure 2.3.3. 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 2.5.

2.4 Java Object.toString() Algorithm

The Java Object.toString() function is used by the initialization algorithm presented in Section 2.3 for generating the PRNG seed. In this section, we take a 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 [41] 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 [13] disassembler we get

\[
hashCode(object) := \begin{cases} 
(a \times x_n + b) \mod m & \text{first object access} \\
\text{hash is given from a history table} & \text{otherwise}
\end{cases}
\] (2.1)

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.

2.5 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.

2.5.1 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 [9]. 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 [29] 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 ciphertext $y$ 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:

$$SP := k^0 \longrightarrow k^1 := R(E_{k^0}(p)) \longrightarrow \ldots \longrightarrow EP := k^{t-1} := R(E_{k^{t-2}}(p))$$ \hspace{1cm} (2.2)

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 [47] 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 plaintext 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 [9] 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 plaintext \( p' \) and its ciphertext \( c' \), 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 plaintext, not a specifically chosen plaintext $p$ 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 keystream 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 plaintext, we recover the keystream with which it is encrypted. This keystream is the same for a given internal state of the stream cipher, 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 plaintexts. 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 $T M^2 = N^2$.

**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:

$$y := \text{session id} := MD5(f(x_n)) \quad (2.3)$$

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 2.3 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: $R_i(y_{0..127}) := y_{63} \oplus i$ 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.
2.5.2 The Seed Attack

Some installations of Java Servlets use the java.security.SecureRandom PRNG, rather than java.util.Random. As outlined in Section 2.4 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 2.3, 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 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:

$$E = \log_2(T) + \log_2(\Delta_{hash}) + \log_2(\Delta_{sid})$$ (2.4)

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 with a small computing cluster.
2.6 Conclusions

This Chapter 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 [7].
Chapter 3

Analysis of the Linux Random Number Generator

3.1 Introduction

Randomness is a crucial resource for cryptography, and random number generators are therefore critical building blocks of almost all cryptographic systems. The security analysis of almost any system assumes a source of random bits, whose output can be used, for example, for the purpose of choosing keys or choosing random nonces. Weak random values may result in an adversary ability to break the system, as was demonstrated by breaking the Netscape implementation of SSL [22], or predicting Java session-ids [26].

Since a physical source of randomness is often too costly, most systems use a pseudo-random number generator. The state of the generator is seeded, and periodically refreshed, by entropy which is gathered from physical sources (such as from timing disk operations, or from a human interface). The state is updated using an algorithm which updates the state and outputs pseudo-random bits.
This Chapter\footnote{The research described in this chapter was previously published as [27]} studies the Linux pseudo-random number generator (which we denote as the LRNG). This is the most popular open source pseudo-random number generator, and it is embedded in all running Linux environments, which include desktops, servers, PDAs, smart phones, media centers, and even routers. In Chapter 4 we study the Microsoft Windows Random Number Generator (which we denote as WRNG) and compare between the two generators.

3.1.1 The Linux Pseudo-Random Number Generator (LRNG)

The Linux kernel is an open source project developed in the last 15 years by group of developers led by Linus Torvalds. The kernel is the common element in all various Linux distributions, on all types of devices.

The output of the LRNG can be used by internal kernel functionalities which use random bits, and by calls to its application programming interface (API). The Linux kernel uses random data for various purposes, such as generating random identifiers, computing TCP sequence numbers, producing passwords, and generating SSL private keys. Within the kernel, the interface for receiving random values from the LRNG is the function \texttt{get\_random\_bytes(*buf, nbytes)}.

The API to the LRNG is through two device drivers named \texttt{/dev/random} and \texttt{/dev/urandom}. Both devices let users read pseudo-random bits. The difference between the two is in the stated level of security of the random bits, and the resulting delay. The first device (\texttt{/dev/random}) outputs “extremely secure” bits\footnote{This wording is according to the LRNG designer. Essentially, this type of output is only available when enough physical entropy is gathered. Our discussion of the resulting security is in Section 3.3.4.} and may block the user until such bits can be generated by the system. The second device (\texttt{/dev/urandom}) outputs less secure bits but its output is never blocked. Section 3.2.4 explains the difference between the two devices.
Why reverse-engineering the LRNG is not easy. The LRNG is part of an open source project and therefore one might assume that its source code is available for public scrutiny and that its security can be easily analyzed (or at least, is not based on “security by obscurity”). However, the LRNG is not well documented and there is no clear description of the implemented algorithm. The LRNG is composed of about 2500 lines of code, and in addition, hundreds of code patches were applied to the code during the last five years (and consequently, the available documentation does not always reflect the current code). One example of the complexity of the LRNG code is the fact that for 17 months the LRNG code included a bug in which entropy addition used a vector of size $4 \times n$ instead of $n$. We also note that throughout our analysis we were not helped by any of the LRNG authors.

These factors turned the reverse-engineering of the LRNG into a challenging task. We therefore combined static reverse-engineering of the source code of the Linux kernel with dynamic tracing to present a clear algorithmic representation of the LRNG (see Section 3.3). Dynamic reverse-engineering is not simple in this case, due to two main restrictions. The first is the fact that any kernel change requires a new build and installation. This process takes a couple of hours to finish, and performing it dozens of times makes the process very tedious. The second, and more troubling restriction, is the fact that any change made to the kernel may also result in some influence on the kernel “noise generation” and hence on the LRNG behavior. We therefore implemented a user-mode simulator of the LRNG as part of our analysis. It can be downloaded from the authors’ web page.

As a final note on the complexity of the implementation, we add that the complexity of the algorithm, the lack of documentation, and the high volume of changes to the LRNG code, resulted in dozens of programming bugs. Many of these bugs resulted in security vulnerabilities during the last five years.

The basic structure of the LRNG. At a high level, the LRNG can be described as three asynchronous components. The first component translates system events into bits
which represent the underlying entropy. The second component adds these bits to the
generator “pool”. When bits are read from the generator, the third component applies three
consecutive SHA-1 operations to generate the output of the generator and the feedback which
is entered back into the pool.

Each sample of “randomness” originating from system events is collected as two 32-bit
words. The first word measures the time of the event and the second word is the event
value, which is usually an encoding of a pressed key, a mouse move or a drive access. In
order to keep track of the amount of physical randomness which is added to the pool, the
LRNG holds a counter for counting an estimate of this value, which is calculated as a
function of the frequencies of the different events. The LRNG denotes this value as entropy
(see Section 3.2.4 for the exact definition) although it is different than the classical entropy
definition by Shannon [51].

Analysis of open-source security packages. It is hard to examine the security of im-
plementations of security packages, and, in the case of proprietary software, one usually has
to trust the software authors. However, even in the case of open source security packages it
is hard to trust security, since the fact that source code can be read does not imply that it
is actually examined by security experts. For example, the work of Nguyen [46] examined
the (open) source code of the GNU Privacy Guard secure email software (GnuPG or GPG),
and identified several cryptographic flaws. The most serious of these flaws has been present
in GPG for almost four years.

3.1.2 Our Contributions

This Chapter describes research conducted on analyzing the LRNG. It provides the following
contributions:

• Publication of a description of the LRNG algorithm. As described above, a considerable
amount of work was required in order to analyze the LRNG code and provide a high-level description of the underlying algorithm.

- An attack which breaks the forward-security of the LRNG. Namely, we show how, given a state of the generator, it is possible to reconstruct previous states. The time complexity of this attack is $2^{64}$ or $2^{96}$, depending on the attack variant. The memory complexity of the attack is $O(1)$.

- An analysis of the amount of entropy which is added to the generator in one typical implementation.

- An analysis of the security of an implementation on a disk-less system (an OpenWRT based router).

- We also identify some vulnerabilities in the current implementation (including an easy deny of service attack) and provide recommendations for improving future implementations of pseudo-random number generators.

3.2 The Structure of the Linux Random Number Generator

Our study is based on version 2.6.10 of the Linux kernel, which was released on December 24, 2004.

3.2.1 General Structure

The generation of random numbers in Linux is composed of three asynchronous procedures. In the first procedure, operating system entropy is collected from various events inside the
kernel. In the second procedure, entropy is fed into an LFSR-like pool, using a mixing function. When random bits are requested, the third procedure occurs, output is generated and the pool is updated. The only non-linear cryptographic operation used by these procedures is the SHA-1 hash function. (We note that the recent attacks on the collision resistance of SHA-1 seem irrelevant for the purpose of attacking the LRNG.)

**Pools and counters.** Figure 3.2.1 describes the LRNG flow. The internal state is kept in three entropy pools: *primary*, *secondary* and *urandom*, whose sizes are 512, 128 and 128 bytes, respectively. Entropy sources add data to the primary pool; output from the primary pool is extracted and fed to the secondary and urandom pools, while the LRNG output is extracted from the secondary pool or from the urandom pool. During the extraction operation, the inner state of a pool is modified in a feedback manner.

![Diagram of LRNG flow](image)

Figure 3.2.1: The general structure of the LRNG: Entropy is collected (C) from four sources and is added (A) to the primary pool. Entropy is extracted (E) from the secondary pool or from the urandom pool. Whenever entropy is extracted from a pool, some of it is also fed back into this pool (broken line). The secondary pool and the urandom pool draw in entropy from the primary pool.

---

3 As is described below, system entropy might also be added to the secondary pool, if the LRNG estimates that the primary pool has full entropy.
Each pool has its own entropy estimation counter. This is an integer value between zero and the pool size in bits, which indicates the current estimated entropy of the pool. When output is extracted from the pool this counter is decremented, and when entropy is added the counter is incremented. An entropy counter is always decremented by the number of extracted bits. Incrementing the counter is more complex. If the added bits originate from one of the entropy sources, then their entropy is estimated, and the counter is incremented accordingly. The entropy estimation uses the timing of the last few events of the same entropy source (see discussion below). If the entropy bits are transferred from the primary pool, the entropy counter of the receiving pool is incremented by the number of transferred bits.

The entropy counter of the secondary pool plays a crucial role when extracting entropy using the blocking interface, /dev/random. Its task is to determine whether there is enough entropy in the pool to supply the requested amount of random data. If the answer is negative, the LRNG tries to transfer entropy from the primary pool to the secondary pool, and if this fails, it blocks and waits until some entropy input arrives and increments the entropy counter.

**Adding physical entropy.** Entropy bits are added to the primary pool from external sources. Desktop and server PCs can use four different sources: mouse and keyboard activity, disk I/O operations, and specific interrupts. When such an event occurs, it produces a 32-bit word representing its timing and a 32-bit word encoding its attributes (e.g., which key was pressed). In addition, the differences between the timings of successive events of the same type are used to estimate the entropy provided by this event. In Section 3.2.4 we define this procedure in detail.

Due to the asynchronous nature of the system, collected entropy cannot be simply added to the pools but is rather collected and batched. A few times a minute the batched data is added to the pools (in a process described in Section 3.2.5). The default operation is to add entropy to the primary pool. If this pool is full (its entropy count equals 4096) entropy
is added to the secondary pool. When the secondary pool is full the process returns to
the primary pool, and so on. Entropy is never added to the urandom pool. This process
increments the entropy counter of the respective pool by the estimated entropy amount.

**Generating output.** Random bits are extracted from one of the three pools: they are
extracted from the urandom pool when the user uses `/dev/urandom` and when the kernel
calls `get_random_bytes`; from the secondary pool when the user uses `/dev/random`; and
from the primary pool when one of the two other pools does not have enough entropy and
needs re-filling. The process of entropy extraction includes three steps: updating the pool’s
contents, extracting random bits to be output, and decrementing the entropy counter of the
pool. This process involves hashing the pool contents using `SHA-1`, and adding the results
to the pool. We study each of the LRNG steps in the following sections.

### 3.2.2 Initialization

Operating system startup includes a sequence of routine actions. This sequence includes the
initialization of the LRNG with constant operating system parameters and with the time-
of-day, and additional disk operations and system events which affect the LRNG using the
interface for adding external entropy (discussed in Section 3.2.5). This sequence of operations
might be easily predicted by an adversary, especially in systems which do not have a hard
drive. If no special actions are taken, the LRNG state might include very limited entropy.
(For example, the time of day is given as a count of seconds and of micro-seconds, each
represented as a 32-bit value. In reality these values have very limited entropy as one can
find computer uptime within an accuracy of a minute, which leads to a brute-force search of
only $60 \times 10^6 < 2^{26}$ different options.)

To solve this problem, the LRNG simulates continuity along shutdowns and startups.
This is done by saving a random-seed at shutdown and writing it back to the pools at
startup. A script that is activated during system startups and shutdowns uses the read and write capabilities of the /dev/urandom interface to perform this operation.

