Anti-Emulation Through Time-Lock Puzzles

Tim Ebringer – The University of Melbourne[1]

Abstract

A common anti-emulation trick is to introduce loops that take a relatively long time to compute. The loop may in fact take so long to emulate that the antivirus scanner gives up.

This paper formalises this approach, using a well-known system from the cryptographic literature called time-lock puzzles. In essence, a packed binary can be quickly created by an attacker which is guaranteed to require a predefined and easily adjustable number of computationally expensive operations to rebuild a cryptographic key. This key is then used in a strong cryptographic cipher to decrypt the next stage.

Although this approach bears some similarity to the brute-force guessing of keys used by the 1998 IDEA.6155 virus, it permits a completely adjustable workload, and guarantees no shortcuts are possible.

It could pose a serious nuisance to AV emulators if such a method was included as the middle stage of a polymorphic packer. This could be mitigated by blacklisting the packer, since there is no reason why legitimate software would be packed in a way that significantly delays execution, though care would need to be taken as the "puzzle" solving code is exactly the same as RSA encryption/decryption.

Introduction

The packer has now become ubiquitous in the malware scene. An "in-the-wild" trojan that has not been horrifically post-processed into awful blobs of self-modifying assembler is nowadays a quaint curiosity. It didn't used to be like this. Software engineers used to respect and appreciate the packer as a clever piece of technology that allowed their application to fit on a single floppy disk, rather than two, thereby reducing distribution costs.

How times change. It is now relatively rare to see malware that has not had some kind of obfuscating transformation applied as a post-processing step. In the anti-virus community, software tools that perform this work are usually referred to as "packers". Malware authors have adopted such tools as a cheap way to try and avoid signature based detection in anti-virus scanners.

To combat the use of packers in malware, anti-virus vendors have constructed emulators which allow the sample to execute in a protected "sandbox", therefore hopefully permitting the unpacking

[1] The bulk of this work was performed whilst Ebringer was an employee of CA, working in the CA Labs research division.

process to proceed far enough for a signature scan to be effective. Whilst not without its drawbacks, it has so far been a relatively effective protection method against an increasing volume of packers.

Malware authors have responded to emulation with a variety of techniques. Many packers offer options to try and detect and respond to being executed in a virtualized or emulated environment. A trend that exists in some packers, particularly underground packers such as Tibs, is to expand the number of unpacking operations. Since running under emulation tends to be around an order of magnitude slower, the emulator is faced with a difficult decision: give up and potentially let a nasty file through, or continue to emulate, irritating the user and damaging the anti-virus engine's reputation for speed.

The earliest reference in the literature to a quasi time-lock approach appears to be a paper by Ször describing the IDEA.6155 virus (1). In this particular beauty, the virus attempts a brute-force decryption of one of its layers. Since a strong cryptographic cipher is not used in this stage, it is likely that a cryptanalytic attack could be mounted.

This paper describes the use and implementation of cryptographic operations to achieve a delay in execution which, with the present state-of-the-art cryptography and number theory, cannot be short-circuited. The cryptographic operation used is known as a time-lock puzzle, and was originally conceived by Rivest et. al (2).

Even in the original paper, the authors struggled to find a plausible use for it. To actually use the construction as a "time-lock" requires predicting the speed of CPUs in the future, resulting, at best, in a fuzzy release-date. This assumes that someone cares enough to want what is allegedly wrapped up in the puzzle to bother to compute the puzzle in the first place. It is not obvious that in the majority of situations, this would have a clear advantage over, say, leaving the information with a legal firm with instructions to release it on a particular date.

Although this paper proposes a practical use for time-lock puzzles, the original authors would probably be dismayed that there is still not a widespread usage that appears to be of net benefit to humanity.

Time-lock puzzles

Time-lock puzzles were first proposed by Rivest et. al. (2). Essentially the "puzzle" relies on the difficulty of computing

$$a^{2^t} \pmod{n}$$

If the factorization of n is not known, the best known method to compute this equation is to simply compute $a^2 \pmod{n}$ a total of t times. As modular squaring is not a particularly cheap operation, being $O((\lg n)^2)$ in the most common implementation, this can take a long time for even relatively small values of t (see Figure 1). Importantly, each squaring depends on the result of the previous, meaning that *it is not possible to parallelize the operation*. A multiple-core CPU, or even a massively parallel supercomputer, will not help at all.

