Cryptanalysis of the Windows Random Number Generator

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science

by

Leo Dorrendorf

Supervised by

Prof. Danny Dolev

and

Dr. Benny Pinkas

School of Engineering and Computer Science
The Hebrew University of Jerusalem
Israel

December 24, 2007

Acknowledgements

I would like to thank my advisors for their support, insight, and patience:
Danny Dolev, Zvi Gutterman, Benny Pinkas and Scott Kirkpatrick.

I would like to express my special gratitue to Michael Kara-Ivanov, whose understanding and encouragement helped me complete this research.

I want to thank everyone who ever taught me about reverse-engineering or shared knowledge of operating systems: Nethanel Elzas, Vadim Penzin, Yaakov Belenky, and Nathan Fain.

Special thanks to Tzachi Reinman and Yaron Sella for reviewing drafts of this thesis.

I owe thanks to many friends, colleagues, reviewers and listeners,
and others...

Abstract

Random numbers are essential in every cryptographic protocol. The quality of a system's random number generator (RNG) is therefore vital to its security. In Microsoft Windows, the operating system provides an RNG for security purposes, through an API function named CryptGenRandom. This is the cryptographic RNG used by the operating system itself and by important applications like the Internet Explorer, and the only RNG recommended for security purposes on Windows. This is the most common security RNG in the world, yet its exact algorithm was never published.

This work provides a description of the Windows RNG, based on examining the binary code of Windows 2000. We reconstructed the RNG's algorithm, and present its exact description and an analysis of the design. Our analysis shows a number of weaknesses in the design and implementation, and we demonstrate practical attacks on the RNG. We propose our recommendations for users and implementors of RNGs on the Windows platform. In addition, we describe the reverse-engineering process which led to our findings.

Table of Contents

1 Introduction **9**
- 1.1 Contributions . . . 9
- 1.2 Outline . . . 10

2 Background **11**
- 2.1 The Role of Randomness in Security . . . 11
- 2.2 The Gap Between Theory and Practice . . . 11
- 2.3 True RNGs versus PRNGs . . . 12
  - 2.3.1 True Random Number Generators . . . 12
  - 2.3.2 Pseudo Random Number Generators . . . 12
- 2.4 PRNG Security Requirements . . . 13
- 2.5 Algebraic PRNGs . . . 14
- 2.6 Applied Cryptography PRNGs . . . 15
- 2.7 Operating System PRNGs . . . 15
- 2.8 Public Information on the WRNG . . . 16

3 Related Work **17**
- 3.1 Overview of RNG Research . . . 17
- 3.2 RNG Attacks . . . 18
  - 3.2.1 Netscape SSL . . . 18
  - 3.2.2 Kerberos 4.0 . . . 18
  - 3.2.3 Java Virtual Machine . . . 18
  - 3.2.4 Linux RNG . . . 19
- 3.3 Suggested RNG Designs . . . 19
  - 3.3.1 Barak-Halevi Construction . . . 19
  - 3.3.2 Yarrow and Fortuna . . . 20

4 The Structure of the Generator **22**
- 4.1 Using the WRNG . . . 22
  - 4.1.1 Windows Crypto API . . . 22
  - 4.1.2 The CryptGenRandom API . . . 23
- 4.2 Internal Structure . . . 24
  - 4.2.1 Main Loop . . . 24

4.2.2 Internal Generator . . . 25
4.2.3 Entropy Collector . . . 25
4.3 Sub-Components . . . 27
4.3.1 SHA-1 Hash Function . . . 27
4.3.2 RC4 Stream Cipher . . . 27
4.3.3 VeryLargeHash Function . . . 29
4.3.4 Kernel Security Device Driver (KSecDD) . . . 29
4.3.5 CircularHash Component . . . 30

5 Interaction with the System **31**
5.1 Scope . . . 31
5.2 Synchronous Entropy Collection . . . 32
5.3 Invocation by the System . . . 33

6 Analysis of the Design **34**
6.1 Uninitialized Variables . . . 34
6.2 Complexity . . . 35
6.3 Misleading Documentation . . . 36
6.4 FIPS-186 Conformance . . . 36
6.5 Pseudo-Randomness . . . 36

7 Cryptanalysis and Attacks **37**
7.1 State Extraction Attack . . . 38
7.2 Attack on Backward Security . . . 38
7.3 Attacks on Forward Security . . . 39
7.3.1 Attack I: Inverting RC4 . . . 40
7.3.2 Attack II: Recovering Stack Components . . . 40
7.3.3 Attack III: Improved Search . . . 41
7.4 Combining the attacks . . . 43

8 Recommendations for Safe Use **44**
8.1 Recommendations to Users of the Current WRNG . . . 44
8.1.1 Consume More Random Bytes . . . 44
8.1.2 Drain All Random Bytes . . . 45
8.1.3 Patch the WRNG . . . 45
8.1.4 Other Modes of Invoking the WRNG . . . 45
8.2 Possible Fixes . . . 46
8.2.1 Replace RC4 . . . 46
8.2.2 Collect Entropy More Often . . . 46
8.2.3 Move into the Kernel . . . 46
8.2.4 Switch to a Proven Design . . . 46

9 Reverse Engineering **47**
9.1 Basic Analysis . . . 48
9.2 Running under a Debugger . . . 48
9.3 Static Analysis . . . 49
9.4 Intrusive Analysis . . . 50
9.5 Kernel Debugging . . . 50

10 Future Work **51**
10.1 Unexplored Attack Venues . . . 51
10.1.1 Ciphertext-only attack . . . 51
10.1.2 Entropy Input Correlation . . . 51
10.2 Other Operating Systems . . . 52
10.2.1 Windows XP . . . 52
10.2.2 Windows Vista . . . 52
10.2.3 Other Platforms . . . 52

11 Conclusions **54**

List of Figures

4.1 The CryptGenRandom API 23
4.2 Sample code using CryptGenRandom 23
4.3 The main loop of the WRNG 24
4.4 The internal generator 25
4.5 Entropy collection and re-keying 26
4.6 RC4 Key Scheduling Algorithm (KSA) 28
4.7 RC4 output generation 28
4.8 Running RC4 in reverse 28
4.9 VeryLargeHash algorithm 29

7.1 The 2^{23} attack on forward security 42

9.1 Windows 2000 binaries' versions and MD5 signatures 49

List of Tables

4.1 Entropy sources 27

5.1 CryptGenRandom component scopes 32

Chapter 1

Introduction

1.1 Contributions

The purpose of this work is to reconstruct the algorithm of the Windows RNG (WRNG), and analyse its security.

The work began when no exact description of the WRNG was published. Since the source code was not publicly available, reverse-engineering had to be used. We fully reverse-engineered the Windows 2000 RNG binaries, and as a result, were the first to publish the exact algorithm.

The algorithm is presented in this thesis in concise pseudocode; the generator's full code is about 10,000 lines of assembly, or what could be 1,000 lines of C. We include an overview of reverse-engineering tools and techniques that could be used to validate our results, or for studying any other binaries.

Having obtained an exact description of the WRNG, we analyzed its design. We examined the algorithm, the implementation-specific parameters, and the WRNG's interaction with the operating system. In the process, we used both static analysis (reading code) and dynamic analysis, examining the WRNG at run-time in different inspection environments.

The analysis was validated by simulator tools, which generate the same output as the WRNG, given its internal state. We provide the method using which an attacker with access to the process memory can save a snapshot of the WRNG state for the simulator.

Analyzing the security of the algorithm, we found a non-trivial attack: given the internal state of the generator at some time, its states at previous times could be recovered in a matter of seconds on a home computer. This attack breaks the forward security of the generator, demonstrating a fundamental flaw in its design, as it is well known how to prevent such attacks.

We present our opinion on fixing the WRNG design in the future, and also give our recommendations for safe use of the present WRNG.

1.2 Outline

The rest of this thesis goes as follows.
Chapter 2 provides the background for this work, explaining the importance of RNGs for security, the difference between RNGs and pseudo-random number generators (PRNGs), and describing PRNG security requirements. Chapter 3 presents related work, including earlier PRNG attacks and accepted PRNG designs. Chapter 4 shows the internal structure of the WRNG in detail. Chapter 5 describes the interaction of the WRNG with the rest of the system. Chapter 6 examines several details of the WRNG implementation and the claims about its security. Chapter 7 shows a cryptanalysis of the WRNG and attacks that compromise its outputs. Chapter 8 contains our recommendations for safe use of the present WRNG, and proposes fixes for future versions. Chapter 9 outlines the reverse-engineering process used to obtain a high-level description of the WRNG from its binaries. Chapter 10 presents research goals that were not achieved in this work, and directions for further research.

Chapter 2

Background

2.1 The Role of Randomness in Security

Randomness is an essential resource for cryptography. Any cryptographic system relies on secrets, and can only be as strong as the random number generator used to create those secrets. As soon as an attacker (eavesdropper or intruder) can guess the secrets, the best cryptographic schemes become useless.

In practical cryptogrphy, RNGs are used to create keys, seeds and nonces, all of which must be unpredictable. Although the theory of random number generation is well researched, building a secure RNG in practice has proven to be quite difficult. Unfortunately, the problem of RNG design is often overlooked: the security analysis of most systems starts out by taking unpredictable random numbers for granted.

2.2 The Gap Between Theory and Practice

The generation of secure random numbers is a well-studied issue in cryptography [55, 5]. Nevertheless constructing a real-world implementation can be quite complex. There are many reasons for this gap between theory and practice:

- *Performance.* Random number generators with provable security guarantees incur a high computation overhead. Therefore weaker designs without security proofs are more common in practice.
- *Real world attacks.* Actual implementations are prone to many attacks which do not exist in the clean cryptographic formulation used to design and analyze PRNGs.
- *Seeding and reseeding the generator.* To be secure, generators must be initialized with a truly random seed. Furthermore, the state of the generator must be periodically refreshed with random inputs, as discussed in Section 2.4. Finding good entropy sources is not simple. The developer of an RNG must identify and use many independent random sources with high enough entropy.

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

2.3 True RNGs versus PRNGs

This section deals with the difference between theoretical and physical RNGs, which themselves are sources of entropy, and pseudo-random number generators (PRNGs) that can be implemented in software and use external sources of entropy.

2.3.1 True Random Number Generators

Ideal random number generators are sources of truly random data - completely unpredictable and uncorrelated bits. Hardware random number generators are a real-life approximation of that model. True randomness exists in the real world, for example some quantum phenomena are unpredictable. There are hardware devices that provide random bits based on photon counting [31]. Cheaper and simpler RNGs can be implemented on regular PC hardware; for example, Intel included an RNG based on thermal noise in their 800 family of chipsets [30], and measuring air turbulence in hard drives by timing I/O operations was suggested in [12].

Unfortunately, hardware RNGs have several weaknesses. They are subject to bias, and need a post-processing layer to remove it. They are vulnerable to environmental interfence and intentional sabotage making them predictable [25]. The extra hardware incurs a cost to the user. Finally, reliance on a hardware RNG limits the portability of an application.

For all these reasons, most applications compromise on true unpredictability in favor of pseudo-random number generation implementable in software.

2.3.2 Pseudo Random Number Generators

A pseudo-random number generator is a further approximation of a true RNG. It's a deterministic machine which produces output that *looks* truly random to an outside observer, and is random enough for practical purposes. Unlike true RNGs, PRNGs can be built in software.

In theoretical cryptography, a pseudo-random number generator is defined as a deterministic function of the following form:

$$F : \{0,1\}^k \rightarrow \{0,1\}^n \tag{2.1}$$

F is a one-way function mapping a short input to a longer output, so that $n > k$.

If the input x is chosen uniformly at random, then no efficient algorithm can distinguish between $F(x)$ and a truly random string of the same length. This property implies that the output appears to be uniformly distributed, and is unpredictable without the knowledge of x [23].

The first k bits of the result serve as the input on the next iteration, and the extra $n - k$ bits are output to the user. Designating F's first input as the initial state, and each next input as the next state, we create a state machine with F as the transition function. Repeated application makes it into a pseudo-random number generator.

The security of this construction hinges on the secrecy of the internal state. The requirement that the transition function be one-way assures that an attacker cannot expose the internal state by observing the outputs. If F is one-way under some assumption (for example, the computational difficulty of integer factorization), the resulting PRNG is provably secure under the same assumption[1].

