Recipes for Cryptography, Authentication, Networking, Input Validation & More

Covers Unix & Windows

Secure Programming Cookbook

for C and C++

O'REILLY®

John Viega, Zachary Girouard & Matt Messier

Secure Programming Cookbook

for C and C++

Other computer security resources from O'Reilly

Related titles

- 802.11 Security
- Building Internet Firewalls
- Computer Security Basics
- Java Cryptography
- Java Security
- Linux Security Cookbook
- Network Security with OpenSSL
- Practical Unix and Internet Security
- Secure Coding: Principles & Practices
- Securing Windows NT/2000 Servers for the Internet
- SSH, The Secure Shell: The Definitive Guide
- Web Security, Privacy, and Commerce
- Database Nation
- Building Secure Servers with Linux

Security Books Resource Center

security.oreilly.com is a complete catalog of O'Reilly's books on security and related technologies, including sample chapters and code examples.

oreillynet.com is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

Conferences

O'Reilly & Associates brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

Secure Programming Cookbook

for C and C++

John Viega and Matt Messier

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 11

Random Numbers

Security-critical applications often require well-chosen random numbers, for purposes ranging from cryptographic key generation to shuffling a virtual deck of cards. Even though problems with random numbers seem as if they should be few and far between, such problems are disturbingly common. Part of the problem is that computers are fundamentally deterministic and therefore are not very good at doing anything unpredictable. However, input from a user can introduce real randomness into a system.

This chapter discusses how to get secure random numbers for your application. We describe how to take a single, secure, random number (a seed), and stretch it into a big stream of random numbers using a secure pseudo-random number generator. We talk about how to get random data in lots of different representations (e.g., an integer in a particular range or a printable string). We also discuss how to get real randomness in an environment that is fundamentally deterministic, and we give advice on figuring out how to estimate how much randomness exists in a piece of data.

11.1 Determining What Kind of Random Numbers to Use

Problem

Your application has a need for random numbers. You must figure out what you need to do to get adequate randomness as cheaply as possible, yet still meet your security properties. To do that, you need to understand what kinds of options are available to you and what the trade-offs are.

Solution

There are essentially three classes of solutions:

Insecure random number generators
: More properly, these are noncryptographic pseudo-random number generators. You should generally assume that an attacker could predict the output of such a generator.

Cryptographic pseudo-random number generators (PRNGs)
: These take a single secure seed and produce as many unguessable random numbers from that seed as necessary. Such a solution should be secure for most uses as long as a few reasonable conditions are met (the most important being that they are securely seeded).

Entropy harvesters
: These are sometimes "true" random number generators—although they really just try to gather entropy from other sources and present it directly. They are expected to be secure under most circumstances, but are generally incredibly slow to produce data.

For general-purpose use, the second solution is excellent. Typically, you will need entropy (i.e., *truly* random data) to seed a cryptographic pseudo-random number generator and will not need it otherwise, except in a few specific circumstances, such as when generating long-term keys.

You should generally avoid the first solution, as the second is worthwhile even when security is not an issue (particularly because we've seen numerous systems where people assumed that the security of their random numbers wasn't an issue when it actually turned out to be).

Entropy is highly useful in several situations. First, there's the case of seeding a random number generator, where it is critical. Second, any time where you would like information-theoretic levels of security (i.e., absolutely provable secrecy, such as is theoretically possible with a one-time pad), then cryptographic randomness will not do. Third, there are situations where a PRNG cannot provide the security level required by a system. For example, if you want to use 256-bit keys throughout your system, you will need to have 256 bits of entropy on hand to make it a full-strength system. If you try to leverage an OS-level PRNG (e.g., */dev/random* on Unix systems), you will not get the desired security level, because such generators currently never produce data with more than 160 bits of security (many have a 128-bit ceiling).

In addition, a combination of the second and third class of solution is often a good practical compromise. For example, you might want to use entropy if it is available, but if it is not, fall back on a cryptographic solution. Alternatively, you might want to use a cryptographic solution that occasionally gets its seed changed to minimize the chance of a compromise of the internal state of the generator.

Note that cryptographic pseudo-random number generators always produce an identical stream of output when identically seeded. If you wish to repeat a stream of numbers, you should avoid reseeding the generator (or you need to do the exact same reseeding at the exact right time).

Discussion

Most common "random number generators," which we will call *noncryptographic pseudo-random number generators*, are not secure. They start with a seed (which needs to be random in and of itself to have any chance of security) and use that seed to produce a stream of numbers that look random from the point of view of a statistician who needs random-looking but reproducible streams of data.

From the point of view of a good cryptographer, though, the numbers produced by such a generator are not secure. Generally, noncryptographic generators leak information about their internal state with each output, meaning that a good cryptographer can start predicting outputs with high accuracy after seeing a few random numbers. In a real system, you generally do not even need to see the outputs directly, instead inferring information about the outputs from the behavior of the program (which is generally made even easier with a bit of reverse engineering of the program).

Traditional noncryptographic pseudo-random number generators include the `rand()` and `random()` functions you'd expect to see in most libraries (so-called linear congruential generators). Other noncryptographic generators include the "Mersenne Twister" and linear feedback shift registers. If a random number generator is not advertised as a cryptographic random number generator, and it does not output high-entropy data (i.e., if it stretches out a seed instead of harvesting randomness from some external input to the machine), do not use it.

Cryptographic pseudo-random number generators are still predictable if you somehow know their internal state. The difference is that, assuming the generator was seeded with sufficient entropy and assuming the cryptographic algorithms have the security properties they are expected to have, cryptographic generators do not quickly reveal significant amounts of their internal state. Such generators are capable of producing a lot of output before you need to start worrying about attacks.

In the context of random number generation, *entropy* refers to the inherent "unknowability" of inputs to external observers. As we discuss in Recipe 11.19, it is essentially impossible to determine how unknowable something is. The best we can do is to establish conservative upper limits, which is, in and of itself, quite difficult.

If a byte of data is truly random, then each of the 2^8 (256) possibilities are equally likely, and an attacker would be expected to make 2^7 guesses before correctly identifying the value. In this case, the byte is said to contain 8 bits of entropy (it can contain no more than that). If, on the other hand, the attacker somehow discovered that

the byte is even, he reduces the number of guesses necessary to 2^7 (128), in which case the byte has only 7 bits of entropy.

We can have fractional bits of entropy. If we have one bit, and it has a 25% chance of being a 0 and a 75% chance of being a 1, the attacker can do 50% better at guessing it than if the bit were fully entropic. Therefore, there is half the amount of entropy in that bit.

In public key cryptography, *n*-bit keys contain far fewer than *n* bits of entropy. That is because there are not 2^n possible keys. For example, in RSA, we are more or less limited by the number of primes that are *n* bits in size.

Random numbers with lots of entropy are difficult to come by, especially on a deterministic computer. Therefore, it is generally far more practical to gather enough entropy to securely seed a cryptographic pseudo-random number generator. Several issues arise in doing so.

First, how much entropy do you need to seed a cryptographic generator securely? The short answer is that you should try to give as much entropy as the random number generator can accept. The entropy you get sets the maximum security level of your data protected with that entropy, directly or indirectly. For example, suppose you use 256-bit AES keys, but chose your key with a PRNG seeded with 56 bits of entropy. Any data encrypted with the 256-bit AES key would then be no more secure than it would have been had the data been encrypted with a 56-bit DES key.

Then again, it's incredibly hard to figure out how much entropy a piece of data contains, and often, estimates that people believe to be conservative are actually large overestimates. For example, the digits of π appear to be a completely random sequence that should pass any statistical test for randomness with flying colors. Yet they are also completely predictable.

We recommend that if you have done a lot of work to figure out how much entropy is in a piece of data and you honestly think you have 160 bits there, you still might want to divide your estimate by a factor of 4 to 8 to be conservative.

Because entropy is so easy to overestimate, you should generally cryptographically postprocess any entropy collected (a process known as *whitening*) before using it. We discuss whitening in Recipe 11.16.

Second, most cryptographic pseudo-random number generators take a fixed-size seed, and you want to maximize the entropy in that seed. However, when collecting entropy, it is usually distributed sparsely through a large amount of data. We discuss methods for turning data with entropy into a seed in Recipe 11.16. If you have an entropy source that is supposed to produce good random numbers (such as a hardware generator), you should test the data as discussed in Recipe 11.18.

Tips on Collecting Entropy

Follow these guidelines when collecting entropy:

- Make sure that any data coming from an entropy-producing source is postprocessed with cryptography to remove any lingering statistical bias and to help ensure that your data has at least as many bits of entropy input as bits you want to output. (See Recipe 11.16.)
- Make sure you use enough entropy to seed any pseudo-random number generator securely. Try not to use less than 128 bits.
- When choosing a pseudo-random number generator, make sure to pick one that explicitly advertises that it is cryptographically strong. If you do not see the word "cryptographic" anywhere in association with the algorithm, it is probably not good for security purposes, only for statistical purposes.
- When selecting a PRNG, prefer solutions with a refereed proof of security bounds. Counter mode, in particular, comes with such a proof, saying that if you use a block cipher bit with 128-bit keys and 128-bit blocks seeded with 128 bits of pure entropy, and if the cipher is a pseudo-random permutation, the generator should lose a bit of entropy after 2^{64} blocks of output.
- Use postprocessed entropy for seeding pseudo-random number generators or, if available, for picking highly important cryptographic keys. For everything else, use pseudo-randomness, as it is much, much faster.

Finally, you need to realize that even properly used cryptographic pseudo-random number generators are only good for a certain number of bytes of output, though usually that's a pretty large number of bytes. For example, AES in counter (CTR) mode (when used as a cryptographic pseudo-random number generator) is only good for about 2^{64} bytes before reseeding is necessary (granted, this is a very large number).

There are situations where you may want to use entropy directly, instead of seeding a cryptographic pseudo-random number generator, particularly when you have data that needs to be independently secured. For example, suppose you are generating a set of ten keys that are all very important. If we use a PRNG, the maximum security of all the keys combined is directly related to the amount of entropy used to seed the PRNG. In addition, the security decreases as a potential attacker obtains more keys. If a break in the underlying PRNG algorithm were to be found, it might be possible to compromise all keys that have ever been issued at once!

Therefore, if you are generating very important data, such as long-term cryptographic keys, generate those keys by taking data directly from an entropy source if possible.

See Also

Recipes 11.16, 11.18, 11.19

11.2 Using a Generic API for Randomness and Entropy

Problem

You would like to have a standard API for getting cryptographic randomness or entropy, which you can then bind to any underlying implementation. Many recipes in this book rely on random numbers and use the API in this recipe without concern for what implementation is behind it.

Solution

The API in this recipe is exactly what you need. In this recipe, we show the API and how to use it. In the next few recipes, we discuss how to bind it to third-party randomness infrastructures.

Discussion

At an API level, this recipe is only going to look at how to fill a buffer with random bytes. To get random values for other data types, see Recipes 11.10 through 11.14.

Here we are going to build a random number generation API where there is only a single generator per application, or perhaps even a single generator for the entire machine. Either way, we expect that the application will have to initialize the API. Note that the initialization may need to seed a cryptographic pseudo-random number generator, so the initialization part might hang. If that is a problem, launch a thread to call the initialization routine, but be aware that asking for any cryptographically strong pseudo-random numbers at all will cause your program to abort if the system has not been initialized. The initialization routine is simply:

```
void spc_rand_init(void);
```

Because we know well that people will often forget to perform initialization, implementations of this API should automatically check to see if this routine has been called when using other API calls, and call it at that point if not.

After initialization, we will provide two universally available options for reading data, as well as a third option that will not always be available:

- Get cryptographically strong random numbers, as generated from a well-seeded pseudo-random number generator.

- Get entropy if it is available, and if it is not, fall back on cryptographically strong random numbers (using any available entropy).
- Get data that should be highly entropic that has never passed through a pseudo-random number generator. Note that this function is not always available and that it will hang until enough entropy is available.

The first function, which always produces cryptographically strong randomness, has the following signature:

```
unsigned char *spc_rand(unsigned char *buf, size_t b);
```

It places `b` bytes into memory, starting at the location `buf`, and returns `buf` (this is done to minimize the chance of someone misusing the API). This function always returns unless it causes your program to abort, which it does only if `spc_rand_init()` has never successfully returned.

The second function, which returns entropy if it is available, and otherwise produces cryptographically strong randomness, has the following signature:

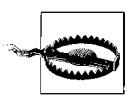
```
unsigned char *spc_keygen(unsigned char *buf, size_t b);
```

The arguments are the same as for `spc_rand()`. The name change reflects the fact that this is meant to be the function you will generally use for generating long-term key material, unless you want to insist that key material come directly from entropy, in which case you should use the `spc_entropy()` function. For all other uses, we recommend using `spc_rand()`.

The `spc_entropy()` function mimics the first two functions:

```
unsigned char *spc_entropy(unsigned char *buf, size_t b);
```

However, note that this function will block until it has enough entropy collected to fill the buffer. For Windows, this function is only usable using the code in this book if you use EGADS, as discussed in Recipe 11.8.



The functions `spc_keygen()` and `spc_entropy()` should cryptographically postprocess (whiten) any entropy they use before outputting it, if that's not already done by the underlying entropy sources. Often, it will be done for you, but it will not hurt to do it again if you are not sure. (See Recipe 11.16 for how to do it.)

See Also

Recipes 11.8, 11.10, 11.11, 11.12, 11.13, 11.14, 11.16

11.3 Using the Standard Unix Randomness Infrastructure

Problem

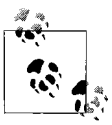
You want to use random numbers on a modern-day Unix machine.

Solution

On most modern Unix systems, there are two devices from which you can read: */dev/random*, which is expected to produce entropy, and */dev/urandom*, which is expected to provide cryptographically secure pseudo-random values. In reality, these expectations may not always be met, but in practice, it seems reasonably safe to assume that they are.

We strongly recommend accessing these devices through the API we present in Recipe 11.2.

Discussion



If you need a cryptographically strong random number source that is nonetheless reproducible, */dev/random* will not suit your purposes. Use one of the other PRNGs discussed in this chapter.

Most modern Unix operating systems have two devices that produce random numbers: */dev/random* and */dev/urandom*. In theory, */dev/random* may block and should produce data that is statistically close to pure entropy, while */dev/urandom* should return immediately, providing only cryptographic randomness.

The real world is somewhat messy, though. First, your application may need to run on a system that does not have these devices. (In that case, see Recipe 11.19, where we discuss solutions to this problem.*) Any reasonable version of Linux, FreeBSD, OpenBSD, or NetBSD will have these devices. They are also present on Mac OS X 10.1 or later, Solaris 9 or later, AIX 5.2 or later, HP-UX 11i or later, and IRIX 6.5.19 or later. As of this writing, only dead or officially "about to die" Unix variants, such as Tru64 and Ultrix, lack these devices. Note that each operating system tends to have its own implementation of these devices. We haven't looked at them all, so we cannot, in general, vouch for how strong and efficient these generators are, but we

* If you want to interoperate with such platforms (there are still plenty of systems without */dev/random* and */dev/urandom*), that reinforces the utility of using our API; simply link against code that implements our API using the solution from Recipe 11.8 instead of the solution from this recipe.

don't think you should worry about this issue in practice. (There are almost always bigger fish to fry.)

Second, depending on the operating system, the entropy produced by */dev/random* may be reused by */dev/urandom*. While few (if any) Unix platforms try to guarantee a clean separation of entropy, this is more of a theoretical problem than a practical problem; it is not something about which we personally would worry. Conversely, depending on the operating system, use of */dev/urandom* can drain entropy, denying service to the */dev/random* device.

Finally, most operating systems do not actually guarantee that */dev/urandom* is properly seeded. To understand why, you need to know something about what generally goes on under the hood. Basically, the randomness infrastructure tries to cull randomness from user input. For example, tiny bits of entropy can be derived from the time between console keystrokes. Unfortunately, the system may start up with very little entropy, particularly if the system boots without user intervention.

To avoid this problem, most cryptographic pseudo-random number generators stash away output before the system shuts down, which is used as a seed for the pseudo-random number generator when it starts back up. If the system can reboot without the seed being compromised (a reasonable assumption unless physical attacks are in your threat model, in which case you have to mitigate risk at the physical level), */dev/urandom* will produce good results.

The only time to get really paranoid about a lack of entropy is before you are sure the infrastructure has been seeded well. In particular, a freshly installed system may not have any entropy at all. Many people choose to ignore such a threat, and it is reasonable to do so because it is a problem that the operating system should be responsible for fixing.

However, if you want to deal with this problem yourself, be aware that all of the operating systems that have a */dev/random* device (as far as we can determine) monitor all keyboard events, adding those events to their internal collection of entropy. Therefore, you can use code such as that presented in Recipe 11.20 to gather sufficient entropy from the keyboard, then immediately throw it away (because the operating system will also be collecting it). Alternatively, you can collect entropy yourself using the techniques discussed in Recipes 11.22 and 11.23, then run your own cryptographic pseudo-random number generator (see Recipe 11.5).

The */dev/random* and */dev/urandom* devices behave just like files. You should read from these devices by opening the files and reading data from them. There are a few common "gotchas" when using that approach, however. First, the call to read data may fail. If you do not check for failure, you may think you got a random number when, in reality, you did not.

Second, people will occasionally use the API functions improperly. In particular, we have seen people who assume that the `read( )` or `fread( )` functions return a value or

a pointer to data. Instead, they return –1 on failure, and otherwise return the number of bytes read.

When using standard C runtime functions, we recommend using `read()`. If you are reading from */dev/urandom*, `read()` will successfully return unless a signal is delivered during the call (in which case the call should be made again), the operating system is misconfigured, or there is some other catastrophic error. Therefore, if `read()` is unsuccessful, retry when the value of `errno` is `EINTR`, and fail unconditionally otherwise. You should also check that the return value is equal to the number of bytes you requested to read, because some implementations may limit the amount of data you can read at once from this device. If you get a short read, merely continue to read until you collect enough data.

When using */dev/random*, things are the same if you are performing regular blocking reads. Of course, if not enough entropy is available, the call will hang until the requested data is available or until a signal interrupts the call.

If you don't like that behavior, you can make the file descriptor nonblocking, meaning that the function will return an error and set `errno` to `EAGAIN` if there isn't enough data to complete the entire read. Note that if some (but not all) of the requested data is ready, it will be returned instead of giving an error. In that case, the return value of `read()` will be smaller than the requested amount.