During shutdown the script reads 512 bytes from /dev/urandom and writes them to a file, and during startup these bits are written back to the /dev/urandom device. This device is defined such that writing to it modifies the primary pool and not the urandom pool (as one could expect from its name). The resulting operations applied to the primary pool are pretty much identical to the effect of receiving these 512 bytes as the encoding of system events, and adding them to the primary pool using the usual procedure for adding entropy, which is outlined in Section 3.2.5. The only difference is that the added bytes do not increment the entropy estimation. The secondary pool and the urandom pool are refreshed by the primary pool, and therefore the script affects all three pools.

It is important to note that this script is part of a Linux distribution package, such as RedHat, and not part of the kernel code itself (this is also the reason that the script must interact with the pools using a device driver rather than reading and writing from/to the pools). The author of the LRNG ([52]) instructs Linux distribution developers to add this script in order to ensure the unpredictability of the LRNG at system startups. This implies that the security of the LRNG is not completely stand-alone, but dependent on an external component, which can be predictable in certain Linux distributions.

**Security implications:** Some Linux distributions, such as the KNOPPIX distribution which is a bootable PC system on a CD or a DVD [1], or the OpenWRT Linux distribution for routers [2], do not use a script of this type and therefore initialize the LRNG from scratch in each reboot. This might result in an initial LRNG state which is rather predictable. It is also obvious that when the seed is saved in a file on the hard disk after system shutdown, anyone who can physically access the disk can read that file and learn the seed, or alternatively replace the seed with a different value (say, a previous value of the seed for which the adversary already recorded the output of the generator). Access permissions are not too
helpful here, since they are related to the operating system and not to the hard disk.

### 3.2.3 Collecting Entropy

In a PC environment, the LRNG collects entropy from events originating from the keyboard, mouse, disk and system interrupts. When such an event occurs, two 32-bit words are used as input to the entropy pools. The first word encodes the timing of the event in \textit{jiffies} (namely, the number of milliseconds from the time the machine was booted) or in cpu-cycles granularity (currently cpu-cycles granularity is only used on SMP). The second word encodes the event type. For example, in case of a keyboard event the word encodes the key that was pressed. Table 3.1 presents the number of unknown bits per each type of event. Note that the actual entropy of these events is much lower, as most of them are predictable to a large extent. Appendix A describes in detail how each value is calculated.

<table>
<thead>
<tr>
<th></th>
<th>Keyboard</th>
<th>Mouse</th>
<th>Hard Drive</th>
<th>Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>8</td>
<td>12</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.1: The number of unknown bits in operating system events.

In other environments, the LRNG gathers entropy from the available resources. For example, an OpenWRT router does not include a hard disk, mouse and keyboard and therefore these cannot be used as an entropy sources. On the other hand, the router collects entropy from network events.

### 3.2.4 Estimating the Entropy Amount

One of the fundamental issues in using physical entropy is estimating the amount of entropy which is added to the generator, and more generally, estimating the current amount of entropy in the pools.

As noted above, the difference between \texttt{/dev/random} and \texttt{/dev/urandom} is that the
/dev/random interface does not return more bits than its current entropy estimation and thus might block. The /dev/urandom interface, and the kernel interface (get_random_bytes), return any number of pseudo-random bits, according to the request. This difference implies that entropy estimation is important mainly for the /dev/random interface.

The LRNG estimates the amount of entropy of an event as a function of its timing only, and not of the event type. The estimate is done in the following manner:

**Definition 1.** Let $t_n$ denote the timing of event number $n$. Define

$$\delta_n = t_n - t_{n-1}$$
$$\delta^2_n = \delta_n - \delta_{n-1}$$
$$\delta^3_n = \delta^2_n - \delta^2_{n-1}$$

Note that $t_n$, $\delta_n$, $\delta^2_n$, $\delta^3_n$ are each 32 bits long.

The amount of entropy added by the event is defined to be $\log_2 \left( \min( |\delta_n|, |\delta^2_n|, |\delta^3_n| )_{[19-30]} \right)$ where $S_{[a-b]}$ denotes bits $a$ to $b$ (inclusive) of $S$ (where location 0 is the MSB).

If $\min( |\delta_n|, |\delta^2_n|, |\delta^3_n| )_{[19-30]} = 0$ then the estimate is 0. (Even if the estimate is 0, the event is used to update the state of the LRNG. The entropy count, however, is only updated if the entropy estimate of the event is positive.)

This estimation is relevant only in the case of adding entropy from OS sources to the pools. When a user writes data to one of the device drivers (/dev/random or /dev/urandom) the entropy counter is not changed. When $n$ bits are extracted from a pool the entropy estimation is decremented by $n$. When bits are transferred from one pool to another the first is decremented and the second incremented—both by the amount of transferred bits.
3.2.5 Updating the Pools

The mechanism for updating the pools is based on a TGFSR (Twisted Generalized Feedback Shift Register [38, 39]). The main advantage of the TGFSR is its extended cycle length for any initial seed. The period of a TGFSR with a state of 128 words (on a 32-bit PC) can be $2^{128 \times 32} - 1$ steps.

**Definition 2 (TGFSR).** A series $a_i \in \{0, 1\}^w$ and a matrix $A_{w \times w}$ are a TGFSR based on a primitive polynomial $x^p + x^{p-t_1} + \cdots + x^{p-t_m} + 1$ ($1 \leq t_1, \ldots, t_m < p$) if and only if

$$a_i = a_{i-p+t_1} \oplus \cdots \oplus a_{i-p+t_m} \oplus a_{i-p} A$$

$$i = p, p+1, \ldots$$

The only input to the TGFSR is the initial value of the state (which is a $p \times w$ bit seed), and in each iteration the internal state is used to generate the new state.

The shift register used in the LRNG is based on the TGFSR (and its implementation which is described in [38]) but is different from it since it adds entropy in each iteration. We define the pools of the LRNG as arrays of length $m$ words ($m = 32$ or $m = 128$) which are indexed by an index $j$.

**Adding entropy.** Entropy is added in each round of state and output computation, as well as when entropy is added from external sources, or from the primary pool to the secondary and urandom pools. Entropy is added by running the algorithm `add(pool1, j, g)` and updating the value of the index $j$, where $g$ is the new entropy word which is added to the pool. Figure 3.2.2 defines the update operation for a pool of size 32 words.

Each pool is updated based on a primitive polynomial. The polynomial is chosen according to the size of the pool, and therefore the secondary and the urandom pools share the same polynomial, which is $x^{32} + x^{26} + x^{20} + x^{14} + x^7 + x + 1$. The primary pool polynomial is $x^{128} + x^{103} + x^{76} + x^{51} + x^{25} + x + 1$, and the entropy addition for that pool is identical.
Algorithm add(pool, j, g):
    temp := g
    temp := temp xor pool[j]
    temp := temp xor pool[(j+1) mod 32]
    temp := temp xor pool[(j+7) mod 32]
    temp := temp xor pool[(j+14) mod 32]
    temp := temp xor pool[(j+20) mod 32]
    temp := temp xor pool[(j+26) mod 32]
    temp := (temp >> 3) xor table[temp & 7]
    // the last 3 bits of temp choose a table
    // entry which is xored to (temp >> 3)
    pool[j] := temp
    // table[] is defined as follows
    // table[0] = 0x0
    // table[1] = 0x3b6e20c8
    // table[2] = 0x76dc4190
    // table[3] = 0x4db26158
    // table[4] = 0xedb88320
    // table[5] = 0xd6d6a3e8
    // table[6] = 0xa00ae278
    // table[7] = 0xa00ae278

Figure 3.2.2: Pseudo-code of the entropy addition algorithm for the urandom and secondary pools. pool is the pool to add entropy to, g is the added entropy, table is a table with eight words, and j is the current position in pool.

to that of the smaller pools, except for using this polynomial for updating the pool (namely, xoring g with entries j, j + 1, j + 25, j + 51, j + 76 and j + 103 modulo 128).

Entropy addition can be analyzed by assuming that the generator is reseeded in each iteration. Alternatively, one might analyze it by assuming that the TGFSR is used to encrypt the entropy input. The LRNG reseeding process changes the elementary properties of the TGFSR. The long period can no longer be guaranteed, and the process is no longer a linear function of the initial state.

3.2.6 Extracting Random Bits

Entropy is extracted from the secondary pool in case of /dev/random and from the urandom pool in case of /dev/urandom or get_random_bytes. It is also extracted from the primary
pool for the purpose of refreshing the other pools.

Extracting entropy from a pool is not a simple operation. It involves hashing the extracted bits, modifying the pool’s state and decrementing the entropy estimate by the number of extracted bits.

Figures 3.2.3 and 3.2.4 present a pseudo-code and a diagram of the extraction algorithm. Entropy extraction is done in blocks of 10 bytes. The process is described for the case of the urandom or secondary pools, which are 32 words long. For simplicity the description does not include the steps of decrementing the entropy estimation, and the entropy refilling process. The algorithm applies the SHA-1 function to the first 16 words, and adds part of the result to location \( i \). It then applies a variant of SHA-1, which we denote as SHA-1’ (see below), to the right half of the pool, and adds parts of the result to locations \( i - 1 \) and \( i - 2 \). Finally, it applies SHA-1’ to the 16 words ending at location \( i - 2 \), and uses the result to compute the output in the following way: The output of SHA-1’ is 5 words (20 bytes) long. These words are folded as described in Figure 3.2.5, and the resulting 10 bytes are the output of the iteration. This output is copied to the target (user or kernel) buffer, and the number of bytes to be copied is updated. The loop continues until the requested number of bytes are output.

The SHA-1 variant being used. The first hash function used in the procedure is the original SHA-1 function (see [3] for the exact definition of SHA-1). Each of the following hash operations in this iteration, which we denote as SHA-1’, use for their five initial constant values (IV) the five output words of the previous hash result. (We note that if the LRNG had used the original SHA-1 function, the attacks in Section 3.3.1 would have been considerably more efficient, as is described there.)

Extracting randomness from the primary pool. Bits are extracted from the primary pool when it is required to refresh one of the other pools. In this case, the algorithm is slightly
Algorithm `Extract(pool, nbytes, i)`:

```markdown
while nbytes > 0
    tmp := SHA−1(pool[0..15])
    // the result is 5 words long
    add(pool, i, tmp[0])
    tmp := SHA−1'(pool[16..31])
    add(pool, i−1 mod 32, tmp[2])
    add(pool, i−2 mod 32, tmp[4])
    tmp := SHA−1'(pool[(i−2−15) mod 32 ...
    (i−2) mod 32])
    tmp := folding(tmp[0..4])
    // the result is 2.5 words long
    output(tmp, min(nbytes, 10))
    nbytes := nbytes − min(nbytes, 10)
    i := i − 3 mod 32
end while
```

Figure 3.2.3: Pseudo-code of the extraction algorithm. *pool* is a 32 word pool from which entropy is extracted, *nbytes* is the number of requested bytes, and *i* is the current position in *pool*.

Figure 3.2.4: Extraction algorithm

*input*: $W_0, W_1, W_2, W_3, W_4$

*output*: $W_0 \oplus W_3, W_1 \oplus W_4, W_2_{[0−15]} \oplus W_2_{[16−31]}$

Figure 3.2.5: Folding operation. Folding 5 words (160 bits) to 2.5 words (80 bits). $W_i$ denotes the $i^{th}$ word, $W_i_{[l−m]}$ denotes bits $l − m$ (inclusive) of the $i^{th}$ word.
different since the primary pool is longer. The SHA-1 (or SHA-1’) operation is applied to each of the eight 16 word chunks of the primary pool, and once more to the 16 words ending at location $i - 8$. The first eight SHA-1 operations update eight words in the pool, and the last operation generates the output.

3.3 Analysis

Our analysis of the security of the LRNG is composed of four parts: (1) a cryptanalytic attack on the forward security of the LRNG, (2) an analysis of the entropy added by system events, (3) observations on the insecurity of the LRNG in the OpenWRT Linux distribution for routers, and (4) observations on security engineering aspects of the LRNG, including a denial-of-service attack.

3.3.1 Forward Security

We first observe that the output which is extracted from a pool is calculated as the last state in the Extract algorithm. Namely, it is computed after the state of the pool is updated. This means that if the state of the pool at time $t$ is known then it is easy to compute the output which was extracted from the pool during its last state transition (i.e., the output which was computed in the transition from time $t - 1$ to time $t$). This is a flaw in the forward security of the LRNG, since it enables anyone who observes the state of the LRNG at a certain time to compute the last output of the LRNG. We show below how to mount a stronger attack on the forward security of the LRNG and compute, given the state at time $t$, previous states, which consequently enable to compute previous outputs of the LRNG.

We describe below how to reverse the state of a single pool, assuming that in its last update it was not refreshed with new entropy. Let us recall that the states of the urandom and secondary pools are updated when random bits are extracted from the LRNG, and that
if the entropy estimates of these pools are low these pools attempt to be refreshed with output from the primary pool. Only the secondary pool, however, is blocked if no such refresh is available. The state of the primary pool is updated with randomness from system events whenever it is updated. The attack is therefore mostly relevant to the urandom pool which is often used for extracting many bits while not receiving any entropy updates. The attack is also relevant to the secondary pool, if the attack starts from a time in which the value of the entropy counter is high, and to the primary pool, if the entropy which is added to the pool is mostly predictable.