To make the initial state secret from the potential attacker, it must be seeded by secret and uniformly distributed data. This *seed* can be requested from the human operator, but human input might not be sufficiently random. It can be gathered from system inputs such as timing of I/O operations, process performance tables etc.

Most implementations also reseed the state periodically, in part to prevent the transition function from entering short cycles, and in part to increase the entropy of the internal state as seen from the attacker's perspective.

2.4 PRNG Security Requirements

PRNGs must fulfill a number requirements in order to be cryptographically secure. In the accepted attack model, the attacker has access to any part of the PRNG output, knows the PRNG code, and can control some of the entropy sources. Furthermore, he can compromise the PRNGs internal state, gaining full access for a limited time. It is possible to build PRNGs that recover from such attacks by refreshing themselves with inputs from entropy sources not compromised by the attacker. We list here the major security requirements for PRNGs, in common terminology established e.g. by [2, 35][2].

Pseudo-randomness An external observer cannot distinguish the output of the PRNG from true random. Even if given almost all of the output, the observer cannot predict the remainder. By extension, the observer must not be able to compute the PRNG's internal state. This requirement is easily achieved if the PRNG output is a one-way function of the state, and the state itself evolves between iterations.

Backward Security An attacker that exposes the generator's internal state, cannot predict future outputs of the RNG. Since PRNGs are deterministic machines, the attacker will be able to simulate the PRNG algorithm and reproduce its output. The only means to prevent complete compromise is to refresh the internal state periodically with randomness from external entropy sources. Once enough entropy is collected from sources not compromised by the attacker, he loses the ability to simulate the PRNG and expose future outputs.

This property is also called *break-in recovery* or *prediction resistance*.

[1] One implementation of this principle is the Blum-Blub-Shub PRNG, described in Section 2.5.

[2] There are other requirements, like input security: the PRNG must not expose sensitive information by leaking its entropy inputs. For example, careless use of keypress data can expose login passwords.

Forward Security An attacker that exposes the generator's internal state, cannot recover previous outputs of the RNG. This requirement is easily achieved if the PRNG's transition function is one-way.

This property is also called *backtracking resistance*.

Resilience to Sustained State Exposure A sustained state exposure attack is an extension of the attack on backward security: an attacker exposes the generator's internal state, then continues to watch the generator's outputs. As long as the entropy inputs refreshing the state are within attacker's brute-force search capabilities, he can test every option of inputs and compute every possible resulting state, comparing the simulated outputs with the observed output. In effect, the attacker sustains the exposure of the internal state. Recovering from this attack is difficult. The attacker's capabilities are unknown (but assumed limited), therefore the generator must be refreshed an unknown amount of entropy to recover. This leads to PRNG designs like Fortuna, with multiple entropy "pools" of different sizes (see Section 3.3.2).

This property is also called *resilience to iterative guessing attacks*.

2.5 Algebraic PRNGs

Historically, simple algebraic PRNGs have been offered by programming languages' default libraries since the 1960's [37]. One prominent type is the Linear Congruential Generator (LCG), based on the following recurrence:

$$X_{i+1} = (a * X_i + b) \bmod n \tag{2.2}$$

Where a,b,n are integer constants, and the initial value X_0 is usually derived from user input or machine time. After decades of use, it was shown that LCGs (including multivariate LCGs) are very weak: their output often covers a small portion of the possible space, is has correlations, and exposes their internal state after a short series of outputs [40, 38, 52]. The use of LCGs can compromise an otherwise secure system [4].

This makes LCGs unsuitable for cryptographic purposes. LCGs may still be the default PRNGs in some programming languages (such as `rand()` in `C` and `C++`), so programmers should use these functions with caution.

A notable algebraic PRNG is the Mersenne Twister [41]. It is very efficient and produces high-quality output, usable for any purpose except cryptography: an attacker can compute its internal state after seeing a series of outputs[3].

The Blum-Blum-Shub generator [5] uses integer squaring over a complex modulus as the transition function. For output, it takes a few of the least-significant bits of the state. Breaking its security is provably as hard as factoring large integers, but because

[3]For example, 624 outputs are needed to compute the internal state for MT19937, the standard Mersenne Twister instantiation.

of the high computational cost and small output size at each iteration, this PRNG is mostly considered impractical.

2.6 Applied Cryptography PRNGs

Today, some cryptographic primitives are widely believed to have good security, despite the absense of formal proofs. For example, the SHA-1 hash function [51] is believed to be a strong one-way function: given its output, the original input cannot be computed by practical means.

Based on such assumptions, it is possible to construct efficient PRNGs with output quality that passes the most stringent statistical tests. Indeed, most modern-day PRNGs are based on standard hash functions, block ciphers, and stream ciphers. The design decisions for applied cryptography RNGs are:

State representation What constitutes the PRNG internal state, which variable types and sizes represent it; where is it stored and how is it protected.

State transition Which function updates the state. Hash functions, block ciphers and stream ciphers are possible alternatives; note that the function must be one-way in order to satisfy the forward security property (see Section 2.4).

Output generation Which function of the state produces the output. This function being one-way is necessary (but not sufficient) for ensuring the pseudo-randomness of the output, and the secrecy of the internal state. Hash functions are natural candidates.

Initial seeding Which data is used to seed the initial state. Human inputs and other simple options like system clocks are poor sources of entropy [25]. The PRNG designer must find other system data with enough randomness. This problem is especially severe immediately after initialization, when the system state is highly predictable. Many PRNGs therefore store some random output in temporary storage, and seed themselves with its contents on re-initialization.

Reseeding and entropy collection When and how to refresh the internal state. Refreshing is needed for backward security (see Section 2.4). As with initial seeding, sources with high enough entropy must be found. The timing of entropy collection can be synchronous (done when the PRNG is invoked), or asynchronous (done peridocally or in response to external events).

2.7 Operating System PRNGs

PRNGs can benefit from integration with the OS in many ways. The operating system can help keep the PRNG's internal state secret. From a cryptographic point of view, the operating system can initialize and refresh the PRNG with OS data hidden from

users. In an environment with multiple users and multiple processes, the PRNG can benefit from entropy generated by all the interactions. Finally, the designers of an operating system should be versed in security and cryptography, and can be expected to implement an efficient and secure generator. 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.

Indeed, all major operating systems provide PRNGs for security use. All Windows versions since Windows 95 have a Crypto API function `CryptGenRandom` for cryptographically secure random numbers. Its design is proprietary and has not been published fully before this work. Linux, Mac OS, FreeBSD and OpenBSD provide access to PRNGs through device nodes `/dev/random` and `/dev/urandom`. The Linux RNG is open source, and its design is analyzed in [28]. All the others are based on the closely related Yarrow and Fortuna designs (see Section 3.3.2).

2.8 Public Information on the WRNG

Before this work, the most informative public sources on the WRNG were its official MSDN documentation [46] and a description in "Writing Secure Code" [29]. Both provided some insight on the building blocks, and presented Microsoft's claims of high security, but did not specify the PRNG's exact algorithm or its mode of interaction with the operating system.

The WRNG is used by the operating system, by operating system applications such as the Internet Explorer, and by applications written by independent developers. Its output is used for cryptographic and security purposes, including keys, Initial Values, salts, and possibly TCP sequence numbers [46].

According to [29], the WRNG was first introduced in Windows 95 and the same design was used since then in all Windows operating systems, including Windows 2000, CE and XP. That statement was written before Windows Vista was released. The Vista version contains the same WRNG interface, but Microsoft claims its design has changed [6].

According to figures by OneStat and Net Applications[4] [5], as of July 2007 Microsoft Windows XP and 2000 accounted together for 85% - 90% of the market. Therefore the subject of this thesis, the WRNG of Windows 2000 and XP, is the most frequently used pseudo-random number generator, with billions of instances running at any given time.

Unfortunately, our work shows that the Windows pseudo-random number generator has several unnecessary flaws.

[4] http://www.onestat.com/html/aboutus_pressbox54-windows-vista-global-usage-share.html
[5] http://marketshare.hitslink.com/report.aspx?qprid=10&qpmr=24&qpdt=1&qpct=3&qptimeframe=M&qpsp=102

Chapter 3

Related Work

3.1 Overview of RNG Research

The field of random number generation has been the subject of extensive research, and is the topic of a heated discussion to this day. The different aspects of research include mathematical methods for random number generation and evaluation, cryptographic formulations, practical design and implementations, and real-world attacks.

A comprehensive overview of the mathematical methods for random number generation, analysis and evaluation is given by Knuth [37]. Bruce Schneier devotes a chapter to PRNGs in his book Applied Cryptography [54].

Peter Gutmann examines many PRNG designs and implementations, with an emphasis on system entropy sources and a guide to designing and implementing a PRNG without access to special hardware or privileged system services [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 [36]. Additional discussion of PRNGs, as well as new PRNG designs appear in [34, 16]. Issues related to operating system entropy sources were discussed in a recent NIST workshop on random number generation [33, 26].

The US Government introduced standards for random number generation and validation in [17, 18]. The recent NIST Special Publication 800-90 [3] includes the specifications of four different random number generators. According to an article by Bruce Schneier[1], one of them, based on ellyptic curve cryptography, contains a potential backdoor discovered by Dan Shumow and Niels Ferguson[2], and has been the subject of heated debate.

The rest of this chapter discusses real-world attacks that compromised systems through their PRNGs, and designs of current operating system PRNGs.

[1] http://www.wired.com/politics/security/commentary/securitymatters/2007/11/securitymatters_1115
[2] http://rump2007.cr.yp.to/15-shumow.pdf

3.2 RNG Attacks

3.2.1 Netscape SSL

In 1996, Goldberg and Wagner demonstrated an RNG attack on the Netscape SSL [22]. SSL (and its successors TLS and OpenSSL) is a widely accepted network encryption protocol, with specifications available online[3]. In the Netscape browser, SSL was supposed to protect private communications from eavesdropping and impersonation attacks, most importantly - to encrypt passwords and credit card numbers sent over the network. Netscape did not release the specifications of the PRNG, but the authors were able to reverse-engineer it.

Netscape used its own PRNG to generate the encryption keys. The authors examined the PRNG algorithm and found that the PRNG was seeded using a small number of very weak system parameters; in the *strongest* implementation, the inputs for the seed were the system time, the process ID and the parent process ID of the browser. Even though the inputs were mixed using a strong hash function (MD5), an attacker would be able to guess the original values using brute force or several improved approaches.

The authors emphasize the importance of peer review to the development of secure software. The faulty SSL implementation was fixed in the next version of Netscape.

3.2.2 Kerberos 4.0

An attack similar to the one on Netscape was shown in 1997, on the MIT implementation of the Kerberos 4.0 authentication protocol [14]. Kerberos generated session keys used for proof of identity and message authentication, with a PRNG seeded with parameters such as time, process ID, host ID, and counters. As the authors show, these inputs do not have sufficient entropy and can be guessed using brute-force search and precomputation.

3.2.3 Java Virtual Machine

More recently, Zvi Gutterman and Dalia Malkhi have analyzed the RNG used by Tomcat in the Java Servlets mechanism [27]. Since HTTP is stateless, many commercial sites use mechanisms like cookies and URL rewriting to keep a session state at the client side. Stateful sessions are useful for keeping track of shopping baskets, customer preferences, previous transactions and more. To prevent impersonation and session stealing, the server generates a session ID token, represented by a large random number, and encodes it in a cookie or a URL. An impersonator should have difficulty guessing the correct token, because of the large search space. This is only true if the RNG generating that token is strong.

The authors analyzed the Java Virtual Machine PRNG, used by Tomcat servers to generate session ID tokens. Java provides two APIs for random number generation: `java.util.Random`, which is an LCG, and `java.security.SecureRandom`, which is a stronger PRNG with a 160-bit state, and uses SHA-1for transition function. The attack

[3] http://wp.netscape.com/eng/ssl3

is based on both generators being seeded with very few entropy inputs, including the system time in milliseconds and an object hash that could take only one of a few hundred values. By exhaustively trying all options for the seed, and comparing resulting outputs with those of the real generator, an attacker can recover the state of the PRNG and from then on guess all session ID tokens, allowing him to impersonate customers of commercial websites and to hijack their transactions.

The authors estimate the search space size at around 2^{42}, which makes the attack feasible on a home computer. In addition, they present a time-memory tradeoff for similar attacks on RNGs, which uses precomputation to shorten the search time.

3.2.4 Linux RNG

The Linux operating system includes an entropy-based PRNG [59], The exact algorithm used by the Linux RNG (rather than the source code, which is open but complicated) was published in [28]; following that paper's notation, we denote the Linux RNG as LRNG.

