“Constant time” compare in Python

You may be familiar with the following piece of code to implement the constant time comparison function for strings:

```
def constant_time_compare(val1, val2):
    if len(val1) != len(val2):
        return False
    result = 0
    for x, y in zip(val1, val2):
        result |= x ^ y
    return result == 0
```

The idea behind this code is to compare all bytes of input using a flag value that will be flipped in any of the comparisons fail. Only when all the bytes were compared, is the ultimate result of the method returned. This is used to thwart attacks that use the time of processing queries to guess secret values.

Unfortunately, because of CPython specifics, this code **doesn’t work** for its intended purpose.

Sensitive code should always use `hmac.compare_digest()` (https://docs.python.org/2/library/hmac.html#hmac.compare_digest) method and you should not write code that needs to be side channel secure in Python.

With the tl;dr version out, let’s investigate why.

Timing side channel

Many attacks against cryptographic implementations don’t actually use maths to compromise the systems. The Bleichenbacher Million Messages attack (http://archiv.infsec.ethz.ch/education/fs08/secsem/bleichenbacher98.pdf), POODLE (https://www.openssl.org/~bodo/ssl-poodle.pdf) and Lucky 13 (http://www.isg.rhul.ac.uk/tls/TLStiming.pdf) attacks use some kind of a side channel to guess the contents of encrypted messages, either the timing of responses or contents of responses (different TLS Alert description field values).

Side channel attacks don’t impact only cryptographic protocols, other places where secret values need to be compared to values that are controlled by attacker, like checking password equality, API tokens and HMAC value validation need to be performed in constant time too.

Already back in 00’s, differences in timing as low as 100ns could be distinguished over LAN environment. See research by Crosby at al. in Opportunities And Limits Of Remote Timing Attacks (https://www.cs.rice.edu/~dwallach/pub/crosby-timing2009.pdf) and Brumley and Boneh in Remote TIming Attacks are Practical (http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf). Currently we also have to worry about cross VM or cross-process attacks where ability to distinguish between single cycles may be possible.

Measuring timing differences

Let’s see what happens if we use the simple way to compare two strings in python, the == operator.

Benchmarking code:

timing-eq_cmp-perf.py

```
import perf

setup = """
str_a = b'secret API key'

alt_s = b'XXXXXXXXXXXXXX'

str_b = str_a[:{0}] + alt_s[{0}:]

assert len(str_a) == len(str_b)
"""

fun = """str_a == str_b"""

if __name__ == "__main__":
    total_runs = 128
    runs_per_process = 4
    runner = perf.Runner(values=runs_per_process,
                         warmups=16,
                         processes=total_runs//runs_per_process)
    vals = list(range(14))  # length of str_a
    for delta in vals:
        runner.timeit("eq_cmp delta={0:#04x}".format(delta),
                      fun,
                      setup=setup.format(delta))
```

Running it will simulate what timings does the attacker see when the difference from the expected value is at different positions in the attacker provided string.

command

```
PYTHONHASHSEED=1 python3 timing-eq_cmp-perf.py \
-o timing-eq_cmp-perf-1.py --fast
```

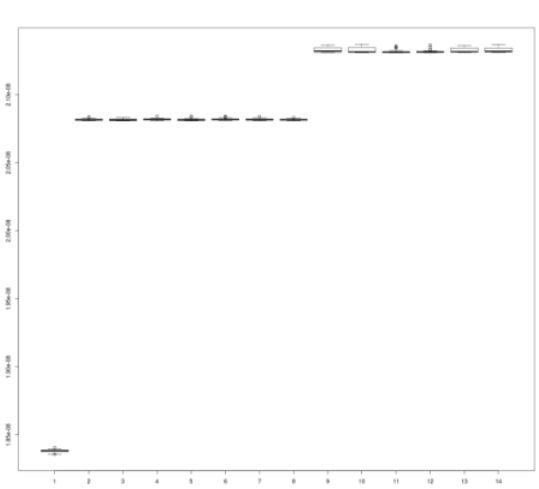
output

```
.................
eq_cmp delta=0x00: Mean +- std dev: 18.4 ns +- 0.0 ns
.................
eq_cmp delta=0x01: Mean +- std dev: 20.8 ns +- 0.0 ns
.................
eq_cmp delta=0x02: Mean +- std dev: 20.8 ns +- 0.0 ns
.................
eq_cmp delta=0x03: Mean +- std dev: 20.8 ns +- 0.0 ns
.................
eq_cmp delta=0x04: Mean +- std dev: 20.8 ns +- 0.0 ns
.................
eq_cmp delta=0x05: Mean +- std dev: 20.8 ns +- 0.0 ns
.................
eq_cmp delta=0x06: Mean +- std dev: 20.8 ns +- 0.0 ns
.................
eq_cmp delta=0x07: Mean +- std dev: 20.8 ns +- 0.0 ns
.................
eq_cmp delta=0x08: Mean +- std dev: 21.3 ns +- 0.0 ns
.................
eq_cmp delta=0x09: Mean +- std dev: 21.3 ns +- 0.0 ns
.................
eq_cmp delta=0x0a: Mean +- std dev: 21.3 ns +- 0.0 ns
.................
eq_cmp delta=0x0b: Mean +- std dev: 21.3 ns +- 0.0 ns
.................
eq_cmp delta=0x0c: Mean +- std dev: 21.3 ns +- 0.0 ns
.................
eq_cmp delta=0x0d: Mean +- std dev: 21.3 ns +- 0.0 ns
```

