Python Radio 39: Secret Unbreakable Code

Simon Quellen Field

Simon Quellen Field

Follow

10 min read

·

May 19, 2025

Listen

Share

More

A One-Time Pad is Cryptographically Secure.

Press enter or click to view image in full size

Secret agent reading coded message.

MidJourney

How sure are you that governments can’t break the encryptions commonly in use? If you were a journalist in a repressive dictatorship, would you trust your freedom or your life to them?

There is a well-known method that is proven to be safe against cryptographic attacks. It is called a one-time pad.

The idea is simple. Each person has a pad of paper that has random letters on it. They code the message by adding the next random letter to each letter of the original message. Then they burn that page or the pad.

In the 1940s, the information theorist Claude Shannon proved this was unbreakable by cryptanalytic methods.

Four things have to be true for it to be secure. The pad must be truly random (no pseudo-random sequences such as those computers generate). You can never reuse a page of the pad. Both people must keep the pad secret. And the pad must be at least as long as the message to be encoded.

The last two things are the concern of the two people, and the code and hardware I present here don’t address them.

Never reusing a page sounds simple. We just delete the page from memory. But is it truly deleted? Can the NSA or the FSB read the deleted text and then be able to decode old messages?

We handle that by writing random text over the old page instead of deleting it. The truly paranoid can do this in a loop, so that the page gets overwritten by new random data many times. My code does it just once.

Enter the Hardware

That leaves the first requirement: that we make the pad from truly random letters. The computer can’t do that by itself. It needs hardware help.

Here is my design for a hardware random bit generator:

Press enter or click to view image in full size

Schematic for hardware random bit generator.

Schematic by the author

This design uses the unpredictable breakdown of a Zener diode that is due to the quantum mechanics of electron tunneling.

I ordered the parts and started writing the code.

But as I was writing, it occurred to me that I have another handy source of random entropy.

The Benefits of Radio Static

One of the problems we encountered when dealing with the data link using the 433 MHz transmitters and receivers was the problem of noise. When the signal was strong, the receiver performed well. The signal-to-noise ratio was excellent, even at half a kilometer.

But when we weren’t transmitting, we got flooded with noise. The receiver would increase its sensitivity until it heard something, even when there was nothing to hear.

And I already had the receiver. All I needed to do was listen to the output when the transmitter was off. That radio noise has all the entropy we need. We don’t even need an antenna.

I connected the power, ground, and output of the receiver to an ESP32-S3 and collected some data using the analog-to-digital port.

Some Software Assistance

Either hardware source has plenty of entropy (randomness). But we won’t assume that there isn’t some bias in there somewhere. Many of the samples will be zero, since I didn’t bother to add a DC bias to bring negative samples into the range of the ADC. We can handle all of that in software if we collect more entropy than we need.

There is a Python library full of hash routines that have been exhaustively tested for their cryptographic properties. We can collect 800 bits of entropy and hand that to the SHA256 hashing algorithm to get 256 bits of well-characterized random numbers.

We are now ready to walk through the code.

From machine import Pin, ADC, freq

From os import statvfs, remove, stat

From hashlib import sha256

From sys import print\_exception

Freq(240\_000\_000)

Green\_led = Pin(34, Pin.OUT)

Hex\_digit = {

“0”: 0,

“1”: 1,

“2”: 2,

“3”: 3,

“4”: 4,

“5”: 5,

“6”: 6,

“7”: 7,

“8”: 8,

“9”: 9,

“a”: 10,

“b”: 11,

“c”: 12,

“d”: 13,

“e”: 14,

“f”: 15,

}

Def df():

S = statvfs(‘//’) # Note: ‘//’ represents the root filesystem

Block\_size = s[0]

Free\_blocks = s[3]

Print(f”Block size: {block\_size}”)

Print(f”Free blocks: {free\_blocks}”)

Return int(free\_blocks \* block\_size)

Class OTP:

Def \_\_init\_\_(self):

Self.code = “”

Self.unused = 0

B = None

Try:

With open(“one\_time\_pad.nxt”, “rb”) as f:

B = f.read()

Except OSError as e:

Pass

If b:

Self.unused = int.from\_bytes(b, ‘little’)

Else:

Print(f”Next address is starting at {self.unused}”)

Self.last\_unused = self.unused

Def in\_hex(self, b):

S = “”

For x in b:

S += f”{x:02x}”

Return s

Def update(self):

From time import ticks\_cpu

B = self.unused.to\_bytes(4, ‘little’)

With open(“one\_time\_pad.nxt”, “wb”) as f:

f.write(b)

if self.unused == 0:

return

size = self.unused – self.last\_unused

random = (int.from\_bytes(self.code[0:4], ‘little’) + ticks\_cpu()) % 1\_000\_000

with open(“one\_time\_pad.otp”, “r+b”) as f:

b = b’’

while len(b) == 0:

f.seek(random, 0)

b = f.read(4)

random = (int.from\_bytes(b, ‘little’) + ticks\_cpu()) % 1\_000\_000

f.seek(random, 0)

b = f.read(size)

f.seek(self.last\_unused, 0)

backward = bytes(reversed(b))

f.write(backward) # Clobber used-up pad bytes with random data

def delete\_one\_time\_pad(self):

try:

remove( “one\_time\_pad.nxt”)

except OSError as e:

pass

try:

remove( “one\_time\_pad.otp”)

except OSError as e:

pass

# @micropython.native

Def create\_one\_time\_pad(self):

Self.unused = 0

Leave\_free = 512 \* 1\_024

Free = df() / 2 # Don’t use more than half the remaining flash

Print(f”Filesystem free space: {free}”)

If free < leave\_free:

Return

Adc = ADC(Pin(10))

Try:

Stat(“one\_time\_pad.otp”)

Print(“One time pad already exists”)

Return

Except OSError:

Pass

With open(“one\_time\_pad.otp”, “wb”) as f:

From time import ticks\_ms, ticks\_diff

Start = ticks\_ms()

Count = 0

While free > leave\_free:

# Collect 100 bytes of random noise (800 bits of entropy)

Noise\_string = b’’

For characters in range(25):

Noise = 0

For foo in range(20):

Noise += int(adc.read())

Noise\_string += noise.to\_bytes(4, ‘little’)

Sha = sha256(noise\_string).digest()

f.write(sha)

free -= len(sha)

left = free – leave\_free

count += len(sha)

elapsed\_seconds = ticks\_diff(ticks\_ms(), start) / 1\_000

rate = count / elapsed\_seconds

mins = int((left / rate) // 60)

hrs = int(mins // 60)

mins -= hrs \* 60

# print(f”{left:6d}, [time left:{hrs:2d}:{mins:02d}] {self.in\_hex(sha)}”)

Print(f”time left:{hrs:2d}:{mins:02d}”, end=”\r”)

Self.unused = 0

Self.last\_unused = 0

Self.update()

Def decrypt(self, pwd, hex\_msg):

Pl = len(pwd)

Msg = hex\_msg[0:8]

For x in range(8, len(hex\_msg), 2):

Msg += chr((hex\_digit[hex\_msg[x]] << 4) | hex\_digit[hex\_msg[x+1]])

Unused = int(msg[0:8], 16)

L = len(msg) – 8

Out = “”

With open(“one\_time\_pad.otp”, “rb”) as f:

f.seek(unused, 0)

self.code = f.read(l)

for x in range(l):

p = ord(pwd[x % pl])

out += chr(p ^ self.code[x] ^ ord(msg[x+8]))

try:

msg\_length = int(out[0:8], 16) + 8

except ValueError as e:

print(“Decode failed. Don’t decode on the same device you encode on.”)