The input of the attack is the state of a pool at time $t$ (denoted as pool$_t$). It computes the state of the pool at time $t - 1$ (denoted as pool$_{t-1}$). Now, given pool$_{t-1}$ it is easy to compute the random value which was output in the extract operation that transitioned the pool state from pool$_{t-2}$ to pool$_{t-1}$. In other words, the forward security requirement is not satisfied since it is possible to compute a previous output of the LRNG. The same analysis can be continued and compute the state and the corresponding outputs at times $t - 2, t - 3$ and so on, until the last time that the pool received an entropy update.

Below we describe two methods for reversing the state of the LRNG. The first method is a generic attack which has a complexity of $2^{96}$, which is much better than an exhaustive search (whose complexity is $2^{1024}$ for the case of a 32 word pool), but is rather impractical. The second attack is almost practical, with a complexity of $2^{64}$, but it is only applicable when the index $j$ is in a specific range (covering 18 of the 32 possible values of $j$ in a 32 word pool). This means that a lucky attacker, which starts its attack when the value of $j$ is at the end of this range, can compute five previous states of the pool (and the corresponding outputs) with a complexity of about $2^{64}$. Both attacks use $O(1)$ memory.

The Extract algorithm is described in Figure 3.2.3 and is used for advancing the pool and extracting an output from it. To simplify the analysis, let us first assume that the pool is either the secondary or the urandom pool, and is therefore 32 words long, and that the
add operation is a simple addition modulo $2^{32} - 1$, instead of the TGFSR operation detailed in Section 3.2.5.

The analysis starts with knowledge of the state of the pool at time $t$ (namely, pool$_t$), and of the value of the index $j$. (To simplify the notation, we denote by $j_i$, or simply by $j$, the value of the index $j$ at the beginning of the computation of the Extract operation that transitions the pool from time $t-1$ to time $t$.)

A generic attack: As a warmup we describe a generic attack which is based on a simple observation: all but three words of the pool (those indexed by $j$, $j-1$ and $j-2$) are identical in both pool$_t$ and pool$_{t-1}$. Given pool$_t$ there are therefore only $2^{96}$ possible candidate values, or “guesses”, for pool$_{t-1}$ — those obtained by copying the values of the words in locations different from $[j-2,j]$ and going over all possible values for the 96 bits in words $j$, $j-1$ and $j-2$. The transition from pool$_{t-1}$ to pool$_t$ is deterministic and defined by the Extract algorithm in Figure 3.2.3. The attack therefore applies this algorithm to each candidate value of pool$_{t-1}$, and checks if the result is equal to pool$_t$. If the two values are not equal then the candidate value is dismissed. Otherwise, the candidate value is put in a short list of possible values for pool$_{t-1}$ (we show immediately that this list is indeed short).

Note that although the Extract algorithm uses three invocations of SHA-1, which cause the bulk of the complexity, all but a fraction of $2^{-32}$ of the candidates for pool$_{t-1}$ can be dismissed after a single application of SHA-1. The complexity of the search is therefore about $2^{96}$ applications of SHA-1.

Estimating the accuracy of the attack. We know that the true value of pool$_{t-1}$ is in the computed short list, but that there might be some additional “false positives”, i.e., values for locations $[j-2,j]$ in time $t-1$ which are different from the true value, but for which applying the Extract algorithm results in the right value for pool$_t$. Fortunately, we do not expect many false positives. There are $2^{96} - 1$ false candidates for pool$_{t-1}[j-2,j]$, and
each of them has a probability of $2^{-96}$ to become a false positive (assuming that we model SHA-1 as a random function and therefore the probability of computing the right value of pool$_t[j-2, j]$ is $2^{-96}$). The number of false positives is therefore $k$ with probability of about $\binom{n}{k} n^{-k} (1 - 1/n)^{n-k}$, where $n = 2^{96} - 1$. Namely, with probability $e^{-1}$ there are no false positives ($k = 0$), with probability $e^{-1}$ there is a single false positive, with probability of about $0.5e^{-1}$ there are two false positives, and so on.

Given the short list of possible values of pool$_{t-1}$, the previous procedure can be applied for each value in the list to compute all possible values of pool$_{t-2}$. Applying this procedure to the correct value of pool$_{t-1}$ always results in one or more candidates for pool$_{t-2}$, which include the correct value of pool$_{t-2}$ and possibly additional false positives (according to the distribution that was detailed above). However, applying the procedure to a value which is a false positive for pool$_{t-1}$, results, with probability $e^{-1}$, with no candidate for pool$_{t-2}$. If this event happens then it can be concluded that the tested value for pool$_{t-1}$ is a false positive, and this value can be removed from the list. (If this is not the case then with probability $e^{-1}$ the tested value results in a single candidate for pool$_{t-2}$, with probability $0.5e^{-1}$ it results in two candidates for pool$_{t-2}$, etc.) The number of possible values for pool$_{t-k}$ is therefore a random variable. In Appendix B we show that the sequence $\{|\text{pool}_{t-k}| - k\}_{k=1,2,...}$ is a martingale, that $E(|\text{pool}_{t-k}|) = k$, and that the probability that pool$_{t-k} > k + b$ is at most $1/b$. We can therefore conclude that for all practical purposes the procedure outputs a list of reasonable size of the possible values of the state at time $t - k$.

A more efficient attack. We now show that for 18 of the possible 32 values of the index $j$, it is possible to reverse the pool by a procedure with a complexity of $2^{64}$. This procedure is applicable if the value of $j$ is in the range $[16, 31]$, and for $j = 1, 2$. We detail the procedure for the case of $j \in [18, 31]$. In this case the words which are affected by the state transition are located in the upper half of the pool, while the first half of the pool does not change from time $t - 1$ to time $t$ (namely, pool$_t[0, 15] = \text{pool}_{t-1}[0, 15]$). It is therefore possible, given pool$_t$, to
apply SHA-1 to words $[0, 15]$ and compute the value that was added to location $j$. Given this value it is possible to compute pool$_{t-1}[j]$ from pool$_t[j]$. In addition, the initialization vector for the second application of SHA-1 is computable from pool$_{t-1}[0, 15]$. It is therefore possible to go over all $2^{64}$ potential values of pool$_{t-1}[j-2, j-1]$, apply SHA-1 to pool$_{t-1}[16, 32]$ and compute the resulting values that were added to locations $j-2$ and $j-1$. If these values are not equal to the difference between the values of these locations in time $t$ and in time $t-1$, it is safe to dismiss the “guess” of pool$_{t-1}[j-2, j-1]$. The true value of pool$_{t-1}$ is never dismissed, while false positives appear with the same probability distribution as in the previous analysis (the only difference is the value of $n$, which is $2^{64}$ instead of $2^{96}$). The number of possible candidates for pool$_{t-k}$ behaves according to the same distribution as in the previous procedure, as is analyzed in Appendix B. The expected number of candidates for pool$_{t-k}$ is therefore only $k$. The complexity of the attack is $O(2^{64})$ operations and $O(1)$ memory.

We note here that the pool could have been reversed with a complexity of $2^{64}$ or less operations for any value of $j$, if the Extract algorithm had used the original SHA-1 function rather than the version which changes the initialization vector after the first invocation of the function. The effect of the SHA-1 variant which is used in the LRNG is that one cannot compute the values added to locations $j-1$ and $j-2$ before computing pool$_{t-1}[0, 15]$. For $j \in [3, 15]$ this means that we must check all options for pool$_{t-1}[j-2, j]$.

**Reversing the primary pool.** Assume that we are given the state of the primary pool, and that we know which entropy value was added to the pool when it was advanced to this state. The only difference of this case from the case of the shorter pools is that the size of the pool is 128 words. The generic attack can still be applied with a $2^{96}$ complexity. The more efficient attack can be applied if $j$ is in the range $[120, 127]$. 44
The add operation. The previous analysis assumed that the add operation in the Extract algorithm is an addition modulo $2^{32} - 1$. However, this operation is implemented as is described in Section 3.2.5 and therefore the value added to location $j$ is a function of the current values of several locations in the pool, as is depicted in Figure 3.2.2. This change obviously does not affect the generic attack, since pool$_t$ is still a deterministic function of pool$_{t-1}$. The second, more efficient, attack is also not affected: the value added to location $j$ is a function of the result of applying SHA-1 to pool$_{t-1}[0, 15] = \text{pool}_t[0, 15]$ and computing a function of the SHA-1 result, of pool$_{t-1}[j]$, and of five other locations in pool$_{t-1}$ which are not changed in the transition to pool$_t$. Given pool$_t[j]$, we can examine its three most significant bits and identify the entry of table which was xored to temp in the transition (this is easy since each of the eight entries in table has a different value to its three most significant bits). The index of the entry identifies the three least significant bits of temp (before it was shifted to the right). It is now possible to reverse the operation that was performed in the second to last line of add, and xor the result with $g$ and with the five pool entries which where used to generate it. The result is pool$_{t-1}[j]$.

3.3.2 Entropy Measurements

When entropy is added to the LRNG pool, each event adds two 32 bit words. In Section 3.2.3 we described the actual active range of bits within the type-value field and Appendix A provides the details for each case.

We now turn to the entropy of the timing of the encoded events. As mouse and keyboard events are usage dependent and network interrupts are not commonly used as an entropy resource for the LRNG, we ran a trial which measured the entropy which is added to the LRNG by hard disk events. Our trial included over ten days of measurements on a single Linux machine, with a total of over 140,000 entropy addition events. The system was mostly idle, and the measurements were in a setting that recorded their results on a different
We mark by $t_n$ the time of event $n$, and the difference between two consecutive events is defined as $\delta_n := t_n - t_{n-1}$. Table 3.2 presents the frequency of different $\delta_n$ values. Note that all but 0.8% of the values are in the range [77, 82]. All other values are larger by about four orders of magnitude. The resulting entropy is only $H := 1.03$ bits per event, which is to say that event timing, which is encoded as a 32 bit field, had in this setting an entropy of at most a single bit. The recordings furthermore show that there is a correlation between consecutive $\delta_n$ values (and therefore the entropy is even lower). The entropy of pairs of events is 1.53 bits per pair, and consequently, the conditional entropy of $\delta_n$ given $\delta_{n-1}$ is only 0.5 bits.\footnote{Closer examination reveals patterns such as the following one: For about 84% of the measurements it holds that $\delta_{i+1} = \delta_i$. Given this event, the conditional probability that $\delta_{i+2} = \delta_i$ is about 90%, and the conditional probability that $\delta_{i+2} \neq \delta_i$ but $\delta_{i+3} = \delta_i$ is 9.9%. Consequently, for only 0.1% of the cases in which $\delta_{i+1} = \delta_i$ we get that both $\delta_{i+2}$ and $\delta_{i+3}$ are different from $\delta_i$.}

We also checked the entropy estimate that would have been calculated by the LRNG for these measurements, using the procedure outlined in Section 3.2.4. This estimate turns out to be very conservative. Only 2224 of the 140,000 measurements resulted in a positive addition to the entropy estimate. These events occurred, more or less, only when $\delta_n$ had a very large value, and consequently the $n$ and $n+1$ measurements had a large entropy estimate (namely, the 1100 events of Table 3.2 for which $\delta_n \gg 84$, each resulted in a pair of consecutive measurements which contributed to the entropy count). The total estimate of

<table>
<thead>
<tr>
<th>$\delta_n$</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>77</td>
<td>243</td>
</tr>
<tr>
<td>78</td>
<td>1730</td>
</tr>
<tr>
<td>79</td>
<td>11468</td>
</tr>
<tr>
<td>80</td>
<td>113402</td>
</tr>
<tr>
<td>81</td>
<td>11786</td>
</tr>
<tr>
<td>82</td>
<td>625</td>
</tr>
<tr>
<td>83</td>
<td>0</td>
</tr>
<tr>
<td>84</td>
<td>0</td>
</tr>
<tr>
<td>$&gt;84$</td>
<td>1112</td>
</tr>
</tbody>
</table>

Table 3.2: Frequency of time differences between two consecutive HD entropy addition computer, without affecting the disk of the examined machine.
the added entropy was 17500 bits. Given that the measurements were taken over a period of ten days, there was an average delay of about 15 minutes between pairs of events which had a positive contribution to the entropy count. The average contribution of these pairs was about 16 entropy bits. This is quite a severe bottleneck for the blocking interface to the LRNG.

### 3.3.3 Analysis of the OpenWRT Linux Distribution

OpenWRT [2] is a Linux distribution for wireless routers. It provides many cryptographic services such as SSL termination, a SSH server, and a wireless encryption. The security of all these services is dependent upon the output of the LRNG.