Already we can see that the difference in timing when the first different byte is at fist position or the second position is quite huge, looking at a box plot (https://en.wikipedia.org/wiki/Box_plot) of the specific values makes it quite obvious:

R script

```
a = read.csv(file="timing-eq_cmp-perf-1.csv", header=FALSE)
data = as.matrix(a)
boxplot(t(data))
```



(https://securitypitfalls.files.wordpress.com/2018/07/timing-eq_cmp-perf-1-boxplot1.png)

Let's see how does it compare to the `"constant_time"_compare`. First the code:

timing-ct_eq_cmp-perf.py

```
import perf

setup = """
def constant_time_compare(val1, val2):
    if len(val1) != len(val2):
        return False
    result = 0
    for x, y in zip(val1, val2):
        result |= x ^ y
    return result == 0

str_a = b'secret API key'

alt_s = b'XXXXXXXXXXXXXX'

str_b = str_a[:{0}] + alt_s[{0}:]

assert len(str_a) == len(str_b)
"""

fun = """constant_time_compare(str_a, str_b)"""

if __name__ == "__main__":
    total_runs = 128
    runs_per_process = 4
    runner = perf.Runner(values=runs_per_process,
                         warmups=16,
                         processes=total_runs//runs_per_process)
    vals = list(range(14))  # length of str_a
    for delta in vals:
        runner.timeit("ct_eq_cmp delta={0:#04x}".format(delta),
                      fun,
                      setup=setup.format(delta))
```

The test run:

command

```
PYTHONHASHSEED=1 python3 timing-ct_eq_cmp-perf.py \
-o timing-ct_eq_cmp-perf-1.json --fast
```

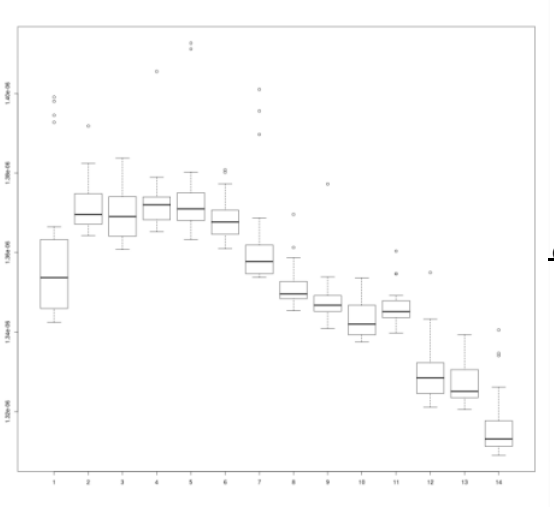
output

```
.................
ct_eq_cmp delta=0x00: Mean +- std dev: 1.36 us +- 0.02 us
.................
ct_eq_cmp delta=0x01: Mean +- std dev: 1.37 us +- 0.01 us
.................
ct_eq_cmp delta=0x02: Mean +- std dev: 1.37 us +- 0.01 us
.................
ct_eq_cmp delta=0x03: Mean +- std dev: 1.37 us +- 0.01 us
.................
ct_eq_cmp delta=0x04: Mean +- std dev: 1.37 us +- 0.01 us
.................
ct_eq_cmp delta=0x05: Mean +- std dev: 1.37 us +- 0.00 us
.................
ct_eq_cmp delta=0x06: Mean +- std dev: 1.36 us +- 0.01 us
.................
ct_eq_cmp delta=0x07: Mean +- std dev: 1.35 us +- 0.01 us
.................
ct_eq_cmp delta=0x08: Mean +- std dev: 1.35 us +- 0.01 us
.................
ct_eq_cmp delta=0x09: Mean +- std dev: 1.34 us +- 0.00 us
.................
ct_eq_cmp delta=0x0a: Mean +- std dev: 1.35 us +- 0.00 us
.................
ct_eq_cmp delta=0x0b: Mean +- std dev: 1.33 us +- 0.01 us
.................
ct_eq_cmp delta=0x0c: Mean +- std dev: 1.33 us +- 0.01 us
.................
ct_eq_cmp delta=0x0d: Mean +- std dev: 1.32 us +- 0.01 us
```

The results don't look too bad, but there's definitely a difference between the first and last one, even accounting for one standard deviation between them. Let's see the box plot:

R script

```
a = read.csv(file="timing-ct_eq_cmp-perf-1.csv", header=FALSE)
data = as.matrix(a)
boxplot(t(data))
```



(https://securitypitfalls.files.wordpress.com/2018/07/timing-ct_eq_cmp-perf-1.png)

That doesn't look good. Indeed, if we compare the distributions for the different delta values using the Kolmogorov–Smirnov test (https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Smirnov_test), we'll see that results for all deltas are statistically different:

R code

```
a = read.csv(file="timing-ct_eq_cmp-perf-1.csv", header=FALSE)
data = as.matrix(a)
r = c()
for (i in c(1:length(data[,1]))){
  r[i] = ks.test(data[1,], data[i,])$p.value}
which(unlist(r) < 0.05/(length(data1[,1])-1))
```

"output"

