Randomness and the Netscape Browser

How secure is the World Wide Web?

Ian Goldberg and David Wagner

Ian and David are PhD students in the computer science department at the University of California, Berkeley. They can be reached at iang@cs.berkeley.edu *or* daw@cs.berkeley.edu.

As the World Wide Web gains broad public appeal, companies are becoming interested in using the Web not just to advertise, but also to take orders for their merchandise and services. Since ordering a product online requires the customer to transmit payment information (such as a credit-card number) from a client program to the company's server program through the Internet, there's need for cryptographic protection. By encrypting payment information before transmitting it, a customer can ensure that no one except the company from which he is purchasing can decode that sensitive data.

Netscape Communications has been at the forefront of the effort to integrate cryptographic techniques into Web servers and browsers. Netscape's Web browser supports the Secure Sockets Layer (SSL), a cryptographic protocol developed by Netscape to provide secure Internet transactions. Given the popularity of Netscape's browser and the widespread use of its cryptographic protocol on the Internet, we decided to study Netscape's SSL implementation in detail.

Our study revealed serious flaws in Netscape's implementation of SSL that make it relatively easy for an eavesdropper to decode the encrypted communications. Although Netscape has fixed these problems in a new version of their browser (as of this writing, Netscape 2.0 beta1 and Netscape Navigator 1.22 Security Update are available), these weaknesses provide several lessons for people interested in producing or purchasing secure software.

Back to Basics

At its most basic level, SSL protects communications by encrypting messages with a secret key--a large, random number known only to the sender and receiver. Because you can't safely assume that an eavesdropper doesn't have complete details of the encryption and decryption algorithms, the protocol can be considered secure only if someone who knows all of the details of these algorithms is unable to recover a message without trying every possible key. Ultimately, security rests on the infeasibility of trying all possible decryption-key values.

The security of SSL, like that of any other cryptographic protocol, depends crucially on the unpredictability of this secret key. If an attacker can predict the key's value or even narrow down the number of keys that must be tried, the protocol can be broken with much less effort than if truly random keys had been used. Therefore, it is vital that the secret keys be generated from an unpredictable random-number source.

Randomness is not a black-and-white quality: some streams of numbers are more random than others. The only truly random number sources are those related to physical phenomena such as the rate of radioactive decay of an element or the thermal noise of a semiconductor diode. Barring the use of external devices, computer programs that need random numbers must generate these numbers themselves. However, since CPUs are deterministic, it is impossible to algorithmically generate truly random numbers.

Many common computer applications (games, for instance) use any readily available source of randomness to provide an initial value, called a "seed," to a pseudorandom number generator (PRNG). PRNGs operate by repeatedly scrambling the seed. Typically, the seed is a short, random number that the PRNG expands into a longer, random-looking bitstream. A typical game might seed a PRNG with the time of day; see Figure 1.

For a simple game, the seed only needs to change each time the game is run. Though the seed will be predictable, this is not a major concern in applications where security is not an issue. In cryptographic applications, however, the seed's unpredictability is essential--if the attacker can narrow down the set of possible seeds, his job is made significantly easier. Since the function used by the PRNG to turn a seed into a pseudorandom number sequence is assumed to be known, a smaller set of possible seeds yields a correspondingly small set of sequences produced by the PRNG.

A good method to select seed values for the PRNG is an essential part of a cryptographic system such as SSL. If the seed values for the PRNG can easily be guessed, the level of security offered by the program is diminished significantly, since it requires less work for an attacker to decrypt an intercepted message.

Netscape's Implementation

Because Netscape would not release detailed information about this section of its program, we resorted to the tedious task of reverse-engineering Netscape's algorithm by manually decompiling their executable program.

The method Netscape uses to seed its PRNG is shown in pseudocode in Figure 2. This algorithm was derived from Version 1.1 of the international version of Netscape's Solaris 2.4 browser. Most other versions of Netscape for UNIX use the same algorithm; the Microsoft Windows and Macintosh versions have slightly different details (for example, they use a particular system timer instead of the process ID), but the techniques employed are fundamentally the same across all architectures and operating systems.

In Figure 2, it's important to note that *mklcpr()* and *MD5()* are fixed, unkeyed algorithms that will presumably be known by an adversary. The seed generated depends only on the values of *a* and *b*, which in turn depend on just three quantities: the time of day, the process ID, and the parent process ID. Thus, an adversary who can predict these three values can apply the well-known MD5 algorithm to compute the exact seed generated.

Figure 3 shows the key-generation algorithm, also reverse-engineered from Netscape's browser. An attacker who can guess the PRNG seed value can easily determine the encryption keys used in Netscape's secure transactions.