An OpenWRT router has very limited entropy sources. There are no keyboard, mouse or hard drive attached to the router. The WRT uses its flash memory (of size $4 - 16$ MBytes) as a special file system, but this file system does not provide entropy to the LRNG.

In addition, the WRT implementation does not save any LRNG state between reboots. Hence, the only entropy source for the LRNG is network interrupts, which might be observable by an adversary (!). This is true in particular for wireless routers where most network interrupts are caused by wireless activity, which is easily visible by an external adversary.

Given this data we can conclude that the WRT implementation of the LRNG is weak. The state of the LRNG is reset in every reboot to a predictable value (composed of the time of day and a constant string), and the only source of entropy is, to a large extent, observable by external adversaries. The result is that an adversary can simulate the state and the output of the LRNG. We note that security can be improved by saving LRNG output at shutdown and loading it into the state at reboot. This would require potential attackers to either eavesdrop to all network traffic from the time the router was initialized, or obtain access to the LRNG state.
3.3.4 Security Engineering

Denial of service. There is no limitation on the number of bits a user can read from the random devices per time unit. However, the secure interface `/dev/random` blocks its output when the entropy estimate is low, until additional “noise” is added to the pools. These facts together suggest two denial of service attacks which block all users from reading `/dev/random` bits.

The first attack is simply to read bits from `/dev/random`. As there is no limit and no prioritization, this results in blocking other users, and might delay them for a long period of time. We tested this attack using simple `dd if=/dev/random` and were able to block other readers of `/dev/random`.

Furthermore, an attack can even be mounted remotely, by triggering system requests for random bytes (`get_random_bytes`) in a significantly higher rate than that of the entropy input events. Since the urandom non-blocking pool (from which these bits are taken) is refilled from the (blocking) primary pool, this attack will result in denial-of-service for the primary and secondary pools. A simple way for an adversary to issue this attack may be to set many TCP connections. For each connection, a TCP-syn-cookie is generated, which requires 128 bytes from the non-blocking pool, hence reducing the entropy count.

Solution. As entropy is a limited and valuable resource, its consumption must be controlled. The common solution in operating systems for such a resource is through the definition of a new quota per user or group for the consumption of random bits.

Guessable passwords. Usually, the first user-operation in a computer system is user login, and the first input entered by the user is the password, or a user-name and password pair.

In a scenario of a disk-less system, without a random seed that is saved between startups, we can imagine a situation where an attacker knows the initial state of the LRNG, and where
the sequence of updates of the LRNG during system reboot is quite predictable. The state of the LRNG might therefore be, to a large extent, a deterministic function of the initial password entered by the user. If the attacker is able to read random bits fast enough, it might be able to identify the password by going over all possible password values, and checking which one results in the LRNG output which was observed.

This attack, which was noted by Kelsey et al. [35], is particularly relevant if the LRNG does not obtain input from a hard disk or from interrupts resulting from network events (as is the case for example with one of the most popular network cards, the 3Com PCI 3c905B Cyclone 100baseTx). In this case the input to the LRNG comes mostly from the user’s input. We attempted to use this observation to extract the user password from the output of the LRNG of the KNOPPIX Linux distribution [1], which is bootable from a CD and therefore does not save the LRNG state. We were unsuccessful in this attack, largely because the examined system used a hard disk which provided considerable amount of entropy during the boot process.

**Solution.** It is better to remove the influence of the values of keyboard events on the LRNG. Keyboard entropy should be based on the timing of its events, and not on the type-values. (The timing of keyboard events might also reveal information about data entered by the user, but this seems like a second-order risk compared to the risk from the values of keyboard events.)

**An adversary can create noise that directly affects the LRNG output.** Normally, there is a separation between the input and output of the LRNG, but when the primary pool is full, the batched entropy is added directly to the secondary pool, from which it is output when `/dev/random` is used. This direct flow between the input and the output of the LRNG might provide an adversary with the ability to create noise that directly affects the generator’s output.
Solution. It is best to always flush the batched entropy to the primary pool, even if it is full. This makes a strict separation between the input and the output sides of LRNG.

The LRNG state reveals the previous LRNG output. The Extract algorithm (Fig. 3.2.3) first updates the pool and then computes its output. The result of this design decision is that an adversary which learns the internal state of the LRNG learns the state of the pool which was used to compute the last LRNG output, and can easily compute this output (and hence break the forward-security of the LRNG).

Solution. It is best to switch the order of operations. The state update will then take place after LRNG output.

3.4 Conclusions and Recommendations

This Chapter analyzes the Linux random number generator. The LRNG algorithm is complex and includes a large state made of three different storage pools, a complex mechanism for adding entropy from system events, and an extraction algorithm based on a shift register and several SHA-1 operations.

We showed that these layers add complexity to the implementation but do not prevent attacks on the forward security of the LRNG. In addition we described weaknesses in the OpenWRT Linux distribution.

Our study was conducted on the latest (at the time) Linux kernel, labeled version 2.6.10, which was released on December 24, 2004. Since then the kernel kept developing. Lately, version 2.6.15 was released in January 2006, and patches are being published since then\(^5\).

Limitations. As far as we could tell the different Linux distributions for PCs (e.g., RedHat, Debian, Slakware) have little if any effect on the LRNG structure, since all distributions use

\(^5\)See http://www.linuxhq.com/kernel/file/drivers/char/random.c for the incremental changes in random.c.
the same kernel source. Changes occur only within the system up and down times, and to our findings are only cosmetic. As there are hundreds of different distributions this statement may be not true for all of them.

Our study does not cover all Linux kernel options. For example, we did not take into account multi-cpu hardware configurations, or unique hardware configurations such as the Qtronix keyboard and mouse device\(^6\) whose entropy collection method is different than the one described here.

**Open source security.** The LRNG is an open source project which enables an adversary to read the entire source code and even trace changes inside the source configuration management system. This feature gives powerful tools to the adversary. On the other hand, open source benefits security by enabling security audits, and enabling easy changes to the code. It is rather easy to add patches to the current LRNG code in order to prevent the attacks we described in this Chapter (this would have been much harder, if at all possible, for closed source PRNGs).

“Open” is not a synonym for “secure”. We feel that the open source community should have a better policy for security sensitive software components. These components should not be treated as other source elements. We suggest to add a better quality assurance procedure for the cryptographic elements of the kernel. For example, the PRNG must pass statistical tests which can be put into the kernel build process. Open source must also have, in our opinion, a clear and updated documentation of the algorithms used in the code. Such documentation could have saved us from the trouble of reverse engineering the code, and would have provided better access for other researchers to review the security of the LRNG.

\(^6\)http://lxr.linux.no/source/drivers/char/qtronix.c?v=2.6.10
3.4.1 Recommendations

Following our analysis of the LRNG, we suggest the following recommendations for the design of pseudo-random number generators.

- **Fixing the LRNG.** The issues which were reported in this Chapter should be fixed. In particular, the LRNG code should be changed to prevent attacks on its forward security. The OpenWRT implementation should be changed to provide more entropy to the LRNG, or at least save its state during shutdown.

- **Implementing a quota for the consumption of random bits.** Random bits are a limited resource, and attackers can easily mount a denial-of-service attack (even remotely) by consuming random bits at a high rate. The common solution for this type of problem is to implement a quota system which limits the effect of each user, or each process, on the operation of other users of the same system. Such a quota system should be added to the Linux kernel.

- **Adopting the Barak-Halevi construction.** The Barak-Halevi (BH) construction and its analysis [6] are attractive in their simplicity, which clearly identifies the role of every component of the system, and enables a simple implementation. In comparison, the current LRNG construction is an overkill in some aspects (like the size of the pools or the number of \texttt{SHA-1} invocations), but its complexity does not improve its security but rather hides its weaknesses. We suggest that future constructions of pseudo-random number generators follow the BH construction (and in general, try to “keep it simple”).

- Since randomness is often consumed in a multi-user environment, it makes sense to generalize the BH model to such environments. Ideally, each user should have its own random-number generator, and these generators should be refreshed with different data which is all derived from the entropy sources available to the system (perhaps after
going through an additional PRNG). This architecture should prevent denial-of-service attacks, and prevent one user from learning about the randomness used by other users.\textsuperscript{7}

\textsuperscript{7}In the current centralized system this property is not guaranteed. A user which learns the state of the generator can simulate future outputs of the generator (assuming that no external entropy is added). If the user later reads random bits it can identify their location in the simulated generator output and learn the values used by other users.
Chapter 4

Analysis Of The Windows Random Number Generator

4.1 Introduction

This Chapter\(^1\) studies the pseudo-random number generator used in Microsoft Windows systems, which we denote as the WRNG. The WRNG is the most frequently used pseudo-random number generator, with billions of instances running at every given time. It is used by calling the function \texttt{CryptGenRandom}. According to the book “Writing Secure Code” \([31]\), published by Microsoft, the WRNG was first introduced in Windows 95 and was since embedded in all Windows based operating systems such as Windows XP or Windows 2000, and in all their variants.\(^2\) According to \([31]\) the design of the WRNG has not changed between the different version of the operating system.\(^3\)

---

\(^1\)The research described in this chapter was previously published as [16]
\(^2\)The statement in [31] was written before Windows Vista was released. The documentation of \texttt{CryptGenRandom} states that it is supported by Windows Vista, but we have not verified this statement.
\(^3\)Our checks show, however, some variations between the implementation of the WRNG in Windows 2000 and in prior versions of the Windows operating system. For example, the code distributed with Windows 2000 uses the type of the operating system to set the number of bytes which are output between two entropy based rekeys of the generator. In Windows 2000 rekeys are done after 16 KBytes of output, while in earlier versions of Windows they are done after outputting only 512 bytes.
In this chapter we examine the generator that is implemented in the Windows 2000 operating system (service pack 4). Windows 2000 is the second most popular operating system, especially in enterprises, with a market share of 4.5%-6% as of April 2007.\footnote{See http://marketshare.hitslink.com/report.aspx?qprid=5, http://www.onestat.com/html/aboutus_pressbox46-operating-systems-market-share.html.}

**WRNG usage.** The WRNG is used by calling the Windows system function `CryptGenRandom` with the parameters `Buffer` and `Len`. Programs call the function with the required length of the pseudo-random data that they need, and receive as output a buffer with this amount of random data. The function is used by internal operating system applications such as the generation of TCP sequence numbers, by operating system applications, such as the Internet Explorer browser, and by applications written by independent developers.

**Our contributions.** This chapter describes the following results:

- We present a detailed analysis of the Windows pseudo random number generator. The analysis is accompanied by a concise pseudo-code for the entire implementation of the WRNG (the complete pseudo-code is about 1000 lines of code), and by a user-mode simulator of the WRNG. The analysis is based on examination of the binary code of the operating system, see details below.

- We present an attack on the forward security of the WRNG. We show how an adversary can compute past outputs and states from a given state of the WRNG, with a complexity of $2^{23}$ computation (namely, in a matter of seconds on a home computer).

- We present an attack on the backward security of the WRNG. We show that given the inner state of the WRNG an adversary can compute future outputs and states with a complexity of $O(1)$ computation.
• We analyze the way in which the operating system uses the WRNG and note that a
different copy of the WRNG is run, in user-mode, for every process, and that typical
invocations of the WRNG are seldom refreshed with additional entropy. Therefore,
the backward and forward security attacks, which only work while there is no entropy
based rekeying, are highly effective. Furthermore, we also found that part of the state
of the generator is initialized with values that are rather predictable.

Attack model. Our results suggest the following attack model: The attacker must obtain
the state of the generator at a certain time. This can be done by attacking a specific
application and obtaining the state of the WRNG run by this process, or by launching a
buffer overflow attack or a similar attack providing administrator privileges, and obtaining
the state of the generators run by all processes. After learning the state the attacker does
not need any additional information from the attacked system. It can learn all previous
and future outputs of the generator, and subsequently, learn cryptographic keys, such as
SSL keys, used by the attacked system. This attack is more powerful and more efficient
than known attacks which require the attacker to control the attacked machine at the time
it is generating cryptographic keys, observe these keys, and relay them to the attacker (in
particular, the latter attacks cannot reveal keys which were used before the attacker obtained
access to the machine; they therefore require the attacker to attack a machine before the
time it is used by the attack target).

Gap between theory and practice. The generation of pseudo-random numbers is a well
studied issue in cryptography, see, e.g., [50, 11]. One might therefore be surprised to learn
that constructing an actual implementation of a pseudo-random number generator is quite
complex. There are many reasons for this gap between theory and practice:

• Performance. Provably secure generators might incur high computation complexity.
Therefore even a simple PRNG such as the Blum-Blum-Shub generator [11] is rarely used in practice.

- **Real world attacks.** Actual implementations are prone to many attacks which do not exist in the clean cryptographic formulation which is used to design and analyze pseudo-random generators (consider, for example, timing attacks and other side-channel attacks).

