Cryptanalysis of the Random Number Generator of the Windows Operating System

Leo Dorrendorf
School of Engineering and Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
dorrel@cs.huji.ac.il

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

Benny Pinkas*
Department of Computer Science
University of Haifa
31905 Haifa, Israel
benny@pinkas.net

November 4, 2007

Abstract

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

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

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

The implication of these findings is that a buffer overflow attack or a similar attack can be used to learn a single state of the generator, which can then be used to predict all random

*Research supported in part by the Israel Science Foundation (grant number 860/06).

values, such as SSL keys, used by a process in all its past and future operation. This attack is more severe and more efficient than known attacks, in which an attacker can only learn SSL keys if it is controlling the attacked machine at the time the keys are used.

1 Introduction

Almost all cryptographic systems are based on the use of a source of random bits, whose output is used, for example, to choose cryptographic keys or choose random nonces. The security analysis (and proofs of security) of secure systems are almost always based on the assumption that the system uses some random data (e.g., a key) which is uniformly distributed and unknown to an attacker. The use of weak random values may result in an adversary being able to break the system (e.g., weak randomness may enable the adversary to learn the cryptographic keys used by the system). This was demonstrated for example by the analysis of the implementation of SSL in Netscape [11], or in an attack predicting Java session-ids [15].

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

Security properties. A pseudo-random number generator must be secure against external and internal attacks. An attacker might know the algorithm (or code) which defines the generator, might know the output of the generator, might be able at some point to examine the generators's state, and might have partial knowledge of the entropy used for refreshing the state. We list here the most basic security requirements that must be provided by pseudo-random generators, using common terminology (e.g., of [2]).

- *Pseudo-randomness.* The generator's output looks random to an outside observer.
- *Forward security.* An adversary which learns the internal state of the generator at a specific time cannot learn anything about previous outputs of the generator.
- *Backward security* (also known as *break-in recovery*). An adversary which learns the state of the generator at a specific time does not learn anything about future outputs of the generator, provided that sufficient entropy is used to refresh the generator's state.

Regarding *backward security*, note that the generator operates as a deterministic process and therefore knowledge of the state of the generator at a specific time can be used to compute all future outputs of the generator (by simply simulating the operation of the algorithm run by the generator). Consequently, backward security can only be provided if the state of the generator is periodically refreshed with data ("entropy") which is sufficiently random.

Forward security, on the other hand, is concerned with ensuring that the state of the generator does not leak information about previous states and outputs. If a generator does not provide forward security then an attacker which learns the state at a certain time can learn previous outputs of the generator, and consequently, past transactions of the user of the system. (Consider, for example, an attacker which uses a computer in an Internet cafe and learns keys used by *previous* users of the machine. Another option for an attacker is to decide which machine to attack only *after* observing which machines are interesting as targets; e.g., machines which were used by a specific user or were used for specific transactions.) Forward security can be easily guaranteed by ensuring that the function which advances the state is one-way. It is well known how to construct forward-secure generators (for an early usage of such generators see, e.g., [3]; see also [4] for a comprehensive discussion and a generic transformation of any standard generator to one which provides forward security). Forward security is also a mandatory requirement of the German evaluation guidances for pseudo-random number generators (AIS 20) [1]. The fact that the random number generator used by Windows 2000 does not provide forward security demonstrates that the design of the generator is flawed.

The random number generator used by Windows. This paper studies the pseudo-random number generator used in Microsoft Windows systems, which we denote as the WRNG. The WRNG is the most frequently used pseudo-random number generator, with billions of instances running at every given time. It is used by calling the function `CryptGenRandom`. According to the book "Writing Secure Code" [17], published by Microsoft, the WRNG was first introduced in Windows 95 and was since embedded in all Windows based operating systems such as Windows XP or Windows 2000, and in all their variants.[1] According to [17] the design of the WRNG has not changed between the different version of the operating system.[2]

In this work we examine the generator that is implemented in the Windows 2000 operating system (service pack 4). Windows 2000 is the second most popular operating system, especially in enterprises, with a market share of 4.5%-6% as of April 2007.[3]

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

Our contributions. This paper describes the following results:

- We present a detailed analysis of the Windows pseudo random number generator. The analysis is accompanied by a concise pseudo-code for the entire implementation of the WRNG (the

[1] The statement in [17] was written before Windows Vista was released. The documentation of `CryptGenRandom` states that it is supported by Windows Vista, but we have not verified this statement.

[2] Our checks show, however, some variations between the implementation of the WRNG in Windows 2000 and in prior versions of the Windows operating system. For example, the code distributed with Windows 2000 uses the type of the operating system to set the number of bytes which are output between two entropy based rekeys of the generator. In Windows 2000 rekeys are done after 16 KBytes of output, while in earlier versions of Windows they are done after outputting only 512 bytes.

[3] See `http://marketshare.hitslink.com/report.aspx?qprid=5`, `http://www.onestat.com/html/aboutus_pressbox46-operating-systems-market-share.html`.

complete pseudo-code is about 1000 lines of code), and by a user-mode simulator of the WRNG. The analysis is based on examination of the binary code of the operating system, see details below.

- We present an attack on the forward security of the WRNG. We show how an adversary can compute past outputs and states from a given state of the WRNG, with an overhead of 2^{23} computation (namely, in a matter of seconds on a home computer).
- We present an attack on the backward security of the WRNG. We show that given the inner state of the WRNG an adversary can compute future outputs and states with an overhead of $O(1)$ computation.
- We analyze the way in which the operating system uses the WRNG and note that a different copy of the WRNG is run, in user-mode, for every process, and that typical invocations of the WRNG are seldom refreshed with additional entropy. Therefore, the backward and forward security attacks, which only work while there is no entropy based rekeying, are highly effective. Furthermore, we also found that part of the state of the generator is initialized with values that are rather predictable.