Given an integer file descriptor, the following code makes the associated descriptor nonblocking:

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

void spc_make_fd_nonblocking(int fd) {
  int flags;

  flags = fcntl(fd, F_GETFL);  /* Get flags associated with the descriptor. */
  if (flags == -1) {
    perror("spc_make_fd_nonblocking failed on F_GETFL");
    exit(-1);
  }
  flags |= O_NONBLOCK;
  /* Now the flags will be the same as before, except with O_NONBLOCK set.
   */
  if (fcntl(fd, F_SETFL, flags) == -1) {
    perror("spc_make_fd_nonblocking failed on F_SETFL");
    exit(-1);
  }
}
```

Here, we will demonstrate how to use */dev/random* and */dev/urandom* properly by binding them to the API we developed in Recipe 11.2. We will implement `spc_entropy()` by reading from */dev/random* in nonblocking mode. We will implement

spc_rand( ) by reading from */dev/urandom*. Finally, we will implement spc_keygen( ) by reading as much data as possible from */dev/random* in a nonblocking fashion, then falling back to */dev/urandom* when */dev/random* is dry.

Note that we need to open */dev/random* on two file descriptors, one blocking and one not, so that we may avoid race conditions where spc_keygen( ) expects a function to be nonblocking but spc_entropy( ) has set the descriptor to blocking in another thread.

In addition, we assume that the system has sufficient entropy to seed */dev/urandom* properly and */dev/random*'s entropy is not reused by */dev/urandom*. If you are worried about either of these assumptions, see the recipes suggested earlier for remedies.

Note that you can expect that */dev/random* output is properly postprocessed (whitened) to remove any patterns that might facilitate analysis in the case that the data contains less entropy than expected.

This code depends on the spc_make_fd_nonblocking( ) function presented earlier.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>

static int spc_devrand_fd           = -1,
           spc_devrand_fd_noblock =   -1,
           spc_devurand_fd          = -1;

void spc_rand_init(void) {
  spc_devrand_fd         = open("/dev/random",  O_RDONLY);
  spc_devrand_fd_noblock = open("/dev/random",  O_RDONLY);
  spc_devurand_fd        = open("/dev/urandom", O_RDONLY);

  if (spc_devrand_fd == -1 || spc_devrand_fd_noblock == -1) {
    perror("spc_rand_init failed to open /dev/random");
    exit(-1);
  }
  if (spc_devurand_fd == -1) {
    perror("spc_rand_init failed to open /dev/urandom");
    exit(-1);
  }
  spc_make_fd_nonblocking(spc_devrand_fd_noblock);
}

unsigned char *spc_rand(unsigned char *buf, size_t nbytes) {
  ssize_t       r;
  unsigned char *where = buf;

  if (spc_devrand_fd == -1 && spc_devrand_fd_noblock == -1 && spc_devurand_fd == -1)
    spc_rand_init( );
  while (nbytes) {
    if ((r = read(spc_devurand_fd, where, nbytes)) == -1) {
```

```
      if (errno == EINTR) continue;
      perror("spc_rand could not read from /dev/urandom");
      exit(-1);
    }
    where  += r;
    nbytes -= r;
  }
  return buf;
}

unsigned char *spc_keygen(unsigned char *buf, size_t nbytes) {
  ssize_t       r;
  unsigned char *where = buf;

  if (spc_devrand_fd == -1 && spc_devrand_fd_noblock == -1 && spc_devurand_fd == -1)
    spc_rand_init( );
  while (nbytes) {
    if ((r = read(spc_devrand_fd_noblock, where, nbytes)) == -1) {
      if (errno == EINTR) continue;
      if (errno == EAGAIN) break;
      perror("spc_rand could not read from /dev/random");
      exit(-1);
    }
    where  += r;
    nbytes -= r;
  }
  spc_rand(where, nbytes);
  return buf;
}

unsigned char *spc_entropy(unsigned char *buf, size_t nbytes) {
  ssize_t       r;
  unsigned char *where = buf;

  if (spc_devrand_fd == -1 && spc_devrand_fd_noblock == -1 && spc_devurand_fd == -1)
    spc_rand_init( );
  while (nbytes) {
    if ((r = read(spc_devrand_fd, (void *)where, nbytes)) == -1) {
      if (errno == EINTR) continue;
      perror("spc_rand could not read from /dev/random");
      exit(-1);
    }
    where  += r;
    nbytes -= r;
  }
  return buf;
}
```

See Also

Recipes 11.2, 11.5, 11.8, 11.19, 11.20, 11.22, 11.23

11.4 Using the Standard Windows Randomness Infrastructure

Problem

You want to use random numbers on a Windows system.

Solution

Use `CryptGenRandom( )` unless you absolutely need entropy, in which case see Recipe 11.8 and Recipes 11.20 through 11.23.

Discussion

Microsoft allows you to get cryptographically strong pseudo-random numbers using the CryptoAPI function `CryptGenRandom( )`. Unfortunately, there is no provision for any way to get entropy. The system does collect entropy behind the scenes, which it uses to improve the quality of the cryptographically strong pseudo-random numbers it gets.

Therefore, if this interface is being used to bind to the API we describe in Recipe 11.2, we can only implement `spc_rand( )` and `spc_keygen( )`, both of which will be exactly the same. If you want to try to get actual entropy on Windows, the only solution as of this writing is to use EGADS, which we discuss in Recipe 11.8. Alternatively, you can collect it yourself, as discussed in Recipes 11.20 through 11.23.

To use `CryptGenRand( )`, you must first acquire an `HCRYPTPROV` context. To do this, use the function `CryptAcquireContext( )`, which we discuss in some detail in Recipe 5.25. With an `HCRYPTPROV` context in hand, you can call `CryptGenRandom( )`, which will return `TRUE` if it is successful; otherwise, it will return `FALSE`, but it should never fail. `CryptGenRandom( )` has the following signature:

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

This function has the following arguments:

`hProv`
: Handle to a cryptographic service provider obtained via `CryptAcquireContext( )`.

`dwLen`
: Number of bytes of random data required. The output buffer must be at least this large.

`pbBuffer`
: Buffer into which the random data will be written.

Here we show how to use this function by binding it to the API from Recipe 11.2:

```
#include <windows.h>
#include <wincrypt.h>

static HCRYPTPROV hProvider;

void spc_rand_init(void) {
  if (!CryptAcquireContext(&hProvider, 0, 0, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT))
    ExitProcess((UINT)-1);  /* Feel free to properly signal an error instead. */
}

unsigned char *spc_rand(unsigned char *pbBuffer, size_t cbBuffer) {
  if (!hProvider) spc_rand_init();
  if (!CryptGenRandom(hProvider, cbBuffer, pbBuffer))
    ExitProcess((UINT)-1); /* Feel free to properly signal an error instead. */
  return pbBuffer;
}

unsigned char *spc_keygen(unsigned char *pbBuffer, size_t cbBuffer) {
  if (!hProvider) spc_rand_init();
  if (!CryptGenRandom(hProvider, cbBuffer, pbBuffer))
    ExitProcess((UINT)-1);
  return pbBuffer;
}
```

See Also

Recipes 5.25, 11.2, 11.8, 11.20, 11.21, 11.22, 11.23

11.5 Using an Application-Level Generator

Problem

You are in an environment where you do not have access to a built-in, cryptographically strong pseudo-random number generator. You have obtained enough entropy to seed a pseudo-random generator, but you lack a generator.

Solution

For general-purpose use, we recommend a pseudo-random number generator based on the AES encryption algorithm run in counter (CTR) mode (see Recipe 5.9). This generator has the best theoretical security assurance, assuming that the underlying cryptographic primitive is secure. If you would prefer a generator based on a hash function, you can run HMAC-SHA1 (see Recipe 6.10) in counter mode.

In addition, the keystream of a secure stream cipher can be used as a pseudo-random number generator.

Discussion

Stream ciphers are actually cryptographic pseudo-random number generators. One major practical differentiator between the two terms is whether you are using the output of the generator to perform encryption. If you are, it is a stream cipher; otherwise, it is a cryptographic pseudo-random number generator.

Another difference is that, when you are using a stream cipher to encrypt data, you need to be able to reproduce the same stream of output to decrypt the encrypted data. With a cryptographic PRNG, there is generally no need to be able to reproduce a data stream. Therefore, the generator can be reseeded at any time to help protect against internal state guessing attacks, which is analogous to rekeying a stream cipher.

The primary concern with a good cryptographic PRNG at the application level is internal state compromise, which would allow an attacker to predict its output. As long as the cryptographic algorithms used by a PRNG are not broken and the generator is not used to produce more output than it is designed to support, state compromise is generally not feasible by simply looking at the generator's output. The number of outputs supported by a generator varies based on the best attacks possible for whatever cryptographic algorithms are in use.

The risk of state compromise is generally not a big deal when dealing with something like */dev/random*, where the generator is in the kernel. The only way to compromise the state is to be inside the kernel. If that's possible, there are much bigger problems than */dev/urandom* or `CryptGenRandom( )` producing data that an attacker can guess.

In the user space, state compromise may be more of an issue, though. You need to work through the threats about which you are worried. Threats are likely to come only from code on the local machine, but what code? Are you worried about malicious applications running with the same permissions being able to somehow peer inside the current process to get the internal state? If so, perhaps you should have a separate process that only provides entropy and runs with a set of permissions where only itself and the superuser would be a concern (this is the recommended approach for using the EGADS package discussed in Recipe 11.8).

If state compromise is a potential issue, you might have to worry about more than an attacker guessing future outputs. You might also have to worry about an attacker *backtracking*, which means compromising previous outputs the generator made. Reseeding the generator periodically, as discussed in Recipe 11.6, can solve this problem. At best, an attacker should only be able to backtrack to the last reseeding (you can reseed without new entropy to mix in).

In practice, few people should have to worry very much about state compromise of their cryptographic PRNG. As was the case at the operating system level, if such

attacks are a realistic threat, you will usually have far bigger threats, and mitigating those threats will help mitigate this one as well.

There is a lot that can go wrong when using a pseudo-random number generator. Coming up with a good construct turns out to be the easy part. Here are some things you should closely consider:

- Pseudo-random number generators need to be seeded with an adequate amount of entropy; otherwise, they are still potentially predictable. We recommend at least 80 bits. See the various recipes elsewhere in this chapter for information on collecting entropy.
- Be careful to pay attention to the maximum number of outputs a generator can produce before it will need to be reseeded with new entropy. At some point, generators start to leak information and will generally fall into a cycle. Note, though, that for the configurations we present, you will probably never need to worry about the limit in practice. For example, the generator based on AES-128 leaks a bit of information after 2^{64} 16-byte blocks of output, and cycles after 2^{128} such blocks.
- When adding entropy to a system, it is best to collect a lot of entropy and seed all at once, instead of seeding a little bit at a time. We will illustrate why by example. Suppose that you seed a generator with one bit of entropy. An attacker has only one bit to guess, which can be done accurately after two outputs. If the attacker completely compromises the state after two outputs, and we then add another bit of entropy, he can once again guess the state easily. If we add one bit 128 times, there is still very little security overall if the generator state is compromised. However, if you add 128 bits of entropy to the generator all at once, an attack should essentially be infeasible.
- If an attacker can somehow compromise the internal state of a pseudo-random number generator, then it might be possible to launch a backtracking attack, where old generator outputs can be recovered. Such attacks are easy to thwart; see Recipe 11.6.

In the following three subsections, we will look at three different techniques for pseudo-random number generators: using a block cipher such as AES, using a stream cipher directly, and using a cryptographic hash function such as SHA1.

Using generators based on block ciphers

If you are in an environment where you have use of a good block cipher such as AES, you have the makings of a cryptographically strong pseudo-random number generator. Many of the encryption modes for turning a block cipher into a stream cipher are useful for this task, but CTR mode has the nicest properties. Essentially, you create random outputs one block at a time by encrypting a counter that is incremented after every encryption operation.

The seed should be at least as large as the key size of the cipher, because it will be used to key a block cipher. In addition, it is useful to have additional seed data that sets the first plaintext (counter) value.

Our implementation is based on the code in Recipe 5.5 and has two exported routines. The first initializes a random number generator:

```
void spc_bcprng_init(SPC_BCPRNG_CTX *prng, unsigned char *key, int kl,
                     unsigned char *x, int xl);
```

This function has the following arguments:

`prng`
: Pointer to a context object that holds the state for a block cipher–based PRNG. The caller may allocate the context object either dynamically or statically; this function will initialize it.

`key`
: Buffer that should contain entropic data. This data is used to key the block cipher, and it is the required portion of the seed to the generator.

`kl`
: Length of the key buffer in bytes; must be a valid value for the algorithm in use.

`x`
: Buffer that may contain extra seed data, which we recommend you use if you have available entropy. If the specified size of this buffer is zero, this argument will be ignored. Note that if the buffer is larger than `SPC_BLOCK_LEN` (see Recipe 5.5) any additional data in the buffer will be ignored. Therefore, if you have sparse amounts of entropy, compress it to the right length before calling this function, as discussed in Recipe 11.16.

`xl`
: Length of the extra seed buffer in bytes. It may be specified as zero to indicate that there is no extra seed data.

Once you have an instantiated generator, you can get cryptographically strong pseudo-random data from it with the following function:

```
unsigned char *spc_bcprng_rand(SPC_BCPRNG_CTX *prng, unsigned char *buf, size_t l);
```

This function has the following arguments:

`prng`
: Pointer to the generator's context object.

`buf`
: Buffer into which the random data will be written.

`l`
: Number of bytes that should be placed into the output buffer.

This function never fails (save for a catastrophic error in encryption), and it returns the address of the output buffer.

Here is an implementation of this generator API, which makes use of the block cipher interface we developed in Recipe 5.5:

```
/* NOTE: This code should be augmented to reseed after each request
/* for pseudo-random data, as discussed in Recipe 11.6
/*
#ifndef WIN32
#include <string.h>
#include <pthread.h>
#else
#include <windows.h>
#endif

/* if encryption operations fail, you passed in a bad key size or are using a
 * hardware API that failed.  In that case, be sure to perform error checking.
*/

typedef struct {
  SPC_KEY_SCHED ks;
  unsigned char ctr[SPC_BLOCK_SZ];
  unsigned char lo[SPC_BLOCK_SZ]; /* Leftover block of output */
  int           ix;               /* index into lo */
  int           kl;               /* The length of key used to key the cipher */
} SPC_BCPRNG_CTX;

#ifndef WIN32
static pthread_mutex_t spc_bcprng_mutex = PTHREAD_MUTEX_INITIALIZER;

#define SPC_BCPRNG_LOCK()   pthread_mutex_lock(&spc_bcprng_mutex);
#define SPC_BCPRNG_UNLOCK() pthread_mutex_unlock(&spc_bcprng_mutex);
#else
static HANDLE hSpcBCPRNGMutex;

#define SPC_BCPRNG_LOCK()   WaitForSingleObject(hSpcBCPRNGMutex, INFINITE)
#define SPC_BCPRNG_UNLOCK() ReleaseMutex(hSpcBCPRNGMutex)
#endif

static void spc_increment_counter(SPC_BCPRNG_CTX *prng) {
  int i = SPC_BLOCK_SZ;

  while (i--)
    if (++prng->ctr[i]) return;
}

void spc_bcprng_init(SPC_BCPRNG_CTX *prng, unsigned char *key, int kl,
                     unsigned char *x, int xl) {
  int i = 0;

  SPC_BCPRNG_LOCK();
  SPC_ENCRYPT_INIT(&(prng->ks), key, kl);
  memset(prng->ctr, 0, SPC_BLOCK_SZ);
```

```
  while (xl-- && i < SPC_BLOCK_SZ)
    prng->ctr[i++] = *x++;
  prng->ix = 0;
  prng->kl = kl;
  SPC_BCPRNG_UNLOCK( );
}

unsigned char *spc_bcprng_rand(SPC_BCPRNG_CTX *prng, unsigned char *buf, size_t l) {
  unsigned char *p;

  SPC_BCPRNG_LOCK( );
  for (p = buf;  prng->ix && l;  l--) {
    *p++ = prng->lo[prng->ix++];
    prng->ix %= SPC_BLOCK_SZ;
  }
  while (l >= SPC_BLOCK_SZ) {
    SPC_DO_ENCRYPT(&(prng->ks), prng->ctr, p);
    spc_increment_counter(prng);
    p += SPC_BLOCK_SZ;
    l -= SPC_BLOCK_SZ;
  }
  if (l) {
    SPC_DO_ENCRYPT(&(prng->ks), prng->ctr, prng->lo);
    spc_increment_counter(prng);
    prng->ix = l;
    while (l--) p[l] = prng->lo[l];
  }
  SPC_BCPRNG_UNLOCK( );
  return buf;
}
```

If your block cipher has 64-bit blocks and has no practical weaknesses, do not use this generator for more than 2^{35} bytes of output (2^{32} block cipher calls). If the cipher has 128-bit blocks, do not exceed 2^{68} bytes of output (2^{64} block cipher calls). If using a 128-bit block cipher, it is generally acceptable not to check for this condition, as you generally would not reasonably expect to ever use that many bytes of output.

To bind this cryptographic PRNG to the API in Recipe 11.2, you can use a single global generator context that you seed in `spc_rand_init( )`, requiring you to get a secure seed. Once that's done (assuming the generator variable is a statically allocated global variable named `spc_prng`), you can simply implement `spc_rand( )` as follows:

```
unsigned char *spc_rand(unsigned char *buf, size_t l) {
  return spc_bcprng_rand(&spc_prng, buf, l);
}
```

Note that you should probably be sure to check that the generator is seeded before calling `spc_bcprng_rand( )`.

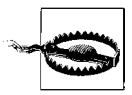
Using a stream cipher as a generator

As we mentioned, stream ciphers are themselves pseudo-random number generators, where the key (and the initialization vector, if appropriate) constitutes the seed. If you are planning to use such a cipher, we strongly recommend the SNOW 2.0 cipher, discussed in Recipe 5.2.

Because of the popularity of the RC4 cipher, we expect that people will prefer to use RC4, even though it does not look as good as SNOW. The RC4 stream cipher does make an acceptable pseudo-random number generator, and it is incredibly fast if you do not rekey frequently (that is particularly useful if you expect to need a heck of a lot of numbers). If you do rekey frequently to avoid backtracking attacks, a block cipher–based approach may be faster; time it to make sure.

RC4 requires a little bit of work to use properly, given a standard API. First, most APIs want you to pass in data to encrypt. Because you want only the raw keystream, you must always pass in zeros. Second, be sure to use RC4 in a secure manner, as discussed in Recipe 5.23.