- **Seeding and reseeding the generator.** Generators are secure as long as they are initialized with a truly random seed. Finding such a seed is not simple. Furthermore, the state of the generator must be periodically refreshed with a fresh random seed in order to prevent backward security attacks. The developer of a generator must therefore identify and use random sources with sufficient entropy.

- **Lack of knowledge.** In many cases the developers of the system do not have sufficient knowledge to use contemporary results in cryptography.

These factors demonstrate the importance of providing a secure pseudo-random generator by the operating system.\(^5\) The designers of the operating system can be expected to be versed with the required knowledge in cryptography, and know how to extract random system data to seed the generator. They can therefore implement an efficient and secure generator. Unfortunately, our work shows that the Windows pseudo-random generator has several unnecessary flaws.

### 4.1.1 Related Work

**Existing PRNG implementations.** In the past, PRNGs were either a separate program or a standard library within a programming language. The evolution of software engineering

\(^5\)Indeed, given the understanding that writing good cryptographic functions is hard, operating systems tend to provide more and more cryptographic functionality as part of the operating system itself. For example, Linux provides implementations of hash functions as part of its kernel.
and operating systems introduced PRNGs which are part of the operating system. From a cryptographic point of view, this architecture is advantageous since it enables to initialize the PRNG with operating system data (which has more entropy and is hidden from users).

Implementations of PRNGs can be either blocking or non-blocking. A blocking implementation does not provide output until it collects sufficient amount of system based entropy. A non-blocking application is always willing to provide output. The PRNG of the FreeBSD operating system is described in [43]. FreeBSD implements a single non-blocking device and the authors declare their preference of performance over security. The PRNG used in OpenBSD is described in [14], which also includes an overview of the use of cryptography in this operating system. Castejon-Amenedo et al. [32] propose a PRNG for UNIX environments. Their system is composed of an entropy daemon and a buffer manager that handles two devices—blocking and non-blocking. The buffer manager divides entropy equally between the two devices, such that there is no entropy that is used in both. A notable advantage of this scheme is the absolute separation between blocking and non-blocking devices, which prevents launching a denial-of-service attack on the blocking device by using the non-blocking device (such an attack is possible in Linux, as is shown in Chapter 3).

Analysis of PRNGs. A comprehensive discussion of the system aspects of PRNGs, as well as a guide to designing and implementing a PRNG without the use of special hardware or access to privileged system services, is given by Gutmann [24]. Issues related to operating system entropy sources were discussed in a recent NIST workshop on random number generation [33, 25]. An extensive discussion of PRNGs, which includes an analysis of several possible attacks and their relevance to real-world PRNGs, is given by Kelsey et al. in [35]. Additional discussion of PRNGs, as well as new PRNG designs appear in [34, 18].

The recent work of Barak and Halevi [6] presents a rigorous definition and an analysis of the security of PRNGs, as well as a simple PRNG construction. That work suggests
separating the entropy extraction process, which is information-theoretic in nature, from the output generation process. Their construction is based on a cryptographic pseudo-random generator $G$, which can be implemented, for example, using AES in counter mode, and which does not use any input except for its seed. The state of the PRNG is the seed of $G$. Periodically, an entropy extractor uses system events as an input from which it extracts random bits. The output of the extractor is xored into the current state of $G$. The construction is much simpler than most existing PRNG constructions, yet its security was proved in [6] assuming that the underlying building blocks are secure. We note that our analysis shows that the WRNG construction, which is much more complex than that of [6], suffers from weaknesses which could have been avoided by using the latter construction.

Outline. The rest of the chapter goes as follows. Section 4.2 provides a detailed description of the WRNG. Section 4.3 presents cryptanalytic attacks on the generator, while Section 4.4 describes the interaction between the operating system and the generator, and its security implications. Section 4.5 compares the WRNG to the generator used by Linux, and Section 4.6 contains conclusions and suggestions for further research.

4.2 The Structure of the Windows Random Number Generator

We start by discussing the process of analyzing the binary code. Then we describe the main loop of the generator, the functions called by this loop, the initialization of the state, and the usage of the generator by the operating system. We conclude this section by listing observations about the structure of the generator.
4.2.1 Analyzing the Binary Code

The algorithm employed by the WRNG, and its design goals, were never published. There are some published hints about the inner structure of the WRNG [31]. However, the exact design and security properties were not published.

Our entire research was conducted on the binary version supplied with each running Windows system. We did not have access to the source code of the generator. We examined the Windows 2000 operating system, which is the second most popular operating system. The research was conducted on Windows 2000 Service Pack 4 (with the following DLL and driver versions: ADVAPI32.DLL 5.0.2195.6876, RSAENH.DLL 5.0.2195.6611 and KSECDD.SYS 5.0.2195.824). The entire inspected binary code is over 10,000 lines of assembly code.

Our study required static and dynamic analysis of the binary code. Static analysis is the process where the binary assembly code is manually translated into pseudo-code written in a high level programming language. In the dynamic analysis phase the binary is run while a debugger is tracing the actual commands which are run, and the values of memory variables. The combined process of dynamic and static analysis enables us to focus only on relevant functions and better understand the meaning of variables and functions.

We used several tools in our analysis: the Interactive Disassembler (IDA) tool [23] which is an editor for static code analysis, the OllyDbg tool [55] for dynamic study of our user mode runtime environment, and the WinDBG tool [40] as our kernel debugging tool. (See also the book “Reversing” [17] which provides an excellent introduction to the field of code analysis.) To verify our findings and demonstrate our attacks we developed four tools:

- **CaptureCryptGenRandom**: captures the current WRNG state into a file.

- **NextCryptGenOutputs**: calculates future outputs of the WRNG from a given state.

- **PreviousCryptGenOutputs0**: calculates previous outputs and states of the WRNG from a given state and knowledge of the initial State and \( R \) variables (this attack, and
the roles of State and R, are described in Section 4.3.2).

- **PreviousCryptGenOutputs**: calculates previous outputs and states of the WRNG from a given state alone. (This attack is described in Section 4.3.2. It has a complexity of \(O(2^{23})\).)

These tools validate our findings. We currently do not publish the tools online. They can be provided upon request.

## 4.2.2 The Structure of the Generator

The algorithm used by the generator is based on two common cryptographic primitives, the RC4 stream cipher (described in Appendix C), and the SHA-1 hash function, which maps arbitrary inputs to a 20 byte long output.

### The main loop of the WRNG

The main loop, presented in Figure 4.2.1, generates 20 bytes of output in each iteration. The main state of the WRNG is composed of two registers, R and State, which are updated in each iteration and are used to calculate the output. The loop operates on data in chunks of 20 bytes: each of the registers used in the main loop, R, State and T, is 20 bytes long. This is also the length of the result of the internal function call `get_next_20_rc4_bytes` and of the output of SHA-1. The output is generated in increments of 20 bytes.

The main loop uses the two variables, R and State, to store a state. It calls an internal function `get_next_20_rc4_bytes` to obtain 20 bytes of pseudo-random data, and uses them to update R and State. It generates 20 bytes of output by applying a variant of SHA-1 to State, and then updates State again using part of this output and using R. (The only difference between the variant of SHA-1 used here and the standard implementation of SHA-1 is a different ordering of the IV vector. We therefore use the notation SHA-1 in...
CryptGenRandom(Buffer, Len)  
// output Len bytes to buffer
while (Len > 0) {
    R := R ⊕ get_next_20_rc4_bytes()  
    State := State ⊕ R
    T := SHA-1(State)  
    Buffer := Buffer | T
    // | denotes concatenation
    R[0..4] := T[0..4]
    // copy 5 least significant bytes
    State := State + R + 1
    Len := Len - 20
}

Figure 4.2.1: The main loop of the WRNG. It has input parameters Len, which is the number of bytes to be output, and Buffer, which gets the output. All internal variables are 20 bytes long and uninitialized. Buffer is assumed to be empty and the WRNG output is concatenated to it in each round of the loop. The function SHA-1’est a variant of SHA-1 where the Initialization Vector (IV) is ordered differently.

most of the discussion.)

The function get_next_20_rc4_bytes. The function get_next_20_rc4_bytes6 keeps a state which is composed of eight instances of the RC4 stream cipher. (See Appendix C for a description of RC4). In each call, the function selects one RC4 state in a round-robin fashion, uses it to generate 20 bytes of output, and returns them to its caller. In the next call it uses the next RC4 stream. After an RC4 instance generates 16Kbytes of output it is refreshed with entropy gathered from the system, as is described below.

The function is described in Figure 4.2.2 (this description assumes a static variable i which is initialized to zero before the first call). We can also imagine this function as storing eight output streams from eight independent invocations of RC4. The function holds a pointer i which points to one of the streams, and for each stream (numbered i) it holds a

6This function is called NewGenRandom in Windows 2000. We use instead the name get_next_20_rc4_bytes which describes the functionality of the function more clearly.
counter $c_i$ which points to a location in the stream (in the code of Figure 4.2.2 this counter is denoted by $RC4[i].accumulator$). When the function is called it returns the 20 bytes numbered $c_i$ to $c_i + 19$ from the stream pointed to by $i$. It then sets $c_i = c_i + 20$, and advances $i$ in a round-robin fashion.

get_next_20_rc4_bytes()
{
    // if |output of RC4 stream| >= 16Kbytes then refresh state
    while (RC4[i].accumulator >= 16384) {
        RC4[i].rekey(); // refresh with system entropy
        RC4[i].accumulator = 0;
        i = (i + 1) % 8;
    }
    result = RC4[i].prng_output(20);
    // 20 byte output from i'th instance
    RC4[i].accumulator += 20;
    i = (i + 1) % 8;
    return(result);
}

Figure 4.2.2: Function get_next_20_rc4_bytes().

Initializing $R$ and State. The WRNG does not explicitly initialize $R$ and State. However, as with any other stack parameter which is not initialized by the program, these variables are implicitly initialized with the latest values stored in the memory address allocated to them. We describe in Section 4.4 some analysis of the actual values with which these variables are initialized, and note that they are highly correlated. We are not sure about the reason for this use of uninitialized variables.

Initializing and refreshing each instance of RC4. All instances of RC4 are initialized and refreshed by the same mechanism, which collects system entropy and uses it to rekey an RC4 instance. The collected system entropy is composed of up to 3584 bytes of data from different operating system sources. Entropy collection is synchronous and is only done when
an RC4 stream is initialized, or reaches the 16 Kbyte threshold. We were not able to see a way to predict all 3584 bytes of these parameters by a practical brute force attack.

The pseudocode for the state refreshment mechanism is described in Figure 4.2.3. It is composed of the following stages:

- The entire 3584 bytes of collected entropy are hashed (using a function called VeryLargeHash) to produce an 80-byte digest. The function is implemented by a series of SHA-1 operations, designed to ensure that a change of a single input bit affects all output bits.

- The output of VeryLargeHash is fed into the RC4 algorithm as a key, and is used to encrypt a cleartext which is read from a Windows registry key named seed (which is 80 bytes long). This registry key is used by all instances of the WRNG run on the same machine and is stored at HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\RNG\Seed, in the HKEY_LOCAL_MACHINE directory.

- The result of the last encryption is 80 bytes long. It is fed to another RC4 encryption as a key, and is used to encrypt additional 256 bytes, which are read from a Windows device driver called KSecDD. The KSecDD device driver serves as just an additional entropy source. The result is 256 bytes long.

- The result of the final encryption is used as a key for the RC4 instance that is used in the WRNG internal state. This RC4 instance is initialized using the RC4 key scheduling algorithm (KSA), described in Appendix C.

**Initializing all RC4 instances.** The WRNG uses eight instances of RC4, all of which are initialized using the procedure described above. Initialization starts with the first call to read bytes from an instance. Note that the initializations of different RC4 instances used by one
instance of the WRNG are run one right after the other, and therefore most of the 3584 bytes of system parameters used for initialization will be equal in two successive initializations.

Additional rekey calls of each of the eight RC4 instances are made after it outputs 16 Kbytes of data. Since there are eight RC4 instances the generator always outputs $8 \times 16 = 128$ KBytes of output between two rekey calls.

**Scope.** Windows is running one WRNG instance *per process*. Therefore, two applications (e.g., Windows Word and Internet Explorer) have two separate states. The RC4 states and auxiliary variables of a specific process reside in DLL space which is allocated upon the first invocation of the Crypto API, and remains allocated until it is unloaded. The state variables $R$ and $State$, on the other hand, are stored on the stack. If a process has several threads, then they all share the same RC4 states stored in the DLL space, but each of them has its own stack, and therefore its own copy of $R$ and $State$. It is interesting to note that $R$ and $State$ are never explicitly initialized, and instead are initialized with the last values that are stored in the stack locations allocated to them. We will describe in Section 4.4 an analysis which shows that there is correlation between the states used in different instances of the WRNG.

Scoping is both good and bad. It separates between two processes. Therefore breaking one WRNG, or learning its state, does not affect applications using another WRNG. On the
downside, the fact that there is only one consumer per WRNG, together with the very long period between rekeys, make it very likely that the WRNG state will rarely be refreshed.