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

Gap between theory and practice. The generation of pseudo-random numbers is a well studied issue in cryptography, see, e.g., [25, 5]. One might therefore be surprised to learn that constructing an actual implementation of a pseudo-random number generator is quite complex. There are many reasons for this gap between theory and practice:

- *Performance.* Provably secure generators might incur high computation overhead. Therefore even a simple PRNG such as the Blum-Blum-Shub generator [5] is rarely used in practice.
- *Real world attacks.* Actual implementations are prone to many attacks which do not exist in the clean cryptographic formulation which is used to design and analyze pseudo-random generators (consider, for example, timing attacks and other side-channel attacks).
- *Seeding and reseeding the generator.* Generators are secure as long as they are initialized with a truly random seed. Finding such a seed is not simple. Furthermore, the state of the generator must be periodically refreshed with a fresh random seed in order to prevent backward security attacks. The developer of a generator must therefore identify and use random sources with sufficient entropy.

- *Lack of knowledge.* In many cases the developers of the system do not have sufficient knowledge to use contemporary results in cryptography. by many programming languages (such as the `C` and `C++` []).

These factors demonstrate the importance of providing a secure pseudo-random generator by the operating system.[4] The designers of the operating system can be expected to be versed with the required knowledge in cryptography, and know how to extract random system data to seed the generator. They can therefore implement an efficient and secure generator. Unfortunately, our work shows that the Windows pseudo-random generator has several unnecessary flaws.

1.1 Related Work

Existing PRNG implementations. In the past, PRNGs were either a separate program or a standard library within a programming language. The evolution of software engineering and operating systems introduced PRNGs which are part of the operating system. From a cryptographic point of view, this architecture is advantageous since it enables to initialize the PRNG with operating system data (which has more entropy and is hidden from users).

Implementations of PRNGs can be either blocking or non-blocking. A blocking implementation does not provide output until it collects sufficient amount of system based entropy. A non-blocking application is always willing to provide output. The PRNG of the FreeBSD operating system is described in [23]. FreeBSD implements a single non-blocking device and the authors declare their preference of performance over security. The PRNG used in OpenBSD is described in [6], which also includes an overview of the use of cryptography in this operating system. Castejon-Amenedo et al. [18] propose a PRNG for UNIX environments. Their system is composed of an entropy daemon and a buffer manager that handles two devices—blocking and non-blocking. The buffer manager divides entropy equally between the two devices, such that there is no entropy that is used in both. A notable advantage of this scheme is the absolute separation between blocking and non-blocking devices, which prevents launching a denial-of-service attack on the blocking device by using the non-blocking device (such an attack is possible in Linux, as is shown in [16]).

The Linux PRNG. The Linux operating system includes an internal entropy based PRNG named `/dev/random` [26], which, following [16], we denote as the LRNG. The exact algorithm used by the LRNG (rather than the source code, which is open) was published in [16], where several security weaknesses of this generator were also presented. We discuss in detail in Section 6 the differences between the LRNG and the WRNG. We note here that the attack on the WRNG is more efficient, and that in addition, unlike the LRNG, the WRNG refreshes its state very rarely and is therefore much more susceptible to attacks on its forward and backward security. On the other, the WRNG is not susceptible to denial of service attacks, which do affect the LRNG.

Analysis of PRNGs. A comprehensive discussion of the system aspects of PRNGs, as well as a guide to designing and implementing a PRNG without the use of special hardware or access to privileged system services, is given by Gutmann [13]. Issues related to operating system entropy sources were discussed in a recent NIST workshop on random number generation [19, 14]. An extensive discussion of PRNGs, which includes an analysis of several possible attacks and their

[4] Indeed, given the understanding that writing good cryptographic functions is hard, operating systems tend to provide more and more cryptographic functionality as part of the operating system itself. For example, Linux provides implementations of hash functions as part of its kernel.

relevance to real-world PRNGs, is given by Kelsey et al. in [21]. Additional discussion of PRNGs, as well as new PRNG designs appear in [20, 8].

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

Outline. The rest of the paper goes as follows. Section 2 provides a detailed description of the WRNG. Section 3 presents cryptanalytic attacks on the generator, while Section 4 describes the interaction between the operating system and the generator, and its security implications. Section 6 compares the WRNG to the generator used by Linux, and Section 7 contains conclusions and suggestions for further research.

2 The Structure of the Windows Random Number Generator

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

2.1 Analyzing the Binary Code

The algorithm employed by the WRNG, and its design goals, were never published. There are some published hints about the inner structure of the WRNG [17]. However, the exact design and security properties were not published.

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

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

We used several tools in our analysis: the Interactive Disassembler (IDA) tool [12] which is an editor for static code analysis, the OllyDbg tool [27] for dynamic study of our user mode runtime environment, and the WinDBG tool [22] as our kernel debugging tool. (See also the book

"Reversing" [7] which provides an excellent introduction to the field of code analysis.) To verify our findings and demonstrate our attacks we developed four tools:

- `CaptureCryptGenRandom`: captures the current WRNG state into a file.
- `NextCryptGenOutputs`: calculates future outputs of the WRNG from a given state.
- `PreviousCryptGenOutputs0`: calculates previous outputs and states of the WRNG from a given state and knowledge of the initial State and R variables (this attack, and the roles of State and R, are described in Section 3.2).
- `PreviousCryptGenOutputs23`: calculates previous outputs and states of the WRNG from a given state alone. (This attack in described in Section 3.2. It has an overhead of $O(2^{23})$.)