If your RC4 implementation has the API discussed in Recipe 5.23, seeding it as a pseudo-random number generator is the same as keying the algorithm. RC4 can accept keys up to 256 bytes in length.



Because of limitations in RC4, you should throw away the first 256 bytes of RC4 output, as discussed in Recipe 5.23.

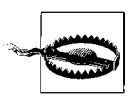
After encrypting 256 bytes and throwing the results away, you can then, given an RC4 context, get random data by encrypting zeros. Assuming the RC4 API from Recipe 5.23 and assuming you have a context statically allocated in a global variable named `spc_prng`, here's a binding of RC4 to the `spc_rand( )` function that we introduced in Recipe 11.2:

```
/* NOTE: This code should be augmented to reseed after each request
/* for pseudo-random data, as discussed in Recipe 11.6
/*
#ifndef WIN32
#include <pthread.h>

static pthread_mutex_t spc_rc4rng_mutex = PTHREAD_MUTEX_INITIALIZER;

#define SPC_RC4RNG_LOCK( )   pthread_mutex_lock(&spc_rc4rng_mutex)
#define SPC_RC4RNG_UNLOCK( ) pthread_mutex_unlock(&spc_rc4rng_mutex)
#else
#include <windows.h>

static HANDLE hSpcRC4RNGMutex;

#define SPC_RC4RNG_LOCK( )   WaitForSingleObject(hSpcRC4RNGMutex, INFINITE)
```

```
#define SPC_RC4RNG_UNLOCK( ) ReleaseMutex(hSpcRC4RNGMutex)
#endif

#define SPC_ARBITRARY_SIZE 16

unsigned char *spc_rand(unsigned char *buf, size_t l) {
  static unsigned char zeros[SPC_ARBITRARY_SIZE] = {0,};
  unsigned char       *p = buf;

#ifdef WIN32
  if (!hSpcRC4RNGMutex) hSpcRC4RNGMutex = CreateMutex(0, FALSE, 0);
#endif

  SPC_RC4RNG_LOCK( );
  while (l >= SPC_ARBITRARY_SIZE) {
    RC4(&spc_prng, SPC_ARBITRARY_SIZE, zeros, p);
    l -= SPC_ARBITRARY_SIZE;
    p += SPC_ARBITRARY_SIZE;
  }
  if (l) RC4(&spc_prng, l, zeros, p);

  SPC_RC4RNG_UNLOCK( );
  return buf;
}
```

Note that, although we don't show it in this code, you should ensure that the generator is initialized before giving output.

Because using this RC4 API requires encrypting zero bytes to get the keystream output, in order to be able to generate data of arbitrary sizes, you must either dynamically allocate and zero out memory every time or iteratively call RC4 in chunks of up to a fixed size using a static buffer filled with zeros. We opt for the latter approach.



RC4 is only believed to be a strong source of random numbers for about 2^{30} outputs. After that, we strongly recommend that you reseed it with new entropy. If your application would not conceivably use that many outputs, it should generally be okay not to check that condition.

Using a generator based on a cryptographic hash function

The most common mistake made when trying to use a hash function as a cryptographic pseudo-random number generator is to continually hash a piece of data. Such an approach gives away the generator's internal state with every output. For example, suppose that your internal state is some value X, and you generate and output Y by hashing X. The next time you need random data, rehashing X will give the same results, and any attacker who knows the last outputs from the generator can figure out the next outputs if you generate them by hashing Y.

One very safe way to use a cryptographic hash function in a cryptographic pseudo-random number generator is to use HMAC in counter mode, as discussed in Recipe 6.10. Here we implement a generator based on the HMAC-SHA1 implementation from Recipe 6.10. You should be able to adapt this code easily to any HMAC implementation you want to use.

```
/* NOTE: This code should be augmented to reseed after each request
/* for pseudo-random data, as discussed in Recipe 11.6
/*
#ifndef WIN32
#include <string.h>
#include <pthread.h>
#else
#include <windows.h>
#endif

/* If MAC operations fail, you passed in a bad key size or you are using a hardware
 * API that failed.  In that case, be sure to perform error checking.
 */
#define MAC_OUT_SZ 20

typedef struct {
  SPC_HMAC_CTX      ctx;
  unsigned char ctr[MAC_OUT_SZ];
  unsigned char lo[MAC_OUT_SZ];   /* Leftover block of output */
  int           ix;               /* index into lo. */
} SPC_MPRNG_CTX;

#ifndef WIN32
static pthread_mutex_t spc_mprng_mutex = PTHREAD_MUTEX_INITIALIZER;

#define SPC_MPRNG_LOCK()   pthread_mutex_lock(&spc_mprng_mutex)
#define SPC_MPRNG_UNLOCK() pthread_mutex_unlock(&spc_mprng_mutex)
#else
static HANDLE hSpcMPRNGMutex;

#define SPC_MPRNG_LOCK()   WaitForSingleObject(hSpcMPRNGMutex, INFINITE)
#define SPC_MPRNG_UNLOCK() ReleaseMutex(hSpcMPRNGMutex)
#endif

static void spc_increment_mcounter(SPC_MPRNG_CTX *prng) {
  int i = MAC_OUT_SZ;

  while (i--)
    if (++prng->ctr[i])
      return;
}

void spc_mprng_init(SPC_MPRNG_CTX *prng, unsigned char *seed, int l) {
  SPC_MPRNG_LOCK();
  SPC_HMAC_Init(&(prng->ctx), seed, l);
  memset(prng->ctr, 0, MAC_OUT_SZ);
  prng->ix = 0;
```

```
  SPC_MPRNG_UNLOCK( );
}

unsigned char *spc_mprng_rand(SPC_MPRNG_CTX *prng, unsigned char *buf, size_t l) {
  unsigned char *p;

  SPC_MPRNG_LOCK( );
  for (p = buf;  prng->ix && l;  l--) {
    *p++ = prng->lo[prng->ix++];
    prng->ix %= MAC_OUT_SZ;
  }
  while (l >= MAC_OUT_SZ) {
    SPC_HMAC_Reset(&(prng->ctx));
    SPC_HMAC_Update(&(prng->ctx), prng->ctr, sizeof(prng->ctr));
    SPC_HMAC_Final(p, &(prng->ctx));
    spc_increment_mcounter(prng);
    p += MAC_OUT_SZ;
    l -= MAC_OUT_SZ;
  }
  if (l) {
    SPC_HMAC_Reset(&(prng->ctx));
    SPC_HMAC_Update(&(prng->ctx), prng->ctr, sizeof(prng->ctr));
    SPC_HMAC_Final(prng->lo, &(prng->ctx));
    spc_increment_mcounter(prng);
    prng->ix = l;
    while (l--) p[l] = prng->lo[l];
  }
  SPC_MPRNG_UNLOCK( );
  return buf;
}
```

This implementation has two publicly exported functions. The first initializes the generator:

```
void spc_mprng_init(SPC_MPRNG_CTX *prng, unsigned char *seed, int l);
```

This function has the following arguments:

`prng`
: Context object used to hold the state for a MAC-based PRNG.

`seed`
: Buffer containing data that should be filled with entropy (the seed). This data is used to key the MAC.

`l`
: Length of the seed buffer in bytes.

The second function actually produces random data:

```
unsigned char *spc_mprng_rand(SPC_MPRNG_CTX *prng, unsigned char *buf, size_t l);
```

This function has the following arguments:

`prng`
: Context object used to hold the state for a MAC-based PRNG.

`out`
: Buffer into which the random data will be placed.

`l`
: Number of random bytes to be placed into the output buffer.

If your hash function produces *n*-bit outputs and has no practical weaknesses, do not use the generator after you run the MAC more than $2^{n/2}$ times. For example, with SHA1, this generator should be not be a problem for at least $2^{80} \times 20$ bytes. In practice, you probably will not have to worry about this issue.

To bind this cryptographic pseudo-random number generator to the API in Recipe 11.2, you can use a single global generator context that you seed in `spc_rand_init( )`, requiring you to get a secure seed. Once that is done (assuming the generator variable is a statically allocated global variable named `spc_prng`), you can simply implement `spc_rand( )` as follows:

```
unsigned char *spc_rand(unsigned char *buf, size_t l) {
  return spc_bcprng_rand(&spc_prng, buf, l);
}
```

Note that, although we don't show it in the previous code, you should ensure that the generator is initialized before giving output.

See Also

Recipes 5.2, 5.5, 5.9, 5.23, 6.10, 11.2, 11.6, 11.8, 11.16

11.6 Reseeding a Pseudo-Random Number Generator

Problem

You have an application-level pseudo-random number generator such as the ones presented in Recipe 11.5, and you want to reseed it, either because you have new entropy to mix in or because you would like to prevent against backtracking attacks.

Solution

Create a new seed by getting a sufficient number of bytes from the generator to seed the generator. If mixing in entropy, compress the entropy down to the seed size if necessary, as discussed in Recipe 11.16, then XOR the compressed seed with the generator output. Finally, reseed the generator with the resulting value.

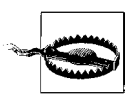
Discussion

There are two common reasons why you may want to reseed a PRNG. First, your threat model may include the possibility of the internal state of your PRNG being compromised, and you want to prevent against an attacker's being able to figure out numbers that were output before the state compromise. Reseeding, if done right, essentially transforms the internal state in a way that preserves entropy while making it essentially impossible to backtrack. Protecting against backtracking attacks can be done cheaply enough, so there is no excuse for not doing it.

Second, you may want to add entropy into the state. This could serve a number of purposes. For example, you might want to add entropy to the system. Remember, however, that cryptographic generators have a maximum amount of entropy they can contain, so adding entropy to a generator state can look unnecessary.

When available, however, reseeding with entropy is a good conservative measure, for several reasons. For one reason, if you have underestimated the amount of entropy that a generator has, adding entropy is a good thing. For another, if the generator has lost any entropy, new entropy can help replenish it. Such entropy loss is natural because cryptographic algorithms are not as good as their theoretical ideals. In addition, because we generally do not know the exact strength of our algorithms, it is hard to determine how quickly entropy gets lost. (Note, however, that if the algorithms are as strong as believed, it should be quite slowly.)

While a generator based on AES or HMAC-SHA1, implemented as discussed in Recipe 11.5, probably never loses more than a miniscule amount of entropy before 2^{64} outputs, it is always good to be conservative and assume that it drains quickly, particularly if you have entropy to spare.



When adding entropy to a system, it is best to collect a lot of entropy and seed all at once, instead of seeding a little bit at a time. We will illustrate why by example. Suppose you seed a generator with one bit of entropy. An attacker has only one bit to guess, which can be done accurately after two outputs. If the attacker completely compromises the state after two outputs, and we then add another bit of entropy, he can once again guess the state easily.

If we add one bit 128 times, there is still very little security overall if the generator state is compromised. However, if you add 128 bits of entropy to the generator all at once, an attack should essentially be infeasible.

The actions you should take to reseed a generator are different depending on whether you are actually adding entropy to the state of the generator or just trying to thwart a backtracking attack. However, the first step is the same in both cases.

1. Figure out how big a seed you need. At the very least, you need a seed that is as many bits in length as bits of entropy you think are in the generator. Generally, this will be at least as large as the key size of the underlying primitive (or the output size when using a one-way hash function instead of a cipher).
2. If you need to introduce new entropy, properly compress the data containing entropy. In particular, you must transform the data into a seed of the proper size, with minimal loss of entropy. One easy way to do that is to process the string with a cryptographic hash function (truncating the hash output to the desired length, if necessary). Then XOR the compressed entropy with the seed output by the generator.
3. Take the value and use it to reseed the generator. If you are using a counter-based generator, you can either reset the counter or choose not to do so. In fact, it is preferable to take a bit of extra output from the generator so that the counter can be set to a random value.

For example, using the block cipher–based PRNG from Recipe 11.5, here is a function that reseeds the generator, given new, uncompressed data containing entropy:

```
void spc_bcprng_reseed(SPC_BCPRNG_CTX *prng, unsigned char *new_data, size_t l) {
  size_t        i;
  unsigned char m[SPC_MAX_KEYLEN + SPC_BLOCK_SZ];

  SPC_BCPRNG_LOCK( );
  if (prng->kl > SPC_MAX_KEYLEN) prng->kl = SPC_MAX_KEYLEN;
  spc_bcprng_rand(prng, m, prng->kl + SPC_BLOCK_SZ);
  while (l > prng->kl) {
    for (i = 0;  i < prng->kl;  i++) m[i] ^= *new_data++;
    l -= prng->kl;
    spc_bcprng_init(prng, m, prng->kl, m + prng->kl, SPC_BLOCK_SZ);
    spc_bcprng_rand(prng, m, prng->kl + SPC_BLOCK_SZ);
  }
  for (i = 0;  i <l;  i++) m[i] ^= *new_data++;
  spc_bcprng_init(prng, m, prng->kl, m + prng->kl, SPC_BLOCK_SZ);
  SPC_BCPRNG_UNLOCK( );
}
```

To handle compression of the data that contains entropy, we avoid using a hash function. Instead, we break the data up into chunks no larger than the required seed size, and reseed multiple times until we have run out of data. This is an entropy-preserving way of processing the data that does not require the use of a cryptographic hash function.

See Also

Recipes 11.5, 11.16

11.7 Using an Entropy Gathering Daemon–Compatible Solution

Problem

Your application needs randomness, and you want it to be able to run on Unix-based platforms that lack the */dev/random* and */dev/urandom* devices discussed in Recipe 11.3—for example, machines that need to support legacy operating systems.

Solution

Use a third-party software package that gathers and outputs entropy, such as the Entropy Gathering and Distribution System (EGADS). Then use the Entropy Gathering Daemon (EGD) interface to read entropy. EGD is a tool for entropy harvesting and was the first tool to export this API.

When implementing our randomness API from Recipe 11.2, use entropy gathered over the EGD interface in places where entropy is needed; then, to implement the rest of the API, use data from that interface to seed an application-level cryptographic pseudo-random number generator (see Recipe 11.5).

Discussion

A few entropy collection systems exist as processes outside the kernel and distribute entropy through the EGD socket interface. Such systems set up a server process, listening on a Unix domain socket. To read entropy, you communicate over that interface using a simple protocol.

One such system is EGADS (described in the next recipe and available from *http://www.securesoftware.com/egads*). Another system is EGD itself, which we do not recommend as of this writing for several reasons, primarily because we think its entropy estimates are too liberal.

Such entropy collection systems usually are slow to collect good entropy. If you can interactively collect input from a user, you might want to use one of the techniques in Recipe 11.19 instead to force the user to add entropy to the system herself. That approach will avoid arbitrary hangs as you wait for crucial entropy from an EGD-compatible system.

The EGD interface is more complex than the standard file interface you get when dealing with the */dev/random* device. Traditionally, you would just read the data needed. With EGD, however, you must first write one of five commands to the socket. Each command is a single byte of data:

0x00
: Query the amount of entropy believed to be available. This information is not at all useful, particularly because you cannot use it in any decision to read data without causing a race condition.

0x01
: Read data if available. This command takes a single-byte argument specifying how many bytes of data should be read, if that much data is available. If not enough entropy is available, any available entropy may be immediately returned. The first byte of the result is the number of bytes being returned, so do not treat this information as entropy. Note that you can never request or receive more than 255 bytes of entropy at a time.

0x02
: Read data when available. This command takes the same argument as the previous command. However, if not enough entropy is available, this command will block until the request can be fulfilled. In addition, the response for the command is simply the requested bytes; the initial byte is not the number of bytes being returned.

0x03
: Write entropy to the internal collector. This command takes three arguments. The first is a two-byte value (most significant byte first) specifying how many bits of entropy are believed to be in the data. The second is a one-byte value specifying how many bytes of data are to be written. The third is the entropic data itself.

0x04
: Get the process identifier of the EGD process. This returns a byte-long header that specifies how long the result is in bytes, followed by the actual process identifier, most significant byte first.

In this recipe, we implement the randomness interface from Recipe 11.2. In addition, we provide a function called `spc_rand_add_entropy()`, which provides an interface to the command for providing the server with entropy. That function does not allow the caller to specify an entropy estimate. We believe that user-level processes should be allowed to contribute data to be put into the mix but shouldn't be trusted to estimate entropy, primarily because you may have just cause not to trust the estimates of other processes running on the same machine that might be adding entropy. That is, if you are using an entropy server that gathers entropy slowly, you do not want an attacker from another process adding a big known value to the entropy system and claiming that it has 1,000 bits of entropy.

In part because untrusted programs can add bad entropy to the mix, we recommend using a highly conservative solution where such an attack is not likely to be effective. That means staying away from EGD, which will use estimates from any untrusted

process. While EGADS implements the EGD interface, it ignores the entropy estimate supplied by the user. It does mix the entropy into its state, but it assumes that it contains no entropy.

The following code implements the `spc_entropy( )` and `spc_keygen( )` functions from Recipe 11.2 using the EGD interface. We omit `spc_rand( )` but assume that it exists (it is called by `spc_keygen( )` when appropriate). To implement `spc_rand( )`, see Recipe 11.5.

When implementing `spc_entropy( )` and `spc_keygen( )`, we do not cryptographically postprocess the entropy to thwart statistical analysis if we do not have as much entropy as estimated, as you can generally expect servers implementing the EGD interface to do this (EGADS certainly does). If you want to be absolutely sure, you can do your own cryptographic postprocessing, as shown in Recipe 11.16.

Note that the following code requires you to know in advance the file on the filesystem that implements the EGD interface. There is no standard place to look for EGD sockets, so you could either make the location of the socket something the user can configure, or require the user to run the collector in such a way that the socket lives in a particular place on the filesystem.

