9. Pseudorandom Number Generators

9. Pseudorandom Number Generators

9.1. Randomness and entropy

As we've seen in the previous sections, cryptography often requires randomness. For example, symmetric keys are usually randomly-generated bits, and random IVs and nonces are required to build secure block cipher chaining modes.

In cryptography, when we say "random," we usually mean "random and unpredictable." For example, flipping a biased coin that comes up heads 99% of the time is random, but you can predict a pattern–for a given coin toss, if you guess heads, it's very likely you're correct. A better source of randomness for cryptographic purposes would be flipping a fair coin, because the outcome is harder to predict than the outcome of the biased coin flip. Consider generating a random symmetric key: you would want to use outcomes of the fair coin to generate the key, because that makes it harder for the attacker to guess your key than if you had used outcomes of the biased coin to generate the key.

We can formalize this concept of unpredictability by defining *entropy*, a measure of uncertainty or surprise, for any random event. The biased coin has low entropy because you expect a given outcome most of the time. The fair coin has high entropy because you are very uncertain about the outcome. The specifics of entropy are beyond the scope of this class, but an important note is that the uniform distribution (all outcomes equally likely) produces the greatest entropy. In cryptography, we generally want randomness with the most entropy, so ideally, any randomness should be bits drawn from a uniform distribution (i.e. the outcomes of fair coin tosses).

However, true, unbiased randomness is computationally expensive to generate. True randomness usually requires sampling data from an unpredictable physical process, such as an unpredictable circuit on a CPU, random noise signals, or the microsecond at which a user presses a key. These sources may be biased and predictable, making it even more challenging to generate unbiased randomness.

Instead of using expensive true randomness each time a cryptographic algorithm requires randomness, we instead use *pseudo-randomness*. Pseudorandom numbers are generated deterministically using an algorithm, but they look random. In particular, a good pseudorandom number algorithm generates bits that are *computationally indistinguishable* from true random bits–there is no efficient algorithm that would let an attacker distinguish between pseudorandom bits and truly random bits.

9.2. Pseudorandom Number Generators (pRNGs)

A *pseudorandom number generator (pRNG)* is an algorithm that takes a small amount of truly random bits as input and outputs a long sequence of pseudorandom bits. The initial truly random input is called the *seed*.

The pRNG algorithm is deterministic, so anyone who runs the pRNG with the same seed will see the same pseudorandom output. However, to an attacker who doesn't know the seed, the output of a secure pRNG is computationally indistinguishable from true random bits. A pRNG is not completely indistinguishable from true random bits–given infinite computational time and power, an attacker can distinguish pRNG output from truly random output. If the pRNG takes in an n-bit seed as input, the attacker just has to input all 2^n possible seeds and see if any of the 2^n outputs matches the bitstring they received. However, when restricted

to any practical computation limit, an attacker has no way of distinguishing pRNG output from truly random output.

It would be very inefficient if a pRNG only outputted a fixed number of pseudorandom bits for each truly random input. If this were the case, we would have to generate more true randomness each time the pRNG output has all been used. Ideally, we would like the pRNG to take in an initial seed and then be available to generate as many pseudorandom bits as needed on demand. To achieve this, the pRNG maintains some internal state and updates the state any time the pRNG generates new bits or receives a seed as input.

Formally, a pRNG is defined by the following three functions:

- *Seed(entropy)*: Take in some initial truly random entropy and initialize the pRNG's internal state.
- *Reseed(entropy)*: Take in some additional truly random entropy, updating the pRNG's internal state as needed.
- *Generate(n)*: Generate n pseudorandom bits, updating the internal state as needed. Some pRNGs also support adding additional entropy directly during this step.

9.3. Rollback resistance

In the previous section, we defined a secure pRNG as an algorithm whose output is computationally indistinguishable from random if the attacker does not know the seed and internal state. However, this definition does not say anything about the consequences of an attacker who does manage to compromise the internal state of a secure pRNG.