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

2.2 The Structure of the Generator

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

The main loop of the WRNG. The main loop, presented in Figure 2.1, generates 20 bytes of output in each iteration. The main state of the WRNG is composed of two registers, R and State, which are updated in each iteration and are used to calculate the output. The loop operates on data in chunks of 20 bytes: each of the registers used in the main loop, R, State and T, is 20 bytes long. This is also the length of the result of the internal function call `get_next_20_rc4_bytes` and of the output of SHA-1. The output is generated in increments of 20 bytes.

```
1  CryptGenRandom(Buffer, Len)
2  // output Len bytes to buffer
3     while (Len>0) {
4        R := R ⊕ get_next_20_rc4_bytes()
5        State := State ⊕ R
6        T := SHA-1'(State)
7        Buffer := Buffer | T
8          // | denotes concatenation
9        R[0..4] := T[0..4]
10         // copy 5 least significant bytes
11       State := State + R + 1
12       Len := Len − 20
13    }
```

Figure 2.1: The main loop of the WRNG. It has input parameters *Len*, which is the number of bytes to be output, and *Buffer*, which gets the output. All internal variables are 20 bytes long and uninitialized. *Buffer* is assumed to be empty and the WRNG output is concatenated to it in each round of the loop. The function SHA-1' is a variant of SHA-1 where the Initialization Vector (IV) is ordered differently.

The main loop uses the two variables, R and State, to store a state. It calls an internal function `get_next_20_rc4_bytes` to obtain 20 bytes of pseudo-random data, and uses them to update R and State. It generates 20 bytes of output by applying a variant of SHA-1 to State, and then updates State again using part of this output and using R. (The only difference between the variant of SHA-1 used here and the standard implementation of SHA-1 is a different ordering of the IV vector. We therefore use the notation SHA-1 in most of the discussion.)

The function `get_next_20_rc4_bytes`. The function `get_next_20_rc4_bytes`[5] keeps a state which is composed of eight instances of the RC4 stream cipher. (See Appendix A for a description of RC4). In each call, the function selects one RC4 state in a round-robin fashion, uses it to generate 20 bytes of output, and returns them to its caller. In the next call it uses the next RC4 stream. After an RC4 instance generates 16Kbytes of output it is refreshed with entropy gathered from the system, as is described below.

The function is described in Figure 2.2 (this description assumes a static variable `i` which is initialized to zero before the first call). We can also imagine this function as storing eight output streams from eight independent invocations of RC4. The function holds a pointer `i` which points to one of the streams, and for each stream (numbered i) it holds a counter c_i which points to a location in the stream (in the code of Figure 2.2 this counter is denoted by `RC4[i].accumulator`). When the function is called it returns the 20 bytes numbered c_i to $c_i + 19$ from the stream pointed to by `i`. It then sets $c_i = c_i + 20$, and advances `i` in a round-robin fashion.

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

Figure 2.2: Function `get_next_20_rc4_bytes()`.

Initializing R and State. The WRNG does not explicitly initialize R and State. However, as with any other stack parameter which is not initialized by the program, these variables are implicitly initialized with the latest values stored in the memory address allocated to them. We describe in Section 4 some analysis of the actual values with which these variables are initialized, and note that they are highly correlated. We are not sure about the reason for this use of uninitialized variables.

[5]This function is called `NewGenRandom` in Windows 2000. We use instead the name `get_next_20_rc4_bytes` which describes the functionality of the function more clearly.

Initializing and refreshing each instance of RC4. All instances of RC4 are initialized and refreshed by the same mechanism, which collects system entropy and uses it to rekey an RC4 instance. The collected system entropy is composed of up to 3584 bytes of data from different operating system sources. Entropy collection is synchronous and is only done when an RC4 stream is initialized, or reaches the 16 Kbyte threshold. We list in Table 1 the different operating system entropy sources. We were not able to see a way to predict all 3584 bytes of these parameters by a practical brute force attack.

The pseudocode for the state refreshment mechanism is described in Figure 2.3. It is composed of the following stages:

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

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

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

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

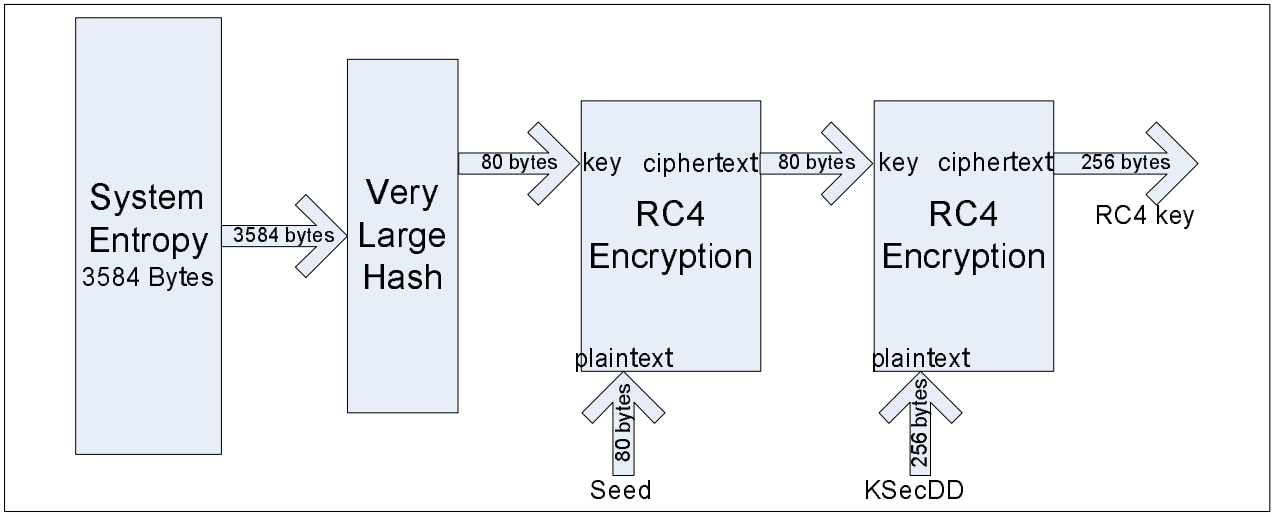

Figure 2.3: RC4 rekeying function

The Function VeryLargeHash The algorithm of VeryLargeHash is based on series of SHA-1 calls, performed on an input buffer (Buf) of any given length (Len), and a fixed-length 80-byte argument (Seed). The Buf holds the entropy buffer while Seed is the Windows global entropy parameter (which is common to all WRNG instances). Note that any change of a single input bit affects all output bits.

```
VeryLargeHash(Buf, Len, Seed) {
   k := Len / 4            // lengths are in bytes
   digest1 := SHA-1(Seed[00..19] | Buf[0..k − 1] | Seed[20..39] | Buf[k..2k − 1])
   digest2 := SHA-1(Seed[20..39] | Buf[k..2k − 1] | Seed[00..19] | Buf[0..k − 1])
   digest3 := SHA-1(Seed[40..59] | Buf[2k..3k − 1] | Seed[60..79] | Buf[3k..4k − 1])
   digest4 := SHA-1(Seed[60..79] | Buf[3k..4k − 1] | Seed[40..59] | Buf[2k..3k − 1])
   result[00..19] := SHA-1(digest1 | digest3)
   result[20..39] := SHA-1(digest2 | digest4)
   result[40..59] := SHA-1(digest3 | digest1)
   result[60..79] := SHA-1(digest4 | digest2)
   return result
}
```

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

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

Scope. Windows is running one WRNG instance *per process*. Therefore, two applications (e.g., Windows Word and Internet Explorer) have two separate states. The RC4 states and auxiliary variables of a specific process reside in DLL space which is allocated upon the first invocation of the Crypto API, and remains allocated until it is unloaded. The state variables R and State, on the other hand, are stored on the stack. If a process has several threads, then they all share the same RC4 states stored in the DLL space, but each of them has its own stack, and therefore its own copy of R and State. It is interesting to note that R and State are never explicitly initialized, and instead are initialized with the last values that are stored in the stack locations allocated to them. We will describe in Section 4 an analysis which shows that there is correlation between the states used in different instances of the WRNG.

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

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

3 Analysis I: Cryptanalytic Attacks

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

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

3.1 Attack on Backward Security

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

3.2 Attacks on Forward Security

The WRNG depends on RC4 for generating streams of pseudo-random output, which are then added to the state of the generator. RC4 is a good stream cipher, but it does not provide any forward security. Namely, given the current state of an RC4 cipher it easy to compute its previous states and previous outputs. (This process is described in Appendix A. See also [4].) We use this fact to mount attacks on the forward security of the WRNG. Suppose an adversary learns the state of the generator at time t and wishes to compute the state at time $t-1$. We show here three methods of computing this state, with an overhead of $O(1)$, $O(2^{40})$, and $O(2^{23})$, respectively, and where the first attack also assumes knowledge of the initial values of State and R. The attack with $O(2^{23})$ overhead is based on observing that State is updated using consecutive addition and exclusive-or operations, which, up to the effect of carry bits, cancel each other.

An instant attack when the initial values of State and R are known. Suppose that the attacker knows the initial values of the variables State and R. (As argued in Section 4 this is a reasonable assumption.) The attacker also knows the current values of the eight RC4 registers. Since RC4 does not provide any forward security, the attacker can compute *all* previous states and outputs of the RC4 registers, until the first invocation of the WRNG. (It can learn the total number of invocations of the WRNG from a static variable named `stream_counter`, found in a static offset in memory — offset `7CA1FFA8` in the DLL of the version of Windows we examined.) Since each state of the WRNG is a function of the previous values of State and R and of the output of the RC4 registers, the attacker can now compute the states and outputs starting from

the first step and continuing until the current time. We implemented this attack in the tool `PreviousCryptGenOutputs0`.

An attack with an overhead of 2^{40}. Let us denote by R^t and S^t the values of R and State just *before* the beginning of the tth iteration of the main loop. (We refer here to the main loop of the WRNG, as it is described in Figure 2.1.) Let us 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, $R^t = R^{t,4}, S^t = S^{t,4}$). Let RC^t denote the output of `get_next_20_rc4_bytes` in the tth iteration. Each of these values is 160 bits long. Let us also denote by X_L the leftmost 120 bits of variable X, and by X_R its 40 rightmost bits.