Of course, the socket should live in a "safe" directory, where only the user running the entropy system can write files (see Recipe 2.4). Clearly, any user who needs to be able to use the server must have read access to the socket.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/uio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>

#define EGD_SOCKET_PATH "/home/egd/socket"

/* NOTE: this needs to be augmented with whatever you need to do in order to seed
 * your application-level generator.  Clearly, seed that generator after you've
 * initialized the connection with the entropy server.
 */

static int spc_egd_fd = -1;

void spc_rand_init(void) {
  struct sockaddr_un a;

  if ((spc_egd_fd = socket(PF_UNIX, SOCK_STREAM, 0)) == -1) {
    perror("Entropy server connection failed");
    exit(-1);
  }
  a.sun_len    = sizeof(a);
  a.sun_family = AF_UNIX;
```

```
  strncpy(a.sun_path, EGD_SOCKET_PATH, sizeof(a.sun_path));
  a.sun_path[sizeof(a.sun_path) - 1] = 0;
  if (connect(spc_egd_fd, (struct sockaddr *)&a, sizeof(a))) {
    perror("Entropy server connection failed");
    exit(-1);
  }
}

unsigned char *spc_keygen(unsigned char *buf, size_t l) {
  ssize_t              nb;
  unsigned char        nbytes, *p, tbytes;
  static unsigned char cmd[2] = {0x01,};

  if (spc_egd_fd == -1) spc_rand_init( );
  for (p = buf;  l;  l -= tbytes) {
    /* Build and send the request command to the EGD server */
    cmd[1] = (l > 255 ? 255 : l);
    do {
      if ((nb = write(spc_egd_fd, cmd, sizeof(cmd))) == -1 && errno != EINTR) {
        perror("Communication with entropy server failed");
        exit(-1);
      }
    } while (nb == -1);

    /* Get the number of bytes in the result */
    do {
      if ((nb = read(spc_egd_fd, &nbytes, 1)) == -1 && errno != EINTR) {
        perror("Communication with entropy server failed");
        exit(-1);
      }
    } while (nb == -1);
    tbytes = nbytes;

    /* Get all of the data from the result */
    while (nbytes) {
      do {
        if ((nb = read(spc_egd_fd, p, nbytes)) == -1) {
          if (errno == -1) continue;
          perror("Communication with entropy server failed");
          exit(-1);
        }
      } while (nb == -1);
      p      += nb;
      nbytes -= nb;
    }

    /* If we didn't get as much entropy as we asked for, the server has no more
     * left, so we must fall back on the application-level generator to avoid
     * blocking.
     */
    if (tbytes != cmd[1]) {
      spc_rand(p, l);
      break;
    }
```

```
  }
  return buf;
}

unsigned char *spc_entropy(unsigned char *buf, size_t l) {
  ssize_t              nb;
  unsigned char        *p;
  static unsigned char cmd = 0x02;

  if (spc_egd_fd == -1) spc_rand_init( );
  /* Send the request command to the EGD server */
  do {
    if ((nb = write(spc_egd_fd, &cmd, sizeof(cmd))) == -1 && errno != EINTR) {
      perror("Communcation with entropy server failed");
      exit(-1);
    }
  } while (nb == -1);

  for (p = buf;  l;  p += nb, l -= nb) {
    do {
      if ((nb = read(spc_egd_fd, p, l)) == -1) {
        if (errno == -1) continue;
        perror("Communication with entropy server failed");
        exit(-1);
      }
    } while (nb == -1);
  }

  return buf;
}

void spc_egd_write_entropy(unsigned char *data, size_t l) {
  ssize_t              nb;
  unsigned char        *buf, nbytes, *p;
  static unsigned char cmd[4] = { 0x03, 0, 0, 0 };

  for (buf = data;  l;  l -= cmd[3]) {
    cmd[3] = (l > 255 ? 255 : l);
    for (nbytes = 0, p = cmd;  nbytes < sizeof(cmd);  nbytes += nb) {
      do {
        if ((nb = write(spc_egd_fd, cmd, sizeof(cmd) - nbytes)) == -1) {
          if (errno != EINTR) continue;
          perror("Communication with entropy server failed");
          exit(-1);
        }
      } while (nb == -1);
    }

    for (nbytes = 0;  nbytes < cmd[3];  nbytes += nb, buf += nb) {
      do {
        if ((nb = write(spc_egd_fd, data, cmd[3] - nbytes)) == -1) {
          if (errno != EINTR) continue;
          perror("Communication with entropy server failed");
          exit(-1);
```

```
            }
          } while (nb == -1);
        }
      }
    }
```

See Also

- EGADS by Secure Software, Inc.: *http://www.securesoftware.com/egads*
- Recipes 2.4, 11.2, 11.3, 11.5, 11.16, 11.19

11.8 Getting Entropy or Pseudo-Randomness Using EGADS

Problem

You want to use a library-level interface to EGADS for gathering entropy or getting cryptographically strong pseudo-random data. For example, you may need entropy on a system such as Microsoft Windows, where there is no built-in API for getting it.

Solution

Use the EGADS API as described in the following "Discussion" section.

Discussion

EGADS, the Entropy Gathering and Distribution System, is capable of performing many functions related to random numbers. First, it provides a high-level interface for getting random values, such as integers, numbers in a particular range, and so on. Second, EGADS does its own entropy collection, and has a library-level API for accessing the collector, making it a simple API to use for any of your randomness needs.

EGADS supports a variety of Unix variants, including Darwin, FreeBSD, Linux, OpenBSD, and Solaris. In addition, it supports Windows NT 4.0, Windows 2000, and Windows XP. Unfortunately, EGADS does not support Windows 95, Windows 98, or Windows ME because it runs as a service (which is a subsystem that does not exist on these versions of Windows). EGADS is available from *http://www.securesoftware.com/egads*.

EGADS is a good solution for the security-minded because it is conservative. It contains a conservative entropy collector and a conservative pseudo-random number generator. Both of these components have provable security properties that rely only

on the strength of the AES cryptographic algorithm. EGADS does a good job of protecting against compromised entropy sources, which other PRNGs tend not to do. It also provides a good amount of protection against backtracking attacks, meaning that if the internal generator state does get compromised, few if any of the previous generator outputs will be recoverable.

To use EGADS, you must install the package, start up the server that comes with it, include *egads.h*, and link against the correct library, which will typically be *libegads.so* on Unix (*libegads.dyld* on Darwin) and *egads.lib* on Windows.

Before you can use any of the functions in the EGADS package, you must first initialize a PRNG context by calling `egads_init( )`:

```
void egads_init(prngctx_t *ctx, char *sockname, char *rfile, int *err);
```

This function has the following arguments:

`ctx`
: PRNG context object that is to be initialized. The caller should allocate the object either statically or dynamically.

`sockname`
: If not specified as `NULL`, this is the address of the server. On Unix, this is the name of the Unix domain socket created by the EGADS server. On Windows, this is the name of the mailslot object created by the EGADS service. If specified as `NULL`, which is normally how it should be specified, the compiled-in default will be used.

`rfile`
: Name of a file from which entropy can be read. On Unix, this defaults to */dev/random* if it is specified as `NULL`. This argument is always ignored on Windows.

`err`
: If any error occurs, an error code will be stored in this argument. A value of 0 indicates that no error occurred; otherwise, one of the `RERR_*` constants defined in *egads.h* will be returned. `NULL` may not be specified here.

The function `egads_entropy( )` establishes a connection to the entropy gateway and obtains the requested number of bytes of raw entropy. If not enough entropy is currently available to satisfy the request, this function will block until there is. Its signature nearly matches that of `spc_entropy( )` from Recipe 11.2:

```
void egads_entropy(prngctx_t *ctx, char *buf, int nbytes, int *err);
```

This function has the following arguments:

`ctx`
: PRNG context object that has been initialized.

`out`
: Buffer into which the entropy data will be placed.

`nbytes`
: Number of bytes of entropy that should be written into the output buffer. You must be sure that the output buffer is sufficiently large to hold the requested data.

`err`
: If any error occurs, an error code will be stored in this argument. A value of 0 indicates that no error occurred; otherwise, one of the `RERR_*` constants defined in *egads.h* will be returned. `NULL` may be not be specified here.

The function `PRNG_output()` allows you to get byte strings of cryptographically random data. Its signature nearly matches that of `spc_rand()` from Recipe 11.2:

```
void PRNG_output(prng_ctx *ctx, char *buf, int64 nbytes);
```

This function has the following arguments:

`ctx`
: PRNG context object that has been initialized.

`buf`
: Buffer into which the entropy data will be placed.

`nbytes`
: Number of bytes of random data that should be written into the output buffer. You must be sure that the output buffer is sufficiently large to hold the requested data.

The function `egads_destroy()` resets a PRNG context object. Before the memory for the context object is freed or goes out of scope (because it is statically allocated on the stack), `egads_destroy()` must be called on a successfully initialized context object. This ensures that the connection to the EGADS server or service is broken, and that any other memory or state maintained by EGADS that is associated with the context object is cleaned up.

```
void egads_destroy(prngctx_t *ctx);
```

This `ctx` argument is the successfully initialized PRNG context that is to be destroyed. It is the caller's responsibility to free any memory used to allocate the object

The rest of the EGADS API allows you to retrieve pseudo-random values of particular data types. All functions in this API take a final argument that, on completion of the call, contains the success or failure status. On failure, the error argument contains an integer error code. On success, it will be 0.

```
void egads_randlong(prngctx_t *ctx, long *out, int *error);
void egads_randulong(prngctx_t *ctx, unsigned long *out, int *error);
void egads_randint(prngctx_t *ctx, int *out, int *error);
void egads_randuint(prngctx_t *ctx, unsigned int *out, int *error);
void egads_randrange(prngctx_t *ctx, int *out, int min, int max, int *error);
```

The egads_randlong( ) function gets a pseudo-random long value, whereas egads_randulong( ) gets a pseudo-random unsigned long value from 0 to ULONG_MAX inclusive. The functions egads_randint( ) and egads_randuint( ) do the same things, but on integer types instead of longs. To get a random integer in a specified range, use the function egads_randrange( ). The min and max arguments are both inclusive, meaning that they are both values that can possibly be returned.

```
void egads_randreal(prngctx_t * ctx, double *out, int *error);
void egads_randuniform(prngctx_t *ctx, double *out, double min, double max,
                       int *error);
void egads_gauss(prngctx_t *ctx, double *out, double mu, double sigma,
                 int *error);
void egads_normalvariate(prngctx_t *ctx, double *out, double mu, double sigma,
                         int *error);
void egads_lognormalvariate(prngctx_t *ctx, double *out, double mu, double sigma,
                            int *error);
void egads_paretovariate(prngctx_t *ctx, double *out, double alpha, int *error);
void egads_weibullvariate(prngctx_t *ctx, double *out, double alpha, double beta,
                          int *error);
void egads_expovariate(prngctx_t *ctx, double *out, double lambda, int *error);
void egads_betavariate(prngctx_t *ctx, double *out, double alpha, double beta,
                       int *error);
void egads_cunifvariate(prngctx_t *ctx, double *out, double mean, double arc,
                       int *error);
```

The egads_randreal( ) function produces a real number between 0 and 1 (inclusive) that is uniformly distributed across that space. To get a real number in a particular range, use the function egads_randuniform( ). For those needing random data in a nonuniform distribution, there are numerous functions in the previous API to produce random floats in various common distributions. The semantics for these functions should be obvious to anyone who is already familiar with the particular distribution.

```
void egads_randstring(prngctx_t *ctx, char *out, int len, int *error);
```

The function egads_randstring( ) generates a random string that can contain any printable character. That is, it produces characters between ASCII 33 and ASCII 126 (inclusive) and thus contains no whitespace characters. The output buffer must be allocated by the caller, and it must be at least as long as the specified length plus an additional byte to accommodate the terminating zero that the function will write to the buffer.

```
void egads_randfname(prngctx_t *ctx, char *out, int len, int *error);
```

The function egads_randfname( ) produces a random string suitable for use as a filename. Generally, you are expected to concatenate the generated string with a base path. This function expects the destination buffer to be allocated already, and to be allocated with enough space to hold the string plus a terminating NULL, which this function will add.

See Also

- EGADS by Secure Software, Inc.: *http://www.securesoftware.com/egads*
- Recipe 11.2

11.9 Using the OpenSSL Random Number API

Problem

Many functions in the OpenSSL library require the use of the OpenSSL pseudo-random number generator. Even if you use something like */dev/urandom* yourself, OpenSSL will use its own API under the hood and thus must be seeded properly.

Unfortunately, some platforms and some older versions of OpenSSL require the user to provide a secure seed. Even modern implementations of OpenSSL merely read a seed from */dev/urandom* when it is available; a paranoid user may wish to do better.

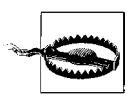
When using OpenSSL, you may want to use the provided PRNG for other needs, just for the sake of consistency.

Solution

OpenSSL exports its own API for manipulating random numbers, which we discuss in the next section. It has its own cryptographic PRNG, which must be securely seeded.

To use the OpenSSL randomness API, you must include *openssl/rand.h* in your code and link against the OpenSSL crypto library.

Discussion



Be sure to check all return values for the functions below; they may return errors.

With OpenSSL, you get a cryptographic PRNG but no entropy gateway. Recent versions of OpenSSL try to seed its PRNG using */dev/random*, */dev/urandom*, and EGD, trying several well-known EGD socket locations. However, OpenSSL does not try to estimate how much entropy its PRNG has. It is up to you to ensure that it has enough before the PRNG is used.

On Windows systems, a variety of sources are used to attempt to gather entropy, although none of them actually provides much real entropy. If an insufficient amount of entropy is available, OpenSSL will issue a warning, but it will keep going

anyway. You can use any of the sources we have discussed elsewhere in this chapter for seeding the OpenSSL PRNG. Multiple API functions are available that allow seed information to be passed to the PRNG.

One such function is `RAND_seed()`, which allows you to pass in arbitrary data that should be completely full of entropy. It has the following signature:

```
void RAND_seed(const void *buf, int num);
```

This function has the following arguments:

`buf`
: Buffer containing the entropy to seed the PRNG.

`num`
: Length of the seed buffer in bytes.

If you have data that you believe contains entropy but does not come close to one bit of entropy per bit of data, you can call `RAND_add()`, which is similar to `RAND_seed()` except that it allows you to provide an indication of how many bits of entropy the data has:

```
void RAND_add(const void *buf, int num, double entropy);
```

If you want to seed from a device or some other file (usually, you only want to use a stored seed), you can use the function `RAND_load_file()`, which will read the requested number of bytes from the file. Because there is no way to determine how much entropy is contained in the data, OpenSSL assumes that the data it reads from the file is purely entropic.

```
int RAND_load_file(const char *filename, long max_bytes);
```

If –1 is specified as the length parameter to this function, it reads the entire file. This function returns the number of bytes read. The function can be used to read from the */dev/random* and */dev/urandom* devices on Unix systems that have them, but you must make sure that you don't specify –1 for the number of bytes to read from these files; otherwise, the function will never return!

To implement PRNG state saving with OpenSSL, you can use `RAND_write_file()`, which writes out a representation of the PRNG's internal state that can be used to reseed the PRNG when needed (e.g., after a reboot):

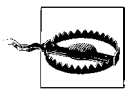
```
int RAND_write_file(const char *filename);
```

If there is any sort of error, `RAND_write_file()` will return –1. Note that the system may write a seed file without enough entropy, in which case it will also return –1. Otherwise, this function returns the number of bytes written to the seed file.

To obtain pseudo-random data from the PRNG, use the function `RAND_bytes()`:

```
int RAND_bytes(unsigned char *buf, int num);
```

If the generator is not seeded with enough entropy, this function could produce output that may be insecure. In such a case, the function will return 0. Make sure that you always check for this condition!



Do not, under any circumstances, use the API function, `RAND_pseudo_bytes()`. It is not a cryptographically strong PRNG and therefore is not worth using for anything that has even a remote possibility of being security-relevant.

You can implement `spc_rand()`, the cryptographic pseudo-randomness function from Recipe 11.2, by simply calling `RAND_bytes()` and aborting if that function returns 0.

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>

unsigned char *spc_rand(unsigned char *buf, size_t l) {
  if (!RAND_bytes(buf, l)) {
    fprintf(stderr, "The PRNG is not seeded!\n");
    abort();
  }
  return buf;
}
```

See Also

Recipe 11.2

11.10 Getting Random Integers

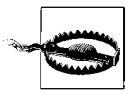
Problem

Given a pseudo-random number generation interface that returns an array of bytes, you need to get random values in the various integer data types.

Solution

For dealing with an integer that can contain any value, you may simply write bytes directly into every byte of the integer.

Discussion



Do not use this solution for getting random floating-point values; it will not produce numbers in a uniform distribution because of the mechanics of floating-point formats.

To get a random integer value, all you need to do is fill the bytes of the integer with random data. You can do this by casting a pointer to an integer to a binary string, then passing it on to a function that fills a buffer with random bytes. For example, use the following function to get a random unsigned integer, using the `spc_rand()` interface defined in Recipe 11.2:

```
unsigned int spc_rand_uint(void) {
  unsigned int res;

  spc_rand((unsigned char *)&res, sizeof(unsigned int));
  return res;
}
```

This solution can easily be adapted to other integer data types simply by changing all the instances of `unsigned int` to the appropriate type.

See Also

Recipe 11.2

11.11 Getting a Random Integer in a Range

Problem

You want to choose a number in a particular range, with each possible value equally likely. For example, you may be simulating dice rolling and do not want any number to be more likely to come up than any other. You want all numbers in the range to be possible values, including both endpoints. That is, if you ask for a number between 1 and 6, you'd like both 1 and 6 to be as likely as 2, 3, 4, or 5.

Solution

There are multiple ways to handle this problem. The most common is the least correct, and that is to simply reduce a random integer (see Recipe 11.10) modulo the size of the range and add to the minimum possible value. This can lead to slight biases in your random numbers, which can sometimes lead to practical attacks, because it means that some outputs are more likely than others.

We discuss more exact solutions in the next section.

Discussion

In all cases, you will start with a function that gives you a random unsigned number that can be any value, such as `spc_rand_uint()` from Recipe 11.10. You will mold numbers returned from this function into numbers in a specific range.

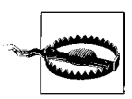
If you need random numbers in a particular range, the general approach is to get a number between zero and one less than the number of values in the range, then add the result to the smallest possible value in the range.

Ideally, when picking a random number in a range, you would like every possible value to be equally likely. However, if you map from an arbitrary unsigned integer into a range, where the range does not divide evenly into the number of possible integers, you are going to run into problems.

Suppose you want to create a random number in a range using an unsigned 8-bit type. When you get a random unsigned 8-bit value, it can take on 256 possible values, from 0 to 255. If you are asking for a number between 0 and 9 inclusive, you could simply take a random value and reduce it modulo 10.

The problem is that the numbers 0 through 5 are more likely values than are 6 through 9. 26 possible values will reduce to each number between 0 and 5, but only 25 values will yield 6 through 9.

In this example, the best way to solve this problem is to discard any random numbers that fall in the range 250-255. In such a case, simply get another random value and try again. We took this approach in implementing the function `spc_rand_range()`. The result will be a number greater than or equal to a minimum value and less than or equal to maximum value.