An additional desirable property of a secure pRNG is *rollback resistance*. Suppose a pRNG has been used to generate 100 bits, and an attacker is somehow able to learn the internal state immediately after bit 100 has been generated. If the pRNG is rollback-resistant, then the attacker cannot deduce anything about any previously-generated bit. Formally, the previously-generated output of the pRNG should still be computationally indistinguishable from random, even if the attacker knows the current internal state of the pRNG.

Not all secure pRNGs are rollback-resistant, but rollback resistance is an important property for any practical cryptographic pRNG implementation. Consider a cryptosystem that uses a single pRNG to generate both the secret keys and the IVs (or nonces) for a symmetric encryption scheme. The pRNG is first used to generate the secret keys, and then used again to generate the IVs. If this pRNG was not rollback-resistant, then an attacker who compromises the internal state at this point could learn the value of the secret key.

9.4. HMAC-DRBG

There are many implementations of pRNGs, but one commonly-used pRNG in practice is HMAC-DRBG[1], which uses the security properties of HMAC to build a pRNG.

HMAC-DRBG maintains two values as part of its internal state, K and V. K is used as the secret key to the HMAC, and V is used as the "message" input to the HMAC.

To generate a block of pseudorandom bits, HMAC-DRBG computes HMAC on the previous block of pRNG output. This can be repeated to generate as many pseudorandom bits as needed. Recall that the output of HMAC looks random to an attacker who doesn't know the key. As long as we keep the internal state (which includes K) secret, an attacker cannot distinguish the output of the HMAC from random bits, so the pRNG is secure.

We also use HMAC to update the internal state K and V each time. If additional true randomness is provided, we add it to the "message" input to HMAC.

[1] DRBG stands for Deterministic Random Bit Generator

Algorithm 1 Generate(n): Generate n pseudorandom bits, with no additional true random input

output = ‘ ’

while len(output) $< n$ **do**

$V = \text{HMAC}(K, V)$

output = output$\|V$

end while

$K = \text{HMAC}(K, V \| \texttt{0x00})$

$V = \text{HMAC}(K, V)$

return output$[0 : n]$

At line 3, we are repeatedly calling HMAC on the previous block of output. The while loop repeats this process until we have at least n bits of output. Once we have enough output, we update the internal state with two additional HMAC calls, and then return the first n bits of pseudorandom output.

Next, let's see how to seed the pRNG. The seed and reseed algorithms use true randomness as input to the HMAC, and uses the output of slight variations on the HMAC input to update K and V.

Algorithm 2 Seed(s): Take some truly random bits s and initialize the internal state.

$K = 0$

$V = 0$

$H = \text{HMAC}(K, V \| s \| \texttt{0x00})$

$V = \text{HMAC}(K, V)$

$K = \text{HMAC}(K, V \| s \| \texttt{0x01})$

$V = \text{HMAC}(K, V)$

The reseed algorithm is identical to the seed algorithm, except we don't need to reset K and V to 0 (steps 1-2).

Finally, if we want to generate pseudorandom output and add entropy at the same time, we combine the two algorithms above:

Algorithm 3 Generate(n): Generate n pseudorandom bits, with additional true random input s.

output = ‘ ’

while len(output) $< n$ **do**

$V = \text{HMAC}(K, V)$

output = output$\|V$

end while

$K = \text{HMAC}(K, V \| s \| \texttt{0x00})$

$V = \text{HMAC}(K, V)$

$K = \text{HMAC}(K, V \| s \| \texttt{0x01})$

$V = \text{HMAC}(K, V)$

return output$[0 : n]$

The specific design decisions of HMAC-DRBG, such as why it uses `0x00` and `0x01`, are not so important. The main takeaway is that because HMAC output is indistinguishable from random, the output of HMAC-DRBG (which is essentially lots of HMAC outputs) is also indistinguishable from random.

The use of the cryptographic hash function in both the seeding and reseeding algorithms means that HMAC-DRBG can accept an arbitrary long initial seed. For example, if each bit of the input seed really only has 0.1 bits of entropy (e.g. because it is a highly biased coin), using 2560 bits of seed material will leave HMAC-DRBG with 256b of actual entropy for its internal operations. Furthermore, adding in additional strings that contain *no* entropy (such as a string of 0 bits or the number π) doesn't make the internal state worse.