**Implementation in user mode.** The WRNG is running in user mode, rather than in the kernel. A kernel based implementation would have kept the internal state of the WRNG hidden from applications, whereas a user mode implementation enables each process to access the state of the WRNG instance assigned to it.

### 4.3 Analysis I: Cryptanalytic Attacks

We demonstrate here attacks on the backward security and forward security of the generator. Namely, show how an adversary that obtains the state of the WRNG (i.e., the values of the variables $R$ and State and the states of the eight RC4 registers) is able to compute future and past states and outputs of the generator. Computing future states is easy, as is computing past states if the adversary knows the initial values of the variables State and $R$. We also show two attacks which compute previous states without knowledge of the initial values of State and $R$. The computational complexity of these two attacks is $2^{40}$ and $2^{23}$, respectively.

The attacks we describe can be applied by an adversary which learns the state of the generator. This is a very relevant threat for several reasons. First, new buffer overflow attacks are found each week. These attacks enable an adversary to capture the memory space of a certain process or of the entire computer system. Second, since the WRNG runs in user mode a malicious user running an application can learn the WRNG state without violating any security privileges (this happens since the WRNG memory space is not blocked from that user).
4.3.1 Attack on Backward Security

Suppose that an adversary learns the state of the WRNG at a specific time. The next state of the WRNG, as well as its output, are a deterministic function of this data. The adversary can therefore easily compute the generator’s output and its next state, using a simulation of the generator’s algorithm (similar to the one we constructed). The adversary can then compute the following output and state of the simulator, as a function of the state it just computed. It can continue doing so until the next refresh of the generator using system entropy.

4.3.2 Attacks on Forward Security

The WRNG depends on RC4 for generating streams of pseudo-random output, which are then added to the state of the generator. RC4 is a good stream cipher, but it does not provide any forward security. Namely, given the current state of an RC4 cipher it easy to compute its previous states and previous outputs. (This process is described in Appendix C. See also [8].) We use this fact to mount attacks on the forward security of the WRNG. Suppose an adversary learns the state of the generator at time \( t \) and wishes to compute the state at time \( t - 1 \). We show here three methods of computing this state, with a complexity of \( O(1) \), \( O(2^{40}) \), and \( O(2^{23}) \), respectively, and where the first attack also assumes knowledge of the initial values of State and \( R \). The attack with \( O(2^{23}) \) complexity is based on observing that State is updated using consecutive addition and exclusive-or operations, which, up to the effect of carry bits, cancel each other.

An instant attack when the initial values of State and \( R \) are known. Suppose that the attacker knows the initial values of the variables State and \( R \). (As argued in Section 4.4 this is a reasonable assumption.) The attacker also knows the current values of the eight RC4 registers. Since RC4 does not provide any forward security, the attacker can compute...
all previous states and outputs of the RC4 registers, until the first invocation of the WRNG. (It can learn the total number of invocations of the WRNG from a static variable named stream_counter, found in a static offset in memory — offset 7CA1FFA8 in the DLL of the version of Windows we examined.) Since each state of the WRNG is a function of the previous values of State and R and of the output of the RC4 registers, the attacker can now compute the states and outputs starting from the first step and continuing until the current time. We implemented this attack in the tool PreviousCryptGenOutputs0.

**An attack with a complexity of** \(2^{40}\). Let us denote by \(R^t, S^t\) the values of \(R\) and State just before the beginning of the \(t\)th iteration of the main loop. (We refer here to the main loop of the WRNG, as it is described in Figure 4.2.1.) Let us denote by \(R^{t,i}, S^{t,i}\) the values just before the execution of the \(i\)th line of code in the \(t\)th iteration of the main loop (namely, \(R^t = R^{t,4}, S^t = S^{t,4}\)). Let \(RC^t\) denote the output of `get_next_20_rc4_bytes` in the \(t\)th iteration. Each of these values is 160 bits long. Let us also denote by \(X_L\) the leftmost 120 bits of variable \(X\), and by \(X_R\) its 40 rightmost bits.

Given \(R^t\) and \(S^t\) our goal is to compute \(R^{t-1}, S^{t-1}\). We also know the state of all eight RC4 registers, and since RC4 does not have any forward security we can easily compute \(RC^{t-1}\). We do not assume any knowledge of the output of the generator. We observe the following relations between the values of \(R\) and \(S\) before and after code lines in which they are changed:

\[
\begin{align*}
S^{t-1,11} &= S^t - R^t - 1 \\
R^{t-1,9} &= R_L | *^{40} \quad \text{(where \(*^{40}\) is a 40-bit string which is unknown at this stage)} \\
R^{t-1} &= R^{t-1,9} \oplus RC^{t-1} \\
S^{t-1} &= S^{t-1,5} = S^{t-1,11} \oplus R^{t-1,9} = (S^t - R^t - 1) \oplus \underbrace{(R_L | R^{t-1,9})}_{S^{t-1,11}} \\
\end{align*}
\]
We also observe the following relation:

\[
R^t_R = SHA-1 \left( S^{t-1,11} \right)_R = SHA-1 \left( S^t - R^t - 1 \right)_R
\]

These relations define \( R_{t-1}^L \) and \( S_{t-1}^L \), but they do not reveal the rightmost 40 bits of these variables (namely \( R_{t-1}^R \) and \( S_{t-1}^R \)), and do not even enable us to verify whether a certain “guess” of these bits is correct. Let us therefore examine the previous iteration, and in particular the process of generating \( R_{t-1}^R \), and use it to compute \( R_{t-1}^R \) (then, \( S_{t-1}^R \) can easily be computed).

\[
R_{t-1}^R = SHA-1 \left( S^{t-2,11} \right)_R
= SHA-1 \left( S^{t-1} - R_{t-1}^t - 1 \right)_R
= SHA-1 \left( (S^t - R^t - 1) \oplus (R_L^t \mid R_{R}^{t-1,9}) - (R_L^t \mid R_{R}^{t-1,9}) \oplus RC_{t-1} \right)_R
\]

Note also that \( R_{R}^{t-1,9} = R_{R}^{t-1} \oplus RC_{R}^{t-1} \). Consequently, we know every value in this equation, except for \( R_{R}^{t-1} \). We can therefore go over all \( 2^{40} \) possible values of \( R_{R}^{t-1} \), and disregard any value for which this equality does not hold. For the correct value of \( R_{R}^{t-1} \) the equality always holds, while for each of the remaining \( 2^{40} - 1 \) values it holds with probability \( 2^{-40} \) (assuming that the output of SHA-1 is uniformly distributed). We therefore expect to have \( O(1) \) false positives, namely incorrect candidates for the value of \( R_{R}^{t-1} \) (see below an analysis of the expected number of false positives after several invocations of this attack).

An attack with a complexity of \( 2^{23} \). A close examination of the relation between the addition and exclusive-or operations reveals a more efficient attack. Note that \( R^{t-1,9} = \)
\( R_t^{t-1} \oplus RC_{t-1} \) and therefore we can obtain the following equation:

\[
R_t^{t-1} = \text{SHA-1} \left( S_t^{t-2,11} \right)_R = \text{SHA-1} \left( S_t^{t-1} - R_t^{t-1} - 1 \right)_R
\]

Note also that

\[
S_t^{t-1} = (S_t^{t} - R_t^{t-1}) \oplus RC_{t-1}^{t-1} \oplus R_t^{t-1}
\]

Let us use the notation \( Z = (S_t^{t} - R_t^{t-1}) \oplus RC_{t-1}^{t-1} \). We are interested in computing \( R_t^{t-1} = \text{SHA-1} \left( (Z \oplus R_t^{t-1}) - R_t^{t-1} - 1 \right)_R \). Denote by \( r_i \) the \( i \)th least significant bit of \( R_t^{t-1} \).

We know all of \( Z \), and the 120 leftmost bits of \( R_t^{t-1} \), and should therefore enumerate over all possible values of the righthand side of the equation, resulting from the \( 2^{40} \) possible values of \( r_{39}, \ldots, r_0 \). (We will see that typically there are much fewer than \( 2^{40} \) such values.)

Use the notation \( 0_Z \) and \( 1_Z \) to denote the locations of the bits of \( Z \) which are equal to 0 and to 1, respectively.

\[
(Z \oplus R_t^{t-1}) - R_t^{t-1} - 1 = \left( \sum_{i \in 0_Z} 2^i r_i + \sum_{i \in 1_Z} 2^i (1 - r_i) \right) - \sum_{i=0\ldots159} 2^i r_i - 1
\]

\[
= Z - 2 \cdot \sum_{i \in 1_Z} 2^i r_i - 1
\]

\[
= Z - 2 \cdot (R_t^{t-1} \land Z) - 1
\]

where \( \land \) denotes bit-wise AND. Therefore,

\[
R_t^{t-1} = \text{SHA-1} \left( Z - 2 \cdot (R_t^{t-1} \land Z) - 1 \right)_R
\]

The equation above shows that the only bits of \( R_t^{t-1} \) which affect the result are bits \( r_i \) for which the corresponding bit \( z_i \) equals 1. The attack can therefore be more efficient: Consider, for example, the case that the 20 least significant bits of \( Z \) are 1, the next 20 bits are 0, and the other bits have arbitrary values. The attack enumerates over all \( 2^{20} \) options for
r_{19}, \ldots, r_0. For each possible option it computes the expression detailed above for R_{R^{-1}}^{t-1}. It then compares the 20 least significant bits of the result to r_{19}, \ldots, r_0. If they are different it disregards this value of r_{19}, \ldots, r_0, and if they are equal it saves it. As before, the correct value is always retained, while each of the other 2^{20} - 1 values is retained with probability 1/2^{20}. We therefore expect O(1) false positives.

In the general case, the attack enumerates over all possible values of the bits of R_{R^{-1}}^{t-1} which affect the result, namely r_i for which 0 \leq i \leq 39 and i \in 1_Z. In case there are \ell such bits, the attack takes 2^{\ell} time. Therefore, assuming that Z is random, the expected complexity of the attack is \sum_{\ell=0}^{40} 2^{\ell} \Pr(|1_Z R| = \ell) = \sum_{\ell=0}^{40} 2^{\ell} \binom{40}{\ell} 2^{-40} = (3/2)^{40} \approx 2^{23}. As before, the number of false positives is O(1), since for every value of \ell we examine 2^{\ell} - 1 incorrect values, and each one of them is retained with probability 2^{-\ell}.

We implemented this attack in the tool PreviousCryptGenOutputs23. The average running time of recovering a previous state is about 19 seconds on a 2.80MHz Pentium IV (without any optimization). The tool can recover all previous states until the time the generator was initialized, as is detailed below.\footnote{We note that there exist much faster implementations of SHA-1, and consequently of the attack. For example, recent experiments on the Sony PS3 machine show that on that platform it is possible to compute 86-87 million invocations of SHA-1 per second (applying the function to 20 byte long inputs). In this implementation, computing 2^{23} invocations of SHA-1 should take less than 1/10 of a second. (The overall complexity of the attack is, of course, somewhat greater.)}

Repeatedly applying the attack on forward security. The procedures detailed above provide a list of O(1) candidate values for the state of the generator at time t-1. They can of course be applied again and again, revealing the states, and consequently the outputs, of the generator at times t-1, t-2, etc. As for the number of false positives, in each of the attacks we have 2^{\ell} - 1 possible false positives, and each of them passes the test with probability 2^{-\ell}. The analysis of this case is identical to the analysis of the number of false positives in an attack on the forward security of the Linux random number generator (see 3). In that
analysis it was shown that the number of false positives can be modeled as a martingale, and that its expected value at time \( t - k \) is only \( k \). (The number of false positives does not grow exponentially since for any false positive for the value of the state at time \( t - k \), it happens with constant probability that the procedure detailed above does not result in any suggestion for the state at time \( t - k - 1 \). In this case we can dismiss this false positive and should not explore its preimages.)

Of course, if the attacker knows even a partial output of the generator at some previous time \( t - k \) it can use this knowledge to identify the true state of the generator at that time, and remove all false positives.