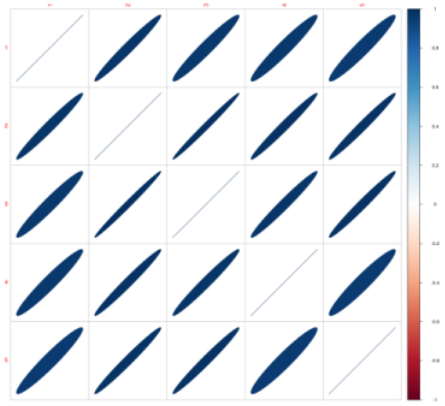
```
[1]  2  3  4  5  6  7  9 10 11 12 13 14
```

Which means that the distributions are statistically distinguishable. (The 0.05 p-value is divided by the amount of performed tests because we're applying the Bonferroni correction (https://en.wikipedia.org/wiki/Bonferroni_correction))

To make sure, we re-run the test 4 more times and check for correlation between medians (as the distributions are unimodal, median is robust statistic (https://en.wikipedia.org/wiki/Robust_statistics)).

R code

```
require(corrplot)

a = read.csv(file="timing-ct_eq_cmp-perf-1.csv", header=FALSE)
data = as.matrix(a)
vals = cbind(apply(data, 1, median))

for (i in 2:5) {
  name = paste("timing-ct_eq_cmp-perf-", i, ".csv", sep="")
  a = read.csv(file=name, header=FALSE)
  data = as.matrix(a)
  vals = cbind(vals, apply(data, 1, median))
}
corrplot(cor(vals, method="spearman"), method="ellipse")
```



(https://securitypitfalls.files.wordpress.com/2018/07/timing-ct_eq_cmp-perf-corrplot.png)

There is a very strong correlation between all the different runs, so indeed, it does look like the function is leaking timing information.
Note, we're using the `"spearman"` correlation statistic (https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient) as the values are not normally distributed.

Let's compare it to the `hmac.compare_digest()` method:

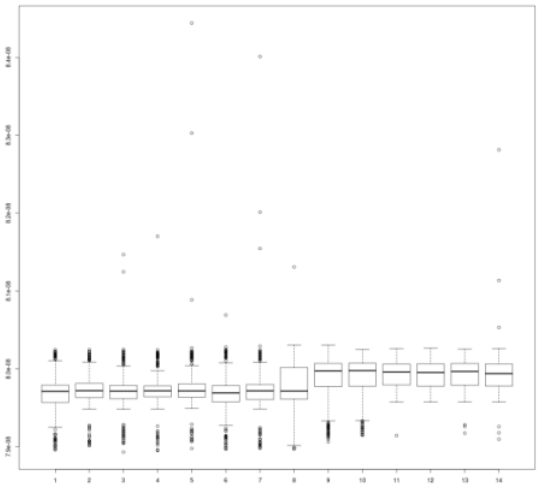
```
import perf

setup = """
from hmac import compare_digest

str_a = b'secret API key'

str_b = b''.join((str_a[:{0}], b'X' * (14 - {0})))

assert len(str_a) == len(str_b)
assert len(str_a) == 14
"""

fun = """compare_digest(str_a, str_b)"""

if __name__ == "__main__":
    total_runs = 128
    runs_per_process = 4
    runner = perf.Runner(values=runs_per_process,
                         warmups=64,
                         processes=total_runs//runs_per_process)
    vals = list(range(14))  # length of str_a
    for delta in vals:
        runner.timeit("compare_digest delta={0:#04x}".format(delta),
                      fun,
                      setup=setup.format(delta))
```

command

```
PYTHONHASHSEED=1 python3 timing-compare_digest-perf.py \
-o timing-compare_digest-perf-1.json --rigorous
```

R code

```
a = read.csv(file="timing-compare_digest-perf-1.csv", header=FALSE)
data = as.matrix(a)
boxplot(t(data))
```



(https://securitypitfalls.files.wordpress.com/2018/07/timing-compare_digest-perf-1.png)

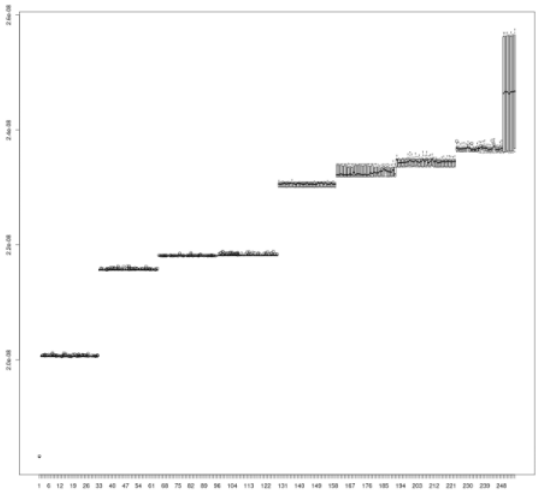
While there is some difference that depends on where the first differing byte is, there is no difference between first and second byte, and the "step" around 8th byte is only around it (when comparing longer strings, I still see just one step at the beginning and one at the end). I have no good explanation for it. That being said, the difference between medians of the 2nd byte and 11th byte is 0.240 ns, for comparison, one cycle of the CPU (4Ghz) on which the test is running takes 0.250 ns. So I'm assuming that it is not detectable over the network, but may be detectable in cross-VM attacks.

To confirm the results I've run the test with simple `==` for 255 byte long strings and with using the `hmac.compare_digest()`.