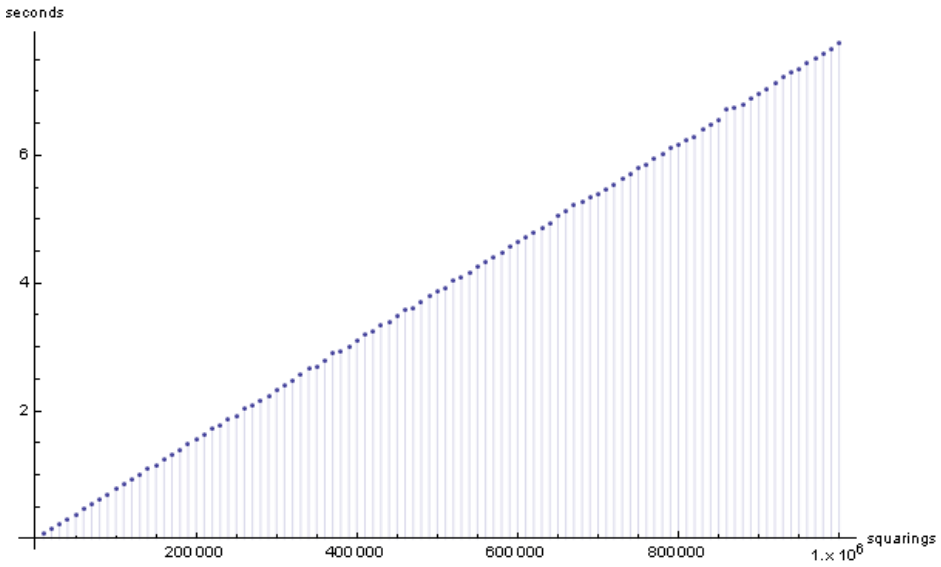


Figure 1: CPU time to compute modular squarings, 2.4GHz Intel Core Duo

If, however, the factorization of n is known, then a short-cut to solving the equation becomes available, which reduces the number of modular exponentiations required to two:

$$r = 2^t \pmod{\varphi(n)}$$

$$b = a^r \pmod{n},$$

where $b = a^{2^t} \pmod{n}$. The workload is then the same as two RSA encryptions. Note that $\varphi(n)$ denotes the Euler-Phi function, and can be computed efficiently if the factorization of n is known[2].

The shortcut is possible because of Euler's formula, which states that $a^{\varphi(n)} \equiv 1 \pmod{n}$, for any a and m with $\gcd(a, m) = 1$.

Let

$$2^t = k \cdot \varphi(n) + r$$

then:

$$a^{2^t} = a^{k\cdot\varphi(n)+r} \pmod{n}$$

$$= \left(a^{\varphi(n)}\right)^k \cdot a^r \pmod{n}$$

$$= 1^k \cdot a^r \pmod{n}$$

$$= a^r \pmod{n}$$

[2] For the interested reader, $\varphi(n)$ denotes the number of integers between 0 and n which are relatively prime to n, i.e. $\varphi(n) = \#\{a: 1 \le a \le n \text{ and } \gcd(a, n) = 1\}$. There is a simple formula for computing $\varphi(n)$ if the factorization of n is known. If $p_1, p_2, \ldots, p_r$ are the distinct primes that divide n, then $\varphi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_r}\right)$. Note that if m is the product of just two distinct primes p and q, then this formula simplifies to $\varphi(n) = (p-1)(q-1)$.

Since $r = 2^t \pmod{\varphi(n)}$, we can easily compute r, so long as $\varphi(n)$ is known.

The interested reader is referred to Rivest's original paper for a more detailed mathematical description. Python code which creates and solves time-lock puzzles is provided as an appendix.

Rivest et. al. showed how an encryption key could be wrapped in a time-lock puzzle. The original authors imagined setting the puzzle work-factor high, as a means to allow information to be released in the future. To illustrate this, they created a puzzle, termed LCS35, which they expected to take 35 years of continuous computing to solve (3).

This paper postulates that by using a greatly reduced work factor, creating puzzles that can be solved in seconds rather than the years originally proposed, an effective anti-emulation layer can be added to a packer. The emulator simply cannot afford the time or resources needed to emulate through the puzzle solving code, and no details of the payload are exposed until the puzzle is complete.

Experiments

We wanted to create a prototype that would trigger the emulation capabilities of anti-virus engines, yet we did not want to create a new packer, nor did we wish to modify existing malware in the process. It was hoped that anti-virus software might detect the EICAR test file in memory as this would enable us to write a time-lock "dropper", a program with an embedded puzzle and encrypted EICAR file which would solve the puzzle and decrypt the file to the current working directory. The experiment was to adjust the puzzle workload and determine, using a binary search, the emulation drop-out point of various anti-virus scanners.