Attacks on Netscape

Unfortunately for Netscape, U.S. regulations prohibit the export of products incorporating strong cryptography. In order to distribute an international version of its browser overseas, Netscape had to weaken the encryption scheme to use keys of just 40 bits, leaving only a million million possible key values. That may sound like a lot of numbers to try, but several people (David Byers, Eric Young, Damien Doligez, Piete Brooks, Andrew Roos, Adam Back, Andy Brown and many others) have been able to try every possible key and recover SSL-encrypted data in as few as 30 hours using spare CPU cycles from many machines. Since nearly all Netscape browsers in use are the free international version, the success of this attack demonstrates a fundamental vulnerability in Netscape that cannot be repaired under current export regulations.

We used the information we uncovered about Netscape's internals to formulate a more intelligent attack against Netscape's encryption scheme. According to Figure 2 and Figure 3, each possibility for the time of day, process ID, and parent-process ID produces a unique seed, which in turn produces a unique encryption key.

When a connection is first established, a challenge value is calculated and sent unencrypted from the Netscape client to the secure server. This allows an attacker to learn that value, which will be useful later.

Netscape's UNIX browsers are more difficult to attack than its browsers for other platforms, since the seeding process used in the UNIX browsers utilizes more-random quantities than does the process used in the browsers for other platforms. We will only discuss attacks on the UNIX browsers; it should be apparent from this discussion how to attack the other versions.

An attacker who has an account on the UNIX machine running the Netscape browser can easily discover the *pid* and *ppid* values used in *RNG_CreateContext()* using the ps command (a utility that lists the process IDs of all processes on the system).

All that remains is to guess the time of day. Most popular Ethernet sniffing tools (including *tcpdump*) record the precise time they see each packet. Using the output from such a program, the attacker can guess the time of day on the system running the Netscape browser to within a second. It is probably possible to improve this guess

significantly. This recovers the *seconds* variable used in the seeding process. (There may be clock skew between the attacked machine and the machine running the packet sniffer, but this is easy to detect and compensate for.)

Of the variables used to generate the seed in Figure 2 (*seconds*, *microseconds*, *pid*, *ppid*), we know the values of *seconds*, *pid*, and *ppid*; only the value of the *microseconds* variable remains unknown. However, there are only one million possible values for it, resulting in only one million possible choices for the seed. We can use the algorithm in Figure 3 to generate the *challenge* and *secret_key* variables for each possible seed. Comparing the computed challenge values to the one we intercepted will reveal the correct value of the secret key. Testing all one million possibilities takes about 25 seconds on an HP 712/80.

Our second attack assumes the attacker does not have an account on the attacked UNIX machine, which means the *pid* and *ppid* quantities are no longer known. Nonetheless, these quantities are rather predictable, and several tricks can be used to recover them.

The unknown quantities are mixed in a way which can cancel out some of the randomness. In particular, even though the *pid* and *ppid* are 15-bit quantities on most UNIX machines, the sum *pid + (ppid << 12)* has only 27 bits, not 30 (see Figure 2). If the value of *seconds* is known, *a* has only 20 unknown bits, and *b* has only 27 unknown bits. This leaves, at most, 47 bits of randomness in the secret key--a far cry from the 128-bit security claimed by the domestic U.S. version.

A little cleverness can reduce the uncertainty even further. First, *ppid* is often 1 (for example, when the user starts Netscape from an X Window system menu); if not, it is usually just a bit smaller than the *pid*. Furthermore, process IDs are not considered secret information by most applications, so some programs will leak information about them. For example, the popular mail-transport agent *sendmail* generates Message-IDs for outgoing mail using its process ID; as a result, sending e-mail to an invalid user on the attacked machine will cause the message to bounce back to the sender; the Message-ID contained in the reply message will tell the sender the last process ID used on that machine. Assuming that the user started Netscape relatively recently, and that the machine is not heavily loaded, this will closely approximate Netscape's *pid*. These observations mean that the amount of unpredictability in the *pid* and *ppid* quantities is quite small.

The most unsophisticated attack on any encryption scheme is to try all possible key values by brute force. Naturally, this approach can be expected to take a long time. For the domestic U.S. version of the Netscape browser, trying every possible 128-bit key is absolutely infeasible. However, the problems with Netscape's seed-generation process make it possible to speed up this process by trying only the keys generated by the possible seed values. Optimizations such as those described earlier should allow even a remote attacker to break Netscape's encryption in a matter of minutes.

Future Impact

Using these weaknesses, we were able to successfully attack Version 1.1 of the Netscape browser. All UNIX versions of the browser are vulnerable. Netscape has confirmed that the Microsoft Windows and Macintosh versions are also subject to this attack. Both the international version (with 40-bit keys) and the domestic version (with 128-bit keys) are vulnerable. In fact, the private keys used by the Netscape server software may be susceptible to this attack as well.