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

$$\begin{aligned}
S^{t-1,11} &= S^t - R^t - 1 \\
R^{t-1,9} &= R^t_L \mid *^{40} \qquad \text{(where } *^{40} \text{ is a 40-bit string which is unknown at this stage)} \\
R^{t-1} &= R^{t-1,9} \oplus RC^{t-1} \\
S^{t-1} &= S^{t-1,5} = S^{t-1,11} \oplus R^{t-1,9} = \underbrace{(S^t - R^t - 1)}_{S^{t-1,11}} \oplus \underbrace{(R^t_L \mid R^{t-1,9}_R)}_{R^{t-1,9}}
\end{aligned}$$

We also observe the following relation:

$$R^t_R = \mathsf{SHA\text{-}1}(S^{t-1,11})_R = \mathsf{SHA\text{-}1}(S^t - R^t - 1)_R$$

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

$$\begin{aligned}
R^{t-1}_R &= \mathsf{SHA\text{-}1}(S^{t-2,11})_R \\
&= \mathsf{SHA\text{-}1}(S^{t-1} - R^{t-1} - 1)_R \\
&= \mathsf{SHA\text{-}1}(\underbrace{(S^t - R^t - 1) \oplus (R^t_L \mid R^{t-1,9}_R)}_{S^{t-1}} - \underbrace{(R^t_L \mid R^{t-1,9}_R) \oplus RC^{t-1}}_{R^{t-1}} - 1)_R
\end{aligned}$$

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

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

$$R^{t-1}_R = \mathsf{SHA\text{-}1}(S^{t-2,11})_R = \mathsf{SHA\text{-}1}(S^{t-1} - R^{t-1} - 1)_R$$

Note also that

$$S^{t-1} = \underbrace{(S^t - R^t - 1)}_{S^{t-1,11}} \oplus \underbrace{RC^{t-1} \oplus R^{t-1}}_{R^{t-1,9}}$$

Let us use the notation $Z = (S^t - R^t - 1) \oplus RC^{t-1}$. We are interested in computing $R_R^{t-1} = \mathsf{SHA\text{-}1}((Z \oplus R^{t-1}) - R^{t-1} - 1)_R$. 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.

$$\begin{aligned}
(Z \oplus R^{t-1}) - R^{t-1} - 1 &= (\sum_{i \in 0_Z} 2^i r_i + \sum_{i \in 1_Z} 2^i (1 - r_i)) - \sum_{i=0 \ldots 159} 2^i r_i - 1 \\
&= Z - 2 \cdot \sum_{i \in 1_Z} 2^i r_i - 1 \\
&= Z - 2 \cdot (R^{t-1} \wedge Z) - 1
\end{aligned}$$

where $\wedge$ denotes bit-wise AND. Therefore,

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

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

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

We implemented this attack in the tool `PreviousCryptGenOutputs23`. The average running time of recovering a previous state is about 19 seconds on a 2.80MHz Pentium IV (without any optimization). The tool can recover all previous states until the time the generator was initialized, as is detailed below.[6]

Repeatedly applying the attack on forward security. The procedures detailed above provide a list of $O(1)$ candidate values for the state of the generator at time $t - 1$. They can of course

[6]We note that there exist much faster implementations of SHA-1, and consequently of the attack. For example, recent experiments on the Sony PS3 machine show that on that platform it is possible to compute 86-87 million invocations of SHA-1 per second (applying the function to 20 byte long inputs) [24]. In this implementation, computing 2^{23} invocations of SHA-1 should take less than 1/10 of a second. (The overall overhead of the attack is, of course, somewhat greater.)

be applied again and again, revealing the states, and consequently the outputs, of the generator at times $t-1, t-2$, etc. As for the number of false positives, in each of the attacks we have $2^\ell - 1$ possible false positives, and each of them passes the test with probability $2^{-\ell}$. The analysis of this case is identical to the analysis of the number of false positives in an attack on the forward security of the Linux random number generator (see [16], Appendix C). In that analysis it was shown that the number of false positives can be modeled as a martingale, and that its expected value at time $t-k$ is only k. (The number of false positives does not grow exponentially since for any false positive for the value of the state at time $t-k$, it happens with constant probability that the procedure detailed above does not result in any suggestion for the state at time $t-k-1$. In this case we can dismiss this false positive and should not explore its preimages.)

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

The effect of the attacks. The WRNG has no forward and backward security: an attacker which learns the state of the generator at time t can easily compute past and future states and outputs, until the times where the state is refreshed with system based entropy. Computing all states and outputs from time t up to time $t+k$ can be done in $O(k)$ work (i.e., $O(k)$ invocations of SHA-1). Computing candidates to all states and outputs from time t to time $t-k$ can be done in $O(2^{23}k^2)$ work. (I.e., in a matter of minutes, depending on k. The $O(2^{23}k^2)$ result is due to the fact that for every $1 \leq j \leq k$ we expect to find j candidate values for time $t-j$, and to each of these we apply the 2^{23} attack to learn its predecessor.) An attacker which learns the state at time t can therefore apply this knowledge to learn all states of the generator in an "attack window", which lasts from the last refresh (or initialization) of the state before time t, to the first refresh after time t.[7] As discussed above, the WRNG keeps a separate state per process, and this state is refreshed only after the generator generates 128 Kbytes of output. Therefore, we can sum up this section with the following statement:

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

In case of a process with low random bit consumption, this window might cover days of usage. In the case of Internet Explorer, we note in Section 4 that it might run 600-1200 SSL connections before refreshing the state of its WRNG. This observation essentially means that, for most users, leakage of the state of the WRNG used by Internet Explorer reveals all SSL keys used by the browser between the time the computer is turned on and the time it is turned off.