return “”

if unused > self.unused:

self.unused = unused # Stay matched with the other device so we can reply

return(out[8:msg\_length])

def encrypt(self, pwd, msg):

pl = len(pwd)

l = ((len(msg) // 32) + 1) \* 32

out = bytearray(b’\x00’ \* (l+8))

out[:8] = bytearray(f”{self.unused:08.8x}”, “utf-8”)

with open(“one\_time\_pad.otp”, “rb”) as f:

f.seek(self.unused, 0)

try:

self.code = f.read(l)

except Exception as e:

print\_exception(e)

for x in range(len(msg)):

p = ord(pwd[x % pl])

out[x+8] = int(p ^ self.code[x] ^ ord(msg[x]))

start = len(msg) + 8

stop = l

for cnt in range(start, stop, 1):

print(cnt)

out[cnt] = int(self.code[cnt])

print(“Test decode: “, end=””)

test = bytearray(b’\x00’ \* len(msg))

for x in range(len(msg)):

p = ord(pwd[x % pl])

test[x] = int(p ^ self.code[x] ^ out[x+8])

for x in test:

if x >= 32 and x < 127:

print(chr(x), end=””)

else:

print(“.”, end=””)

print()

self.unused += l

self.update()

return out[:l]

def ui(self):

Green\_led.on()

While True:

Test = input(“Create, Remove, Encrypt, or Decrypt? “)

If len(test) == 0:

Print(“Type c, r, e, or d”)

Elif test[0].lower() == ‘c’:

Self.create\_one\_time\_pad()

Elif test[0].lower() == ‘r’:

Confirm = input(“Really delete the one time pad that takes a long time to build? “)

If confirm == “yes”:

Self.delete\_one\_time\_pad()

Elif test[0].lower() == ‘e’:

Pwd = input(“Enter password: “)

Msg = input(“Enter message to encrypt: “)

Coded = self.encrypt(pwd, f”{len(msg):08.8x}” + msg)

Print(f”Coded: {coded[0:8].decode()}{self.in\_hex(coded[8:])}”)

Elif test[0].lower() == ‘d’:

Pwd = input(“Enter password: “)

Msg = input(“Enter message to decrypt: “)

If msg != “”:

Decoded = self.decrypt(pwd, msg)

Print(f”Decoded: {decoded}”)

Else:

Print(“Type c, r, e, or d”)

Def main():

Otp = OTP()

Otp.ui()

Main()

We’ll start with the create\_one\_time\_pad() method.

We want a large pad, but we don’t want to completely fill up our flash memory. We find out how much space we have, use only half, and leave at least half a megabyte free.

Now we collect some random bits. We add up 20 bytes from the ADC to get a 4-byte integer. We convert that into a 4-byte string and append it to our string of random characters. We do this 25 times to get 100 bytes of random noise.

We hand that string to the SHA256() method, and write the resulting 32 bytes to our one-time pad. We continue doing this until the pad is full.

This can take about 15 minutes, so we want to give the user some indication of progress. We calculate the rate at which the file grows and give an estimate of the completion time. On smaller devices with only 4 megabytes of flash, it may take five minutes or less.

Finally, we call the update() method.

Update() has two jobs. It keeps track of which part of the pad we have used in a file called one\_time\_pad.nxt. Its second job is to clobber the part we have used. We want to put random data there. But where can we find random data? Oh, yes! We have a whole file full of it!

We take 4 bytes of our last random patch and add the clock to it. Then we use that to seek into the file to get another one. We keep doing that until we get something that isn’t zero. I did this because it was convenient during debugging to kill the program before the whole 15 minutes were up, and the file was shorter than a megabyte. Randomly seeking would often lead off the end of the file, returning zeros.

Once we had enough random bits to write over the part of the pad we had consumed, we reverse the bit string for luck and write over the used part of the pad.

Now we are ready to encrypt a message.

You have probably used two-factor authentication before as a security method. Three-factor authentication is a little better. The three parts are:

Something you know (such as a password).