Very soon after we announced these attacks, Netscape responded with a new version of the browser, which uses more randomness in producing the encryption keys. Since only the older versions are vulnerable, the direct, long-term impact of our attack should be small. Still, we can learn several lessons from this experience.

The Achilles heel of Netscape's security was the way in which it generated random numbers. The cryptography community has long known that generating random numbers requires great care and is easy to do poorly; Netscape learned this lesson somewhat painfully. If you need to generate random numbers for cryptographic purposes, be very careful.

In a narrow sense, the security flaw we found in the Netscape browser serves merely as an anecdote to emphasize the difficulty of generating cryptographically strong random numbers. But there's a broader moral to the story. The security community has painfully learned that small bugs in a security-critical module of a software system can have serious consequences, and that such errors are easy to commit. The only way to catch these mistakes is to expose the source code to scrutiny by security experts.

Peer review is essential to the development of any secure software. Netscape did not encourage outside auditing or peer review of its software--and that goes against everything the security industry has learned from past mistakes. By extension, without peer review and intense outside scrutiny of Netscape's software at the source-code level, there is simply no way consumers can know where there will be future security problems with Netscape's products.

Interestingly, Jim Bidzos of RSA Data Security reports that he offered to review Netscape's security before its initial release, but that Netscape declined. Now the company has changed its tune. "They're asking us to review it this time," Bidzos said.

Since we announced our attack, Netscape has publicly released the source code to its patch for independent scrutiny. But the company has not made available for public review any other security-critical modules of their programs. Until they learn their lesson and open their security programming to public evaluation, many security experts will remain justifiably skeptical of the company's security claims.

The growth of Internet commerce opens wonderful new opportunities for both businesses and consumers, but these opportunities can be safely exploited only after all parties are satisfied with the security of their online financial transactions. We are concerned that companies are hiding information about their security modules and shunning public

review. We hope that the lessons learned from this incident will encourage software companies to openly embrace in-depth public scrutiny of their security software as both a natural and necessary part of the software-development cycle.

Resources and References

A number of resources are available to help programmers in generating cryptographically strong random numbers. One of the best is Colin Plumb's article "Truly Random Numbers" (*Dr. Dobb's Journal*, November 1994), which provides both source code and a discussion of his technique.

Another helpful resource is "Randomness Recommendations for Security" (RFC 1750). (RFCs are widely available memos detailing information of broad relevance to the Internet community.) Also, Bruce Schneier's book *Applied Cryptography* (John Wiley & Sons, 1994) offers a helpful introductory discussion of randomness.

Finally, we have collected links to many resources for generating cryptographically strong random numbers on a supplemental World Wide Web page at http://www.cs.berkeley.edu/~daw/netscape-randomness.html, which includes links to:

- Ron Rivest's "The MD5 Message-Digest Algorithm" (RFC 1321) describes this popular mixing function in detail.
- Adam Back, David Byers, and Eric Young's "Another SSL breakage...". Post to cypherpunks mailing list, August 15, 1995.
- Damien Doligez's "SSL challenge--broken!". Post to cypherpunks mailing list, August 15, 1995.
- Marc Van Heyningen's "Re: What's the Netscape problem." Post to www-security mailing list, September 20, 1995.
- "International Traffic in Arms Regulations," 22 CFR 120-130. Federal Register, vol. 58 no. 139. July 22, 1993.
- Sameer Parekh's "Community ConneXion Corrects Inaccuracies in Netscape Press Release."

Acknowledgments

We owe tremendous appreciation to David Oppenheimer for his thorough commentary on an early draft of this article; his help has greatly improved the explanation of our attack.

Figure 1: A typical C program that uses a PRNG.

```
srand(time(0));
 ...
printf("You rolled the die, and got a %d.\n", 1 + (rand()%6));
```

Figure 2: The Netscape 1.1 seeding process: pseudocode.

```
global variable seed;
```

```
RNG_CreateContext()
   (seconds, microseconds) = time of day; /* Time elapsed since 1970 */
   pid = process ID;  ppid = parent process ID;
   a = mklcpr(microseconds);
   b = mklcpr(pid + seconds + (ppid << 12));
   seed = MD5(a, b);
mklcpr(x) /* not cryptographically significant; shown for completeness
*/
   return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);
MD5() /* a very good standard mixing function, source omitted */
```

Figure 3: The Netscape v1.1 key-generation process: pseudocode.

```
RNG_GenerateRandomBytes()
     x = MD5(seed);
     seed = seed + 1;
     return x;
global variable challenge, secret_key;
create_key()
     RNG_CreateContext();
     tmp = RNG_GenerateRandomBytes();
     tmp = RNG_GenerateRandomBytes();
     challenge = RNG_GenerateRandomBytes();
     secret_key = RNG_GenerateRandomBytes();
```