Some programmers may expect this function to exclude the upper limit as a possible value; however, we implement this function in such a way that it is not excluded.

```
#include <limits.h>
#include <stdlib.h>

int spc_rand_range(int min, int max) {
  unsigned int rado;
  int          range = max - min + 1;

  if (max < min) abort(); /* Do your own error handling if appropriate.*/
  do {
    rado = spc_rand_uint();
  } while (rado > UINT_MAX - (UINT_MAX % range));
  return min + (rado % range);
}
```

You might worry about a situation where performance suffers because this code has to retry too many times. The worst case for this solution is when the size of the range is `UINT_MAX / 2 + 1`. Even in such a case, you would not expect to call `spc_rand_uint( )` very many times. The average number of times it would be called here would be slightly less than two. While the worst-case performance is theoretically unbounded, the chances of calling `spc_rand_uint( )` more than a dozen times are essentially zero. Therefore, this technique will not have a significant performance impact for most applications.

If you are okay with some items being slightly more likely than others, there are two different things you can do, both of which are fairly easy. First, you can perform a modulo operation and an addition to get the integer in the right range, and just not worry about the fact that some values are more likely than others:

```
#include <stdlib.h>

int spc_rand_range(int min, int max) {
  if (max < min) abort( );
  return min + (spc_rand_uint( ) % (max - min + 1));
}
```

Of course, this solution clumps together all the values that are more likely to be chosen, which is somewhat undesirable. As an alternative, you can spread them out by using division and rounding down, instead of a simple modulus:

```
#include <limits.h>

int spc_rand_range(int min, int max) {
  if (max < min) abort( );
  return min + (int)((double)spc_rand_uint( ) *
                     (max - min + 1) / (double)UINT_MAX) % (max - min);
}
```

Note the modulo operation in this solution. That is to prevent getting a value that is out of range in the very rare occasion that `spc_rand_uint( )` returns `UINT_MAX`.

See Also

Recipe 11.10

11.12 Getting a Random Floating-Point Value with Uniform Distribution

Problem

When looking for a random floating-point number, we usually want a value between 0 and 1 that is just as likely to be between 0 and 0.1 as it is to be between 0.9 and 1.

Solution

Because of the way that floating-point numbers are stored, simply casting bits to a float will make the distribution nonuniform. Instead, get a random unsigned integer, and divide.

Discussion

Because integer values are uniformly distributed, you can get a random integer and divide so that it is a value between 0 and 1:

```
#include <limits.h>

double spc_rand_real(void) {
  return ((double)spc_rand_uint( )) / (double)UINT_MAX;
}
```

Note that to get a random number between 0 and *n*, you can multiply the result of `spc_rand_real( )` by *n*. To get a real number within a range inclusive of the range's bounds, do this:

```
#include <stdlib.h>

double spc_rand_real_range(double min, double max) {
  if (max < min) abort( );
  return spc_rand_real( ) * (max - min) + min;
}
```

11.13 Getting Floating-Point Values with Nonuniform Distributions

Problem

You want to select random real numbers in a nonuniform distribution.

Solution

The exact solution varies depending on the distribution. We provide implementations for many common distributions in this recipe.

Discussion

Do not worry if you do not know what a particular distribution is; if you have never seen it before, you really should not need to know what it is. A uniform distribution (as discussed in Recipe 11.12) is far more useful in most cases.

In all cases, we start with a number with uniform distribution using the API from Recipe 11.12.

Note that these functions use math operations defined in the standard math library. On many platforms, you will have to link against the appropriate library (usually by adding `-lm` to your link line).

```
#include <math.h>

#define NVCONST 1.7155277699141

double spc_rand_normalvariate(double mu, double sigma) {
  double myr1, myr2, t1, t2;

  do {
    myr1 = spc_rand_real( );
    myr2 = spc_rand_real( );
    t1 = NVCONST * (myr1 - 0.5) / myr2;
    t2 = t1 * t1 / 4.0;
  } while (t2 > -log(myr2));
  return mu + t1 * sigma;
}

double spc_rand_lognormalvariate(double mu, double sigma) {
  return exp(spc_rand_normalvariate(mu, sigma));
}

double spc_rand_paretovariate(double alpha) {
  return 1.0 / pow(spc_rand_real( ), 1.0 / alpha);
}

double spc_rand_weibullvariate(double alpha, double beta) {
  return alpha * pow(-log(spc_rand_real( )), 1.0 / beta);
}

double spc_rand_expovariate(double lambda) {
  double myr = spc_rand_real( );

  while (myr <= 1e-7)
    myr = spc_rand_real( );
  return -log(myr) / lambda;
}

double spc_rand_betavariate(double alpha, double beta) {
  double myr1, myr2;

  myr1 = spc_rand_expovariate(alpha);
  myr2 = spc_rand_expovariate(1.0 / beta);
  return myr2 / (myr1 + myr2);
}

#define SPC_PI 3.1415926535
```

```
double spc_rand_cunifvariate(double mean, double arc) {
  return (mean + arc * (spc_rand_real( ) - 0.5)) / SPC_PI;
}
```

See Also

Recipe 11.12

11.14 Getting a Random Printable ASCII String

Problem

You want to get a random printable ASCII string.

Solution

If you do not want whitespace characters, the printable ASCII characters have values from 33 to 126, inclusive. Simply get a random number in that range for each character.

If you want to choose from a different character set (such as the base64 character set), map each character to a specific numeric value between 0 and the number of characters you have. Select a random number in that range, and map the number back to the corresponding character.

Discussion

The code presented in this section returns a random ASCII string of a specified length, where the specified length includes a terminating `NULL` byte. We use the printable ASCII characters, meaning that we never output whitespace or control characters.

Assuming a good underlying infrastructure for randomness, each character should be equally likely. However, the ease with which an attacker can guess a single random string is related not only to the entropy in the generator, but also to the length of the output. If you use a single character, there are only 94 possible values, and a guess will be right with a probability of 1/94 (not having entropy can give the attacker an even greater advantage).

As a result, your random strings should use no fewer than 10 random characters (not including the terminating `NULL` byte), which gives you about 54 bits of security. For a more conservative security margin, you should go for 15 to 20 characters.

```
#include <stdlib.h>

char *spc_rand_ascii(char *buf, size_t len) {
```

```
 char *p = buf;

  while (--len)
    *p++ = (char)spc_rand_range(33, 126);
  *p = 0;
  return buf;
}
```

11.15 Shuffling Fairly

Problem

You have an ordered list of items that you would like to shuffle randomly, then visit one at a time. You would like to do so securely and without biasing any element.

Solution

For each index, swap the item at that index with the item at a random index that has not been fully processed, including the current index.

Discussion

Performing a statistically fair shuffle is actually not easy to do right. Many developers who implement a shuffle that seems right to them off the top of their heads get it wrong.

We present code to shuffle an array of integers here. We perform a statistically fair shuffle, using the `spc_rand_range( )` function from Recipe 11.11.

```
#include <stdlib.h>

void spc_shuffle(int *items, size_t numitems) {
  int    tmp;
  size_t swapwith;

  while (--numitems) {
    /* Remember, it must be possible for a value to swap with itself */
    swapwith = spc_rand_range(0, numitems);
    tmp = items[swapwith];
    items[swapwith] = items[numitems];
    items[numitems] = tmp;
  }
}
```

If you need to shuffle an array of objects, you can use this function to first permute an array of integers, then use that permutation to reorder the elements in your array. That is, if you have three database records, and you shuffle the list [1, 2, 3], getting [3, 1, 2], you would build a new array consisting of the records in the listed order.

See Also

Recipe 11.11

11.16 Compressing Data with Entropy into a Fixed-Size Seed

Problem

You are collecting data that may contain entropy, and you will need to output a fixed-size seed that is smaller than the input. That is, you have a lot of data that has a little bit of entropy, yet you need to produce a fixed-size seed for a pseudo-random number generator. At the same time, you would like to remove any statistical biases (patterns) that may be lingering in the data, to the extent possible.

Alternatively, you have data that you believe contains one bit of entropy per bit of data (which is generally a bad assumption to make, even if it comes from a hardware generator; see Recipe 11.19), but you'd like to remove any patterns in the data that could facilitate analysis if you're wrong about how much entropy is there. The process of removing patterns is called *whitening*.

Solution

You can use a cryptographic hash function such as SHA1 to process data into a fixed-size seed. It is generally a good idea to process data incrementally, so that you do not need to buffer potentially arbitrary amounts of data with entropy.

Discussion

Be sure to estimate entropy conservatively. (See Recipe 11.19.)

It is a good idea to use a cryptographic algorithm to compress the data from the entropy source into a seed of the right size. This helps preserve entropy in the data, up to the output size of the message digest function. If you need fewer bytes for a seed than the digest function produces, you can always truncate the output. In addition, cryptographic processing effectively removes any patterns in the data (assuming that the hash function is a pseudo-random function). Patterns in the data can help facilitate breaking an entropy source (in part or in full), particularly when that source does not actually produce as much entropy as was believed.

Most simpler compression methods are not going to do as good a job at preserving entropy. For example, suppose that your compression function is simply XOR. More concretely, suppose you need a 128-bit seed, and you XOR data in 16-byte chunks into a single buffer. Suppose also that you believe you have collected 128 bits of entropy from numerous calls to a 128-bit timestamp operation.

In any particular timestamp function, all of the entropy is going to live in a few of the least significant bits. Now suppose that only two or three of those bits are likely to contain any entropy. The XOR-everything strategy will leave well over 120 bits of the result trivial to guess. The remaining eight bits can be attacked via brute force. Therefore, even if the input had 128 bits of entropy, the XOR-based compression algorithm destroyed most of the entropy.

SHA1 is good for these purposes. See Recipe 6.5 for how to use SHA1.

See Also

Recipes 6.5, 11.19

11.17 Getting Entropy at Startup

Problem

You want to be able to seed a cryptographic pseudo-random number generator securely as soon as a machine boots, without having to wait for interaction from the user or other typical sources of entropy.

Solution

If you have never been able to seed the generator securely, prompt for entropy on install or first use (see Recipes 11.20 and 11.21).

Otherwise, before shutting down the generator, have it output enough material to reseed itself to a file located in a secure part of the filesystem. The next time the generator starts, read the seed file and use the data to reseed, as discussed in Recipe 11.6.

Discussion

It can take a noticeable amount of time for a PRNG to gather enough entropy that it is safe to begin outputting random data. On some systems with */dev/random* as the entropy source, users could be forced to sit around indefinitely, not knowing how to get more entropy into the system.

It would be nice if you did not have to collect entropy every time a program starts up or the machine reboots. You should need to get entropy only once per application, then be able to store that entropy until the next time you need it.

If you have sufficient trust in the local filesystem, you can certainly do this by writing out a seed to a file, which you can later use to initialize the generator when it starts back up. Of course, you need to make sure that there are no possible security issues in file access. In particular, the location you use for saving seed files needs to be a secure location (see Recipe 2.4 for more on how to ensure this programmatically). In addition, you should be sure not to store a seed on a potentially untrusted filesystem, such as an NFS mount, and you should probably use advisory file locking in an attempt to defeat any accidental race conditions on the seed file.

You should also consider the threat of an insider with physical access to the machine compromising the seed file. For that reason, you should always strive to add new entropy to a generator after every startup as soon as enough bits can be collected. Using a seed file should be considered a stopgap measure to prevent stalling on startup.

See Also

Recipes 2.4, 11.6, 11.20, 11.21

11.18 Statistically Testing Random Numbers

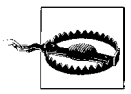
Problem

You are using a hardware random number generator or some other entropy source that hasn't been cryptographically postprocessed, and you would like to determine whether it ever stops producing quality data. Alternatively, you want to have your generator be FIPS 140 compliant (perhaps for FIPS certification purposes).

Solution

FIPS 140-2 tests, which are ongoing throughout the life of the generator, are necessary for FIPS 140 compliance. For actual statistical tests of data produced by a source, the full set of tests provided by FIPS 140-1 are much more useful, even though they are now irrelevant to the FIPS certification process.

Discussion



FIPS 140 tests are useful for proving that a stream of random numbers are weak, but the tests don't demonstrate at all when the numbers are good. In particular, it is incredibly easy to have a weak generator yet still pass FIPS tests by processing data with a cryptographic primitive like SHA1 before running the tests. FIPS 140 is only useful as a safety net, for when an entropy source you think is strong turns out not to be.

FIPS 140 is a standard authored by the U.S. National Institute of Standards and Technology (NIST; see *http://csrc.nist.gov/cryptval/*). The standard details general security requirements for cryptographic software deployed in government systems (primarily cryptographic "providers"). There are many aspects to the FIPS 140 standard, one of which is a set of tests that all entropy harvesters and pseudo-random number generators must be able to run to achieve certification.

FIPS 140-1 was the original standard and had several tests for random number sources; most of these occurred on startup, but one occurred continuously. Those tests only needed to be implemented for the two highest levels of FIPS compliance (Levels 3 and 4), which few applications sought.

In FIPS 140-2, only a single test from FIPS 140-1 remains. This test is mandatory any time a random number generator or entropy source is used.

Although the FIPS 140-1 standard is being obsoleted by 140-2, it is important to note that a module can routinely fail the FIPS 140-1 tests and still be FIPS 140-1 compliant. For Level 3 compliance, the user must be able to run the tests on command, and if the tests fail, the module must go into an error state. For Level 4 compliance, the module must comply with the requirements of Level 3, plus the tests must be run at "power-up." A weak random number generator, such as the one implemented by the standard C library function `rand()`, should be able to get Level 3 certification easily.

FIPS 140-1 testing is a reasonable tool for ensuring that entropy sources are producing quality data, if those entropy sources are not using any cryptographic operations internally. If they are, the entropy source will almost certainly pass these tests, even if it is a very poor entropy source. For the same reason, this set of tests is not good for testing cryptographic PRNGs, because all such generators will pass these tests with ease, even if they are poor. For example, simply hashing an incrementing counter that starts at zero using MD5 will produce a data stream that passes these tests, even though the data in that stream is easily predictable.

FIPS 140-2 testing generally is not very effective unless a failed hardware device starts producing a repeating pattern (e.g., a string of zero bits). The FIPS 140-2 test consists of comparing consecutive generator outputs (on a large boundary size; see the next section). If your "random number generator" consists only of an ever-incrementing 128-bit counter, you will never fail this test.

For this reason, we think the full suite of FIPS 140-1 tests is the way to go any time you really want to test whether an entropy source is producing good data, and it is a good idea to run these tests on startup, and then periodically, when feasible. You should always support the continuous test that FIPS 140-2 mandates whenever you are using hardware random number generators that could possibly be prone to disastrous failure, because it might help you detect such a failure.

FIPS 140-1 power-up and on-demand tests

The FIPS 140-1 standard specifies four statistical tests that operate on 20,000 consecutive bits of output (2,500 bytes).

In the first test, the "Monobit" test, the number of bits set to 1 are counted. The test passes if the number of bits set to 1 is within a reasonable proximity to 10,000. The function `spc_fips_monobit( )`, implemented as follows, performs this test, returning 1 on success, and 0 on failure.

```
#define FIPS_NUMBYTES     2500
#define FIPS_MONO_LOBOUND 9654
#define FIPS_MONO_HIBOUND 10346

/* For each of the 256 possible bit values, how many 1 bits are set? */
static char nb_tbl[256] = {
  0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3,
  4, 3, 4, 4, 5, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4,
  4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2,
  3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5,
  4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 1, 2, 2, 3, 2, 3, 3,
  4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3,
  3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5,
  6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6,
  4, 5, 5, 6, 5, 6, 6, 7, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5,
  6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8
};

int spc_fips_monobit(unsigned char data[FIPS_NUMBYTES]) {
  int i, result;

  for (i = result = 0;  i < FIPS_NUMBYTES;  i++)
    result += nb_tbl[data[i]];
  return (result > FIPS_MONO_LOBOUND && result < FIPS_MONO_HIBOUND);
}
```

The second test is the "Poker" test, in which the data is broken down into consecutive 4-bit values to determine how many times each of the 16 possible 4-bit values appears. The square of each result is then added together and scaled to see whether the result falls in a particular range. If so, the test passes. The function `spc_fips_poker( )`, implemented as follows, performs this test, returning 1 on success and 0 on failure:

```
#define FIPS_NUMBYTES      2500
#define FIPS_POKER_LOBOUND 1.03
```

```
#define FIPS_POKER_HIBOUND 57.4

int spc_fips_poker(unsigned char data[FIPS_NUMBYTES]) {
  int    i;
  long   counts[16] = {0,}, sum = 0;
  double result;

  for (i = 0;  i < FIPS_NUMBYTES;  i++) {
    counts[data[i] & 0xf]++;
    counts[data[i] >> 4]++;
  }
  for (i = 0;  i < 16;  i++)
    sum += (counts[i] * counts[i]);
  result = (16.0 / 5000) * (double)sum - 5000.0;
  return (result > FIPS_POKER_LOBOUND && result < FIPS_POKER_HIBOUND);
}
```

The third and fourth FIPS 140-1 statistical tests are implemented as follows to run in parallel in a single routine. The third test, the "Runs" test, goes through the data stream and finds all the "runs" of consecutive bits that are identical. The test then counts the maximum length of each run. That is, if there are three consecutive zeros starting at the first position, that's one run of length three, but it doesn't count as any runs of length two or any runs of length one. Runs that are longer than six bits are counted as a six-bit run. At the end, for each length of run, the count for consecutive zeros of that run length and the count for consecutive ones are examined. If either fails to fall within a specified range, the test fails. If all of the results are in an appropriate range for the run length in question, the test passes.