An observation about state updates. The update of the variable State in the main loop is based on exclusive-oring and adding R. More precisely, let S^t denote the value of State at the beginning of the tth iteration of the loop. Then $\mathsf{S}^{t+1} = (\mathsf{S}^t \oplus \mathsf{R}) + \mathsf{R}' + 1$, where R' is identical to R, except for the five least significant bytes which are replaced with bytes from the output of the WRNG (which might be known to an attacker). The addition and exclusive-or operations are

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

related (they are identical up to the effect of the carry, which affects addition but not the exclusive-or operation). Therefore S^{t+1} is strongly related to S^t, much more than if, say, it was defined as $\mathsf{S}^{t+1} = \mathsf{S}^t \oplus \mathsf{R}$. These relations are discussed in Section 5. (Note however that we were not able to exploit these relations in order to attack the generator.)

Similarity to the Digital Signature Standard. According to [17] the main algorithm of the WRNG is based on the PRNG used in NIST Digital Signature Standard (DSS) (also known as FIPS 186-2) [9]. We describe this algorithm in Appendix B. The authors of [17] explain that the WRNG is based on the DSS design where system entropy is replacing user input. As we describe in Appendix B the WRNG algorithm is different than the one used in DSS, and is less secure against forward security attacks.

4 Analysis II: The Interaction between the Operating System and the Generator

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

Frequency of entropy based rekeys of the state. Each process has its own copy of a WRNG instance. Since each instance of the WRNG uses eight RC4 streams, its state is refreshed only after it generates 128 Kbytes of output. Between refreshes the operation of the WRNG is deterministic. If one process (say, a web browser) uses very little pseudo-random bits, the WRNG instance that is used by this state will be refreshed very rarely, even if other processes consume many pseudo-random bits from the instances of the WRNG that they use. We described in Section 3 attacks on the forward and backward security of the WRNG which enable an attacker which observes a state of the WRNG to learn all states and outputs of the generator from the time it had its last refresh (or initialization) to the next time it will be refreshed.

Entropy based rekeys in Internet Explorer. We examined the usage of the WRNG by Internet Explorer (IE), which might be the most security sensitive application run by most users (all experiments were applied to IE 6, version 6.0.2800.1106). The examination of Internet Explorer was conducted by hooking all calls to CryptGenRandom using a kernel debugger, and recording the caller and the number of bytes produced. When IE invokes SSL it calls the WRNG through LSASS.EXE, the system security service, which is used by IE exclusively for this purpose (as mentioned before, as a service LSASS.EXE keeps its own state of the WRNG). During an SSL session, there is a varying number of requests (typically, four or more requests) for random bytes. Each request asks for 8, 16 or 28 bytes at a time. We can therefore estimate that each SSL connection consumes about 100-200 bytes of output from the WRNG. This means that the instance of the WRNG used by IE asks for a refresh only after handling about 600-1200 different SSL connections. It is hard to imagine that normal users run this number of SSL connections between the time they turn on their computer and the time they turn it off. Therefore, the attacks presented in Section 3 can essentially learn encryption keys used in all previous and future SSL connections of the attacked PC.

Initializing State and R. The variables State and R are not explicitly initialized by the generator, but rather take the last value stored in the stack location in which they are defined. This means that

in many circumstances these values can be guessed by an attacker knowledgeable in the Windows operating system. This is particularly true if the attacker studies a particular application, such as SSL or SSH, and learns the distribution of the initial values of these variables. Knowledge of these values enables an instant attack on the generator which is even more efficient than the 2^{23} attack we describe (see Section 3).

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

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

We do not know how to explain this "loose" management of the state, and cannot tell whether it is a feature or a bug. In the attacks we describe in Section 3 we show how to compute previous states assuming that State and R retain their state between invocations of the generator. These attacks are relevant even given the behavior we inspected above, for two reasons: (1) We observed that in IE the WRNG almost always retains the values of State and R. When it does not, the values of these variables seem to be rather predictable. The attacker can therefore continue with the attack until it notices that it cannot reproduce the WRNG output anymore. The attacker should then enumerate over the most likely values of State and R until it can continue the attack. (This attack requires an additional analysis of State and R, but it does seem feasible.) (2) Other applications might use the WRNG is such a way that the stack locations in which the values of State and R are stored are never overwritten.

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

The output of VeryLargeHash is used as a key for two RC4 encryption of the variables Seed and KSecDD, respectively, and the result is used to initialize the RC4 state of the WRNG. Even

if an attacker knows the values of Seed and KSecDD, they do not help it to predict the output of VeryLargeHash, and consequently predict the initialization of the RC4 state. The RC4 algorithm itself is known to be vulnerable to related key attacks, and it is known that its first output bytes are not uniformly distributed [10]. We were not able, however, to use these observations to attack the WRNG, since it applies SHA-1 to its state before outputting it.[8]

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

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

5 Analysis of the Update of State

We describe here an analysis of the update of the variable State in the main loop of the generator (the main loop is described in Figure 2.1).

The update of State is based on exclusive-oring and adding the variable R to State. Assume for a minute that the five least significant bytes of R are not set to be equal to the output of the WRNG (i.e., line 7 in Figure 2.1 is removed). Denote the ith bit of State as s_i, and the ith bit of R as r_i. Let 0_S define the set of indices for which $s_i = 0$. State is advanced in the following way:

$$\begin{aligned}
\mathsf{State} &= (\mathsf{State} \oplus \mathsf{R}) + \mathsf{R} + 1 \\
&= (\sum_{i \in 0_S} 2^i r_i + \sum_{i \notin 0_S} 2^i (1 - r_i)) + \sum_{i=0 \ldots 159} 2^i r_i + 1 \\
&= \mathsf{State} + 2 \cdot \sum_{i \in 0_S} 2^i r_i + 1 \\
&= \mathsf{State} + 2 \cdot (\mathsf{R} \ \wedge \ \neg \mathsf{State}) + 1
\end{aligned}$$

(Where $\wedge$ denotes bit-wise AND, and $\neg$ denotes bit-wise negation.)