The LRNG is used by applications such as TCP, PGP, SSL and more. It is accessed through device nodes `/dev/random` and `/dev/urandom` (blocking and non-blocking, respectively), or as a kernel function. The LRNG receives entropy from the kernel on every keyboard, mouse, and disk event, and on interrupts; the entropy of each event is small, but there should be many. On machines without hard drives and human-interface devices, the entropy inputs are weaker. The LRNG solves the problem of low-entropy initialization after reboot by writing 512 bytes of output to a file before shutdown, and reading it as a seed on startup.

The LRNG has an entropy estimation mechanism, keeping counters attached to its internal states, which increase on reseeding events and decrease when output is produced. When the blocking device node `/dev/random` reaches 0 on its entropy counter, it will block until some reseeding events occur. The definition of entropy for the purposes of this mechanism differs from the one given by Shannon in [56].

The authors demonstrated an attack on the forward security of the generator, with an overhead of 2^{64} in most cases and overhead of 2^{96} in all other cases. In addition, they showed that a denial-of-service attack could be mounted by reading excessive amounts of output, causing the blocking device to block permanently. Other results include a problem with input security, and several programming bugs.

The authors note that open source code allows for easy identification and correction of security issues, but is not synonymous with well-documented or secure systems. The flaws they've pointed out are being corrected at the moment of writing.

3.3 Suggested RNG Designs

3.3.1 Barak-Halevi Construction

The recent work by Barak and Halevi [2] presents rigorous definitions of PRNG security. The authors suggest a simple PRNG construction, and argue that it is provably as secure

as a true RNG under the attack model they define.

The Barak-Halevi construction suggests separating the entropy "extraction" process, which is information-theoretic in nature, from the output generation process. The output generator is a cryptographic PRNG such as in Figure 2.3.2, and the entropy extractor is a function of system entropy inputs, the only demand on it being "sufficient randomness" (that is, if the inputs have high entropy, so should the output). The extractor's output is XOR'ed to the PRNG state.

This construction is much simpler than most existing PRNG designs, yet its security is proven assuming that the underlying building blocks are secure. As we show in this thesis, the Windows RNG construction is much more complex than that of [2], and yet suffers from several weaknesses.

The authors stress that "entropy" in a security sense must be defined from the point of view of the attacker. Therefore, attempts to quantify the entropy of a PRNG's internal state or that of the system inputs (as done in the Linux RNG and Yarrow, described in this chapter), are a mistaken approach; it's unknown which entropy sources, or how much of the internal state, the adversary can compromise and to which extent.

An important dilemma regarding the frequency of entropy refreshes is defined:

> On the one hand, refreshing very often pose the risk of using refresh data that does not have enough entropy, thus allowing an attacker that knows the previous internal state to learn the new state by exhaustively searching through the limited space of possibilities... On the other hand, refreshing the state very infrequently means that it takes longer to recover from a compromise.

3.3.2 Yarrow and Fortuna

The Mac OS, FreeBSD and OpenBSD provide access to PRNGs through the device nodes `/dev/random` and `/dev/urandom`. They are all based on two closely related designs, Yarrow and Fortuna.

The Yarrow PRNG design [34] addresses the attack model and security requriements presented in Section 2.4, and aims to be computationally efficient. Its state consists of a key and a running counter; its output generator is a block cipher, encrypting the counter with the key. Upon producing a small amount of output, the generator replaces the key using the same mechanism, without involving entropy.

Yarrow's entropy collector uses two entropy pools, labeled "fast" and "slow". All entropy inputs are divided between the two. The algorithm keeps entropy estimates that increase as each pool receives input. When the estimate reaches a certain threshold, the pool is hashed and the result refreshes the key. The "fast" pool has a low threshold, and serves to quickly recover the key after a compromise. The "slow" pool has a high threshold, and its purpose is to prevent sustained state exposure attacks by collecting so much entropy that any adversary would be unable to use brute force to guess it.

Yarrow additionally stores a seed file to initialize itself with high entropy after reboot. The seed file is updated on every refresh of the generator's key, and must be stored

securely on the file system.

FreeBSD and OpenBSD use the Yarrow design [50, 13]. Mac OS ports one of the BSD versions[4]; its PRNG source code is available online[5] subject to free registration.

Fortuna [16] is Yarrow with minor modifications. Its design admits that using entropy estimates is useless, because entropy can be measured with respect to the attacker only. Therefore it proposes using 32 pools, the i'th pool being used once in every 2^i refreshes, so that pool 0 is used every time. The authors argue that each pool gathers twice as much entropy as the previous one, therefore for any practical attacker there is an i such that the i'th pool exceeds his brute force capability. This approach assumes that the inputs to the different pools are "reasonably independent"; that assumption is criticized by [2], demonstrating a possible attack based on how the entropy inputs are handled. Otherwise, Fortuna conforms to the requirements of [2].

Fortuna uses a hash function as its transition function, instead of the block cipher in Yarrow. This gives it one-way properties that protect it from a forward security attack. This protection is missing in Yarrow. Its only defense is the entropy refreshing its state.

[4] http://developer.apple.com/documentation/Darwin/Reference/ManPages/man4/random.4.html
[5] http://www.opensource.apple.com/darwinsource/10.5/xnu-1228/bsd/dev/random/

Chapter 4

The Structure of the Generator

This chapter describes the internal structure of the Windows RNG. We did not have access to the source code, and so the description was obtained by reverse-engineering the Windows binaries.

The original WRNG consists of about 10,000 lines of assembly code, which would be impossible to describe fully. This chapter presents the algorithm in pseudocode, omitting details such as unused control paths, error handling, thread synchronization and various extreme cases, and in some cases changing marginal details (such as function prototypes) to better convey the meaning.

The version analyzed in this work was Windows 2000 Service Pack 4, version `5.00.2195`. The Windows XP version was examined more superficially; our analysis showed it had the same structure as the Windows 2000 RNG, including the same main loop, but we did not validate it with simulation tools. According to a statement from Microsoft [6], the Windows XP RNG has the same design as in Windows 2000. Windows Vista was unreleased at the time of this research.

4.1 Using the WRNG

4.1.1 Windows Crypto API

The Microsoft Windows RNG is used through functions in the Crypto API. The Windows RNG is not accessible through a device node, as in Linux, Mac OS and the BSD systems.

The Microsoft Windows operating system provides the Cryptographic Application Programming Interface (Crypto API) - a set of interfaces covering the most commonly needed cryptographic functions. The Crypto API documentation dictates the use of the function `CryptGenRandom` for random number generation. `CryptGenRandom` can generate cryptographic keys directly, as buffers of random data. Another Crypto API function, `CryptGenKey`, generates keys specialaized for different cryptographic algorithms, but uses the same PRNG implementation internally [45, 44].

The Crypto API allows choosing a Cryptographic Service Provider (CSP), the software library implementing the Crypto API functions. Some CSPs are installed with the

Figure 4.1: The CryptGenRandom API

```
BOOL WINAPI CryptGenRandom(
    HCRYPTPROV hProv,
    DWORD dwLen,
    BYTE* pbBuffer
);
```

Figure 4.2: Sample code using CryptGenRandom

```
#include <windows.h>
#include <wincrypt.h>
#define SIZE 160
void main() {
    HCRYPTPROV hProv = 0;
    BYTE data[SIZE];
    CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0);
    CryptGenRandom(hProv, SIZE, data);
}
```

Windows system, and others can be imported or created by the user. In this work, we analysed the `CryptGenRandom` implementation found in the Microsoft Strong Cryptographic Provider [48], which the default provider for strong cryptography installed with every Windows system, and used by the Internet Explorer.

4.1.2 The CryptGenRandom API

As shown in Figure 4.1, `CryptGenRandom` receives three parameters: an output buffer, length, and a handle to a CSP [46]. An example program using `CryptGenRandom` is shown in Figure 4.2.

To call `CryptGenRandom`, the Windows SDK (Software Developer's Kit) headers and binaries are needed. The WinSDK is available at the Microsoft downloads site[1]. An explanation on the headers, binaries and linkage needed to compile can be found on MSDN [46]. For convenience, a sample program with a Makefile is provided online[2].

The function does not receive a user-supplied seed to reproduce a sequence of outputs, and is always supposed to generate unpredictable random data.

[1] http://www.microsoft.com/downloads
[2] http://www.cs.huji.ac.il/~dorrel/wrng/test.zip

Figure 4.3: The main loop of the WRNG

```
1  CryptGenRandom(Buffer, Len) //output Len bytes to Buffer
2      while (Len>0) {
3          R := R ⊕ get_next_20_rc4_bytes()
4          State := State ⊕ R
5          T := SHA-1'(State)
6          Buffer := Buffer ‖ T
7          R[0..4] := T[0..4] //copy 5 least significant bytes
8          State := State + R + 1
9          Len := Len − 20
10     }
```

4.2 Internal Structure

The Windows RNG code is logically divided into three componenets:

Main loop The WRNG hashes part of its state with SHA-1, producing 20 bytes of output in a loop. It updates its state using several mathematical operations, and with output from the internal generator.

Internal generator The internal generator uses several instances of the RC4 stream cipher (see Section 4.3.2). On initialization or after generating a predefined amount of output, it invokes the entropy collector to create new RC4 instances.

Entropy collector The entropy collector gathers several thousand bytes of system data, applies complex cryptographic transformations to mix it, and uses the result to re-key the RC4 instances for the internal generator.

4.2.1 Main Loop

The main loop is presented in Figure 4.3. The generator's components updated on each iteration of this loop, and used to calculate the output, are the two registers, R and State. All operations use 20-byte widths: the registers R, State and T, and the return values of the functions, are all 20 bytes long.

The registers R and State are never initialized; this is discussed in Section 6.1. The internal generator, shown as the function `get_next_20_rc4_bytes`, XORs R with its output. The function SHA-1', generating output as a function of State, is a variant of SHA-1 with the Initialization Vector (IV) reversed[3]. The `Buffer` is treated as empty, and the SHA-1' output is concatenated to it. Five bytes of the result are copied back into R, then R is added with carry to State.

[3]This is apparently done in accordance with FIPS-186 [17], as dicussed in Section 6.4. The original SHA-1 is discussed in Section 4.3.1.

Figure 4.4: The internal generator

```
get_next_20_rc4_bytes() // return 20 bytes from internal generator
    while (RC4[i].accumulator >= 16384) {
        //has the i'th instance produced 16KB of output?
        RC4[i].rekey() //refresh with system entropy
        i = (i+1) % 8
    }
    result = RC4[i].process(20) // get 20 bytes from i'th instance
    RC4[i].accumulator += 20
    i = (i+1) % 8
    return(result)
```

4.2.2 Internal Generator

The internal generator, denoted `get_next_20_rc4_bytes`, is described in Figure 4.4. It uses eight instances of the RC4 cipher (see Section 4.3.2). On each invocation it chooses one instance round-robin[4] to produce 20 bytes of output, which is XOR'ed to R in the main loop.

Each instance keeps an `accumulator` variable recording the number of bytes it produces. When the `accumulator` exceeds a fixed threshold of 16384 bytes (16 KB), the entropy collector is invoked and creates a new key for the RC4 instance (therefore the entropy refreshment process is also called re-keying). On the first invocation, the enrtopy collector is used in the same manner to initialize all eight RC4 instances.

Because the generator is always asked for 20 bytes at a time, the accumulators of all eight RC4 instances advance together in synchrony. As a result, all eight RC4 instances re-key at nearly the same time, as discussed in Section 5.2. Note that between the re-keyings, the whole WRNG is completely deterministic.

4.2.3 Entropy Collector

This module collects system entropy, and distils it into keys used to re-initialize RC4 instances in the internal generator of the previous section. It also manages an internal state of its own named `CircularHash`, and saves a random seed in the registry. Its pseudocode is described in Figure 4.5.

The module collects a variety of system data, summarized in Table 4.1, into one large gathering buffer. Although the buffer size is 3584 bytes, debugger runs show that the actual amount of data collected can differ, and reaches about 1600 bytes on a Windows 2000 system. The first 256 bytes are taken from the module's own internal state named `CircularHash`, described in Section 4.3.5. The rest of the data comes from various query functions, from simple ones such as `GetLocalTime` and `GetUserName`, to the more specialized ones such as `NtQuerySystemInformation`, which gets advanced run-time

[4]Shown in the figure by the index `i`, which is used as a static variable initialized to zero.