The fourth test, the "Long Runs" test, also calculates runs of bits. The test looks for runs of 34 bits or longer. Any such runs cause the test to fail; otherwise, it succeeds.

```
#define FIPS_NUMBYTES  2500
#define FIPS_LONGRUN   34
#define FIPS_RUNS_1_LO 2267
#define FIPS_RUNS_1_HI 2733
#define FIPS_RUNS_2_LO 1079
#define FIPS_RUNS_2_HI 1421
#define FIPS_RUNS_3_LO 502
#define FIPS_RUNS_3_HI 748
#define FIPS_RUNS_4_LO 223
#define FIPS_RUNS_4_HI 402
#define FIPS_RUNS_5_LO 90
#define FIPS_RUNS_5_HI 223
#define FIPS_RUNS_6_LO 90
#define FIPS_RUNS_6_HI 223

/* Perform both the "Runs" test and the "Long Run" test */
int spc_fips_runs(unsigned char data[FIPS_NUMBYTES]) {
  /* We allow a zero-length run size, mainly just to keep the array indexing less
   * confusing.  It also allows us to set cur_val arbitrarily below (if the first
   * bit of the stream is a 1, then runs[0] will be 1; otherwise, it will be 0).
   */
  int           runs[2][7] = {{0,},{0,}};
```

```
    int           cur_val, i, j, runsz;
    unsigned char curr;

    for (cur_val = i = runsz = 0;  i < FIPS_NUMBYTES;  i++) {
      curr = data[i];
      for (j = 0;  j < 8;  j++) {
        /* Check to see if the current bit is the same as the last one */
        if ((curr & 0x01) ^ cur_val) {
          /* The bits are different. A run is over, and a new run of 1 has begun */
          if (runsz >= FIPS_LONGRUN) return 0;
          if (runsz > 6) runsz = 6;
          runs[cur_val][runsz]++;
          runsz = 1;
          cur_val = (cur_val + 1) & 1; /* Switch the value. */
        } else runsz++;
        curr >>= 1;
      }
    }

    return (runs[0][1] > FIPS_RUNS_1_LO && runs[0][1] < FIPS_RUNS_1_HI &&
            runs[0][2] > FIPS_RUNS_2_LO && runs[0][2] < FIPS_RUNS_2_HI &&
            runs[0][3] > FIPS_RUNS_3_LO && runs[0][3] < FIPS_RUNS_3_HI &&
            runs[0][4] > FIPS_RUNS_4_LO && runs[0][4] < FIPS_RUNS_4_HI &&
            runs[0][5] > FIPS_RUNS_5_LO && runs[0][5] < FIPS_RUNS_5_HI &&
            runs[0][6] > FIPS_RUNS_6_LO && runs[0][6] < FIPS_RUNS_6_HI &&
            runs[1][1] > FIPS_RUNS_1_LO && runs[1][1] < FIPS_RUNS_1_HI &&
            runs[1][2] > FIPS_RUNS_2_LO && runs[1][2] < FIPS_RUNS_2_HI &&
            runs[1][3] > FIPS_RUNS_3_LO && runs[1][3] < FIPS_RUNS_3_HI &&
            runs[1][4] > FIPS_RUNS_4_LO && runs[1][4] < FIPS_RUNS_4_HI &&
            runs[1][5] > FIPS_RUNS_5_LO && runs[1][5] < FIPS_RUNS_5_HI &&
            runs[1][6] > FIPS_RUNS_6_LO && runs[1][6] < FIPS_RUNS_6_HI);
}
```

The FIPS continuous output test

The FIPS continuous output test requires that random number generators (which would include both entropy sources and PRNGs) have the data they are going to produce broken up into "blocks" of at least 16 bytes. If the generator has a "natural" block size of greater than 16 bytes, that should always get used. Otherwise, any size 16 bytes or greater can be used. We recommend never using blocks larger than 16 bytes (unless required) because the underlying generator uses larger blocks naturally.*

This test collects the first block of output and never gives it to anyone. Instead, it is compared against the second block of output and thrown away. The second block may be output if it is not identical to the first block; otherwise, the system must fail.

* Usually, entropy sources do not have a natural block size that large, if they have one at all (there is usually a somewhat artificial block size, such as the width of the memory you read to query the source).

The second output is also saved and is then compared to the third block. This process continues for all generator outputs.

The following (non-thread-safe) code adds a FIPS-compliant wrapper to the `spc_entropy()` function from Recipe 11.2 (note that this assumes that `spc_entropy()` does not cryptographically postprocess its data, because otherwise the test is all but worthless).

```
#include <stdlib.h>
#include <string.h>
#define RNG_BLOCK_SZ 16

char *spc_fips_entropy(char *outbuf, int n) {
  static int  i, bufsz = -1;
  static char b1[RNG_BLOCK_SZ], b2[RNG_BLOCK_SZ];
  static char *last = b1, *next = b2;
  char        *p = outbuf;

  if (bufsz == -1) {
    spc_entropy(next, RNG_BLOCK_SZ);
    bufsz = 0;
  }
  while (bufsz && n--)
    *p++ = last[RNG_BLOCK_SZ - bufsz--];
  while (n >= RNG_BLOCK_SZ) {
    /* Old next becomes last here */
    *next ^= *last;
    *last ^= *next;
    *next ^= *last;
    spc_entropy(next, RNG_BLOCK_SZ);
    for (i = 0;  i < RNG_BLOCK_SZ;  i++)
      if (next[i] != last[i]) goto okay;
    abort( );
okay:
    memcpy(p, next, RNG_BLOCK_SZ);
    p += RNG_BLOCK_SZ;
    n -= RNG_BLOCK_SZ;
  }
  if (n) {
    *next ^= *last;
    *last ^= *next;
    *next ^= *last;
    spc_entropy(next, RNG_BLOCK_SZ);
    for (i = 0;  i < RNG_BLOCK_SZ;  i++)
      if (next[i] != last[i])
         goto okay2;
    abort( );
okay2:
    memcpy(p, next, n);
    bufsz = RNG_BLOCK_SZ - n;
  }
  return outbuf;
}
```

See Also

- NIST Cryptographic Module Validation Program home page: *http://csrc.nist.gov/cryptval/*
- Recipe 11.2

11.19 Performing Entropy Estimation and Management

Problem

You are collecting your own entropy, and you need to determine when you have collected enough data to use the entropy.

Solution

At the highest level, the solution is to be incredibly conservative in entropy estimation. In the discussion, we will examine general practices and guidelines for particular sources.

Discussion

Fundamentally, the practical way to look at entropy is as a measurement of how much information in a piece of "random" data an attacker can glean about your randomness infrastructure. For example, if you have a trusted channel where you get 128 bits of data, the question we are really asking is this: how much of that data is provided to an attacker through whatever data channels are available to him? The complexity of an attack is based on how much data an attacker has to guess.

Clearly, in the practical sense, a single piece of data can have different amounts of entropy for different people. For example, suppose that we use the machine boot time to the nearest second as a source of entropy. An attacker who has information about the system startup time narrowing it down to the nearest week still has a much harder problem than an attacker who can narrow it down to a 10-second period. The second attacker can try all 10 possible starting values and see if he gets the correct value. The first has far, far more values to try before finding the original value.

In practice, it turns out that boot time is often an even more horrible source of entropy than we have already suggested. The *nmap* tool can often give the system uptime of a remote host with little effort, although this depends on the operating system and the firewall configuration of the host being targeted.

The basic lesson here is that, before you decide how to estimate entropy, you should figure out what your threat model is. That is, what kinds of attacks are you worried

about? For example, it is possible to monitor electromagnetic signals coming from a computer to capture every signal coming from that machine. The CIA has been known to do this with great success. In such a case, there may be absolutely no entropy at all without some sort of measures to prevent against such attacks.

Most people are not worried about such a threat model because the attack requires a high degree of skill. In addition, it generally requires placing another machine in close proximity to the machine being targeted. A more realistic assumption, is that someone with a local (nonroot) account on the machine will try to launch an attack. Quite a bit of the entropy an interactive Unix system typically gathers can be observed by such an attacker, either directly or indirectly.

If you are not worried about people with access to the local system, we believe you should at least assume that attackers will somehow find their way onto the same network segment as the machine that's collecting entropy. You should therefore assume that there is little entropy to be had in network traffic that the machine receives, because other machines on the network may be able to see the same traffic, and even inject new traffic.

Another threat you might want to consider is the possibility of an attacker's finding a way to pollute or destroy one or more entropy sources. For example, suppose you are using a hardware random number generator. The attacker may not have local account access and may not have the resources or know-how for an electromagnetic signal capture attack. However, there may be an easy way to break the physical random number generator and get it to produce a big string of zeros.

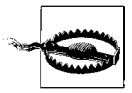
Certainly, you can use FIPS 140 testing as a preventive measure here, as discussed in Recipe 11.18. However, those tests are not very reliable. You might still want to assume that entropy sources may not provide any entropy at all.

Such attacks are probably worst-case in most practical systems. You can prevent against tainted entropy sources by using multiple entropy sources, under the assumption (which is probably pretty reasonable in practice) that an attacker will not have the resources to effectively taint more than one source at once.

With such an assumption, you can estimate entropy as if such attacks are not possible, then subtract out the entropy estimate for the most plentiful entropy source. For example, suppose that you want to collect a 128-bit seed, and you read keyboard input and also read separately from a fast hardware random number generator. With such a metric, you would assume that the hardware source (very likely to be the most plentiful) is providing no entropy. Therefore, you refuse to believe that you have enough entropy until your entropy estimate for the keyboard is 128 bits.

You can come up with more liberal metrics. For example, suppose you are collecting a 128-bit seed. You could have a metric that says you will believe you really have 128 bits of entropy when you have collected at least 160 bits total, where at least 80 of those bits are from sources other than the fastest source. This is a reasonable metric,

because even if a source does fail completely, you should end up with 80 bits of security on a 128-bit value, which is generally considered impractical to attack. (Thus, 80-bit symmetric keys are often considered more than good enough for all current security needs.)



One thing you should do to avoid introducing security problems by underestimating entropy is aggregate each entropy source independently, then mash everything together once you have met your output metric. One big advantage of such a technique is that it simplifies analysis that can lead to cryptographic assurance. To do this, you can have a collector for each entropy source. When you need an output, take the state of each entropy source and combine them somehow.

More concretely, you could use a SHA1 context for each entropy source. When an output is needed and the metrics are met, you can get the state of each context, XOR all the states together, and output that. Of course, remember that in this scenario, you will never have more entropy than the output size of the hash function.

Now assume that the attacker cannot make a source fail; she can only take measurements for guessing attacks. We will talk about estimating the amount of entropy in a piece of data, assuming two different threat models: with the first, the attacker has local but nonprivileged access to the machine,* and in the second, the attacker has access to the local network segment.

In the second threat model, assume this attacker can see everything external that goes on with the application by somehow snooping network traffic. In addition, assume that the attacker knows all about the operational environment of the machine on which the application runs. For example, assume that she knows the operating system, the applications running on the system, approximately when the machine rebooted, and so on. These assumptions mean that a savvy attacker can actually figure out a fair amount about the machine's state from observing network traffic.

Unfortunately, the first problem we encounter when trying to estimate entropy is that, while there is an information-theoretic approach to doing so, it is actually ridiculously difficult to do in practice. Basically, we can model how much entropy is in data only once we have a complete understanding of that data, as well as a complete understanding of all possible channels available to an attacker for measuring the parts of that data that the attacker would not otherwise be able to figure out from patterns in the data.

* If an attacker already has privileged access to a machine, you probably have more important issues than her guessing random numbers.

Particularly when an attacker may have local access to a machine, it can be a hopeless task to figure out what all the possible channels are. Making things difficult is the fact that machines behave very deterministically. This behavior means that the only points where there is the possibility for any entropy at all is when outside inputs are added to the system.

The next problem is that, while a trusted entropy accumulator might be able to take some measurements of the outside data, there may be nothing stopping an attacker from taking measurements of the exact same data. For example, suppose that an operating system uses keyboard strokes as an entropy source. The kernel measures the keystroke and the timestamp associated with the key press. An attacker may not be able to measure keystrokes generated by other users, but he should be able to add his own keystrokes, which the operating system will assume is entropy. The attacker can also take his own timestamps, and they will be highly correlated to the timestamps the operating system takes.

If we need to use our own entropy-gathering on a system that does its own, we trust the operating system's infrastructure, and we use a different infrastructure (particularly in terms of the cryptographic design), measuring entropy that the system also measures will generally not be a problem.

For example, suppose that you have a user interactively type data on startup so that you can be sure there is sufficient entropy for a seed. If an attacker is a local nonprivileged user, you can hope that the exact timings and values of key-press information will contain some data the attacker cannot know and will need to guess. If the system's entropy collection system does its job properly, cryptographically postprocessing entropy and processing it only in large chunks, there should be no practical way to use system infrastructure as a channel of information on the internal state of your own infrastructure. This falls apart only when the cryptography in use is broken, or when entropy estimates are too low.

The worst-case scenario for collecting entropy is generally a headless server. On such a machine, there is often very little trustworthy entropy coming from the environment, because all input comes from the network, which should generally be largely untrusted. Such systems are more likely to request entropy frequently for things like key generation. Because there is generally little entropy available on such machines, resource starvation attacks can be a major problem when there are frequent entropy requests.

There are two solutions to this problem. The first is operational: get a good hardware random number generator and use it. The second is to make sure that you do not frequently require entropy. Instead, be willing to fall back on cryptographic pseudo-randomness, as discussed in Recipe 11.5.

If you take the second approach, you will only need to worry about collecting entropy at startup time, which may be feasible to do interactively. Alternatively, if

you use a seed file, you can just collect entropy at install time, at which point interacting with the person performing the install is not a problem.

Entropy in timestamps

For every piece of data that you think has entropy, you can try to get additional entropy by mixing a timestamp into your entropy state, where the timestamp corresponds to the time at which the data was processed.

One good thing here is that modern processors can generate very high-resolution timestamps. For example, the x86 `RDTSC` instruction has granularity related to the clock speed of the processor. The problem is that the end user often does not see anywhere near the maximum resolution from a timing source. In particular, processor clocks are usually running in lockstep with much slower bus clocks, which in turn are running in lockstep with peripheral clocks. Expert real-world analysis of event timings modulo these clock multiples suggests that much of this resolution is not random.

Therefore, you should always assume that your clock samples are no more accurate than the sampling speed of the input source, not the processor. For example, keyboards and mice generally use a clock that runs around 1 Khz, a far cry from the speed of the `RDTSC` clock.

Another issue with the clock is something known as a *back-to-back attack*, in which depending on the details of entropy events, an attacker may be able to force entropy events to happen at particular moments. For example, back-to-back short network packets can keep a machine from processing keyboard or mouse interrupts until the precise time it is done servicing a packet, which a remote attacker can measure by observing the change in response in the packets he sends.

To solve this problem, assume that you get no entropy when the delta between two events is close to the interrupt latency time. That works because both network packets and keystrokes will cause an interrupt.*

Timing data is generally analyzed by examining the difference between two samples. Generally, the difference between two samples will not be uniformly distributed. For example, when looking at multiple such deltas, the high-order bits will usually be the same. The floor of the base 2 logarithm of the delta would be the theoretical maximum entropy you could get from a single timestamp, measured in bits. For example, if your delta between two timestamps were, in hex, `0x9B` (decimal 155), the maximum number of bits of entropy you could possibly have is 7, because the log of 155 is about 7.28.

* Some operating systems can mitigate this problem, if supported by the NIC.

However, in practice, even that number is too high by a bit, because we always know that the most significant bit we count is a 1. Only the rest of the data is really likely to store entropy.

In practice, to calculate the maximum amount of entropy we believe we may have in the delta, we find the most significant 1 bit in the value and count the number of bits from that point forward. For example, there are five bits following the most significant 1 bit in `0x9B`, so we would count six. This is the same as taking the floor of the log, then subtracting one.

Because of the nonuniform nature of the data, we are only going to get some portion of the total possible entropy from that timestamp. Therefore, for a difference of `0x9B`, six bits is an overestimate. With some reasonable assumptions about the data, we can be sure that there is at least one fewer bit of entropy.

In practice, the problem with this approximation is that an attacker may be able to figure out the more significant bits by observation, particularly in a very liberal threat model, where all threats come from the network.

For example, suppose you're timing the entropy between keystrokes, but the keystrokes come from a computer on the network. Even if those keystrokes are protected by encryption, an attacker on the network will know approximately when each keystroke enters the system.

In practice, the latency of the network and the host operating system generally provides a tiny bit of entropy. On a Pentium 4 using `RDTSC`, we would never estimate this amount at above 2.5 bits for any application. However, if you can afford not to do so, we recommend you do not count it.

The time where you may want to count it is if you are gathering input from a source where the source might actually come from a secure channel over the network (such as a keyboard attacked to a remote terminal), and you are willing to be somewhat liberal in your threat model with respect to the network. In such a case, we might estimate a flat three bits of entropy per character,* which would include the actual entropy in the value of that character.

In summary, our recommendations for timestamps are as follows:

- Keep deltas between timestamps. Do not count any entropy for the first timestamp, then estimate entropy as the number of bits to the right of the most significant bit in the delta, minus one.
- Only count entropy when the attacker does not have a chance of observing the timing information, whether directly or indirectly. For example, if you are timing entropy between keystrokes, be sure that the typing is done on the physical console, instead of over a network.

* Assuming that successive characters are different; otherwise, we would estimate zero bits of entropy.

- If you have to accept data from the network, make sure that it is likely to have some other entropy beyond the timing, and never estimate more than 2.5 bits of entropy per packet with a high-resolution clock (i.e., one running in the GHz range). If your clock has better than millisecond resolution and the processor is modern, it is probably reasonable to assume a half-bit of entropy on incoming network packets.

Entropy in a key press

As with any entropy source, when you are trying to get entropy from a key press, you should try to get entropy by taking a timestamp alongside the key press and estimate entropy as discussed in the previous subsection.

How much entropy should you count for the actual value of the key itself, though?

Of course, in practice, the answer has to do with how likely an attacker is to guess the key you are typing. If the attacker knows that the victim is typing *War and Peace*, there would be very little entropy (the only entropy would be from mistakes in typing or time between timestrokes).

If you are not worried about attacks from local users, we believe that a good, conservative approximation is one bit of entropy per character, if and only if the character is not identical to the previous character (otherwise, estimate zero). This assumes that the attacker has a pretty good but not exact idea of the content being typed.

If an attacker who is sending his own data into your entropy infrastructure is part of your threat model, we think the above metric is too liberal. If your infrastructure is multiuser, where the users are separated from each other, use a metric similar to the ones we discussed earlier for dealing with a single tainted data source.

For example, suppose that you collect keystroke data from two users, Alice and Bob. Keep track of the number of characters Alice types and the number Bob types. Your estimate as to the number of bits of entropy you have collected should be the minimum of those two values. That way, if Bob is an attacker, Alice will still have a reasonable amount of entropy, and vice versa.

If you are worried that an attacker may be feeding you all your input keystrokes, you should count no entropy, but mix in the key value to your entropy state anyway. In such a case, it might be reasonable to count a tiny bit of entropy from an associated timestamp if and only if the keystroke comes from the network. If the attacker may be local, do not assume there is any entropy.