**The effect of the attacks.** The WRNG has no forward and backward security: an attacker which learns the state of the generator at time \( t \) can easily compute past and future states and outputs, until the times where the state is refreshed with system based entropy. Computing all states and outputs from time \( t \) up to time \( t + k \) can be done in \( O(k) \) work (i.e., \( O(k) \) invocations of SHA-1). Computing candidates to all states and outputs from time \( t \) to time \( t - k \) can be done in \( O(2^{23}k^2) \) work. (I.e., in a matter of minutes, depending on \( k \). The \( O(2^{23}k^2) \) result is due to the fact that for every \( 1 \leq j \leq k \) we expect to find \( j \) candidate values for time \( t - j \), and to each of these we apply the \( 2^{23} \) attack to learn its predecessor.) An attacker which learns the state at time \( t \) can therefore apply this knowledge to learn all states of the generator in an “attack window”, which lasts from the last refresh (or initialization) of the state before time \( t \), to the first refresh after time \( t \). As discussed above, the WRNG keeps a separate state per process, and this state is refreshed only after the generator generates 128 Kbytes of output. Therefore, we can sum up this section with

---

8 In general, forward security should be provided by the function which advances the generator, and the use of entropy to refresh the state of the generator is only intended to limit the effect of backward security attacks. In the case of the WRNG, the generator itself provides no forward security. Entropy based refreshes therefore help in providing some limited forward security: the attack can only be applied until the last time the generator was refreshed.
the following statement:

Knowledge of the state of the generator at a single instance in time suffices to predict 128 Kbytes of its output. These random bits are used in the time period lasting from the last entropy refresh before the attack to the first refresh after it.

In case of a process with low random bit consumption, this window might cover days of usage. In the case of Internet Explorer, we note in Section 4.4 that it might run 600-1200 SSL connections before refreshing the state of its WRNG. This observation essentially means that, for most users, leakage of the state of the WRNG used by Internet Explorer reveals all SSL keys used by the browser between the time the computer is turned on and the time it is turned off.

An observation about state updates. The update of the variable State in the main loop is based on exclusive-orring and adding $R$. More precisely, let $S^t$ denote the value of State at the beginning of the $t$th iteration of the loop. Then $S^{t+1} = (S^t \oplus R) + R' + 1$, where $R'$ is identical to $R$, except for the five least significant bytes which are replaced with bytes from the output of the WRNG (which might be known to an attacker). The addition and exclusive-or operations are related (they are identical up to the effect of the carry, which affects addition but not the exclusive-or operation). Therefore $S^{t+1}$ is strongly related to $S^t$, much more than if, say, it was defined as $S^{t+1} = S^t \oplus R$.

Similarity to the Digital Signature Standard. According to [31] the main algorithm of the WRNG is based on the PRNG used in NIST Digital Signature Standard (DSS) (also known as FIPS 186-2) [20]. The authors of [31] explain that the WRNG is based on the DSS design where system entropy is replacing user input. The WRNG algorithm is different than the one used in DSS, and is less secure against forward security attacks.
4.4 Analysis II: The Interaction between the Operating System and the Generator

We describe here how the generator is invoked by the operating system, and how this affects its security.

**Frequency of entropy based rekeys of the state.** Each process has its own copy of a WRNG instance. Since each instance of the WRNG uses eight RC4 streams, its state is refreshed only after it generates 128 Kbytes of output. Between refreshes the operation of the WRNG is deterministic. If one process (say, a web browser) uses very little pseudo-random bits, the WRNG instance that is used by this state will be refreshed very rarely, even if other processes consume many pseudo-random bits from the instances of the WRNG that they use. We described in Section 4.3 attacks on the forward and backward security of the WRNG which enable an attacker which observes a state of the WRNG to learn all states and outputs of the generator from the time it had its last refresh (or initialization) to the next time it will be refreshed.

**Entropy based rekeys in Internet Explorer.** We examined the usage of the WRNG by Internet Explorer (IE), which might be the most security sensitive application run by most users (all experiments were applied to IE 6, version 6.0.2800.1106). The examination of Internet Explorer was conducted by hooking all calls to CryptGenRandom using a kernel debugger, and recording the caller and the number of bytes produced. When IE invokes SSL it calls the WRNG through LSASS.EXE, the system security service, which is used by IE exclusively for this purpose (as mentioned before, as a service LSASS.EXE keeps its own state of the WRNG). During an SSL session, there is a varying number of requests (typically, four or more requests) for random bytes. Each request asks for 8, 16 or 28 bytes at a time. We can therefore estimate that each SSL connection consumes about 100-200
bytes of output from the WRNG. This means that the instance of the WRNG used by IE asks for a refresh only after handling about 600-1200 different SSL connections. It is hard to imagine that normal users run this number of SSL connections between the time they turn on their computer and the time they turn it off. Therefore, the attacks presented in Section 4.3 can essentially learn encryption keys used in all previous and future SSL connections of the attacked PC.

**Initializing State and R.** The variables State and R are not explicitly initialized by the generator, but rather take the last value stored in the stack location in which they are defined. This means that in many circumstances these values can be guessed by an attacker knowledgeable in the Windows operating system. This is particularly true if the attacker studies a particular application, such as SSL or SSH, and learns the distribution of the initial values of these variables. Knowledge of these values enables an instant attack on the generator which is even more efficient than the $2^{23}$ attack we describe (see Section 4.3).

We performed some experiments in which we examined the initial values of State and R when the generator is invoked by Internet Explorer. The results are as follows: (1) In the first experiment IE was started after rebooting the system. In different invocations of the experiment the variables State and R were mapped to different locations in the stack, but their initial values were correlated. (2) In the second experiment IE was restarted 20 times without rebooting the system. All invocations had the same initial values of State and R. (3) In the third experiment we ran 20 sessions of IE in parallel. The initial values of the variables were highly correlated (in all invocations but one, the initial value was within a Hamming distance of 10 or less from the initial value of another invocation).

**Maintaining the state of State and R.** The variables State and R are maintained on the stack. If the WRNG is called several times by the same process, these variables are not kept in static memory between invocations of the WRNG, but are rather assigned to
locations on the stack each time the WRNG is called (in this respect they are different from
the RC4 states, which are kept in static memory and retain their state between invocations).
If State and R are mapped to the same stack locations in two successive WRNG invocations,
and these locations were not overwritten between invocations, then the variables retain their
state. Otherwise, the variables in the new invocation obtain whatever value is on the stack.
We performed several initial experiments to examine the values of State and R when the
WRNG is used by IE. In all but a few invocations they were assigned to the same stack
location and retained their state between invocations. In the few times that they were
assigned to other locations, their values were correlated.

We do not know how to explain this “loose” management of the state, and cannot tell
whether it is a feature or a bug. In the attacks we describe in Section 4.3 we show how to
compute previous states assuming that State and R retain their state between invocations
of the generator. These attacks are relevant even given the behavior we inspected above, for
two reasons: (1) We observed that in IE the WRNG almost always retains the values of State
and R. When it does not, the values of these variables seem to be rather predictable. The
attacker can therefore continue with the attack until it notices that it cannot reproduce the
WRNG output anymore. The attacker should then enumerate over the most likely values
of State and R until it can continue the attack. (This attack requires an additional analysis
of State and R, but it does seem feasible.) (2) Other applications might use the WRNG is
such a way that the stack locations in which the values of State and R are stored are never
overwritten.

Initialization of RC4 states. As noted above, the different RC4 instances used by the
same WRNG instance are initialized one after the other by vectors of system data which are
quite correlated. This is also true, to a lesser extent, for two instances of the WRNG run
by two processes. On the other hand, the VeryLargeHash function which is applied to these
values is based on the SHA-1 hash function, and is likely to destroy any correlation between related inputs. We have not examined the entropy sources in detail, and have not found any potential correlation of the outputs of the VeryLargeHash function.

The output of VeryLargeHash is used as a key for two RC4 encryption of the variables Seed and KSecDD, respectively, and the result is used to initialize the RC4 state of the WRNG. Even if an attacker knows the values of Seed and KSecDD, they do not help it to predict the output of VeryLargeHash, and consequently predict the initialization of the RC4 state. The RC4 algorithm itself is known to be vulnerable to related key attacks, and it is known that its first output bytes are not uniformly distributed [21]. We were not able, however, to use these observations to attack the WRNG, since it applies SHA-1 to its state before outputting it.\footnote{We note that the distribution of the first output bytes of RC4 is known to be slightly biased, and the output of the WRNG is computed by applying SHA-1 to a function of RC4, \textit{State} and \textit{R}. Therefore, an attacker which knows the values of \textit{State} and \textit{R} knows that there is a slight bias in the distribution of the output of the WRNG. However, this bias seems to be too weak to be useful.}

Although we were not able to attack the RC4 initialization process, it seems that a more reasonable initialization procedure would have gathered system entropy once, and used it to generate initialization data to all eight RC4 instances. (Say, by running the final invocation of RC4 in the initialization procedure to generate $8 \times 256 = 2048$ bytes which initialize all eight RC4 instances.)

\textbf{Protecting the state.} As the WRNG is running in user space (and not in protected kernel space), an adversary that wishes to learn the state of a certain application needs only break into the address space of the specific application. This property increases the risk of an attacker learning the state of the WRNG, and consequently applying forward and backward security attacks. (The WRNG is run is user space since each application is running its own WRNG copy. The other option would have been to let the system run a single generator in the kernel, and use it to provide output to all applications.)
4.5 Comparison to the Linux PRNG

The pseudo-random number generator used in the Linux operating system (denoted LRNG) was analyzed in 3. The analysis of the WRNG shows that it differs from the LRNG in several major design issues.

- **Kernel versus User mode.** The LRNG is implemented entirely in kernel mode while a large part of the WRNG is running in user mode.

  **Security implication:** An application which runs in Windows and uses the WRNG can read the entire state of the WRNG, while the LRNG is hidden from Linux applications. This means that, compared to Linux, it is easier for an attacker to obtain a state of the WRNG.

- **Reseeding timeout.** The LRNG is feeding the state with system based entropy in every iteration and whenever system events happen, while the WRNG is reseeding its state only after generating 128 KBytes of output.

- **Synchronization.** The collection of entropy in the LRNG is asynchronous: whenever there is an entropy event the data is accumulated in the state of the generator. In the WRNG the entropy is collected only for a short period of time before the state is reseeded. In the long period between reseedings there is no entropy collection.

- **Scoping.** The LRNG runs a single copy of the generator which is shared among all users running on the same machine. In Windows, on the other hand, a different instance of the generator is run for every process on the machine.

- **Efficiency of attacks.** The best forward security attack on the LRNG requires $O(2^{64})$ work. The attack on the forward security of the WRNG is therefore more efficient by a factor of about $2^{40}$ (it has a complexity of $O(2^{23})$ compared to $O(2^{64})$).
Security implication: The impact of the previous four properties is that forward and backward security attacks are more severe when applied to the WRNG. The attacks are more efficient by twelve orders of magnitude. They reveal the outputs of the generator between consecutive reseedings, and these reseedings are much more rare in the case of the WRNG. In some cases, reseeding the LRNG happens every few seconds, while the WRNG is reseeded every few days, if it is reseeded at all.

- Blocking. The LRNG implements an entropy estimation counter which is used to block it from generating output when there is not enough system entropy within the generator. This has advantages such as stronger resistance to break-in attacks, but also leads to situations where the generator halts until sufficient system entropy is collected. Hence, this also leads to easy denial of service attacks when one consumer of pseudo-random bits can empty the system entropy pools and block other users. The WRNG does not use entropy measurements, and is therefore not blocking.

Security implication: Unlike the LRNG, the WRNG is not vulnerable to denial of service attacks.

4.6 Conclusions

4.6.1 Conclusions

WRNG design. The thesis presents a clear description of the WRNG, the most frequently used PRNG. The WRNG has a complex layered architecture which includes entropy rekeying every 128 KBytes of output, and uses RC4 and SHA-1 as building blocks. Windows runs the WRNG in user space, and keeps a different instance of the generator for every process.

Attacks. The WRNG depends on the use of RC4, which does not provide any forward security. We used this fact to show how an adversary which learns the state of the WRNG
can compute past and future outputs of the generator. The attacker can learn future outputs in $O(1)$ time and compute past outputs in $O(2^{23})$ time. These attacks can be run within seconds or minutes on a modern PC and enable such an attacker to learn the values of cryptographic keys generated by the generator. The attacks on both forward and backward security reveal all outputs until the time the generator is rekeyed with system entropy. Given the way in which the operating system operates the generator, this means that a single attack reveals 128 KBytes of generator output for every process.

*Code analysis.* Our research is based on studying the WRNG by examining its binary code. We were not provided with any help from Microsoft and were only using the binary versions of Windows. To verify our findings we developed a user mode simulator which captures WRNG states and computes future and past outputs of the WRNG. We validated the simulator output against real runs of the WRNG.

*WRNG versus LRNG.* We compared between the pseudo-random generators used by Windows and Linux (WRNG vs. LRNG). The forward security attack on the WRNG is faster by a factor of $O(2^{40})$ compared to the attack on the LRNG. In addition, our findings show that the LRNG has better usage of operating system entropy, uses asynchronous entropy feedings, uses the extraction process as an entropy source, and shares its output between multiple processes. As a result, a forward security attack on the WRNG reveals longer sequences of generator output, compared to an attack on the LRNG.