Figure 4.5: Entropy collection and re-keying

```
RefreshState(rc4instance) //re-key an RC4 instance
    byte[0xE00] buf; //a 3584-byte buffer
    buf := CircularHash | system_entropy
    OldSeed := Seed
    Seed := VeryLargeHash(buf, OldSeed)
    //RC4-encrypt the new Seed with the old Seed as key
    RC4(Seed, OldSeed)
    //call a kernel driver and get 256 bytes
    k := KSecDD(CircularHash)
    //encrypt the result with the new Seed as key
    RC4(k, Seed)
    //init the RC4 instance with the resulting key
    RC4Init(rc4instance, k)
    rc4instance.accumulator := 0
```

statistics from the operating system. Documentation on the individual OS functions is available in [29] and on MSDN.

The collected data passes through the mixing function `VeryLargeHash`, along with a registry variable named `Seed`. `VeryLargeHash` is described in detail in Section 4.3.3. It is designed to ensure that the change of a single input bit affects all output bits.

The registry variable `Seed`[5] is an 80 byte buffer. Modifications to its state are written back to the registry and saved to the hard drive. As a result, the WRNG can initialize itself on system startup with a a function of its former state, and avoids relying exclusively on system entropy (which might be weak after start-up as discussed in Section 2.6). However, the security of this registry key is nonexistent: it can be viewed and modified by any user.

The output of `VeryLargeHash` is written back to `Seed`, and encrypted using RC4 with the old value of `Seed` as key.

The device driver `KSecDD` is invoked with the `CircularHash` as seed, and produces a 256-byte output, which is encrypted using RC4 using the new `Seed` as key. The result re-keys an RC4 instance belonging to the internal generator of previous section, using the RC4 KSA algorithm. See details on the sub-components in Section 4.3.

[5]Full path: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\RNG\Seed`.

Table 4.1: Entropy sources

Source	Bytes requested
CircularHash	256
KSecDD	256
GetCurrentProcessID()	8
GetCurrentThreadID()	8
GetTickCount()	8
GetLocalTime()	16
QueryPerformanceCounter()	24
GlobalMemoryStatus()	16
GetDiskFreeSpace()	40
GetComputerName()	16
GetUserName()	257
GetCursorPos()	8
GetMessageTime()	16
NTQuerySystemInformation calls	
ProcessorTimes	48
Performance	312
Exception	16
Lookaside	32
ProcessorStatistics	up to the remaining length
ProcessesAndThreads	up to the remaining length

4.3 Sub-Components

4.3.1 SHA-1 Hash Function

The SHA-1 hash function [1, 51] is widely used for security purposes. Recent research [60] found weaknesses in its collision-resistance, allowing to find x_1 and x_2 such that SHA-1(x_1) = SHA-1(x_2) more effectively than the function should permit. Nevertheless, there has been no progress on *inverting* SHA-1(that is, given y, finding x so that SHA-1$(x) = y$). Effectively, SHA-1 is a strong one-way function for any practical purpose. Its output y can be given to an attacker without yielding any information on the input x, except for letting the attacker validate his guess of x.

4.3.2 RC4 Stream Cipher

RC4 is a stream cipher designed by Ron Rivest for RSA Security in 1987[6]. Its popularity is due to the small code size, making it easy to implement, and high efficiency. Its state S is a permutation of the values $1 \ldots 256$, with two pointers i and j. Its initialization algorithm is shown in Figure 4.6, and its output generation algorithm is shown in

[6] http://www.rsa.com/rsalabs/node.asp?id=2250

Figure 4.6: RC4 Key Scheduling Algorithm (KSA)

```
RC4_KSA(S, key, keylength)
    for i from 0 to 255
        S[i] := i
    j := 0
    for i from 0 to 255 {
        j := (j + S[i] + key[i mod keylength]) mod 256
        swap(S[i],S[j])
    }
```

Figure 4.7: RC4 output generation

```
RC4_process(S)
    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 4.7. The output is XOR'ed with the cleartext for encryption.

The security of RC4 has been the subject of many studies [19, 39, 20]. The research found statistical deviations, weak key families, vulnerabilities to related key attacks, and some leaks of key information in the first 256 bytes of output. RC4 is still safe to use subject to precautions, like discarding the first output bytes. The WRNG does not take these precautions. We note that most of the RC4 research came after the release of Windows 2000, but before Windows XP, and its WRNG could have been updated.

Most importantly, RC4 has no forward security: given its state, it's possible to compute all previous states (and therefore outputs). It can be run backwards as shown in Figure 4.8. Its use as the transition function of a PRNG has disastrous consequences, described in Chapter 7.

Figure 4.8: Running RC4 in reverse

```
RC4_reverse(S)
    while GeneratingOutput:
        output S[(S[i] + S[j]) mod 256]
        swap(S[i],S[j])
        j := (j - S[i]) mod 256
        i := (i - 1) mod 256
```

Figure 4.9: VeryLargeHash algorithm