Results for `==`:

R code

```
a = read.csv(file="timing-eq_cmp-2-perf-1.csv", header=FALSE)
data = as.matrix(a)
boxplot(t(data))
```



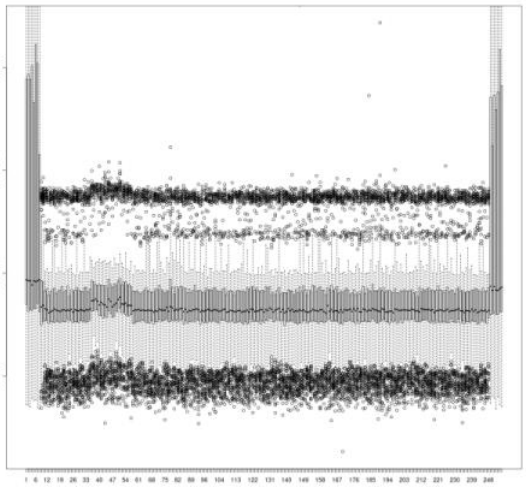
(https://securitypitfalls.files.wordpress.com/2018/07/timing-eq_cmp-2-perf-1.png)

As expected, obvious steps that are directly dependant on the amount of matching data between the two parameters to the operator.

Results for `compare_digest()`:

R code

```
a = read.csv(file="timing-compare_digest-8-perf-1.csv", header=FALSE)
data = as.matrix(a)
boxplot(t(data), ylim=c(min(data), quantile(data, 0.99)))
```

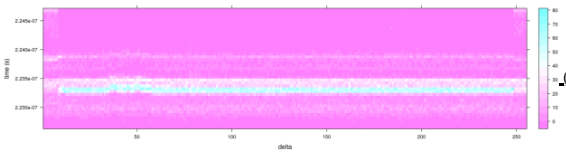


(https://securitypitfalls.files.wordpress.com/2018/07/timing-compare_digest-8-perf-1.png)

They are quite noisy, but what the grouping around 2.239e-7 hints at (the thick horizontal line comprised of circles), is that the distribution is not unimodal (otherwise the outliers would look like the ones below the boxes). Let's see what are the counts for different time bins, as in a histogram, in detail:

R code

```
require("lattice")
a = read.csv(file="timing-compare_digest-8-perf-1.csv", header=FALSE)
data = as.matrix(a)
h <- hist(data, breaks=200,plot=FALSE)
breaks = c(h$breaks)
mids = c(h$mids)
hm <- rbind(hist(data[1,], breaks=breaks, plot=FALSE)$counts)
for (i in c(2:length(data[,1]))) {
  hm <- rbind(hm, hist(data[i,], breaks=breaks, plot=FALSE)$counts)}

d = data.frame(x=rep(seq(1, nrow(hm), length=nrow(hm)), ncol(hm)),
               y=rep(mids, each=nrow(hm)),
               z=c(hm))
levelplot(z~x*y, data=d, xlab="delta", ylab="time (s)",
  ylim=c(min(data), quantile(data, 0.99)))
```



(https://securitypitfalls.files.wordpress.com/2018/07/timing-compare_digest-8-perf-1-levelplot.png)

We can see now, that even though the measurements with delta between 0 and 8 and 249 and 255 look very different on the box plot, it's more because a third mode was added to them rather than one of the other two was removed. Statistical test confirms this:

R code

```
a = read.csv(file="timing-compare_digest-8-perf-1.csv", header=FALSE)
data = as.matrix(a)
r = c()
for (i in c(1:length(data[,1]))){
  r[i] = ks.test(data[19,], data[i,])$p.value}
which(unlist(r) < 0.05/nrow(data))
```

"result"

```
[1]   1   2   3   4   5   6   7   8  36  37  38  39  41  45  46  49  50  51  52
[20]  53  54 249 250 251 252 253 254 255
```

(the deltas between 36 and 54 are a fluke that subsequent quick runs didn't show).

Note about benchmarking

You may have noticed that the data we have collected, has very low amounts of noise. While it is partially the result of use of the perf (https://pypi.org/project/perf/) module instead of the timeit (https://docs.python.org/3/library/timeit.html) library module, it mostly is the result of careful system configuration.

On the benchamrking system, the following tasks were performed:

- 3rd and 4th core were isolated
- kernel RCU was disabled on the isolated cores
- HyperThreading was disabled in BIOS
- Intel TurboBoost was disabled
- Intel power management was disabled (no C-states or P-states other than C0 were allowed)
- CPU frequency was locked in place to 4Ghz (the nominal for the i7 4970K of the workstation used)
- Decreasing maximum perf probe query rate to 1 per second
- Disabling irqbalance and setting default IRQ affinity to un-isolated cores
- ASRL disabled
- Python hash table seed fixed

Those operations can be performed by:

1. Adding `isolcpus=2,3 rcu_nocbs=2,3 processor.max_cstate=1 idle=poll` to the kernel command line
2. Disabling HyperThreading in BIOS
3. Running `python3 -m perf system tune`
4. Disabling ASLR by running `echo 0 > /proc/sys/kernel/randomize_va_space`
5. exporting the `PYTHONSEED` environment variable

Documentation (https://perf.readthedocs.io/en/latest/system.html) of the perf module provides most of the explanations of the particular options, but we diverge in two places: ASLR and Python hash seed. The purpose of the perf module is to test the overall performance of a piece of Python code (and compare it to either compilation or different implementation). Because Python is a language than answers the question "what if everything was a hash table" ;), that means the names of variables, memory positions of variables or code (https://vstinner.github.io/analysis-python-performance-issue.html), number of variables, and particular hash table key have significant impact on performance. But, because we are interested if an attacker is able to tell behaviour of code between two different inputs, and those two inputs will likely be processed by the same process, both the ASLR seed and the Python hash table seeds will be constant from the point of view of the attacker. To speed up finding the expected value for particular inputs I thus opted out of those randomisation mechanisms.