Something you have (in our case, that is our one-time pad).

Something you are (this is where fingerprint readers or iris scans come in).

You can buy fingerprint readers for about $10 that are easy to connect to our ESP32 using a UART port. Mine is on order, so this code doesn’t use a fingerprint scanner yet, although that is an easy software adjustment to make.

The code does use a password, however. So we have two factors, which is good enough for my bank, so maybe it’s good enough for now.

The password and the message to encode get passed to the encrypt() method.

Encrypt() adds the address into the pad as the first 8 bytes. This is not encrypted.

To keep an adversary from knowing how long our message is, we bump the length up to the next 32 bytes. Our encrypted coded message will always be a multiple of 32 bytes long.

Then, for each letter in the message, we XOR it with a letter from the password and a letter from the one-time pad.

The nice feature of the XOR function is that you can run it twice and get the original message back. That is how we decrypt. To make sure we did it right, we decrypt it right away and present it to the user. She already knows the message, so we haven’t given anything away.

Lastly, we call update to save the address and clobber the code.

Decrypt() reads the address (remember, we did not encrypt that) in the first 8 bytes of the coded message. It reads the pad at that address and does the XORs. The first 8 bytes of the message are the message length. We need that so we can ignore the random noise that fills out the length to 32 bytes. Then we return the decoded message.

The only part left is the main() function, which is mostly self-explanatory.

It collects the password and message and adds the message length before calling encrypt(). At the other end, the message recipient will paste the message into decrypt() to read it.

The Results

Let’s look at the code in action. Here is the encoding:

Create, Remove, Encrypt, or Decrypt? E

Enter password: this is a longish password with lots of entropy and yet still something I can remember.

Enter message to encrypt: This is a test of our one-time pad encryption mechanism.

Test decode: 00000038This is a test of our one-time pad encryption mechanism.

Coded: 00000300cb7795fa16de07e904452187a47d68ce8ea15bfbe9bc8aa22f3acf07a584f429a4fbecb00c785d8adb1b51b0081b606d8fe8ff404589622fdd606997e09829b88c019469e8c260942583acc8fc8a37aab28e3da5f829697d

Create, Remove, Encrypt, or Decrypt?

Next, we copy the coded message into the decoding machine:

Create, Remove, Encrypt, or Decrypt? D

Enter password: this is a longish password with lots of entropy and yet still something I can remember.

Enter message to decrypt: 00000300cb7795fa16de07e904452187a47d68ce8ea15bfbe9bc8aa22f3acf07a584f429a4fbecb00c785d8adb1b51b0081b606d8fe8ff404589622fdd606997e09829b88c019469e8c260942583acc8fc8a37aab28e3da5f829697d

Decoded: This is a test of our one-time pad encryption mechanism.

Create, Remove, Encrypt, or Decrypt?

Make sure you don’t try to decrypt the message on the same machine you encrypted it on. That machine has already clobbered the one-time pad used to encrypt the message.

Some Words About Downloading

Our code requires the hashlib library. We install that using the mpremote program running on the host machine (a Windows machine in my case):

Mpremote connect com4 mip install hashlib

We build the one-time pad on one esp32, and then copy the two files to the other esp32:

Mpremote connect com4 fs cp :one\_time\_pad.nxt one\_time\_pad.nxt

Mpremote connect com4 fs cp :one\_time\_pad.otp one\_time\_pad.otp

Mpremote connect com8 fs cp one\_time\_pad.nxt :one\_time\_pad.nxt

Mpremote connect com8 fs cp one\_time\_pad.otp :one\_time\_pad.otp

The colon tells mpremote we are talking about a file on the remote machine. And (of course) you will change com4 and com8 to the ports you use on your own machine. On a Mac or Linux, they will be found in /dev.

Since the one\_time\_pad.otp file is rather large, expect to wait a while for it to copy.

Lastly, you must securely delete the one\_time\_pad.otp from your host machine. Write over it several times with random data.