```
VeryLargeHash(Buf, Len, Seed)
  k := Len / 4
  digest1:=SHA1(Seed[00..20]‖Buf[ 0..k ]‖Seed[20..40]‖Buf[ k..2k])
  digest2:=SHA1(Seed[20..40]‖Buf[ k..2k]‖Seed[00..20]‖Buf[ 0..k ])
  digest3:=SHA1(Seed[40..60]‖Buf[2k..3k]‖Seed[60..80]‖Buf[3k..4k])
  digest4:=SHA1(Seed[60..80]‖Buf[3k..4k]‖Seed[40..60]‖Buf[2k..3k])
  result[00..20] := SHA1(digest1 ‖ digest3)
  result[20..40] := SHA1(digest2 ‖ digest4)
  result[40..80] := SHA1(digest3 ‖ digest1)
  result[60..80] := SHA1(digest4 ‖ digest2)
  return result
```

4.3.3 VeryLargeHash Function

The `VeryLargeHash` function consists of a series of SHA-1 operations, performed on an input buffer of any given length, and a fixed-length 80-byte argument. Its code is described in Figure 4.9. It's used in the entropy collector to update the `Seed`, which is always the 2nd argument, with the contents of the entropy gathering buffer. Note that the change of a single bit of input can affect every bit of the output, assuming the same is correct for SHA-1.

4.3.4 Kernel Security Device Driver (KSecDD)

The `KSecDD` device driver is used by the WRNG entropy collector. The entropy collector passes the contents of the `CircularHash` as the input (see Section 4.3.5), and receives 256 bytes of output from an RNG implemented by one of the `KSecDD`'s methods.

The device driver has no official documentation, but is mentioned in MSDN articles on the Windows CNG (Cryptography Next Generation) API [47] and the and Encrypting File System [49], where it is described as the Microsoft "kernel security support provider interface".

`KSecDD` exports numerous functions through the driver method interface. The WRNG entropy collector makes a request to one of thes methods; reverse-engineering shows that the function `NewGenRandomEx` handles this request (see Section 9.5). This function is quite similar to the WRNG internal generator `get_next_20_rc4_bytes` itself, sharing the design and most of code. A notable similarity is the equal refreshment threshold for the individual RC4 instances. The differences are listed below:

- The kernel component does not work in 20-byte chunks. If asked for a larger output, it will try to produce it using a single RC4 instance, as long as its accumulator allows. It will switch to another instance, in round-robin order, on the next invocation.

- There are only 4 RC4 instances, compared to 8 in the user-mode generator.
- The entropy sources used are kernel APIs and CPU registers.

The kernel-level entropy sources consist of calls to `ZwQuerySystemInformaion` and access to some CPU registers.

`ZwQuerySystemInformaion` is undocumented, but its parameters and mode of use in `NewGenRandomEx` are exactly identical to those of `NTQuerySystemInformaion` in the user-mode entropy collector. APIs with identical names but different prefixes (Zw/Nt/Rtl etc.) are commonly used in the Windows architecture to provide the same functionality at different levels.

Several CPU registers are read but only the `TSC` (Time Stamp Counter) register is used as an entropy source. This register contains the exact number of CPU clock cycles since reset.

4.3.5 CircularHash Component

The entropy collector keeps an internal state component called the `CircularHash`. When gathering entropy, it copies the `CircularHash` contents directly into the gathering buffer. The `CircularHash` is a 256-byte memory area, with some associated settings and functions. It is initialized on the first call to the WRNG by clearing its memory.

On every call to `get_next_20_rc4_bytes`, the function `UpdateCircularHash` is run. It takes a buffer argument, hashes it using the MD4 hash function (an unused alternative setting defines SHA-1 as the hash), and XORs the result to the memory area at the current index. The current index is then advanced by 7 bytes. All indices are taken modulo 256, which explains the name `CircularHash`.

The `CircularHash` variable is updated in a simple and predictable way: it is always passed a null argument, therefore its evolution consists of repeatedly XOR'ing a fixed value (the MD4 hash of the empty string) to a circular buffer.

The `CircularHash` supports several features unused by the WRNG:

- It could be seeded with external inputs
- It could hash part of itself along with the external input
- It could allow saving its state within the Crypto Service Provider
- It could use SHA-1instead of the outdated MD4

To summarize, it appears that the `CircularHash` could be a full-fledged PRNG, especially in combination with the entropy collector. The reason why it is used so weakly, and yet not taken out altogether, is unknown.

Chapter 5

Interaction with the System

We describe here how the WRNG interacts with the operating system, and how that affects its security.

5.1 Scope

This section explains the scope of a WRNG instance.

The WRNG runs in user mode, loading its code and data into the address space of the calling process. The only exception is the `KSecDD` component, which is in kernel space, but is only used in the entropy collection (see Section 4.3.4).

All variables directly affecting the WRNG output reside either on stack (State, R) or in the DLL area (RC4 instances). Since the DLL is loaded into the process address space, there is one set of static variables per process. Inside the process, a call to `CryptGenRandom` creates stack frames on the user's stack. There can be a set of stack variables per thread, assuming that different threads invoke `CryptGenRandom`. All variables that are shared on the machine (`KSecDD`, `Seed`) participate only during entropy collection. In effect, each thread can have a separate WRNG state, but all threads within one process consume random bytes from the same RC4 instances.

Table 5.1 summarizes the scopes and lifetimes of different variable in WRNG code. These correspond to the standard scopes in C and Windows:

Static variables Allocated in fixed offsets in the DLL's data segment. Their initial values are defined in the DLL's binary. Important static variables are the eight RC4 instances, and the `CircularHash` structure (see Section 4.3.5).

Registry variables WRNG uses one registry variable, the `Seed`, an 80-byte buffer accessible for reading and writing to all processes and users. CryptGenRandom makes any changes to the `Seed` permanent by setting a flag that causes it to be saved to the hard drive. See Section 4.2.3 for details.

Kernel variables The `KSecDD` serves as an external source of entropy in the entropy collector, which invokes an PRNG method of this driver. `KSecDD` keeps its own

Table 5.1: CryptGenRandom component scopes

Type	Location	Scope	Lifetime
Volatile	Process stack	single user single process	within one call to CryptGenRandom (from call to return)
Static	Process memory space	single user single process	from first call to CryptGenRandom until the termination of the process
Kernel	Kernel memory space	all users all processes	from boot-up to shutdown
Registry	System registry and optionally, hard drive	all users all processes	permanent

state in in kernel space, therefore it's shared between all the users and processes on a machine. See Section 4.3.4 for details.

Volatile variables All other variables in the WRNG code are volatile local variables, residing on stack.

The effect of the WRNG's scope is both good and bad. It separates between two processes: for example, Microsoft Word and Internet Explorer will have different WRNG instances. Therefore breaking an instance of WRNG by compromising its state, does not affect other applications. On the downside, there is only process consuming random bytes from each WRNG instance. With the very long period between re-keyings, as descrubed in 5.2, it's likely that the WRNG state will be refreshed very rarely.

The WRNG runs multiple instances in user space. An alternative would be letting the system run a single generator in the protected kernel space, using it to provide output to all applications. A kernel-based implementation would have kept the WRNG's internal state safe and hidden, whereas a user mode implementation gives each process access to the state of its own WRNG instance. An adversary that wishes to learn the state of a certain application, needs only break into the address space of that specific application. This increases the risk of an attacker learning the state of the WRNG, and consequently applying forward and backward security attacks, as described in Chapter 7.

5.2 Synchronous Entropy Collection

This section discusses the implications of how the WRNG invokes its entropy collector. See Section 4.2.2 for the pseudocode.

Entropy collection is synchronous: it only happens when the WRNG is invoked. Even then, it happens only when an RC4 instance reaches its 16 kilobyte threshold, or on initialization. No other events can trigger the entropy collector, or update the generator's internal state. During the long periods between re-keyings, some entropy generated by system events accumulates in the OS and CPU statistics, used by the WRNG as entropy sources, but the rest of it is ignored by the WRNG and goes to

waste. On the other hand, when the WRNG does collect entropy, it does so in the thousands of bytes, destroying any chance of sustained exposure attack (described in Section 2.4).

There is a weakness in the WRNG mode of operation: all 8 RC4 instances re-key simultaneously. Since all instances' accumulators are cleared on initialization, and advance 20 bytes at a time, their values always stay within 20 bytes of each other. As a result, all eight accumulators arrive at the threshold nearly at the same time, possibly within the same call to CryptGenRandom. Consequently, the entropy collector is invoked eight times at close intervals. Any system inputs collected for such close time intervals are bound to share a lot of information. A more reasonable implementation would use the entropy collector once to re-key all eight RC4 instances.

Between refreshes the operation of the WRNG is deterministic. Because separate processes have separate WRNG instances, if a process does not request lots of random bytes, its the WRNG instance used by this state will be refreshed very rarely, even if other processes consume lots of random bytes from their instances of WRNG.

As a result, the WRNG produces almost $8 * 16384$ bytes, or 128 kilobytes of output, without refreshing its internal state with any system entropy. This has severe consequences, extending compromise achieved by any attack on the WRNG (see Chapter 7).

5.3 Invocation by the System

We examined the usage of WRNG by Internet Explorer (IE), which might be the most security-sensitive application run by most users[1]. The examination was conducted by hooking all calls to `CryptGenRandom` using a kernel debugger, and recording the calling process and the number of bytes requested.

Internet Explorer calls the WRNG indirectly through `LSASS.EXE`, the system security service. Other requests to the WRNG came from `SVCHOST.EXE`, in apparent connection with TCP traffic. It's possible that `SVCHOST.EXE` uses the WRNG to generate random sequence numbers for TCP.

During SSL initialization, there is a varying number of requests (typically, four or more) for random bytes. Each request asks for 8, 16 or 28 bytes at a time. No random bytes are requested after the initialization is complete. We can therefore estimate that each SSL connection consumes no more than 100-200 bytes of WRNG output. This means that Internet Explorer's instance of the WRNG will refresh its state only after handling about 650-1300 different SSL connections. It is hard to imagine normal users running 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 Chapter 7 can essentially learn encryption keys used in all previous and future SSL sessions on the attacked PC.

[1] All experiments were applied to IE 6, version 6.0.2800.1106.

Chapter 6

Analysis of the Design

This chapter discusses several details of the WRNG implementation, including some of its design choices and claims of security.

6.1 Uninitialized Variables

The variables R and State participate in the main loop of the WRNG, and affect the generated output, but are never explicitly initialized. Both are allocated on stack, and like any other stack variable which is not initialized explicitly, R and State obtain the last values that were stored in the same address, whether in a previous stack frame or just in unused memory. With the Intel stack architecture, the previous stack frame could contain volatile variables, return addresses and function arguments [7], none of which are usually random or unpredictable.

Therefore the initial values of R and State may vary, but they are not derived from a random source. They only depend on the control flow of the code calling the WRNG. An attacker interested in the values of R and State should be able to narrow down the options, and then use exhaustive search. This is especially true if the attacker can simulate the target application. Using the same exact binary in a debugger, he could find out the *exact* values of R and State used by the target. This applies for example to the Internet Explorer, which is widely available in binary form.

We performed experiments to examine the initial values of R and State when the WRNG is invoked by Internet Explorer. We recorded the values and the stack addresses they were mapped to, and checked whether the addresses were the same, and what was the Hamming distance between the values in different invocations. In all experiments, R and State could be mapped to different locations in the stack, but their values were still highly correlated. The results are as follows:

- In the first experiment one IE instance was started after rebooting the system, 20 times. R and State were mapped to different locations in the stack, but their initial values were correlated, belonged to a small set of several options.

- In the second experiment IE was restarted 20 times without rebooting the system. All invocations had the same initial values of R and State.

- In the third experiment we ran 20 sessions of IE in parallel. With one exception, the initial values were within a Hamming distance of 10 bits from each other.

The experiments confirm that R and State are highly predictable.

R and State are maintained on the stack. Unlike the RC4 instances, these variables are not kept allocated between invocations of the WRNG, and may be overwritten by other functions' stack frames. If R and State 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 was on the stack.

We do not know whether this "loose" management of state variables was intentional or the result of oversight. In the attacks we describe in Chapter 7 we show how to compute previous values of R and State, assuming that they retain their state between invocations of the generator. These attacks are relevant even given the behavior we describe above, for two reasons:

1. The attacker can continue with the attack until he notices that he does not reproduce the correct WRNG output anymore. The attacker should then enumerate over the most likely values of R and State until he can continue the attack.

2. Other applications might use the WRNG in such a way that the stack locations in which the values of R and State are stored are never overwritten. This is likely to happen because of their considerable depth on stack.

6.2 Complexity