Expectations of behaviour

You may wonder, why is the Python code so unstable, so data dependant, if the implementation of `hmac.compare_digest()` is doing exactly the same thing (https://github.com/python/cpython/blob/80b762f010097ab8137782e5fbdc89c5c620ed4e/Modules/_operator.c#L727-L762) (xor-ing the values together and then or-ing result with a guard variable)? The problem stems from the fact that the Python `int` and C `unsigned char` are vastly different data types – one is used for arbitrary precision arithmetic while the other can store just 256 unique values. Thus, even such simple operations like xor or or with two small integers are data dependant in Python.

Let's see how much time does the Python VM need for those two small integers. (Unfortunately, it looks like perf uses the slow json module, and because it exports results after every loop iteration, after few hundred results, the export takes more time than benchmarking. To make it fast enough, and not waste few days on exporting the same data over and over again, we will use timeit module.)

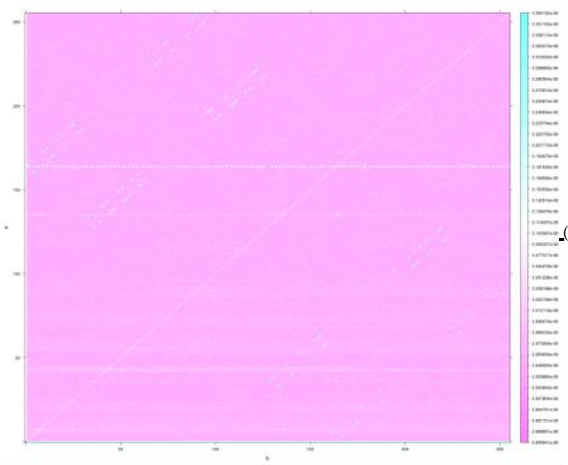
Script:

timing-xor-2-timeit.py

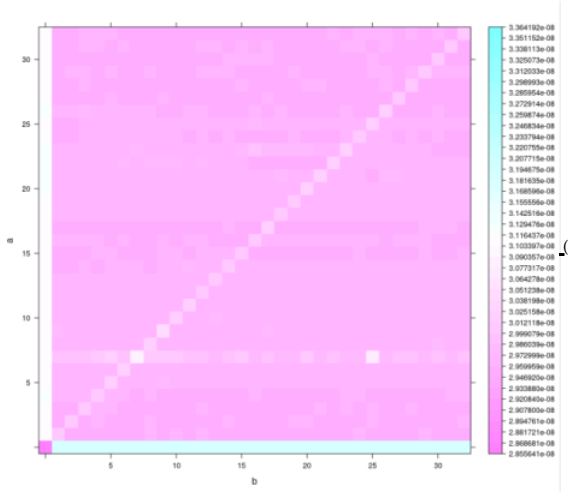
```
import timeit
import sys
import math

setup = """
val_a = {0}

val_b = {1}
"""

fun = """val_a ^ val_b"""

def std_dev(vals):
    avg = sum(vals)/len(vals)
    sum_sq = sum((i - avg)**2 for i in vals)
    return math.sqrt(sum_sq / (len(vals) - 1))

if __name__ == "__main__":
    total_runs = 20
    runs_per_process = 3
    warmups = 16

    runner = timeit.Timer(fun, setup=setup.format(0, 0))
    number, delay = runner.autorange()
    number //= 2
    delay /= 2

    print(("will do {0} iterations per process, "
           "expecting {1:7.2} s per process")
          .format(number, delay), file=sys.stderr)
    print("warmups:", file=sys.stderr, end='')
    sys.stderr.flush()
    for _ in range(warmups):
        timeit.repeat(fun, setup=setup.format(0, 0), repeat=1,
                      number=number)
        print(".", file=sys.stderr, end='')
        sys.stderr.flush()
    print(file=sys.stderr)

    for a in range(256):
        for b in range(256):
            res = []
            for _ in range(total_runs // runs_per_process):
                # drop the first result as a local warmup
                res.extend(i / number for i in
                           timeit.repeat(fun,
                                         setup=setup.format(a, b),
                                         repeat=runs_per_process + 1,
                                         number=number)[1:])
                print(".", file=sys.stderr, end='')
                sys.stderr.flush()
                if std_dev(res)  timing-xor-2-timeit-1.csv
```