### 4.6.2 Recommendations

*Forward security.* The most obvious recommendation is to change the algorithm used by the WRNG to one which provides forward security. This can be done by making local changes to the current implementation of the generator, or by replacing RC4 with a function which provides forward security. Alternatively, it is possible to use the transformation of [8] which transforms any standard generator to one providing forward security. We believe however
that it is preferable to replace the entire algorithm used by the generator with a simpler algorithm which is rigorously analyzed. A good approach is to adopt the Barak-Halevi construction. That construction, suggested in [6], is a simple yet powerful construction of entropy based PRNGs. Its design is much simpler to implement than the current WRNG implementation and, assuming that its building blocks are secure, it provably preserves both forward and backward security. It can be implemented using, say, AES and a simple entropy extractor.

**Frequency of entropy based rekeys.** The generator should rekey its state more often. We also suggest that rekeys are forced based on the amount of time that has passed since the last rekey. It is important to note that entropy based rekeys are required in order to limit the effect of attacks mounted by an adversary which obtains the state of the generator. (In a good generator, forward security and pseudo-randomness are guaranteed by the function which advances the state, and are ensured even if the generator generates megabytes or gigabytes of output between rekeys.) The risk of an adversary getting hold of the state seems to be more dependent on the amount of time the system runs, than on the length of the output of the generator. It therefore makes sense to force rekeys every some time interval, rather than deciding whether to rekey based on the amount of output produced by the generator.

4.6.3 Open Problems

*Extending our research to additional Windows platforms.* Our entire research was conducted on a specific Windows 2000 build. We did several early checks on additional binary versions of Windows but that work is only in its beginning. The important operating systems to examine are the main Windows releases such as Windows XP and Windows Vista, as well as systems which have fewer sources of entropy, such as Windows CE.
State initialization. As we stated in our analysis, the internal RC4 states are initialized and rekeyed with very similar entropy parameters. These are hashed by a procedure which uses SHA-1 and propagates a change in the value of a single input bit to all output bits. The result of this procedure initializes the RC4 algorithm. We were not able to use this finding, but it seems that additional research is needed here. The research should examine the different entropy sources and the hashing algorithm, and check if they result in any related key attack on RC4. We also noted that the state variables State and R are not explicitly initialized but rather take the current values stored in the stack. More research is needed to examine in detail the distribution of these values.
Chapter 5

Summary

Almost all cryptographic systems are based on the use of a source of random bits, whose output is used, for example, to choose cryptographic keys or choose random nonces. The security analysis (and proofs of security) of secure systems are almost always based on the assumption that the system uses some random data (e.g., a key) which is uniformly distributed and unknown to an attacker. The use of weak random values may result in an adversary being able to break the system (e.g., weak randomness may enable the adversary to learn the cryptographic keys used by the system). This was demonstrated for example by the analysis of the implementation of SSL in Netscape [22], or in an attack predicting Java session-ids 2.

Physical sources of randomness are often too costly and therefore most systems use a pseudo-random number generator. The generator is modeled as a function whose input is a short random seed, and whose output is a long stream which is indistinguishable from truly random bits. Implementations of pseudo-random generators often use a state whose initial value is the random seed. The state is updated by an algorithm which changes the state and outputs pseudo-random bits, and implements a deterministic function of the state of the generator. The theoretical analysis of pseudo-random generators assumes that the state is
initialized with a truly random seed. Implementations of pseudo-random generators initialize the state with random bits ("entropy") which are gathered from physical sources, such as timing of disk operations, of system events, or of a human interface. Many implementations also refresh (or "rekey") the state periodically, by replacing the existing state with one which is a function of the existing state and of entropy similar to that used in the initialization.

Chapter 2 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 [10], and emphasizes the need for deploying these with a large internal state.

Linux is the most popular open source project. The Linux random number generator is part of the kernel of all Linux distributions and is based on generating randomness from entropy of operating system events. The output of this generator is used for almost every security protocol, including TLS/SSL key generation, choosing TCP sequence numbers, and file system and email encryption. Although the generator is part of an open source project,
its source code (about 2500 lines of code) is poorly documented, and patched with hundreds of code patches.

We used dynamic and static reverse engineering to learn the operation of this generator. This thesis presents a description of the underlying algorithms and exposes several security vulnerabilities. In particular, we show an attack on the forward security of the generator which enables an adversary who exposes the state of the generator to compute previous states and outputs. In addition we present a few cryptographic flaws in the design of the generator, as well as measurements of the actual entropy collected by it, and a critical analysis of the use of the generator in Linux distributions on disk-less devices.

The pseudo-random number generator (PRNG) used by the Windows operating system is the most commonly used PRNG. The pseudo-randomness of the output of this generator is crucial for the security of almost any application running in Windows. Nevertheless, its exact algorithm was never published.

We examined the binary code of a distribution of Windows 2000, which is still the second most popular operating system after Windows XP. (This investigation was done without any help from Microsoft.) We reconstructed, for the first time, the algorithm used by the pseudo-random number generator (namely, the function \texttt{CryptGenRandom}). We analyzed the security of the algorithm and found a non-trivial attack: given the internal state of the generator, the previous state can be computed in $O(2^{23})$ work (this is an attack on the forward-security of the generator, an $O(1)$ attack on backward security is trivial). The attack on forward-security demonstrates that the design of the generator is flawed, since it is well known how to prevent such attacks.

We also analyzed the way in which the generator is run by the operating system, and found that it amplifies the effect of the attacks: The generator is run in user mode rather than in kernel mode, and therefore it is easy to access its state even without administrator privileges. The initial values of part of the state of the generator are not set explicitly,
but rather are defined by whatever values are present on the stack when the generator is called. Furthermore, each process runs a different copy of the generator, and the state of the generator is refreshed with system generated entropy only after generating 128 KBytes of output for the process running it. The result of combining this observation with our attack is that learning a single state may reveal 128 Kbytes of the past and future output of the generator.

The implication of these findings is that a buffer overflow attack or a similar attack can be used to learn a single state of the generator, which can then be used to predict all random values, such as SSL keys, used by a process in all its past and future operation. This attack is more severe and more efficient than known attacks, in which an attacker can only learn SSL keys if it is controlling the attacked machine at the time the keys are used.
Bibliography


Appendix A

LRNG Entropy Collection

We explain each “noise” source and the different valid values for the 32 bits of the type-value during the entropy addition procedure:

- *keyboard event.* The type-value contains the keyboard press and release codes, valid range between [0, 255].

- *mouse event.*

\[
\text{type-value} := (\text{type} \leftarrow 4) \oplus \text{code} \oplus (\text{code} \rightarrow 4) \oplus \text{value}
\]

Where type describes the event type - pressing or releasing in case of mouse buttons and start movement or end movement in case of mouse movement; code is the mouse button pressed (left, right or middle) or wheel scrolling in case of mouse buttons, and the axis of the movement (horizontal or vertical) in case of mouse movement; value is true/false in case of mouse buttons press or release\(^1\), 1 or \(-1\) for denoting scrolling direction (1 for up, \(-1\) for down) in case of wheel scrolling, and the size of movement in

\(^1\)Each action produces an input for all three buttons to the mouse type-value formula. The button that was active gets an action value = 1 while the others get value = 0.
case of mouse movement. In short, the mouse data is a 32-bit word with the movement size as its main entropy factor. However, only 10 bits are used for movement, another 2 bits are used for the buttons, so in fact only 12 out of the 32 bits are effective.

- **disk event.** Computed at completion of a disk (such as IDE, SCSI, Floppy, block devices) I/O operation. Its type-value is composed of major and minor numbers (major and minor numbers are operating system symbols that together uniquely define a device):

  \[
  \text{type-value} := 0x100 + ((\text{major} \ll 20) | \text{minor})
  \]

  If there is only one IDE disk, the type-value is fixed, since the major and minor numbers are constants. (In most cases the major is 3 (first IDE disk) and the minor is 0 (master), and their combined type-value yields 0x300100.) Assuming an average machine has no more than 8 disks, the type-value actual span is limited to 3 bits.

- **interrupt event.** The result of an interrupt occurrence is the IRQ (interrupt request channel) number, with a valid range of \([0,15]\). It is important to note that as of the current kernel versions only very limited number of hardware device drivers supply interrupt values to the LRNG. In many setups interrupts will not add any entropy events.
Appendix B

LRNG Probability Calculations

The number of false positives generated by the procedure for reversing the pool is a random variable with the following distribution: The probability of having \( k \) false positives, for \( k = 0, \ldots, n \), is 
\[
\binom{n}{k} n^{-k} (1 - 1/n)^{n-k} \approx e^{-1/k!},
\]
where \( n \) is either \( 2^{64} \) or \( 2^{96} \).

Starting from a single state of the pool at time \( t \), let us denote by \( d_i \) the number of false positives at time \( t - i \). Note that for every \( i \) there exists, in addition to the false positives, an additional candidate which is the true value of the pool at time \( t - i \).

In time \( t \) we have one good candidate and no false positives \( (d_0 = 0) \). Now,
\[
E(d_1) = \sum_{k=1}^{n} ke^{-1/k!} = e^{-1} \sum_{k=0}^{n-1} 1/k! = e^{-1} \cdot e = 1.
\]

It also holds that \( E(d_i|d_{i-1} = 1) = E(d_1) = 1 \). As a result, \( E(d_i|d_{i-1} = c) = (c + 1)E(d_{i-1}|d_{i-1} = 1) = c + 1 \). (We multiply \( E(d_i|d_{i-1} = 1) \) by \( c + 1 \) since we obtain false positives at time \( t - i \) from the \( c \) false positives at time \( t - i + 1 \) and in addition from the true value of pool \( t_i+1 \).)

A martingale is a sequence of random variables \( X_0, X_1, X_2, \ldots \) which satisfies the relation
\[
E(X_i|X_{i-1}, \ldots, X_0) = X_{i-1}.
\]
It is known that for a martingale $E(X_i) = E(X_0)$, and there are known tail bounds on the divergence from this expectation (see [42] for details).

Let us define the sequence $z_i = d_i - i$ (the deviation of $d_i$ from the value $i$). The sequence \{d_i\} is not a martingale but the sequence \{z_i\} is, since $E(z_i|z_{i-1} = c) = E(d_i|d_{i-1} = c + i - 1) - i = c + i - 1 + 1 - i = c = z_{i-1}$. We therefore get that $E(z_i) = E(z_0) = 0$ and $E(d_i) = i$.

It is now possible to apply the Kolmogorov-Doob inequality (see [42]), which states that $\Pr(\max(z_i) > b) < E(z_0)/b = 1/b$ (for the purpose of using this inequality we define $z_i = d_i - i + 1$ and therefore $z_0 = 1$). As a corollary we can obtain, for example, that the probability that $d_i$ is greater than $i + 100$ is at most $1/100$. 
RC4 is a stream cipher. Its initialization process is defined in Figure C.0.1. The process of generating output is defined in Figure C.0.2.

for $i$ from 0 to 255
  $S[i] := i$

$j := 0$

for $i$ from 0 to 255
  $j := (j + S[i] + \text{key}[i \mod \text{keylength}]) \mod 256$
  swap($S[i],S[j]$)

Figure C.0.1: RC4 Key Scheduling Algorithm (KSA). The array $key$ holds the key, $keylength$ is the key size in bytes.

$i := 0$

$j := 0$

while GeneratingOutput:
  $i := (i + 1) \mod 256$
  $j := (j + S[i]) \mod 256$
  swap($S[i],S[j]$)
  output $S[(S[i] + S[j]) \mod 256]$

Figure C.0.2: RC4 pseudo random generation algorithm. The output is xored with the clear text for encryption.

RC4 has no forward security. Suppose we are given its state just before the $t$th iteration of
the output generation algorithm (namely, the values of $S^t, i^t$ and $j^t$). It is easy to compute the previous state, and consequently the previous output, by running the following operations:

\[
\begin{align*}
\text{swap}(S[i], S[j]) \\
j &:= (j - S[i]) \mod 256 \\
i &:= (i - 1) \mod 256
\end{align*}
\]

Therefore, given the state of RC4 at a specific time, it is easy to compute all its previous states and outputs.

RC4 is a stream cipher. It is initialized by the following key scheduling algorithm (KSA).
(The array \textit{key} holds the key, \textit{keylength} is the key size in bytes.)

\[
\begin{align*}
\text{for } i \text{ from } 0 \text{ to } 255 \\
S[i] &:= i \\
j &:= 0 \\
\text{for } i \text{ from } 0 \text{ to } 255 \\
j &:= (j + S[i] + \text{key}[i \mod \text{keylength}]) \mod 256 \\
\text{swap}(S[i], S[j])
\end{align*}
\]

Following is the algorithm used by RC4 to generate pseudo random output. This output is xored with the clear text for encryption.

\[
\begin{align*}
\text{i} &:= 0 \\
\text{j} &:= 0 \\
\text{while GeneratingOutput:} \\
i &:= (i + 1) \mod 256 \\
j &:= (j + S[i]) \mod 256 \\
\text{swap}(S[i], S[j]) \\
\text{output } S[(S[i] + S[j]) \mod 256]
\end{align*}
\]