It appears that the WRNG design takes steps to make output generation efficient, such as choosing RC4 (which is very fast) for output generation. In fact, the output generation is quite efficient, using only RC4, SHA-1, and buffer operations such as memory copying, XOR, and addition.

Another design choice was to disable a certain control path. If enabled, it would skip the internal RC4-based generator and provide random bytes directly from the entropy collector, for processing in the main loop. It was likely disabled because of the high computational complexity of the entropy colletor, which would be incurred on every call to the WRNG. Another possibly beneficial but disabled component is the `CircularHash` (see Section 4.3.5), which is only used rudimentally.

These choices are more to the disadvantage of the WRNG: the use of RC4 opens the WRNG to a forward security break-in, and the long interval between re-keying extends the effect of possible attacks.

It should be noted that the overall design of the WRNG is highly complex. It uses two output generators, one processing the output of the other. Its entropy collector

invokes hash functions (SHA-1 and MD4) dozens of times, and uses temporary RC4 instances which are initialized and discarded on the fly several times (see Chapter 4). This complexity makes the WRNG computationally expensive, making it difficult to ensure security by design, and easier for an attacker to find bugs.

6.3 Misleading Documentation

The `CryptGenRandom` documentation on MSDN [46] makes some inaccurate statements:

> With Microsoft CSPs, CryptGenRandom uses the same random number generator used by other security components. This allows numerous processes to contribute to a system-wide seed...
> If an application has access to a good random source, it can fill the pbBuffer buffer with some random data before calling CryptGenRandom. The CSP then uses this data to further randomize its internal seed.

Both statements are wrong. Different processes have separate WRNG instances and do not contribute to each other's entropy (see Section 5.1). As Section 4.2.1 shows, the input buffer (`pbBuffer`) cannot serve as seed. `CryptGenRandom` ignores its input buffer, simply overwriting its contents. This can be validated using a simulator of WRNG.

6.4 FIPS-186 Conformance

According to [29], the WRNG is based on Appendix 3 to the NIST Digital Signature Standard (DSS), also known as FIPS 186 [17]. The authors of [29] explain that the WRNG is based on the DSS design, with system entropy replacing user input.

Attempts to conform to FIPS 186 are visible in the code. As the reverse-engineering shows, the WRNG main loop is named `FIPS186Gen`. The definitions in FIPS 186 also explain the use of SHA-1 with reversed initial value.

Although the WRNG main loop bears superficial resemblance to the FIPS 186 algorithm, on closer examination it differs from FIPS 186 in every operation or line of pseudocode. Therefore Microsoft should not cite conformance to FIPS 186 as a basis for claims of the generator's security.

6.5 Pseudo-Randomness

Despite any problems in the WRNG design, its output quality passes the most stringent tests. According to Microsoft, `CryptGenRandom` has received the US Government certification by the FIPS 140-1 standard [10, 29]. We did not validate these claims: the WRNG uses SHA-1 to process its output, and SHA-1 itself passes the pseudorandomness tests of FIPS-140[18, 51]. It's unclear whether the WRNG's internal design had any role in passing these tests.

Chapter 7

Cryptanalysis and Attacks

This chapter demonstrates attacks on the backward and forward security of the WRNG. We show how an adversary can obtain the internal state of the WRNG (the values of the variables R and State and the eight RC4 instances), and afterwards to compute past and future states and outputs of the generator. Computing future states is simple, as is computing past states if the adversary knows the initial values of the variables State and R. We also show two attacks which recover those values, with computational overhead of 2^{40} and 2^{23}, respectively.

The attacks we describe require an adversary to learn 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, malicious code running in an application can learn the WRNG state without violating any security privileges.

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, or by obtaining administrative privileges, thus exposing the state of the generators run by all processes. After learning the state the attacker does not need to maintain the connection with the victim system. He can use the attack tools offline to learn all previous and future outputs of the generator, and subsequently, learn cryptographic tokens (e.g. SSL keys) used by the victim. This attack is more powerful and more efficient than known attacks, which require the attacker to control the victim machine at the time it is generating cryptographic keys, to observe these keys, and to relay them to the attacker. This is especially true with respect to keys generated before the attacker obtained access to the machine, as our attacks allow breaking into a machine *after* it was used by the victim.

7.1 State Extraction Attack

This section describes how an attacker can obtain a snapshot of WRNG's internal states.

As described in Chapter 4, the following components are relevant for output generation: the eight RC4 instances in DLL space and the buffers R and State on stack.

The RC4 instances are static variables in a DLL. The Windows Portable Executable format allocates DLL addresses at compile time, to save on code remapping [11]. Therefore the RC4 instances have a permanent address that can be found statically[1].

After a call to WRNG completes, its internal state remains in memory. An invocation of WRNG is naturally placed deeper on stack than the calling code. The WRNG makes no effort to clean up the stack variables, so their values linger. An observer with access to the code can compute the stack offset which his code needs to use to obtain R and State[2].

An attacker can compute the offsets, access them by pointers, and copy the values of all eight RC4 states and the stack buffers into his own variables. He can then use these values for simulation, write them to a file, or transmit them over the network, etc. He could also overwrite the WRNG state in memory with another state, effectively "planting" WRNG outputs to be obtained by the victim.

We stress that this attack can only be carried out after another attack gains access to the process memory space.

7.2 Attack on Backward Security

Suppose that an adversary learns the state of the WRNG at a specific time. The next states and outputs are a deterministic function of this data. The adversary can therefore simulate the generator's next states and outputs, by following the generator's algorithm. From the attacker's point of view, he is obtaining future outputs from a current PRNG state, whereas from the victim's point of view, his current outputs are being exposed by a past break-in, hence the name "Backward Security".

The attacker first creates a snapshot of the internal state using the state extraction attack, then simulates the WRNG. There are two alternatives for the simulation. First, the attacker can simply request the real WRNG to produce all output for the length remaining until the next re-keying. Having obtained the output, the attacker restores the original WRNG state from the snapshot he made. The victim will then receive the same outputs as the attacker. Alternatively, the attacker can implement a simulator of the WRNG on any platform or programming language, according to the description in Figures 4.3 and 4.4, and obtain the same outputs as the victim of the attack, offline. We built such a simulator to validate the reverse-engineering and these attacks.

[1] For example, on our Windows 2000 system, the array holding RC4 instances is always at address `0x7CA1FDFC`.

[2] On our system, a function called right after `CryptGenRandom` would find R 140 bytes under its own stack frame, and State directly under R, 160 bytes deep.

This attack compromises all WRNG output up to the next re-keying. When the WRNG refreshes itself with entropy, its RC4 instances are wiped out and replaced completely. Even though the attacker still knows R and State, we see no practical way to predict all the system entropy sources, and keep track of the RC4 state. The attacker can find out the refreshment time, by checking the accumulator fields of the RC4 instances.

The attack on backward security applies to any PRNG. Given the algorithm and the internal state, it's a easy to simulate a deterministic output generator.

Only the entropy refreshing the PRNG's state determines when the PRNG will recover from that attack. What makes it bad in the WRNG is the 128-kilobyte interval between re-keyings (as noted in Section 5.2).

7.3 Attacks on Forward Security

An adversary that learns the state of a PRNG at a specific time, and finds a way to reverse its state transition function, can compute its previous states, therefore compromising its past outputs. From the victim's point of view, his current outputs are exposed due to a *future* break-in, hence the name "Forward Security".

The attacks we demonstrate come in order of increased sophistication: the first attack is easy, but assumes that the stack variables R and State are recovered by some other means. The second attack is a method for recovering past values of R and State, at a cost of 2^{40}. The third attack improves over the second by doing the same at an average cost of 2^{23}, but has a more complex implementation.

Denoting the time of the sattack as t, the last two attacks provide a list of candidates for the generator state at time $t-1$. They should be applied repeatedly, revealing the states (and consequently the outputs) of the generator at times $t-1, t-2, ...$ Each application corresponds to one iteration of the main loop of the WRNG.

Since the transition function can have collisions, several possible states may precede any given state. Therefore at each application of the last two attacks, there can be a small number of false positives, as shown below. As each result must be investigated, the search path branches, becoming a tree.

The number of false positives behaves the same as in the attack on the forward security of the Linux RNG (see [28], Appendix C). It was shown there 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 every false positive for the value of the state at time $t-k$, it happens with constant probability that the search procedures do not find further candidates at time $t-k-1$. In this case the search path is cut off, and produces no more overhead.

If the attacker knows even a partial output of the generator at some previous time $t-k$, he can use this knowledge to identify the true state of the generator at that time, and remove all false positives.

7.3.1 Attack I: Inverting RC4

The WRNG depends on RC4 for generating streams of pseudo-random output, processing that output in its main loop before passing it to the user. RC4 does not have any forward security (see Section 4.3.2) and its transition function is easily reversed; given its current state, it's easy to compute its previous states and outputs as shown in Figure 4.8.

An instant attack is possible when the initial values of R and State are known to the adversary. As argued in Section 6.1, this is a reasonable assumption. Extracting the states of the RC4 instances using the state exposure attack, the adversary can compute all of the previous RC4 states, up to the last re-keying when they were initialized. As in the attack on backward security, the accumulator fields in the RC4 instances indicate exactly how long ago they were initialized.

After obtaining the values of the RC4 instances and R and State at some point in the past, this attack reduces to simulating the WRNG from that point exactly as in the attack on backward security.

7.3.2 Attack II: Recovering Stack Components

The previous attack depends on the attacker somehow knowing earlier values of R and State. As argued in Section 6.1, these values are assumably easy to obtain. In this section, we show how to eliminate this assumption and recover the exact values of R and State, at the cost of 2^{40} incurred by brute-force search.

For intuition, notice that every line of WRNG's main loop is an assignment. It can be seen as an equation, in which a variable's new value is defined in terms of its old value and other operands. Switching the sides of this eqation, we obtain a definition of the former value (which is our goal) in terms of the new value, which we know.

To distinguish between values the same variables at different times, we introduce the following notation:

- Denote by R^t and S^t the values of R and State just *before* the start of the tth iteration of the main loop, as it is described in Figure 4.3)
- Denote by $R^{t,i}, S^{t,i}$ the values just before the execution of the ith line of code in the tth iteration of the main loop (namely, $R^t = R^{t,3}, S^t = S^{t,4}$).
- Let RC^t denote the output of `get_next_20_rc4_bytes` in the tth iteration.
- Each of the values above is 160 bits long. Let us also denote by X_L the leftmost (most significant) 120 bits of variable X, and by X_R its 40 rightmost (least significant) bits.

Given R^t and S^t, our goal is to compute R^{t-1}, S^{t-1}. Given R^{t-1} and S^t, computing S^{t-1} is trivial as shown below, so we will concentrate on finding R^{t-1}.

We assume we're given the states of all eight RC4 instances, 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:

$$\begin{aligned} S^{t-1,8} &= S^t - R^t - 1 \\ R^{t-1,7} &= R_L^t \parallel R_R^{t-1} \qquad \text{(where } \mathrm{R_R^{t-1}} \text{ is a 40-bit unknown)} \\ R^{t-1} &= R^{t-1,7} \oplus RC^{t-1} \\ S^{t-1} &= S^{t-1,5} = S^{t-1,8} \oplus R^{t-1,7} = \underbrace{(S^t - R^t - 1)}_{S^{t-1,8}} \oplus \underbrace{(R_L^t \parallel R_R^{t-1,7})}_{R^{t-1,7}} \end{aligned}$$

We also observe the following relation:

$$R_R^t = \mathsf{SHA\text{-}1}`(S^{t-1,8})_R = \mathsf{SHA\text{-}1}`(S^t - R^t - 1)_R$$

These relations define R_L^{t-1} and S_L^{t-1}, but they do not reveal the rightmost 40 bits of these variables (namely R_R^{t-1} and S_R^{t-1}), 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_R^{t-1}, and use it to compute R_R^{t-1} (then, S_R^{t-1} can easily be computed).

$$\begin{aligned} R_R^{t-1} &= \mathsf{SHA\text{-}1}`(S^{t-2,8})_R \\ &= \mathsf{SHA\text{-}1}`(S^{t-1} - R^{t-1} - 1)_R \\ &= \mathsf{SHA\text{-}1}`(\underbrace{(S^t - R^t - 1) \oplus (R_L^t \parallel R_R^{t-1,7})}_{S^{t-1}} - \underbrace{(R_L^t \parallel R_R^{t-1,7}) \oplus RC^{t-1}}_{R^{t-1}}) - 1)_R \end{aligned}$$

Note also that $R_R^{t-1,7} = R_R^{t-1} \oplus RC_R^{t-1}$. 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).

7.3.3 Attack III: Improved Search

This section improves on the attack in the previous section, reducing the overhead to 2^{23} instead of 2^{40}. The improvement is based on observing that some of the bits in the sought value, R_R^{t-1}, are cancelled out and therefore irrelevant in the brute-force search.

Note that $R^{t-1,7} = R^{t-1} \oplus RC^{t-1}$. Therefore we obtain the following equation:

$$R_R^{t-1} = \mathsf{SHA\text{-}1}`(S^{t-2,8})_R = \mathsf{SHA\text{-}1}`(S^{t-1} - R^{t-1} - 1)_R$$

Note also that

$$S^{t-1} = \underbrace{(S^t - R^t - 1)}_{S^{t-1,8}} \oplus \underbrace{RC^{t-1} \oplus R^{t-1}}_{R^{t-1,7}}$$

Figure 7.1: The 2^{23} attack on forward security