R code

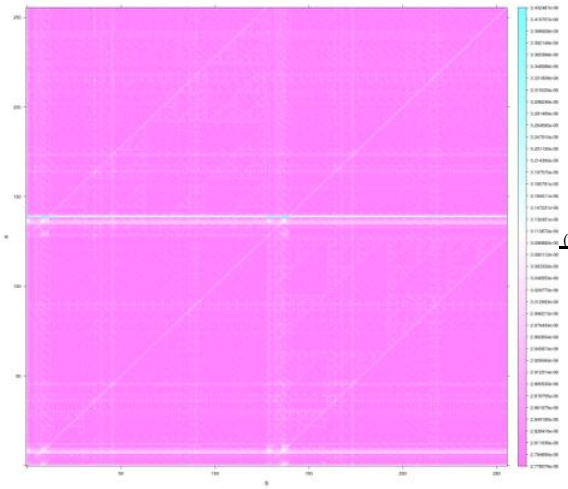
```
require("lattice")
a = read.csv(file="timing-xor-2-timeit-1.csv",
             header=FALSE, col.names=seq(1, 20),
             fill=TRUE)
data = as.matrix(a)
med = apply(data, 1, median, na.rm=TRUE)
# full lines
len = length(med)
columns = ceiling(length(med) / 256)
d = data.frame(x=rep(seq(0, 255), length.out=len, 256),
               y=rep(seq(0, 255), length.out=len, each=256),
               z=med)
my.at = seq(min(med), max(med), length=40)
levelplot(z~x*y, data=d, xlab="b", ylab="a",
          at=my.at, aspects="iso",
          colorkey=list(at=my.at, labels=list(at=my.at)))
```


(https://securitypitfalls.files.wordpress.com/2018/07/timing-xor-2-timeit-1.png)

While there few repeating patterns, there are 4 things that are of particular importance – behaviour when the two numbers are equal (the lighter diagonal), when both are zero, or when one of the operands is zero. The difference between the background and the diagonal is small, just 0.555 ns, but that translates to about 2 cycles at 4GHz. The difference between the 0, 0 and the backgrounds is even smaller, just 0.114 ns, so half a cycle. The difference between the background and the situations when the second variable is non-zero is about 2.24 ns which translates to about 9 cycles. When the first variable is non-zero and the second is, the difference is about 1.39 ns which is about 6 cycles. Here's the zoomed in part of the graph for the small numbers:


(https://securitypitfalls.files.wordpress.com/2018/07/timing-xor-2-timeit-1-zoom.png)

The binary or operator is similarly dependant on values of parameters:


(https://securitypitfalls.files.wordpress.com/2018/08/timing-or-timeit-1.png)

Both of those things put together mean that using the supposedly constant time compare doesn't actually protect against timing attacks, but rather makes them *easier*. The strength of the signal for different inputs is about 100 time stronger, likely allowing them even over Internet, not only over LAN (as is the case for == operator).

Anything else?

Because I started looking into those microbenchmarks to verify the "constant" time CBC MAC and pad check from tlslite-ng (https://github.com/tomato42/tlslite-ng/blob/02852484e5ca9ea5e0c69d525ff45a3a999b386b/tlslite/utils/constanttime.py#L96), needed to protect against Lucky 13 (see the very extensive article (https://www.imperialviolet.org/2013/02/04/luckythirteen.html) by Adam Langley on the topic), I've also checked if it is possible to speed up the process of hashing data. Because on the Python level we don't have the luxury of access to lower level hash APIs, as the developers of OpenSSL have, to implement the CBC check, I wrote code that in fact calculates 256 different hmacs (https://github.com/tomato42/tlslite-ng/blob/02852484e5ca9ea5e0c69d525ff45a3a999b386b/tlslite/utils/constanttime.py#L178-L186) for every record that contains at least 256 bytes of data + padding. That means that for every record processed, the

client and server actually process 64 KiB of additional data. In theory (that is, if the hmac itself is constant time), we could speed the process of checking the mac and de-padding in TLS dramatically, if we could hash the data just once, as OpenSSL is doing in its TLS implementation. You may say, "but hashes are implemented in C, surely they are constant time!". To which I'll answer, "what did we say about trusting assumptions?".

Let's see how our assumptions hold. First code that hashes all of provided data, but returns also a hash from the "middle" of data (in a TLS implementation that would be the real HMAC that we need to compare to the one from record):

timing-hmac-split-perf.py

```
import perf

setup = """
import hmac
from hashlib import sha1

def fun(digest, data, split):
    digest.update(data[:split])
    ret = digest.copy().digest()
    digest.update(data[split:])
    return ret, digest.digest()

str_a = memoryview(b'X'*256)
key = b'a' * 32

val_b = {0}

mac = hmac.new(key, digestmod=sha1)
"""

fun = """fun(mac.copy(), str_a, val_b)"""

if __name__ == "__main__":
    total_runs = 128
    runs_per_process = 4
    runner = perf.Runner(values=runs_per_process,
                         warmups=16,
                         processes=total_runs//runs_per_process)
    vals = list(range(256))  # length of str_a
    for delta in vals:
        runner.timeit("hmac split delta={0:#04x}".format(delta),
                      fun,
                      setup=setup.format(delta))
```

Command to gather the statistics:

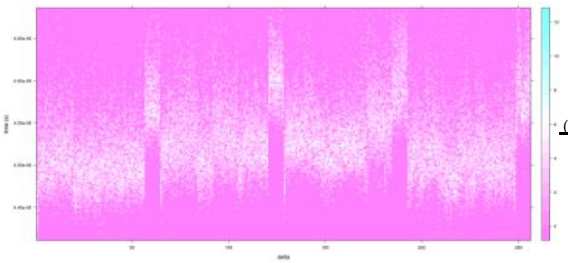
command

```
PYTHONHASHSEED=1 python3 timing-hmac-split-perf.py \
-o timing-hmac-split-perf-1.json
```