Entropy in mouse movements

On most operating systems, moving the mouse produces events that give positional information about the mouse. In some cases, any user on the operating system can see those events. Therefore, if attacks from local users are in your threat model, you should not assume any entropy.

However, if you have a more liberal threat model, there may be some entropy in the position of the mouse. Unfortunately, most mouse movements follow simple trajectories with very little entropy. The most entropy occurs when the pointer reaches the general vicinity of its destination, and starts to slow down to lock in on a target. There is also often a fair bit of entropy on startup. The in-between motion is usually fairly predictable. Nonetheless, if local attacks are not in your threat model, and the attacker can only guess approximately what parts of your screen the mouse went to in a particular time frame based on observing program behavior, there is potentially a fair bit of entropy in each mouse event, because the attacker will not be able to guess to the pixel where the cursor is at any given moment.

For mouse movements, beyond the entropy you count for timestamping any mouse events, we recommend the following:

- If the mouse event is generated from the local console, not from a remotely controlled mouse, and if local attacks are not in your threat model, add the entire mouse event to your entropy state and estimate no more than three bits of entropy per sample (1.5 would be a good, highly conservative estimate).
- If the local user may be a threat and can see mouse events, estimate zero bits.
- If the local user may be a threat but should not be able to see the actual mouse events, estimate no more than one bit of entropy per sample.

Entropy in disk access

Many people believe that measuring how long it takes to access a disk is a good way to get some entropy. The idea is that there is entropy arising from turbulence between the disk head and the platter.

We recommend against using this method altogether.

There are several reasons that we make this recommendation. First, if that entropy is present at all, caching tends to make it inaccessible to the programmer. Second, in 1994, experts estimated that such a source was perhaps capable of producing about 100 bits of entropy per minute, if you can circumvent the caching problem. However, that value has almost certainly gone down with every generation of drives since then.

Entropy in data from the network

As we have mentioned previously in this recipe, while it may be tempting to try to gather entropy from network data, it is very risky to do so, because in any reasonable threat model, an attacker can measure and potentially inject data while on the network.

If there is any entropy to be had at all, it will largely come from the entropy on the recipient's machine, more than the network. If you absolutely have to measure entropy from such a source, never estimate more than 2.5 bits of entropy per packet

with a high-resolution clock (i.e., one running in the GHz range). If your clock has better than millisecond resolution and the processor is modern, it is probably reasonable to assume a half-bit of entropy on incoming network packets, even if the packets are generated by an attacker.

Entropy in the sound device

There is generally some entropy to be had by reading a sound card just from random thermal noise. However, the amount varies depending on the hardware. Sound cards are usually also subject to RF interference. Although that is generally not random, it does tend to amplify thermal noise.

Conservatively, if a machine has a sound card, and its outputs do not fail FIPS-140 tests, we believe it is reasonable to estimate 0.25 bits per sample, as long as an attacker cannot measure the same samples. Otherwise, do not estimate any.

Entropy from thread timing and other system state

Systems effectively gain entropy based on inputs from the environment. So far, we have discussed how to estimate entropy by directly sampling the input sources. If you wish to measure entropy that you are not specifically sampling, it is generally feasible to query system state that is sensitive to external inputs.

In practice, if you are worried about local attacks, you should not try to measure system state indirectly, particularly as an unprivileged user. For anything you can do to measure system state, an attacker can probably get correlated data and use it to attack your results.

Otherwise, the amount of entropy you get definitely depends on the amount of information an attacker can guess about your source. It is popular to use the output of commands such as *ps*, but such sources are actually a lot more predictable than most people think.

Instead, we recommend trying to perform actions that are likely to be indirectly affected by everything going on in the system. For example, you might measure how many times it takes to yield the scheduler a fixed number of times. More portably, you can do the same thing by timing how long it takes to start and stop a significant number of threads.

Again, this works only if local users are not in your threat model. If they are not, you can estimate entropy by looking at the difference between timestamps, as discussed earlier in this recipe. If you want to be conservative in your estimates, which is a good idea, particularly if you might be gathering the same entropy from different sources, you may want to divide the basic estimate by two or more.

See Also

Recipes 11.5, 11.18

11.20 Gathering Entropy from the Keyboard

Problem

You need entropy in a low-entropy environment and can prompt the user to type in order to collect it.

Solution

On Unix, read directly from the controlling terminal (*/dev/tty*). On Windows, process all keyboard events. Mix into an entropy pool the key pressed, along with the timestamp at which each one was processed. Estimate entropy based upon your operating environment; see the considerations in Recipe 11.19.

Discussion

There can be a reasonable amount of entropy in key presses. The entropy comes not simply from which key is pressed, but from when each key is pressed. In fact, measuring which key is pressed can have very little entropy in it, particularly in an embedded environment where there are only a few keys. Most of the entropy will come from the exact timing of the key press.

The basic methodology is to mix the character pressed, along with a timestamp, into the entropy pool. We will provide an example implementation in this section, where that operation is merely hashing the data into a running SHA1 context. If you can easily get information on both key presses and key releases (as in an event-driven system like Windows), we strongly recommend that you mix such information in as well.

The big issue is in estimating the amount of entropy in each key press. The first worry is what happens if the user holds down a key. The keyboard repeat may be so predictable that all entropy is lost. That is easy to thwart, though. You simply do not measure any entropy at all, unless the user pressed a different key from the previous time.

Ultimately, the amount of entropy you estimate getting from each key press should be related to the resolution of the clock you use to measure key presses. In addition, you must consider whether other processes on the system may be recording similar information (such as on a system that has a */dev/random* infrastructure already). See Recipe 11.19 for a detailed discussion of entropy estimation.

The next two subsections contain code that reads data from the keyboard, hashes it into a SHA1 context, and repeats until it is believed that the requested number of bits of entropy has been collected. A progress bar is also displayed that shows how much more entropy needs to be collected.

Collecting entropy from the keyboard on Unix

First, you need to get a file descriptor for the controlling terminal, which can always be done by opening */dev/tty*. Note that it is a bad idea to read from standard input, because it could be redirected from an input source other than */dev/tty*. For example, you might end up reading data from a static file with no entropy. You really do need to make sure you are reading data interactively from a keyboard.

Another issue is that there must be a secure path from the keyboard to the program that is measuring entropy. If, for example, the user is connected through an insecure *telnet* session, there is essentially no entropy in the data. However, it is generally okay to read data coming in over a secure *ssh* connection. Unfortunately, from an application, it is difficult to tell whether an interactive terminal is properly secured, so it's much better to issue a warning about it, pushing the burden off to the user.

You will want to put the terminal into a mode where character echo is off and as many keystrokes as possible can be read. The easiest way to do that is to put the terminal to which a user is attached in "raw" mode. In the following code, we implement a function that, given the file descriptor for the tty, sets the terminal mode to raw mode and also saves the old options so that they can be restored after entropy has been gathered. We do all the necessary flag-setting manually, but many environments can do it all with a single call to `cfmakeraw( )`, which is part of the POSIX standard.

In this code, timestamps are collected using the `current_stamp( )` macro from Recipe 4.14. Remember that this macro interfaces specifically to the x86 `RDTSC` instruction. For a more portable solution, you can use `gettimeofday( )`. (Refer back to Recipe 4.14 for timestamping solutions.)

One other thing that needs to be done to use this code is to define the macro `ENTROPY_PER_SAMPLE`, which indicates the amount of entropy that should be estimated for each key press, between the timing information and the actual value of the key.

We recommend that you be highly conservative, following the guidelines from Recipe 11.19. We strongly recommend a value no greater than 2.5 bits per key press on a Pentium 4, which takes into account that key presses might come over an *ssh* connection (although it is reasonable to keep an unencrypted channel out of the threat model). This helps ensure quality entropy and still takes up only a few seconds of the user's time (people will bang on their keyboards as quickly as they can to finish).

For a universally applicable estimate, 0.5 bits per character is nice and conservative and not too onerous for the user.

Note that we also assume a standard SHA1 API, as discussed in Recipe 6.5. This code will work as is with OpenSSL if you include *openssl/sha.h* and link in *libcrypto*.

```
#include <termios.h>
#include <unistd.h>
```

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif
#include <openssl/sha.h>

#define HASH_OUT_SZ     20
#define OVERHEAD_CHARS  7
#define DEFAULT_BARSIZE (78 - OVERHEAD_CHARS)
#define MAX_BARSIZE     200

void spc_raw(int fd, struct termios *saved_opts) {
  struct termios new_opts;

  if (tcgetattr(fd, saved_opts) < 0) abort();
  /* Make a copy of saved_opts, not an alias. */
  new_opts = *saved_opts;
  new_opts.c_lflag    &= ~(ECHO | ICANON | IEXTEN | ISIG);
  new_opts.c_iflag    &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
  new_opts.c_cflag    &= ~(CSIZE | PARENB);
  new_opts.c_cflag    |= CS8;
  new_opts.c_oflag    &= ~OPOST;
  new_opts.c_cc[VMIN]  = 1;
  new_opts.c_cc[VTIME] = 0;
  if (tcsetattr(fd, TCSAFLUSH, &new_opts) < 0) abort();
}

/* Query the terminal file descriptor with the TIOCGWINSZ ioctl in order to find
 * out the width of the terminal.  If we get an error, go ahead and assume a 78
 * character display.  The worst that may happen is bad wrapping.
 */
static int spc_get_barsize(int ttyfd) {
  struct winsize sz;

  if (ioctl(ttyfd, TIOCGWINSZ, (char *)&sz) < 0) return DEFAULT_BARSIZE;
  if (sz.ws_col < OVERHEAD_CHARS) return 0;
  if (sz.ws_col - OVERHEAD_CHARS > MAX_BARSIZE) return MAX_BARSIZE;
  return sz.ws_col - OVERHEAD_CHARS;
}

static void spc_show_progress_bar(double entropy, int target, int ttyfd) {
  int  bsz, c;
  char bf[MAX_BARSIZE + OVERHEAD_CHARS];

  bsz = spc_get_barsize(ttyfd);
  c   = (int)((entropy * bsz) / target);
  bf[sizeof(bf) - 1] = 0;
  if (bsz) {
    snprintf(bf, sizeof(bf), "\r[%-*s] %d%%", bsz, "",
             (int)(entropy * 100.0 / target));
    memset(bf + 2, '=', c);
```

```
    bf[c + 2] = '>';
  } else
    snprintf(bf, sizeof(bf), "\r%d%%", (int)(entropy * 100.0 / target));
  while (write(ttyfd, bf, strlen(bf)) == -1)
    if (errno != EAGAIN) abort( );
}

static void spc_end_progress_bar(int target, int ttyfd) {
  int bsz, i;

  if (!(bsz = spc_get_barsize(ttyfd))) {
    printf("100%%\r\n");
    return;
  }
  printf("\r[");
  for (i = 0;  i < bsz;  i++) putchar('=');
  printf("] 100%%\r\n");
}

void spc_gather_keyboard_entropy(int l, char *output) {
  int            fd, n;
  char           lastc = 0;
  double         entropy = 0.0;
  SHA_CTX        pool;
  volatile char  dgst[HASH_OUT_SZ];
  struct termios opts;
  struct {
    char         c;
    long long    timestamp;
  }              data;

  if (l > HASH_OUT_SZ) abort( );
  if ((fd = open("/dev/tty", O_RDWR)) == -1) abort( );
  spc_raw(fd, &opts);
  SHA1_Init(&pool);
  do {
    spc_show_progress_bar(entropy, l * 8, fd);
    if ((n = read(fd, &(data.c), 1)) < 1) {
      if (errno == EAGAIN) continue;
      abort( );
    }
    current_stamp(&(data.timestamp));
    SHA1_Update(&pool, &data, sizeof(data));
    if (lastc != data.c) entropy += ENTROPY_PER_SAMPLE;
    lastc = data.c;
  } while (entropy < (l * 8));
  spc_end_progress_bar(l * 8, fd);
  /* Try to reset the terminal. */
  tcsetattr(fd, TCSAFLUSH, &opts);
  close(fd);
  SHA1_Final((unsigned char *)dgst, &pool);
  spc_memcpy(output, (char *)dgst, l);
  spc_memset(dgst, 0, sizeof(dgst));
}
```

Collecting entropy from the keyboard on Windows

To collect entropy from the keyboard on Windows, we will start by building a dialog that displays a brief message advising the user to type random characters on the keyboard until enough entropy has been collected. The dialog will also contain a progress bar and an OK button that is initially disabled. As entropy is collected, the progress bar will be updated to report the progress of the collection. When enough entropy has been collected, the OK button will be enabled. Clicking the OK button will dismiss the dialog.

Here is the resource definition for the dialog:

```
#include <windows.h>

#define SPC_KEYBOARD_DLGID      101
#define SPC_PROGRESS_BARID      1000
#define SPC_KEYBOARD_STATIC     1001

SPC_KEYBOARD_DLGID DIALOG DISCARDABLE  0, 0, 186, 95
STYLE DS_MODALFRAME | DS_NOIDLEMSG | DS_CENTER | WS_POPUP | WS_VISIBLE |
    WS_CAPTION
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL         "Progress1",SPC_PROGRESS_BARID,"msctls_progress32",
                    PBS_SMOOTH | WS_BORDER,5,40,175,14
    LTEXT           "Please type random characters on your keyboard until the \
                    progress bar reports 100% and the OK button becomes active.",
                    SPC_KEYBOARD_STATIC,5,5,175,25
    PUSHBUTTON      "OK",IDOK,130,70,50,14,WS_DISABLED
END
```

Call the function `SpcGatherKeyboardEntropy()` to begin the process of collecting entropy. It requires two additional arguments to its Unix counterpart, `spc_gather_keyboard_entropy()`:

`hInstance`
: Application instance handle normally obtained from the first argument to `WinMain()`, the program's entry point.

`hWndParent`
: Handle to the dialog's parent window. It may be specified as `NULL`, in which case the dialog will have no parent.

`pbOutput`
: Buffer into which the collected entropy will be placed.

`cbOutput`
: Number of bytes of entropy to place into the output buffer. The output buffer must be sufficiently large to hold the requested amount of entropy. The number of bytes of entropy requested should not exceed the size of the hash function used, which is SHA1 in the code provided. SHA1 produces a 160-bit or 20-byte hash. If the requested entropy is smaller than the hash function's output, the hash function's output will be truncated.

SpcGatherKeyboardEntropy( ) uses the CryptoAPI to hash the data collected from the keyboard. It first acquires a context object, then creates a hash object. After the arguments are validated, the dialog resource is loaded by calling CreateDialog( ), which creates a modeless dialog. The dialog is created modeless so that keyboard messages can be captured. If a modal dialog is created using DialogBox( ) or one of its siblings, message handling for the dialog prevents us from capturing the keyboard messages.

Once the dialog is successfully created, the message-handling loop performs normal message dispatching, calling IsDialogMessage( ) to do dialog message processing. Keyboard messages are captured in the loop prior to calling IsDialogMessage( ), however. That's because IsDialogMessage( ) causes the messages to be translated and dispatched, so handling them in the dialog's message procedure isn't possible.

When a key is pressed, a WM_KEYDOWN message will be received, which contains information about which key was pressed. When a key is released, a WM_KEYUP message will be received, which contains the same information about which key was released as WM_KEYDOWN contains about a key press. The keyboard scan code is extracted from the message, combined with a timestamp, and fed into the hash object. If the current scan code is the same as the previous scan code, it is not counted as entropy but is added into the hash anyway. As other keystrokes are collected, the progress bar is updated, and when the requested amount of entropy has been obtained, the OK button is enabled.

When the OK button is clicked, the dialog is destroyed, terminating the message loop. The output from the hash function is copied into the output buffer from the caller, and internal data is cleaned up before returning to the caller.

```
#include <windows.h>
#include <wincrypt.h>
#include <commctrl.h>

#define SPC_ENTROPY_PER_SAMPLE  0.5
#define SPC_KEYBOARD_DLGID      101
#define SPC_PROGRESS_BARID      1000
#define SPC_KEYBOARD_STATIC     -1

typedef struct {
  BYTE  bScanCode;
  DWORD dwTickCount;
} SPC_KEYPRESS;

static BOOL CALLBACK KeyboardEntropyProc(HWND hwndDlg, UINT uMsg, WPARAM wParam,
                                         LPARAM lParam) {
  HWND *pHwnd;

  if (uMsg != WM_COMMAND || LOWORD(wParam) != IDOK ||
      HIWORD(wParam) != BN_CLICKED) return FALSE;

  pHwnd = (HWND *)GetWindowLong(hwndDlg, DWL_USER);
  DestroyWindow(hwndDlg);
```

```
  *pHwnd = 0;
  return TRUE;
}

BOOL SpcGatherKeyboardEntropy(HINSTANCE hInstance, HWND hWndParent,
                              BYTE *pbOutput, DWORD cbOutput) {
  MSG            msg;
  BOOL           bResult = FALSE;
  BYTE           bLastScanCode = 0, *pbHashData = 0;
  HWND           hwndDlg;
  DWORD          cbHashData, dwByteCount = sizeof(DWORD), dwLastTime = 0;
  double         dEntropy = 0.0;
  HCRYPTHASH     hHash = 0;
  HCRYPTPROV     hProvider = 0;
  SPC_KEYPRESS   KeyPress;

  if (!CryptAcquireContext(&hProvider, 0, MS_DEF_PROV, PROV_RSA_FULL,
                           CRYPT_VERIFYCONTEXT)) goto done;
  if (!CryptCreateHash(hProvider, CALG_SHA1, 0, 0, &hHash)) goto done;
  if (!CryptGetHashParam(hHash, HP_HASHSIZE, (BYTE *)&cbHashData, &dwByteCount,
                         0)) goto done;
  if (cbOutput > cbHashData) goto done;
  if (!(pbHashData = (BYTE *)LocalAlloc(LMEM_FIXED, cbHashData))) goto done;

  hwndDlg = CreateDialog(hInstance, MAKEINTRESOURCE(SPC_KEYBOARD_DLGID),
                         hWndParent, KeyboardEntropyProc);
  if (hwndDlg) {
    if (hWndParent) EnableWindow(hWndParent, FALSE);
    SetWindowLong(hwndDlg, DWL_USER, (LONG)&hwndDlg);
    SendDlgItemMessage(hwndDlg, SPC_PROGRESS_BARID, PBM_SETRANGE32, 0,
                       cbOutput * 8);
    while (hwndDlg && GetMessage(&msg, 0, 0, 0) > 0) {
      if ((msg.message == WM_KEYDOWN || msg.message == WM_KEYUP) &&
          dEntropy < cbOutput * 8) {
        KeyPress.bScanCode   = ((msg.lParam >> 16) & 0x000000FF);
        KeyPress.dwTickCount = GetTickCount( );
        CryptHashData(hHash, (BYTE *)&KeyPress, sizeof(KeyPress), 0);
        if (msg.message == WM_KEYUP || (bLastScanCode != KeyPress.bScanCode &&
            KeyPress.dwTickCount - dwLastTime > 100)) {
          bLastScanCode = KeyPress.bScanCode;
          dwLastTime = KeyPress.dwTickCount;
          dEntropy += SPC_ENTROPY_PER_SAMPLE;
          SendDlgItemMessage(hwndDlg, SPC_PROGRESS_BARID, PBM_SETPOS,
                             (WPARAM)dEntropy, 0);
          if (dEntropy >= cbOutput * 8) {
            EnableWindow(GetDlgItem(hwndDlg, IDOK), TRUE);
            SetFocus(GetDlgItem(hwndDlg, IDOK));
            MessageBeep(0xFFFFFFFF);
          }
        }
        continue;
      }
      if (!IsDialogMessage(hwndDlg, &msg)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
```