```
1  FindPreviousR(R^t, S^t, RC^{t-1}) //return all candidates for R^{t-1}
2      results := empty set
3      Z := (S^t − R^t − 1) ⊕ RC^{t-1}
4      n = HammingWeight(Z_R)
5      R_L^{t-1} := (R^t ⊕ RC^{t-1})_L
6      q := ¬Z_R
7      for i = 0...2^n {
8          R_R^{t-1} := q
9          R^{t-1} := R_L^{t-1} ‖ R_R^{t-1}
10         candidate = SHA-1‘(Z − 2 · (R^{t-1} ∧ Z) − 1)_R
11         if (R_R^{t-1} ∧ Z_R) = (candidate ∧ Z_R)
12             results.add(candidate)
13         q := (q + 1) ∨ ¬Z_R
14     }
15     return results
```

Denote $Z = (S^t - R^t - 1) \oplus RC^{t-1}$. We are interested in computing $R_R^{t-1} = \text{SHA-1‘}((Z \oplus R^{t-1}) - R^{t-1} - 1)_R$. Denote by r_i the ith least significant bit of R^{t-1}. We know all of Z, and the 120 leftmost bits of R^{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.

$$\begin{aligned}(Z \oplus R^{t-1}) - R^{t-1} - 1 &= (\sum_{i \in 0_Z} 2^i r_i + \sum_{i \in 1_Z} 2^i (1 - r_i)) - \sum_{i=0\ldots159} 2^i r_i - 1 \\ &= Z - 2 \cdot \sum_{i \in 1_Z} 2^i r_i - 1 \\ &= Z - 2 \cdot (R^{t-1} \wedge Z) - 1\end{aligned}$$

where $\wedge, \vee, \neg$ denote bit-wise AND, OR, and NOT. Therefore,

$$R_R^{t-1} = \text{SHA-1‘}(Z - 2 \cdot (R^{t-1} \wedge Z) - 1)_R \tag{7.1}$$

Equation 7.1 shows that only the bits of R^{t-1} for which the corresponding bit z_i equals 1 affect the result. The attack can therefore be more efficient: on average, only 20 out of the 40 least significant bits of Z are 1.

Denote n the number of bits set in Z_R. Then only 2^n options for R^{t-1} must be checked, with all the irrelevant bits set arbitrarily. It is possible to iterate over these values efficiently, without checking all 2^{40} options, as shown in Figure 7.1: the variable

q is manipulated by having the bits in the irrelevant positions always set. When q is incremented, carry from the relevant bits passes through the irrelevant bits transparently.

When checking the equality in Equation 7.1, the irrelevant bits must be masked out on both sides of the equation. This is shown as the bitwise-and operation in line 11 of Figure 7.1. Note that in the `candidate` value, all these bits are unambigously defined as a result of the SHA-1‘ operation.

As before, the correct candidate value of R_R^{t-1} is always found, while each of the remaining 2^n values may cause a false positive with probability 2^{-n}.

In the general case, the attack takes 2^n time. Therefore, assuming that Z is uniformly distributed, the expected overhead of the attack is:

$$\begin{aligned} \sum_{n=0}^{40} 2^n \Pr(|1_{Z_R}| = n) &= \\ \sum_{n=0}^{40} 2^n \binom{40}{n} * 2^{-40} &\approx (3/2)^{40} \approx 2^{23} \end{aligned}$$

We implemented this attack and found that its running time, without any optimization or parallelization, is about 19 seconds on a 2.80MHz Pentium IV. This timing corresponds to $n = 23$. For other values of n, the timing changes exponentionally: for $n = 22$, it takes 9.5 seconds, and for $n = 24$ it takes 38 seconds etc. Extreme cases are exponentially rare; for example, $n = 40$, which should take about 11 days, happens with probability 2^{-40}. Further optimization is possible, and since the 2^n options are independent, the search can be made fully parallel.

7.4 Combining the attacks

The attacks on the WRNG's forward and backward security combine for devastating effect: an attacker which learns the state of the generator, can *always* recover 128 kilobytes of WRNG output.

The attacker can compute past and future states and outputs of the WRNG, but he is limited to a time window between two points: the last state refreshment and the next state refreshment. At both points, the generator's state is replaced and the attacker cannot proceed. The size of that window is 128 kilobytes, as shown in Section 5.2.

Chapter 8

Recommendations for Safe Use

This chapter contains our recommendations for users of the current WRNG version on workarounds that would make the use of WRNG secure, and for implementors of future Windows PRNGs, on changes to the design that would eliminate the security problems.

8.1 Recommendations to Users of the Current WRNG

A major cause of problems in the WRNG is the long interval between entropy refreshments (see Section 5.2). From the user's perspective, this means his outputs can be stolen by an attacker breaking into the machine in the distant past as well as the distant future. Several of the workarounds in this section try to solve this problem by making the WRNG refresh itself more often. Others propose an entirely different mode of use for the WRNG, as a component in a different PRNG instead of a complete RNG.

8.1.1 Consume More Random Bytes

Consider the following realistic example: a user whose process consumes 16 bytes from the WRNG at regular 1-minute intervals, to create new keys for encrypting communications. With the 128-kilobyte interval between WRNG refreshments, it would take $((128 * 1024)/16)/60 = 136$ hours, or more than five days, before his WRNG would re-key its internal states. An attacker who breaks into this process, and uses the attacks of Chapter 7 to acquire the WRNG state and its past and future outputs, need only repeat the break-in once in every five days to keep the system completely compromised.

Our recommendation is to request e.g. 2048 bytes every minute instead, and simply discard any unneeded bytes. This way the WRNG re-keys itself every hour, demanding about as much effort from the potential attacker as the effort needed to fully control the system without exploiting the WRNG. The specific parameters can be adjusted, of course.

8.1.2 Drain All Random Bytes

Users often need to create a single highly secure cryptographic token, such as a master key in a key ladder. In that case, even a 1-hour attack window as in the previous subsection may be unsatisfactory. The user would want absolutely no chance of past or future break-ins compromising his master key. The solution to this problem is to drain the WRNG completely *before and after* generating the key, causing the WRNG to refresh itself with entropy. The first refresh will erase its past state, which could be compromised by a past break-in, and the second refresh will erase the state used for generating the important key, eliminating the chance of its exposure by a future break-in. To summarize, the algorithm is:

1. Request and discard 128 kilobytes of WRNG output.

2. Request as many bytes as needed to generate the secure token.

3. Request and discard 128 kilobytes of WRNG output.

8.1.3 Patch the WRNG

It is possible to shorten the refreshment interval with a patch.

The refreshment threshold is a static variable in the WRNG's DLL, it can be redefined by patching the WRNG code, or by overwriting it in memory.

For the procedures for patching and direct memory manipulation, see Section 9.4. The two methods allow creating a fixed version of the DLL with a shorter refreshment interval, or changing the interval from within an application after the DLL is loaded.

The lowest practical value for the threshold is 20, which means "re-key each RC4 instance on every use". This would eliminate the security problem, but may not be advisable due to the high complexity of the entropy collector. The patch would cause the generator to run slowly and become computationally expensive, and also the entropy buffers collected on every invocation would be more correlated.

An equivalent solution would be to patch the WRNG to re-enable the unused control path described in Section 6.2. This would cause the entropy collector to be invoked on every call to the WRNG, with the same problem as in the previous paragraph.

8.1.4 Other Modes of Invoking the WRNG

The WRNG may have security problems, but it still has a very strong entropy collector. It's possible to use the WRNG as one of the sources of entropy for another PRNG.

A tempting but wrong option would be to directly invoke the kernel portion of the WRNG (the `KSecDD` device, described in Section 4.3.4), without the user-mode wrapper. Thsi is not advisable: although the kernel part is less vulnerable to the attacks of Section 7, it has the same weaknesses in principle.

8.2 Possible Fixes

Clearly, the WRNG design has many problems. This section summarizes our recommendations for fixing that design, in order of increasing effort.

8.2.1 Replace RC4

As shown in Chapter 7, the use of RC4 in WRNG compromises its forward security. A simple and effective solution would be to replace RC4 with a transition function based on a hash function, which would provide the one-way property RC4 lacks.

8.2.2 Collect Entropy More Often

Collecting entropy more often would mitigate the WRNG's security problems. The developers can fix the WRNG by lowering the refreshment threshold.

A better solution would be to separate entropy refreshment from output generation. The refreshment threshold should depend on a time interval, not on the number of bytes requested; an attacker can directly affect the latter.

Moreover, it should be possible for small system events (keystrokes, mouse movement, disk interrupts etc.) to pass data to the WRNG as entropy inputs, preferably for processing with a lightweight entropy collector. This would create an asynchronous trickle of entropy into the WRNG, complicating things for an attacker.

We do not recommend removing the WRNG's current entropy collector completely. Gathering large amounts of system data at a time is a good strategy to defeat attackers attempting a sustained state exposure attack (see Section 2.4). Instead, a second entropy collector should be added for frequent state refreshments.

8.2.3 Move into the Kernel

Runnnig the WRNG as a kernel-mode service, perhaps with a thin wrapper in user-mode for convenient invocation, would benefit the WRNG security by hiding its state, thus making the initial break-in harder. In the kernel, the WRNG would be shared by all processes on the machine, and could collect entropy directly from the operating system. In our opinion, the overhead incurred by calling into the kernel every time a process needs highly random bytes is worth the extra security.

8.2.4 Switch to a Proven Design

Ultimately, we believe that it is preferable to replace the entire WRNG with a simple algorithm that has been rigorously analyzed. A good approach is to adopt the Barak-Halevi construction, discussed in Section 3.3.1. It is a highly efficient and simple design with a powerful proof of security.

Chapter 9

Reverse Engineering

In this chapter we describe in the reverse-engineering process that led us to our findings.

Software reverse-engineering is the study of code with the purpose of deducting its algorithm. It requires the reverse-engineer to perform intelligent analysis and to understand the reasoning used by the algorithm's original designer, and therefore cannot be automated.

The core of the reverse-engineering process is static analysis: reading the code in an attempt to understand its building blocks, its interconnections, and finally its workings as a whole. We describe the static analysis we've performed on WRNG in Section 9.3.

It's a meticulous process, and difficult even when the object of study is written in a high level language; it is even more difficult if the object has been compiled to machine code (binary or assembly). Typical compilation strips a lot of information during the transition from source to machine code. Therefore reverse engineering is a time consuming process that requires knowledge of operating systems and compilers. Nevertheless it's practical, and with the right skills, tools and insight, it gives us the ability to examine the security of closed-source products such as the Microsoft Windows OS.

The difficulties of static analysis can be mitigated by getting as much information about the target as possible, using other means. In our research on the WRNG, we used all the public information available (see Section 2.8). Other kinds of analysis are also highly effective: basic analysis (Section 9.1) includes simple tools and techniques for examining binaries and executables; dynamic analysis comprises running the target binary under a debugger (Section 9.2); intrusive analysis is a collection of advanced examination techniques that modify the original binary or its mode of use (Section 9.4); finally, kernel-level analysis (Section 9.5) is needed to reverse-engineer the kernel components of the WRNG, and track its global interactions with the system.

We cannot cover reverse-engineering fully in this thesis, as it is a field rich in research and practical applications. Due to constraints of space, we can only outline the different stages of reverse-engineering the WRNG. For literature on reverse-engineering, see [15, 32, 53].

9.1 Basic Analysis

Basic analysis includes a variety of simple techniques that reveal information on the target binary, before it is examined in detail.

Most function names and variable names are stripped during compilation, but the ASCII strings remaining in binary can help understand its purpose. The GNU command-line tool `strings` [21], available in Windows, searches the binary for consecutive ASCII characters. The binary's import and export tables contain the API functions it offers or uses. These can be inspected using tools such as `depends` [43] or `dumpbin` [8].

The Unix tool `strace` and its Windows ports [61] record all system calls made by an executable (if the target is a library, an executable can be created to call the relevant functions). This can completely expose a program's interaction with the operating system. `strace` even searches the system's header files to map constants (enumerations and `#define`'s) to human-readable names. This is how we acquired the function names in Table 4.1, even before we began static analysis.

9.2 Running under a Debugger

Running a process under a debugger is a kind of dynamic analysis. Intrusive analysis, covered in Section 9.4, also falls under that category.

Running a program rather than examining it statically can help focus on relevant code only, and better understand the meaning of variables and functions.

Most development environments offer their own debuggers, but reverse-engineering normally involves debugging without access to the source code. There are specialized debuggers for these circumstances. The shareware debugger `OllyDBG` [62], and the debugger built into `IDA Pro` [24] are highly recommended.

Invoking the WRNG in a debugger shows its control flow and the binaries involved. The control flow starts in the API function `CryptGenRandom` in `ADVAPI32.DLL`, which dispatches the call to the `CPGenRandom` function in the Cryptographic Service Provider (see Section 4.1.1). The CSP for the Microsoft Enhanced Cryptographic Provider is `RSAENH.DLL`. The control flow continues differently in Windows 2000, XP and Vista: in Windows 2000, it remains inside `RSAENH.DLL`, whereas in XP and Vista and jumps back into `ADVAPI32.DLL`, into `SystemFunction036`.

There is a control flow path that the user-mode debugger can't follow. It's a system call, invoking a method in the driver KSecDD (see Section 4.3.4). It is handled by the kernel, not inside the process scope; it takes a kernel debugger (see Section 9.5) to inspect the control flow there.

Figure 9.1 summarizes the relevant binaries in Windows 2000, which we used as reference for our reverse-engineering, and later validated using our tools.

Figure 9.1: Windows 2000 binaries' versions and MD5 signatures

ADVAPI32.DLL	5.0.2195.6876	5469E6799EFE4F3AE60BB2477C6C8098
RSAENH.DLL	5.0.2195.6611	25B6E88F1D5475E2ADDABBA2E9CA0DBA
KSECDD.SYS	5.0.2195.6824	33C865F17883BDE68A8B77D94E7B4CBA
ADVAPI32.LIB	N/A	5E3A359EEDF96087E405FD83F38BE28D

9.3 Static Analysis

Static analysis is the process of reading and examining code, typically in disassembly, trying to map it to the high-level programming constructs that correspond to it in the source. It normally involves commenting or editing the code to add meaningful names and explanations. Static analysis is the most time-consuming and difficult part of reverse-engineering, but it yields the most authoritative information.

Typically, static analysis includes mapping out all relevant code and creating a graph of function calls and data cross-references. All available type information is propagated top-down (from a known function prototype to every mention of its arguments) and bottom-up (from a known function prototypte to every instance of a call to this function). The reverse-engineer must identify the meaning of the building blocks and gradually translate the assembly to a higher-level pseudocode.

The minimum tools required are a command-line disassembler such as `dumpbin` [8] and a text processor, preferably with good search-and-replace capabilities and support for regular expressions (e.g. `Notepad++` [57]). Alternatively, high-end professional tools like `IDA Pro` [24] provide these capabilities as well as debugging, scripting, and much more

One valuable feature available in advanced tools is the support for symbol files. Symbol files contain debugging information, including meaningful function and variable names, which can make reverse-engineering much easier. Microsoft provides Windows symbol files for the developers' convenience, including those for the WRNG binaries [9]. These files reveal for example that the name for the WRNG main loop is `FIPS186Gen`, hinting at its original design.