Now, let us analyze the actual update of State. Denote by $\mathsf{State_L}$ the 120 most significant bits of State, and by $\mathsf{State_R}$ its 40 least significant bits. Denote $\mathsf{R_L}, \mathsf{R_R}$ and out_R similarly. Then State is advanced in the following way (where '|' denotes concatenation):

$$\mathsf{State_L} \mid \mathsf{State_R} = [\mathsf{State_L} + 2 \cdot (\mathsf{R_L} \ \wedge \ \neg \mathsf{State_L})] \underbrace{\mid}_{\leftarrow \text{carry}} [(\mathsf{State_R} \oplus \mathsf{R_R}) + \mathsf{out_R} + 1]$$

[8] We note that the distribution of the first output bytes of RC4 is known to be slightly biased, and the output of the WRNG is computed by applying SHA-1 to a function of RC4, State and R. Therefore, an attacker which knows the values of State and R knows that there is a slight bias in the distribution of the output of the WRNG. However, this bias seems to be too weak to be useful.

Note that the only bits of R_L which affect the result are those which correspond to 0 bits of State. In addition, the carry bit resulting from the addition of rightmost 40 bits might affect the left 120 bits.

As for R, it is updated in the following way, where out is the output of the generator, and RC is the value returned by the function `get_next_20_rc4_bytes` which advances the RC4 ciphers.

$$\mathsf{R_L}|\mathsf{R_R} \;=\; [\mathsf{R_L} \oplus \mathsf{RC_L}] \mid \mathsf{out_R}$$

Note that an observer which has access to the output of the generator knows the rightmost 40 bits of R.

The previous two observations demonstrate the some bits of R do not affect the update of State. On average, only half the bits of R (or R_L) affect the update of State. It is not clear, however, how to use these observations in order to attack the generator. Assume for example that we know $\mathsf{S^t}, \mathsf{R^t}$, the values of State and R at time t, that we also have access to the output of the generator, but that we do not know the output of the RC4 streams. Let $|0_{S_L}|$ denote the number of bits equal to 0 among the bits of $\mathsf{S_L^t}$. Then in the next step there are at most $2^{|0_{S_L}|+1}$ options for the value of $\mathsf{S_L^{t+1}}$, and 2^{40} possible values for $\mathsf{S_R^{t+1}}$. (If $\mathsf{S^t}$ is random we can expect only 2^{100} possible values of $\mathsf{S^{t+1}}$. We can make this number even lower if we can control the number of 0 bits in State to be low, as is possible if we can control the initialization of State (which is, as argued in Section 4, possible in many scenarios.) As for R, there are 2^{120} options for $\mathsf{R_L^{t+1}}$, whereas knowledge of the output uniquely defines $\mathsf{R_R^{t+1}}$.

All this information does not help a lot. The output of the next iteration is defined as $\mathsf{SHA\text{-}1}(\mathsf{S^{t+1}} \oplus \mathtt{R}^{\mathsf{t+1}} \oplus \mathtt{RC}^{\mathsf{t+1}})$, and the unknown value of $\mathsf{RC^{t+1}}$ masks all the information we have about $\mathsf{S^{t+1}}$ and $\mathsf{R^{t+1}}$. If we want to examine the number of possibilities for $\mathsf{S^{t+2}}$ we have to apply the previous calculation starting from, say, 2^{60} possibilities for S_L^{t+1}, and therefore end up with close to 2^{160} options for S^{t+2}.

6 Comparison with the Linux PRNG

The pseudo-random number generator used in the Linux operating system (denoted LRNG) was analyzed in [16]. The analysis of the WRNG shows that it differs from the LRNG in several major design issues.

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

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

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

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

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

- *Efficiency of attacks.* The best forward security attack on the LRNG requires $O(2^{64})$ work. The attack on the forward security of the WRNG is therefore more efficient by a factor of about 2^{40} (it has an overhead of $O(2^{23})$ compared to $O(2^{64})$).

  Security implication: The impact of the previous four properties is that forward and backward security attacks are more severe when applied to the WRNG. The attacks are more efficient by twelve orders of magnitude. They reveal the outputs of the generator between consecutive reseedings, and these reseedings are much more rare in the case of the WRNG. In some cases, reseeding the LRNG happens every few seconds, while the WRNG is reseeded every few days, if it is reseeded at all.

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

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

7 Conclusions

7.1 Conclusions

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

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

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

WRNG versus LRNG. We compared between the pseudo-random generators used by Windows and Linux (WRNG vs. LRNG). The forward security attack on the WRNG is faster by a factor of

$O(2^{40})$ compared to the attack on the LRNG. In addition, our findings show that the LRNG has better usage of operating system entropy, uses asynchronous entropy feedings, uses the extraction process as an entropy source, and shares its output between multiple processes. As a result, a forward security attack on the WRNG reveals longer sequences of generator output, compared to an attack on the LRNG.