```
      }
    }
    if (hWndParent) EnableWindow(hWndParent, TRUE);
  }

  if (dEntropy >= cbOutput * 8) {
    if (CryptGetHashParam(hHash, HP_HASHVAL, pbHashData, &cbHashData, 0)) {
      bResult = TRUE;
      CopyMemory(pbOutput, pbHashData, cbOutput);
    }
  }

done:
  if (pbHashData) LocalFree(pbHashData);
  if (hHash) CryptDestroyHash(hHash);
  if (hProvider) CryptReleaseContext(hProvider, 0);
  return bResult;
}
```

There are other ways to achieve the same result on Windows. For example, you could install a temporary hook to intercept all messages and use the modal dialog functions instead of the modeless ones that we have used here. Another possibility is to be collecting entropy throughout your entire program by installing a more permanent hook or by moving the entropy collection code out of `SpcGatherKeyboardEntropy( )` and placing it into your program's main message-processing loop. `SpcGatherKeyboardEntropy( )` could then be modified to operate in global state, presenting a dialog only if there is not a sufficient amount of entropy collected already.

Note that the dialog uses a progress bar control. While this control is a standard control on Windows, it is part of the common controls, so you must initialize common controls before instantiating the dialog; otherwise, `CreateDialog( )` will fail inexplicably (`GetLastError( )` will return 0, which obviously is not very informative). The following code demonstrates initializing common controls and calling `SpcGatherKeyboardEntropy( )`:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
                   int nShowCmd) {
  BYTE                 pbEntropy[20];
  INITCOMMONCONTROLSEX CommonControls;

  CommonControls.dwSize = sizeof(CommonControls);
  CommonControls.dwICC  = ICC_PROGRESS_CLASS;
  InitCommonControlsEx(&CommonControls);
  SpcGatherKeyboardEntropy(hInstance, 0, pbEntropy, sizeof(pbEntropy));
  return 0;
}
```

See Also

Recipes 4.14, 6.5, 11.19

11.21 Gathering Entropy from Mouse Events on Windows

Problem

You need entropy in a low-entropy environment and can prompt the user to move the mouse to collect it.

Solution

On Windows, process all mouse events. Mix into an entropy pool the current position of the mouse pointer on the screen, along with the timestamp at which each event was processed. Estimate entropy based upon your operating environment; see the considerations in Recipe 11.19.

Discussion

There can be a reasonable amount of entropy in mouse movement. The entropy comes not just from where the mouse pointer is on the screen, but from when each movement was made. In fact, the mouse pointer's position on the screen can have very little entropy in it, particularly in an environment where there may be very little interaction from a local user. Most of the entropy will come from the exact timing of the mouse movements.

The basic methodology is to mix the on-screen position of the mouse pointer, along with a timestamp, into the entropy pool. We will provide an example implementation in this section, where that operation is merely hashing the data into a running SHA1 context.

The big issue is in estimating the amount of entropy in each mouse movement. The first worry is that it is common for Windows to send multiple mouse event messages with the same mouse pointer position. That is easy to thwart, though. You simply do not measure any entropy at all, unless the mouse pointer has actually changed position.

Ultimately, the amount of entropy you estimate getting from each mouse movement should be related to the resolution of the clock you use to measure mouse movements. In addition, you must consider whether other processes on the system may be recording similar information. (See Recipe 11.19 for a detailed discussion of entropy estimation.)

The following code captures mouse events, hashes mouse pointer positions and timestamps into a SHA1 context, and repeats until it is believed that the requested

number of bits of entropy has been collected. A progress bar is also displayed that shows how much more entropy needs to be collected.

Here is the resource definition for the progress dialog:

```
#include <windows.h>

#define SPC_MOUSE_DLGID     102
#define SPC_PROGRESS_BARID  1000
#define SPC_MOUSE_COLLECTID 1002
#define SPC_MOUSE_STATIC    1003

SPC_MOUSE_DLGID DIALOG DISCARDABLE  0, 0, 287, 166
STYLE DS_MODALFRAME | DS_NOIDLEMSG | DS_CENTER | WS_POPUP | WS_VISIBLE |
    WS_CAPTION
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL         "Progress1",SPC_PROGRESS_BARID,"msctls_progress32",
                    PBS_SMOOTH | WS_BORDER,5,125,275,14
    LTEXT           "Please move your mouse over this dialog until the progress \
                    bar reports 100% and the OK button becomes active.",
                    SPC_MOUSE_STATIC,5,5,275,20
    PUSHBUTTON      "OK",IDOK,230,145,50,14,WS_DISABLED
    CONTROL         "",SPC_MOUSE_COLLECTID,"Static",SS_LEFTNOWORDWRAP |
                    SS_SUNKEN | WS_BORDER | WS_GROUP,5,35,275,80
END
```

Call the function `SpcGatherMouseEntropy()`to begin the process of collecting entropy. It has the same signature as `SpcGatherKeyboardEntropy()` from Recipe 11.20. This function has the following arguments:

`hInstance`
: Application instance handle normally obtained from the first argument to `WinMain()`, the program's entry point.

`hWndParent`
: Handle to the dialog's parent window. It may be specified as `NULL`, in which case the dialog will have no parent.

`pbOutput`
: Buffer into which the collected entropy will be placed.

`cbOutput`
: Number of bytes of entropy to place into the output buffer. The output buffer must be sufficiently large to hold the requested amount of entropy. The number of bytes of entropy requested should not exceed the size of the hash function used, which is SHA1 in the code provided. SHA1 produces a 160-bit or 20-byte hash. If the requested entropy is smaller than the hash function's output, the hash function's output will be truncated.

`SpcGatherMouseEntropy()` uses the CryptoAPI to hash the data collected from the mouse. It first acquires a context object, then creates a hash object. After the arguments are validated, the dialog resource is loaded by calling `DialogBoxParam()`, which

creates a modal dialog. A modal dialog can be used for capturing mouse messages instead of the modeless dialog that was required for gathering keyboard entropy in Recipe 11.20, because normal dialog processing doesn't eat mouse messages the way it eats keyboard messages.

Once the dialog is successfully created, the message handling procedure handles WM_MOUSEMOVE messages, which will be received whenever the mouse pointer moves over the dialog or its controls. The position of the mouse pointer is extracted from the message, converted to screen coordinates, combined with a timestamp, and fed into the hash object. If the current pointer position is the same as the previous pointer position, it is not counted as entropy but is added into the hash anyway. As mouse movements are collected, the progress bar is updated, and when the requested amount of entropy has been obtained, the OK button is enabled.

When the OK button is clicked, the dialog is destroyed, terminating the message loop. The output from the hash function is copied into the output buffer from the caller, and internal data is cleaned up before returning to the caller.

```
#include <windows.h>
#include <wincrypt.h>
#include <commctrl.h>

#define SPC_ENTROPY_PER_SAMPLE  0.5
#define SPC_MOUSE_DLGID         102
#define SPC_PROGRESS_BARID      1000
#define SPC_MOUSE_COLLECTID     1003
#define SPC_MOUSE_STATIC        1002

typedef struct {
  double     dEntropy;
  DWORD      cbRequested;
  POINT      ptLastPos;
  DWORD      dwLastTime;
  HCRYPTHASH hHash;
} SPC_DIALOGDATA;

typedef struct {
  POINT ptMousePos;
  DWORD dwTickCount;
} SPC_MOUSEPOS;

static BOOL CALLBACK MouseEntropyProc(HWND hwndDlg, UINT uMsg, WPARAM wParam,
                                      LPARAM lParam) {
  SPC_MOUSEPOS    MousePos;
  SPC_DIALOGDATA  *pDlgData;

  switch (uMsg) {
    case WM_INITDIALOG:
      pDlgData = (SPC_DIALOGDATA *)lParam;
      SetWindowLong(hwndDlg, DWL_USER, lParam);
      SendDlgItemMessage(hwndDlg, SPC_PROGRESS_BARID, PBM_SETRANGE32, 0,
```

```
                        pDlgData->cbRequested);
      return TRUE;

    case WM_COMMAND:
      if (LOWORD(wParam) == IDOK && HIWORD(wParam) == BN_CLICKED) {
        EndDialog(hwndDlg, TRUE);
        return TRUE;
      }
      break;

    case WM_MOUSEMOVE:
      pDlgData = (SPC_DIALOGDATA *)GetWindowLong(hwndDlg, DWL_USER);
      if (pDlgData->dEntropy < pDlgData->cbRequested) {
        MousePos.ptMousePos.x = LOWORD(lParam);
        MousePos.ptMousePos.y = HIWORD(lParam);
        MousePos.dwTickCount  = GetTickCount( );
        ClientToScreen(hwndDlg, &(MousePos.ptMousePos));
        CryptHashData(pDlgData->hHash, (BYTE *)&MousePos, sizeof(MousePos), 0);
        if ((MousePos.ptMousePos.x != pDlgData->ptLastPos.x ||
             MousePos.ptMousePos.y != pDlgData->ptLastPos.y)  &&
            MousePos.dwTickCount - pDlgData->dwLastTime > 100) {
          pDlgData->ptLastPos = MousePos.ptMousePos;
          pDlgData->dwLastTime = MousePos.dwTickCount;
          pDlgData->dEntropy += SPC_ENTROPY_PER_SAMPLE;
          SendDlgItemMessage(hwndDlg, SPC_PROGRESS_BARID, PBM_SETPOS,
                             (WPARAM)pDlgData->dEntropy, 0);
          if (pDlgData->dEntropy >= pDlgData->cbRequested) {
            EnableWindow(GetDlgItem(hwndDlg, IDOK), TRUE);
            SetFocus(GetDlgItem(hwndDlg, IDOK));
            MessageBeep(0xFFFFFFFF);
          }
        }
      }
      return TRUE;
  }

  return FALSE;
}

BOOL SpcGatherMouseEntropy(HINSTANCE hInstance, HWND hWndParent,
                           BYTE *pbOutput, DWORD cbOutput) {
  BOOL           bResult = FALSE;
  BYTE           *pbHashData = 0;
  DWORD          cbHashData, dwByteCount = sizeof(DWORD);
  HCRYPTHASH     hHash = 0;
  HCRYPTPROV     hProvider = 0;
  SPC_DIALOGDATA DialogData;

  if (!CryptAcquireContext(&hProvider, 0, MS_DEF_PROV, PROV_RSA_FULL,
                           CRYPT_VERIFYCONTEXT)) goto done;
  if (!CryptCreateHash(hProvider, CALG_SHA1, 0, 0, &hHash)) goto done;
  if (!CryptGetHashParam(hHash, HP_HASHSIZE, (BYTE *)&cbHashData, &dwByteCount,
                         0)) goto done;
```

```
  if (cbOutput > cbHashData) goto done;
  if (!(pbHashData = (BYTE *)LocalAlloc(LMEM_FIXED, cbHashData))) goto done;

  DialogData.dEntropy     = 0.0;
  DialogData.cbRequested = cbOutput * 8;
  DialogData.hHash        = hHash;
  DialogData.dwLastTime   = 0;
  GetCursorPos(&(DialogData.ptLastPos));

  bResult = DialogBoxParam(hInstance, MAKEINTRESOURCE(SPC_MOUSE_DLGID),
                           hWndParent, MouseEntropyProc, (LPARAM)&DialogData);

  if (bResult) {
    if (!CryptGetHashParam(hHash, HP_HASHVAL, pbHashData, &cbHashData, 0))
      bResult = FALSE;
    else
      CopyMemory(pbOutput, pbHashData, cbOutput);
  }

done:
  if (pbHashData) LocalFree(pbHashData);
  if (hHash) CryptDestroyHash(hHash);
  if (hProvider) CryptReleaseContext(hProvider, 0);
  return bResult;
}
```

There are other ways to achieve the same result on Windows. For example, entropy could be collected throughout your entire program by installing a message hook or by moving the entropy collection code out of `MouseEntropyProc( )` and placing it into your program's main message processing loop. `SpcGatherMouseEntropy( )` could then be modified to operate in global state, presenting a dialog only if there is not a sufficient amount of entropy collected already.

Note that the dialog uses a progress bar control. While this control is a standard control on Windows, it is part of the common controls, so you must initialize common controls before instantiating the dialog; otherwise, `DialogBoxParam( )` will fail inexplicably (`GetLastError( )` will return 0, which obviously is not very informative). The following code demonstrates initializing common controls and calling `SpcGatherMouseEntropy( )`:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
                   int nShowCmd) {
  BYTE                pbEntropy[20];
  INITCOMMONCONTROLSEX CommonControls;

  CommonControls.dwSize = sizeof(CommonControls);
  CommonControls.dwICC  = ICC_PROGRESS_CLASS;
  InitCommonControlsEx(&CommonControls);
  SpcGatherMouseEntropy(hInstance, 0, pbEntropy, sizeof(pbEntropy));
  return 0;
}
```

See Also

Recipes 11.19, 11.20

11.22 Gathering Entropy from Thread Timings

Problem

You want to collect some entropy without user intervention, hoping that there is some inherent, measurable entropy in the environment.

Solution

In practice, timing how long it takes to start up and stop a particular number of threads can provide a bit of entropy. For example, many Java virtual machines exclusively use such a technique to gather entropy.

Because the thread timing data is only indirectly related to actual user input, it is good to be extremely conservative about the quality of this entropy source. We recommend the following methodology:

1. Launch and join on some fixed number of threads (at least 100).
2. Mix in a timestamp when all threads have returned successfully.
3. Estimate entropy based on the considerations discussed in Recipe 11.19.
4. Wait at least a second before trying again, in hopes that there is additional entropy affecting the system later.

The following code spawns a particular number of threads that you can time, in the hope of getting some entropy. This code works on Unix implementations that have the *pthreads* library (the POSIX standard for threads). Linking is different depending on platform; check your local *pthreads* documentation.

```
#include <pthread.h>

static void *thread_stub(void *arg) {
  return 0;
}

void spc_time_threads(unsigned int numiters) {
  pthread_t tid;

  while (numiters--)
    if (!pthread_create(&tid, 0, thread_stub, 0))
      pthread_join(tid, 0);
}
```

On Windows, the idea is the same, and the structure of the code is similar. Here is the same code as presented above, but implemented using the Win32 API:

```
#include <windows.h>

static DWORD WINAPI ThreadStub(LPVOID lpData) {
  return 0;
}

void SpcTimeThreads(DWORD dwIterCount) {
  DWORD  dwThreadId;
  HANDLE hThread;

  while (dwIterCount--) {
    if ((hThread = CreateThread(0, 0, ThreadStub, 0, 0, &dwThreadId)) != 0) {
      WaitForSingleObject(hThread, INFINITE);
      CloseHandle(hThread);
    }
  }
}
```

See Recipe 4.14 for several different ways to get a timestamp. We strongly recommend that you use the most accurate method available on your platform.

See Also

Recipes 4.14, 11.19

11.23 Gathering Entropy from System State

Problem

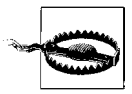
You want to get some information that might actually change rapidly about the state of the kernel, in the hope that you might be able to get a bit of entropy from it.

Solution

The solution is highly operating system–specific. On systems with a */proc* filesystem, you can read the contents of all the files in */proc*. Otherwise, you can securely invoke commands that have some chance of providing entropy (especially if called infrequently). On Windows, the Performance Data Helper (PDH) API can be used to query much of the same type of information available on Unix systems with a */proc* filesystem.

Mix any data you collect, as well as a timestamp taken after the operation completes, into an entropy pool (see Recipe 11.19).

Discussion



We strongly recommend that you do not increase your entropy estimates based on any kernel state collected, particularly on a system that is mostly idle. Much of the time, kernel state changes more slowly than people think. In addition, attackers may be able to query the same data and get very similar results.

The internal state of an operating system can change quickly, but that does not mean there is necessarily any entropy there to collect. See Recipe 11.19 for a discussion about estimating how much entropy you are getting.

Definitely do not query sources like these very often, because you are unlikely to get additional entropy running in a tight loop, and the overhead involved is extremely high.

On systems with a */proc* filesystem, pretty much all of the interesting operating system–specific information you might want to query is available by reading the files in the */proc* directory. The contents of the files in that directory are updated as the user reads from those files. Open the files anew every time you want to poll for possible entropy.

On systems without */proc*, you can try to get information by running commands that might change frequently and capturing all the data in the command. Be sure to call out to any commands you run in a secure manner, as discussed in Recipes 1.7 and 1.8.

When calling commands, state does not actually change very quickly at all, particularly on systems with few users. It is popular to query the *ps* and *df* commands (using the flags that give the most entropy, of course), but there is often almost no entropy in the output they produce.

Other commands that some operating systems may have, where there might be some frequent change (though we would not count on it) include the following:

- *sysctl*: Use the -A flag.
- *iostat*
- *lsof*
- *netstat*: Use the -s flag if you want to see highly detailed information that may change frequently on machines that see a lot of network traffic.
- *pstat*
- *tcpdump*: Ask it to capture a small number of packets.
- *vmstat*

Often, these commands will need to run with superuser privileges (for example, *tcpdump*). Depending on your threat model, such commands can possibly be more useful because they're less subject to attacks from local users.

This approach can be a reasonable way of collecting data from the network. However, note that attackers could possibly feed you packets in a predictable manner, designed to reduce the amount of entropy available from this source.

See Also

Recipes 1.7, 1.8, 11.19