Since static analysis is error-prone, it should be validated. Validation is done by translating the pseudocode created by the reverse-engineer into a program that attempts to simulate the target. When running the resulting simulator along with the target program, the two must produce the same outputs on the same inputs.

The following is an example of reverse-engineering a function. The reverse-engineer is examining a function which processes its input with repetative series of AND, NOT, XOR and shift operations, creates a 20-byte output, and uses the constants `0x5A827999, 0x6ED9EBA1, 0x8F1BBCDC`, and `0xCA62C1D6`. These constants are known to be used in the SHA-1 function; therefore the reverse-engineer should consider SHA-1 as the possible source, especially if it is mentioned in the public documentation. The reverse-engineer should then test this theory by running a sample implementation of SHA-1 on the same input as the suspected function, and comparing the outputs.

9.4 Intrusive Analysis

Intrusive analysis is a kind of dynamic analysis in which the target binary is modified or used outside of its intended context. It can be very effective for a variaty or purposes.

Some functions are difficult to understand because their original context is complicated, because they're deep in the calling graph, or because they have intricate details. They may be easier to understand if they can be run under a "magnifying glass": outside of their original scope, on inputs chosen by the reverse-engineer, and possibly with some modifications. This can be achieved by *assembly ripping*, which involves pasting an arbitrary piece of assembly from the target into a test program, compiling it, and running it repeatedly on different inputs. This technique provides easy access to code and functions that were not meant to be exported. On the downside, it requires dealing with inline assembly and may be time-consuming.

A more effective techinque is to create *new entry points*. If the target is a library (such as a DLL), it can be loaded into the address space of the testing process. Ordinarily, the DLL only allows access to its exported functions. Nevertheless, the test program can take any offset in the DLL space, both to code and data, and cast it to a convenient type, then use the resulting pointer as a regular pointer to data or to function. Data can be read or written, and functions can be called with the usual conventions. This technique is available in languages that allow pointer arithmetic (`C, C++`), and can very efficient as it does not require dealing with inline assembler. We use this technique for the state exposure attack of Section 7.1.

The original binary can be modified in-place to isolate relevant parts, or bypass unwanted control flow branches. This technique is called *patching*. It can be done using any binary editor. Reverse-engineering tools such as `OllyDBG` and `IDA Pro` contain convenient facilities for patching. We used patching to disable the entropy collector of the WRNG temporarily; as a result, we had a simplified version of WRNG for initial analysis.

9.5 Kernel Debugging

A kernel debugger can serve as a regular user-mode debugger, for single-stepping processes and setting breakpoints. In addition, a kernel debugger has full access to the system, and can access CPU registers and the kernel space as well as any process in user-space. We used the Microsoft `WinDBG` [42] to trace the system call into `KSecDD`. After that we could use regular static analysis to examine the relevant parts of that driver.

We also used the kernel debugger to hook any calls to the WRNG API `CryptGenRandom`. As that function always occupies a fixed offset in the address space of the calling process, setting a global breakpoint on that address allowed us to watch every call to the WRNG on the system, and to record the calling process and the number of bytes requested.

Chapter 10

Future Work

This chapter presents research goals that were not achieved in this work, and directions for further research. In particular, we address here potential attacks that are not covered by our research, and operating systems we did not examine.

10.1 Unexplored Attack Venues

In this section we describe potential attacks on the WRNG that we did not manage to implement.

10.1.1 Ciphertext-only attack

We do not see a way to attack the WRNG and compromise its state or outputs, without access to the host machine. The WRNG processes its output with SHA-1, which is not inversible (see Section 4.3.1), so it does not appear possible to compromise the internal state of the WRNG by analyzing its output.

Another approach would be to try guessing the entropy inputs seeding the WRNG, and validate the guess using the outputs. This too does not seem feasible, because of the large size and variety of entropy inputs used in the WRNG.

10.1.2 Entropy Input Correlation

The WRNG uses the same entropy inputs collected several times at short intervals (see Section 5.2). It is possible that given one set of inputs, the other sets would be closely related and could be guessed. We did not perform measurements to check what number of bits differed in different sets of inputs, and we imagine that some correlation would be present. The WRNG could be especially sensitive to such correlations because of RC4's vulnerability to related keys attacks (see Section 4.3.2).

Due to the large size and variety of entropy inputs, and because the inputs pass through SHA-1, we do not believe this could likely be the basis for a practical attack.

10.2 Other Operating Systems

Our research centered on the Windows 2000 version of WRNG. Although it is largely similar to the WRNG in XP and Vista, we did not cover these operating systems fully. There are also unanswered questions about the WRNG operation on unusual platforms.

10.2.1 Windows XP

We examined Windows XP superficially. We reverse-engineered the main loop of its WRNG, and found it largely identical to the one in Windows 2000, but did not delve into the internal random number generator or the entropy collector. Neither did we validate our results with full simulation, as in Windows 2000.

Microsoft acknowledged [6] that our attacks apply to Windows XP. XP Service Pack 3, due in 2008, is promised to correct the problems, but it is not clear how.

10.2.2 Windows Vista

As of the writing of this thesis, we have examined Windows Vista very superficially. It appears it has the same general WRNG structure, with some differences in the main loop, and apparently uses RC4 for the internal random number generation as well. Unlike XP, Microsoft claimed that Vista was not vulnerable to our attacks [6]; however, it's unknown whether this is due to some technicality (like a small difference in the main loop, to which our attacks could be adapted), or a major difference.

In the first Service Pack for Vista, due in 2008, Microsoft promises to use different PRNG designs [58]:

> ...Strengthens the cryptography platform with a redesigned random number generator, which leverages the Trusted Platform Module (TPM), when present, for entropy and complies with the latest standards. The redesigned RNG uses the AES-based pseudo-random number generator (PRNG) from NIST Special Publication 800-90 by default. The Dual Elliptical Curve (Dual EC) PRNG from SP 800-90 is also available for customers who prefer to use it.

These are the PRNGs from NIST Special Publication 800-90 [3], one of which is controversial as noted in Section 3.1. We note that an implementation of the AES-based PRNG is potentially vulnerable to the same issues as the current WRNG, if it is not refreshed with entropy often enough.

10.2.3 Other Platforms

PRNG research is especially important in systems that have limited entropy sources. Systems without a human terminal or hard drives are harder to secure because an attacker should be able to guess or observe all of their entropy inputs. A natural solution would be to provide hardware-based RNGs for these systems, but clearly the OS should be involved as well.

We did not examine random number generation in operating systems like Windows CE or Windows XP Embedded.

The proliferation of mobile phones, advanced set-top-boxes and game consoles, all of which are used for delivery of highly valuable content, raises the importance of security on these platforms. Weak random number generation can compromise DRM and conditional access systems, therefore these devices should be examined closely.

Chapter 11

Conclusions

In this thesis, we present a detailed analysis of the Windows RNG, the security RNG deployed on 90% of computers in the world.

In the introductory chapters, we have highlighted the importance of random number generators for security, and the difficulty of designing and implementing them properly.

We present a short but comprehensive description of the WRNG. We are the first to fully publish its structure, mode of interaction with the system, and a detailed analysis of its design and implementation.

The WRNG has a complex layered architecture, refreshes itself with entropy after every 128 KBytes of output, and uses RC4 and SHA-1 as its building blocks. Windows runs the WRNG in user space, and keeps a different instance of the generator for each process.

We provide a short description of the reverse engineering process we used in order to obtain the algorithm of the WRNG from its binaries. Based on the results of reverse-engineering, we built tools which capture the WRNG state and compute future and past outputs of the WRNG.

The WRNG has design and implementation problems, mainly the inappropriate use of RC4 and the long 128-kilobyte interval between entropy refreshments. We have devised attacks that exploit these problems, showing how an attacker can learn future outputs and compute past outputs with an overhead of (2^{23}) operations. The combined attacks always let the adversary compromise 128 kilobytes of WRNG output.

Bibliography

[1] Donald E. Eastlake 3rd and P. Jones. US secure hash algorithm 1 (SHA1). RFC 3174, Internet Engineering Task Force, September 2001.

[2] Boaz Barak and Shai Halevi. An architecture for robust pseudo-random generation and applications to /dev/random. In *Proc. ACM Conf. on Computing and Communication Security (ACM CCS)*, 2005.

[3] E. Barker and J. Kelsey. Recommendation for random number generation using deterministic random bit generators (revised). SP-800-90, U.S. DoC/National Institute of Standards and Technology, 2007.

[4] Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. "Pseudo-Random" Number Generation within Cryptographic Algorithms: The DSS Case. *Lecture Notes in Computer Science*, 1294:277–??, 1997.

[5] Lenore Blum, Manuel Blum, and Michael Shub. Comparison of two pseudo-random number generators. In R. L. Rivest, A. Sherman, and D. Chaum, editors, *CRYPTO '82*, pages 61–78, New York, 1983. Plenum Press.

[6] Computerworld. Microsoft confirms that XP contains random number generator bug. http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=90 2007.

[7] Intel Corporation. Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture. http://www.intel.com/design/PentiumII/manuals/243190.htm.

[8] Microsoft Corporation. Description of the DUMPBIN utility. http://support.microsoft.com/kb/177429.

[9] Microsoft Corporation. Download Windows Symbol Packages. http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx.

[10] Microsoft Corporation. Microsoft Enhanced Cryptographic Provider: FIPS 140-1 Documentation: Security Policy. http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp238.pdf.

[11] Prasad Dabak, Sandeep Phadke, and Milind Borate. *Undocumented Windows NT*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[12] Don Davis, Ross Ihaka, and Philip Fenstermacher. Cryptographic randomness from air turbulence in disk drives. In *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, pages 114–120, London, UK, 1994. Springer-Verlag.

[13] Theo de Raadt, Niklas Hallqvist, Artur Grabowski, Angelos D. Keromytis, and Niels Provos. Cryptography in openbsd: An overview. In *USENIX Annual Technical Conf., FREENIX Track*, pages 93–101, 1999.

[14] Bryn Dole, Steve Lodin, and E. H. Spafford. Misplaced trust: Kerberos 4 Session Keys. In *Misplaced trust: Kerberos 4 Session Keys*, 1997.

[15] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, April 2005. http://www.wiley.com/go/eeilam.

[16] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, 2003.

[17] FIPS. Digital signature standard (dss), FIPS PUB 186, 1994. http://www.itl.nist.gov/fipspubs/fip186.htm.

[18] FIPS. Security requirements for cryptographic modules, FIPS PUB 140-2, 2001.

[19] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In *SAC '01*, pages 1–24. Springer-Verlag, 2001.

[20] Scott R. Fluhrer and David A. McGrew. Statistical analysis of the alleged rc4 keystream generator. In *FSE '00: Proceedings of the 7th International Workshop on Fast Software Encryption*, pages 19–30, London, UK, 2001. Springer-Verlag.

[21] Free Software Foundation. GNU Binutils. http://www.gnu.org/software/binutils/.

[22] Ian Goldberg and David Wagner. Randomness in the netscape browser. *Dr. Dobb's Journal*, January 1996.

[23] Oded Goldreich. *Foundations of Cryptography*, volume Basic Tools. Cambridge University Press, 2001.

[24] Ilfak Guilfanov. The IDA Pro disassembler and debugger version 5.0, March 2006. http://www.datarescue.com/idabase/.

[25] Peter Gutmann. Software generation of practically strong random numbers. In *Proc. of 7th USENIX Security Symposium*, 1998. An updated version appears in http://www.cypherpunks.to/~peter/06_random.pdf.

[26] Peter Gutmann. Testing issues with os-based entropy sources. http://www.cs.auckland.ac.nz/~pgut001/pubs/nist_rng.pdf, July 2004.

[27] Zvi Gutterman and Dahlia Malkhi. Hold your sessions: An attack on java session-id generation. In *CT-RSA*, pages 44–57, 2005.

[28] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2006.

[29] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2 edition, April 2002.

[30] Intel. Intel 82802 firmware hub: Random number generator programmer's reference manual. http://www.intel.com/design/chipsets/manuals/298029.htm, 1999.

[31] Thomas Jennewein, Ulrich Achleitner, Gregor Weihs, Harald Weinfurter, and Anton Zeilinger. A fast and compact quantum random number generator, 1999.

[32] Kris Kaspersky, Natalia Tarkova, and Julie Laing. *Hacker Disassembling Uncovered*. A-List Publishing, 2003.

[33] John Kelsey. Entropy and entropy sources in x9.82. http://csrc.nist.gov/CryptoToolkit/RNG/Workshop/EntropySources.pdf, July 2004.

[34] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography*, pages 13–33, 1999.

[35] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. *LNCS*, 1372:168–188, 1998.

[36] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption*, pages 168–188, 1998.

[37] D. E. Knuth. *Seminumerical Algorithms*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[38] Hugo Krawczyk. How to predict congruential generators. In *Proceedings on Advances in cryptology*, pages 138–153. Springer-Verlag New York, Inc., 1989.

[39] I. Mantin and A. Shamir. A practical attack on broadcast rc, 2001.

[40] G. Marsaglia. Random numbers fall mainly in the planes. *Proc. N.A.S.*, 61:25–28, 1968.

[41] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[42] Microsoft. Debugging tools for windows, July 2006. http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

[43] Steve P. Miller. Dependency Walker. http://dependencywalker.com/.

[44] MSDN. CPGenKey Function (Windows). http://msdn2.microsoft.com/en-us/library/aa3782

[45] MSDN. CPGenRandom Function (Windows). http://msdn2.microsoft.com/en-us/library/aa378205.aspx.

[46] MSDN. CryptGenRandom Function (Windows). http://msdn2.microsoft.com/en-us/library/aa379942.aspx.

[47] MSDN. Cryptography API: Next Generation, About CNG, CNG Features. http://msdn2.microsoft.com/en-us/library/bb204775.aspx.

[48] MSDN. Microsoft Enhanced Cryptographic Provider. http://msdn2.microsoft.com/en-gb/library/aa386986.aspx.

[49] MSDN. The Windows Server 2003 Family Encrypting File System. http://msdn2.microsoft.com/en-us/library/ms995356.aspx.

[50] Mark R. V. Murray. An implementation of the Yarrow PRNG for FreeBSD. In Samuel J. Leffler, editor, *BSDCon*, pages 47–53. USENIX, 2002.

[51] National Institute of Standards and Technology. FIPS PUB 180-1: Secure hash standard. Technical report, National Institute of Standards and Technology, 1995.

[52] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Commun. ACM*, 31(10):1192–1201, 1988.

[53] Cyrus Peikari and Anton Chuvakin. *Security Warrior*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[54] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[55] Adi Shamir. On the generation of cryptographically strong pseudo-random sequences. In *Proc. ICALP*, pages 544–550. Springer, 1981.

[56] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

[57] SourceForge.net. About Notepad++. http://notepad-plus.sourceforge.net/uk/site.htm.

[58] Microsoft TechNet. Overview of Windows Vista Service Pack 1. http://technet2.microsoft.com/WindowsVista/en/library/417467e7-7845-46d4-85f1-dd4

[59] Ted Ts'o. random.c — linux kernel random number generator. http://www.kernel.org.

[60] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. *LECTURE NOTES IN COMPUTER SCIENCE*, 3621:17–??, 2005.

[61] www.intellectualheaven.com. StraceNT - A System Call Tracer for Windows. http://www.intellectualheaven.com/default.asp?BH=projects&H=strace.htm.

[62] Oleh Yuschuk. Ollydbg 1.1: A 32-bit assembler level analysing debugger for microsoft windows, June 2004. http://www.ollydbg.de/.