And a way to visualise them:

R code

```
require("lattice")
a = read.csv(file="timing-hmac-split-perf-1.csv", header=FALSE)
data = as.matrix(a)
h <- hist(data, breaks=200,plot=FALSE)
breaks = c(h$breaks)
mids = c(h$mids)
hm <- rbind(hist(data[1,], breaks=breaks, plot=FALSE)$counts)
for (i in c(2:length(data[,1]))) {
  hm <- rbind(hm, hist(data[i,], breaks=breaks, plot=FALSE)$counts)}

d = data.frame(x=rep(seq(1, nrow(hm), length=nrow(hm)), ncol(hm)),
               y=rep(mids, each=nrow(hm)),
               z=c(hm))
levelplot(z~x*y, data=d, xlab="delta", ylab="time (s)",
  ylim=c(min(data), quantile(data, 0.99)))
```


(https://securitypitfalls.files.wordpress.com/2018/08/timing-hmac-split-perf-1.png)

Besides the obvious peaks since 56th to 64th byte every 64 bytes (caused by an additional hash block that had to be padded (https://tools.ietf.org/html/rfc4634#section-4) to calculate the intermediate HMAC), there is also a dip for the first byte of the 64 byte block and a second dip for bytes between 20 and 55 of every block. Finally, when the split is about even (in that the intermediate hash is calculated over the first 120 bytes), the whole operation takes measurably longer. In short, if the position of the intermediate hash comes from the last byte of encrypted data (as it does in TLS), calculating HMAC like this has a definite sidechannel leak.

To confirm, let's perform Kolomogorov-Smirnov test:

R code

```
a = read.csv(file="timing-hmac-split-perf-1.csv", header=FALSE)
data = as.matrix(a)
r=c()
for (i in c(1:nrow(data))){
   r[i] = ks.test(data[2,], data[i,])$p.value}
which(unlist(r) < 0.05/(nrow(normalised)-1))
```

(we're testing against second row as the first row (for delta of 0) is *obviously* different from the others so all tests failing wouldn't be unexpected)

result

```
[1]   1  21  23  26  44  57  58  59  60  61  62  63  64  69  71  75  77  78
[19]  80  81  82  83  92  93  94  95  96  98  99 100 103 104 109 114 118 121
[37] 122 123 124 125 126 127 128 130 131 132 133 134 135 136 137 138 139 140
[55] 141 142 143 144 145 146 147 160 161 163 165 169 170 171 172 173 174 175
[73] 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 215
[91] 217 218 249 250 251 252 253 254 255 256
```

Quite obviously different, even with just 128 samples per delta value.

Summary

Moral of the story is, don't use something without testing if it behaves as it claims to. If it does have tests, verify that they check *your* expectations, not only the programmers that wrote it in the first place.

State your assumptions and test them. If values look similar, measure them multiple times, and use statistical methods to compare them.

Test setup

Tests were performed, as previously mentioned, on an Intel i7 4790K CPU. The system was running Linux 4.17.5-1-ARCH with Python 3.6.6-1 and perf 1.5.1 from Archlinux.

Conversion from json files to csv files was performed using `json-to-csv.py` script available at the testing repo, together with raw results, at github (https://github.com/tomato42/python-timing-tests).

Post scriptum

Other operations on integers, including equality are also not constant time:

timing-eq-timeit.py

```
import timeit
import sys
import math


setup = """
val_a = {0}

val_b = {1}
"""


fun = """val_a == val_b"""


def std_dev(vals):
    avg = sum(vals)/len(vals)
    sum_sq = sum((i - avg)**2 for i in vals)
    return math.sqrt(sum_sq / (len(vals) - 1))


if __name__ == "__main__":
    total_runs = 3
    runs_per_process = 3
    warmups = 16

    runner = timeit.Timer(fun, setup=setup.format(0, 0))
    number, delay = runner.autorange()
    number //= 100
    delay /= 100

    print("will do {0} iterations per process, "
          "expecting {1:7.2} s per process"
          .format(number, delay), file=sys.stderr)
    print("warmups:", file=sys.stderr, end='')
    sys.stderr.flush()
    for _ in range(warmups):
        timeit.repeat(fun, setup=setup.format(0, 0), repeat=1,
                      number=number)
        print(".", file=sys.stderr, end='')
        sys.stderr.flush()
    print(file=sys.stderr)

    for a in range(256):
        for b in range(256):
            res = []
            for _ in range(total_runs // runs_per_process):
                # drop the first result as a local warmup
                res.extend(i / number for i in
                           timeit.repeat(fun,
                                         setup=setup.format(a, b),
                                         repeat=runs_per_process+1,
                                         number=number)[1:])
                print(".", file=sys.stderr, end='')
                sys.stderr.flush()
                if std_dev(res)  timing-eq-timeit-1.csv
```

Execution:

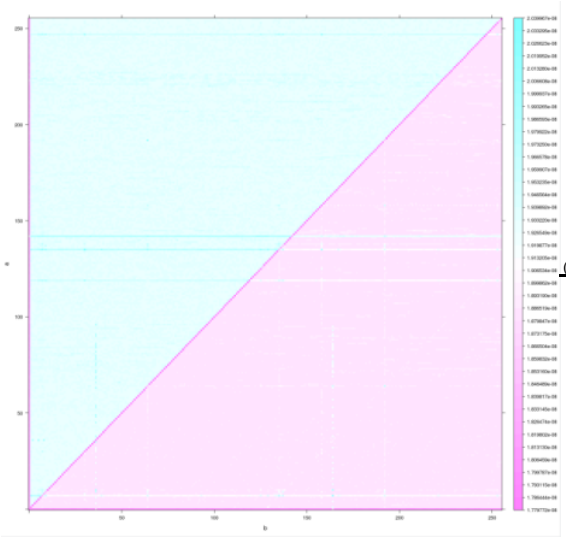
command

```
PYTHONHASHSEED=1 taskset -c 2 python3 \
-u timing-eq-timeit.py > timing-eq-timeit-1.csv
```

Code to create the graph:

R code

```
require("lattice")
a = read.csv(file="timing-eq-timeit-1.csv", header=FALSE,
             col.names=seq(1, 20), fill=TRUE)
data = as.matrix(a)
med = apply(data, 1, median, na.rm=TRUE)
# full lines
len = length(med)
columns = ceiling(length(med) / 256)
d = data.frame(x=rep(seq(0, 255), length.out=len, 256),
               y=rep(seq(0, 255), length.out=len, each=256),
               z=med)
my.at = seq(min(med), max(med), length=40)
levelplot(z~x*y, data=d, xlab="b", ylab="a",
          at=my.at, colorkey=list(at=my.at, labels=list(at=my.at)))
```



(https://securitypitfalls.files.wordpress.com/2018/08/timing-eq-timeit-1.png)

So it looks to me like the xor operation is actually one of the more constant time primitives…
Posted in Articles and tagged constant time, lucky 13, python, side channel, testing, timing attacks on 2018/08/03 by Hubert Kario. 5 Comments

5 comments

1. Alexander Kostadinov says:
   2018/08/03 at 17:45
   Wow, thank you for the highly educational post. I think original constant_time_compare has another flaw as well. It will exit early when string length does not match.

   I'm not a python dev but II'm wondering if this flaw can be removed as well the type conversion to be eliminated like this:

   def constant_time_compare(password, guess):
   len_guess = len(guess)
   len_pass = len(password)
   result = len_pass == len_guess
   generated_password = ''
   for index in range(len_guess):
   generated_password += (password if index < len_pass else guess)[index]
   for index in range(len_guess):
   result = (guess[index] == generated_password[index]) & result
   return result

   Basically timing should be based on the `guess` length always. The only thing I see that could expose password length data is that with a longer guess, we generate `generated_password` using another string. My *hope* though is that time for this would be comparable to getting from the other.
   I wonder if one can get any better without adding artificial noise.

   REPLY

   1. Hubert Kario says:
      2018/08/03 at 18:06
      > It will exit early when string length does not match.

      This is expected, and rather hard to exploit – it only leaks if the secret is as long as the attacker provided data. Given that you should always compare the attacker provided data to some hash (HMAC, scrypt, PBKDF2) the length of that secret value will be constant. Because of Kerckhoffs's principle, we must assume that the attacker knows what hash and algorithm we are using.

      > (password, guess)

      To compare attacker controlled data to a password, you need to have plaintext password stored somewhere. But I'm sure you don't have them and that was just an example… 😅

      > I wonder if one can get any better without adding artificial noise.

      adding artificial noise in general does not work: https://eprint.iacr.org/2015/1129.pdf

      REPLY

      1. Alexander Kostadinov says:
         2018/08/03 at 18:39
         Yeah, forgot we have passwords hashed and thus fixed length. In such case though we have one less thing to worry about. Now what if we avoid this data conversion with comparing characters for equality instead of xor-ing (as in my example above)? Shouldn't this eliminate the value based timing differences?

         wrt noise, I remember this doc. I was thinking more about using high precision timer to sleep such that artificially make operation always take around X milliseconds. Much preferable if language allows making sure something takes constant time. But if it truly doesn't, then what are the options? And what if next version of language introduces some optimizations that invalidate previous previous measurements for constant time? I guess you'd say automated tests 🙂

      2. Hubert Kario says:
         2018/08/03 at 19:34
         > Now what if we avoid this data conversion with comparing characters for equality instead of xor-ing (as in my example above)

         there is no "character" class in Python, and hash outputs are binary strings, which means that single "characters" (bytes, really) are integers

> Shouldn't this eliminate the value based timing differences?

I don't think so. But you have the tools, run the benchmarks 🙂

> I was thinking more about using high precision timer to sleep such that artificially make operation always take around X milliseconds.

For low level operations, the differences are measured in single cycles, not doable from python level. For high level: what is the baseline? How do you ensure it doesn't change? What if thermal throttling kicks in? What if the server is under heavier load that when we measured the baseline?

> But if it truly doesn't, then what are the options?

use language that does – most compiled languages should meet this, I wouldn't be surprised if Cython fixed it (if the compiler wasn't too clever)

but in general, minimise the critical code, delegate it to languages that operate on machine intrinsics (are not interpreted) and then test those modules

> I guess you'd say automated tests 🙂

to quote the Indiana Jones and the Last Crusade: "You have chosen… wisely"

3. Hubert Kario says:
   2018/08/04 at 11:39
   I've updated the article to include the test for the "==" operator: it's quite obviously not constant time too.