7.2 Recommendations

Forward security. The most obvious recommendation is to change the algorithm used by the WRNG to one which provides forward security. This can be done by making local changes to the current implementation of the generator, or by replacing RC4 with a function which provides forward security. Alternatively, it is possible to use the transformation of [4] which transforms any standard generator to one providing forward security. We believe however that it is preferable to replace the entire algorithm used by the generator with a simpler algorithm which is rigorously analyzed. A good approach is to *adopt the Barak-Halevi construction.* That construction, suggested in [2], is a simple yet powerful construction of entropy based PRNGs. Its design is much simpler to implement than the current WRNG implementation and, assuming that its building blocks are secure, it provably preserves both forward and backward security. It can be implemented using, say, AES and a simple entropy extractor.

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

7.3 Open Problems

Extending our research to additional Windows platforms. Our entire research was conducted on a specific Windows 2000 build. We did several early checks on additional binary versions of Windows but that work is only in its beginning. The important operating systems to examine are the main Windows releases such as Windows XP and Windows Vista, as well as systems which have fewer sources of entropy, such as Windows CE.

State initialization. As we stated in our analysis, the internal RC4 states are initialized and rekeyed with very similar entropy parameters. These are hashed by a procedure which uses SHA-1 and propagates a change in the value of a single input bit to all output bits. The result of this procedure initializes the RC4 algorithm. We were not able to use this finding, but it seems that additional research is needed here. The research should examine the different entropy sources and the hashing algorithm, and check if they result in any related key attack on RC4. We also noted that the state variables State and R are not explicitly initialized but rather take the current values stored in the stack. More research is needed to examine in detail the distribution of these values.

Acknowledgments

We would like to thank Dag Arne Osvik and Werner Schindler for their helpful comments regarding this paper.

References

[1] AIS 20: Functionality classes and evaluation methodology for deterministic random number generators. Application Notes and Interpretations of the Scheme (AIS) Version 1, Bundesamt für Sicherheit in der Informationstechnik, December 1999. `http://www.bsi.bund.de/zertifiz/zert/interpr/ais20e.pdf`.

[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] Donald Beaver and Stuart Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Advances in Cryptology - Eurocrypt '92*, pages 307–323, Berlin. Springer-Verlag. LNCS Vol. 658.

[4] Mihir Bellare and Bennet S. Yee. Forward-security in private-key cryptography. In Marc Joye, editor, *CT-RSA*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2003.

[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] 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.

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

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

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

[10] 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.

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

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

[13] 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`.

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

[15] Zvi Gutterman and Dahlia Malkhi. Hold your sessions: An attack on java session-id generation. In *CT-RSA*, volume 3376 of *LNCS*, pages 44–57. Springer-Verlag, 2005.

[16] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *S&P*, pages 371–385. IEEE Computer Society, 2006.

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

[18] Borislav H. Simov Jose Castejon-Amenedo, Richard McCue. Extracting randomness from external interrupts. In *The IASTED Int. Conf. on Communication, Network, and Information Security*, pages 141–146, 2003.

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

[20] 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*, volume 1758 of *LNCS*, pages 13–33. Springer, 1999.

[21] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption*, volume 1372 of *LNCS*, pages 168–188. Springer, 1998.

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

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

[24] Dag Arne Osvik. personal communication, 2007.

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

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

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

A RC4

RC4 is a stream cipher. Its initialization process is defined in Figure A.1. The process of generating output is defined in Figure A.2.

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

```
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 A.1: RC4 Key Scheduling Algorithm (KSA). The array *key* holds the key, *keylength* is the key size in bytes.

```
i := 0
j := 0
while GeneratingOutput:
   i := (i + 1) mod 256
   j := (j + S[i]) mod 256
   swap(S[i],S[j])
   output S[(S[i] + S[j]) mod 256]
```

Figure A.2: RC4 pseudo random generation algorithm. The output is xored with the clear text for encryption.

```
   swap(S[i],S[j])
   j := (j - S[i]) mod 256
   i := (i - 1) mod 256
```

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

B FIPS 186-2

The FIPS 186-2 standard (Appendix 3.1, Random Number Generation for the Digital Signature Standard) describes the following pseudo-random number generator. The variable x_i denotes the output of each iteration, G is a function similar to SHA-1 and XKey is initialized with some chosen secret value. XSEED is an external input, which might be a user input in the context of FIPS 186-2. In the context of the WRNG it has a similar role to that of the RC4 output which is fed into the state.

The state of the generator is advanced in the following way:

$\texttt{XSEED}_i$ = optional user input
$\texttt{XVAL} = (\texttt{XKey} + \texttt{XSEED}_i) \bmod 2^b$
$x_i = \texttt{G(t, XVAL)} \bmod \texttt{q}$
$\texttt{XKEY} = (1 + \texttt{XKEY} + x_i) \bmod 2^b$

We do not see any close relation between this algorithm and the one implemented in the WRNG. Unlike the WRNG, the FIPS 182-2 algorithm seems to provide forward security: Suppose that an attacker knows $\texttt{XKEY}_\texttt{i}$ and $\texttt{XSEED}_\texttt{i}$ and wishes to compute $\texttt{XKEY}_{\texttt{i}-1}$. Then $\texttt{XKEY}_\texttt{i} = 1 + \texttt{XKEY}_{\texttt{i}-1} + \texttt{G}(\texttt{t}, \texttt{XKEY}_{\texttt{i}-1} + \texttt{XSEED}_\texttt{i})$. If G is modeled as a random function, then the only possible attack is to enumerate over all possible values of $\texttt{XKEY}_{\texttt{i}-1}$, and is therefore inefficient.

Table 1: Entropy sources

Source	Size in 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