Additionally, HMAC-DRBG has rollback resistance: if you can compute the previous state from the current state you have successfully reversed the underlying hash function!

9.5. Stream ciphers

As we've seen in the previous section, an attacker without knowledge of the internal state of a secure, rollback-resistant pRNG cannot predict the pRNG's past or future output, and the attacker cannot distinguish the pRNG output from random bits. This sounds very similar to the properties we want in a random, unpredictable one-time pad. In fact, we can use pRNGs to generate a one-time pad that we then use for encrypting messages. This encryption scheme is an example of a class of algorithms known as *stream ciphers*.

Recall that in block ciphers, we encrypted and decrypted messages by splitting them into fixed-size blocks. Stream ciphers use a different approach to encryption, in which we encrypt and decrypt messages as they arrive, one bit at a time. You can imagine a stream cipher operating on an encrypted file being downloaded from the Internet: as each subsequent bit is downloaded, the stream cipher can immediately decrypt the bit while waiting for the next bit to download. This is different from a block cipher, where you might need a block of bits, several blocks of bits, or the entire message to be downloaded before you can start decrypting.

A common class of stream cipher algorithms involves outputting an unpredictable stream of bits, and then using this stream as the key of a one-time pad. In other words, each bit of the plaintext message is XORed with the corresponding bit in the key stream.

The output of a secure pRNG can be used as the key for this one-time pad scheme. Formally, in a pRNG-based stream cipher, the secret key is the initial seed used to seed the pRNG. To encrypt an n-bit message, Alice runs the pRNG until it generates n pseudorandom bits. Then she XORs the pseudorandom bits with the plaintext message. Since the pRNG can generate as many bits as needed, this algorithm can encrypt arbitrary-length messages.

To decrypt the message, Bob uses the same secret key to seed the pRNG. Since the pRNG is deterministic with the same seed, Bob will generate the same pseudorandom bits when he runs the pRNG. Then Bob XORs the pseudoranom bits with the ciphertext to learn the original plaintext.

To avoid key reuse, Alice and Bob can both seed the pRNG with a random IV in addition to their secret key so that the pRNG output is different and unpredictable each time. In short, the pRNG algorithm is:

- Encryption: $Enc(K, M) = \langle IV, PRNG(K, IV) \oplus M \rangle$
- Decryption: $Dec(K, IV, C_2) = PRNG(K, IV) \oplus C_2$

AES-CTR is effectively a stream cipher. Although technically AES appears to be a pseudo-random permutation rather than a pseudo-random generator, in practice the results are similar. As long as the total ciphertext encrypted with a given key is kept to a reasonable level (2^{64}b), the one-time pad output of AES-CTR should be effectively indistinguishable from pRNG output. Beyond this threshold, there is a significant probability

with CTR mode that there will be two blocks with identical ciphertext, which would leak information that the underlying plaintext blocks are different.

Although theoretically we could use any cryptographically secure pRNG (like HMAC-DRBG) as a stream cipher, dedicated stream ciphers (such as the ChaCha20 cipher) have properties that we would consider a disadvantage in a secure pRNG but are actually advantages for stream ciphers. In particular, both AES-CTR mode encryption and ChaCha20 include a counter value in the computation of the stream.

One desirable consequence of including a counter is the ability to encrypt or decrypt an arbitrary point in the message without starting from the beginning. If you have a 1 terabyte file encrypted using either AES-CTR mode or ChaCha20 and you wish to read just the last bytes, you can set the counter to the appropriate point and just decrypt the last bytes, while if you used HMAC-DRBG as the stream cipher, you would need to start at the beginning of the message and compute 1 terabytes of HMAC-DRBG output before you could read the end of the file.[2]

[2]This use of a counter value means that if you view it as a pRNG, AES-CTR and ChaCha20 lack rollback resistance as the key and counter are the internal state. But on the other hand, it is this ability to rollback and jump-forward into the output space that makes them more useful as stream ciphers.