Implementation

The implementation used to provide a proof-of-concept is a two stage build process. This permits rapid regeneration of executable binaries which contain the same payload encrypted using a different key, and a different puzzle.

Architecture

Two programs were constructed; a puzzle-generating program written in Python, and a puzzle-solving program written in portable ANSI C. The dual-architecture is because writing code in Python is blissfully easy, but on the other hand a Win32 PE file has the best chance of triggering emulation, and gives a better idea of how such a packer might be encountered in the wild. The puzzle-generating program takes as parameters a work-factor and an input file and produces a C header file. This header file contains as global variables the parameters of the puzzle and the encrypted payload defined as a byte-array.

The header then becomes part of the source code for "timelock.exe", a program that will solve the puzzle, decrypt the payload, and write the result to disk. The entire process is shown in Figure 2.

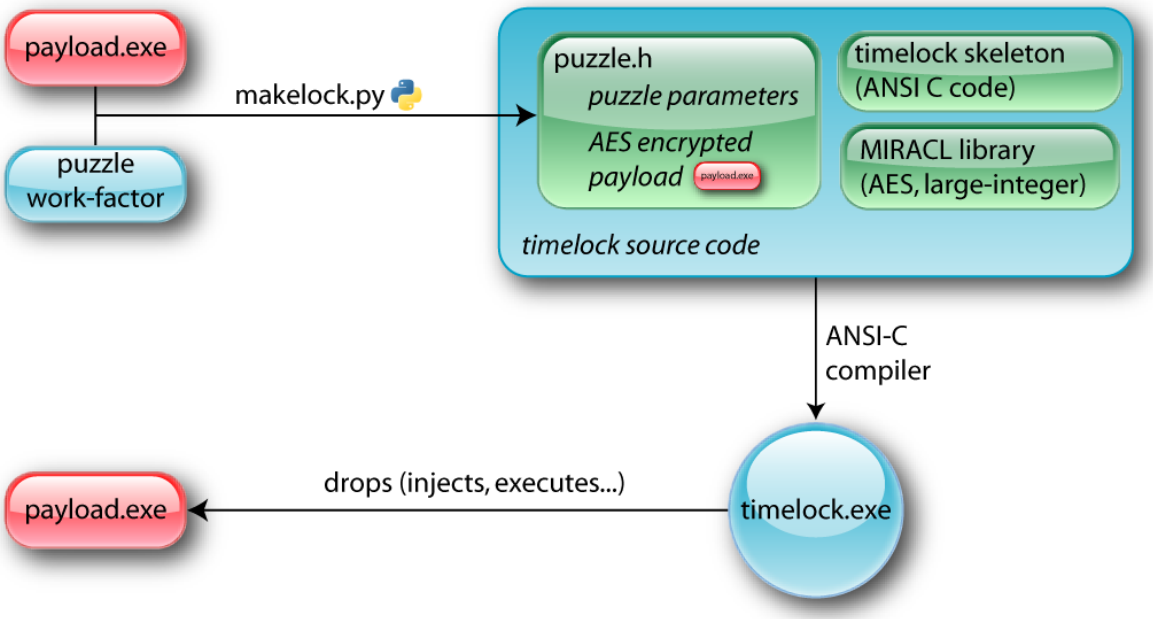


Figure 2: Time-lock puzzle generation

This architecture was selected as it permits rapid generation of "timelock.exe" binaries. The present implementation is portable, but its only effect is to write the decrypted payload to disk. Were it to be differently implemented as something nastier such as an injector, it would lose its portability.

The puzzle modulus bit-length was fixed in makelock.py at 1024-bits, which is common for RSA. However, given the low-security nature of the usage presented in this paper, it would make sense to significantly drop this. A 256-bit modulus, whilst woefully insecure for ordinary RSA, would still allow the puzzle to be solved faster than the modulus factored, and would speed up puzzle generation.

Algorithms

We present the pseudocode for the algorithms used in makelock.py and timelock.exe. For the most part, this follows the presentation of Rivest et. al., and is included for completeness.

Algorithm Genereate a time-lock puzzle wrapping a binary file

INPUT: A work factor t, and an input file f.

OUTPUT: A timelock puzzle $(n, a, c_k, t, E_k(f))$

1. Generate a 128-bit AES encryption key k.
2. Encrypt the payload f using encryption key k as $f' \leftarrow E_k(f)$. Note that in the current implementation, the payload is padded to a multiple of the cipher blocksize with zeros. This could be properly encoded as padding, then later removed, but for PE executables, this is not necessary.
3. Generate two 512-bit primes, p and q.
4. Compute $n \leftarrow p \cdot q$.
5. Compute $\varphi(n) \leftarrow (p-1) \cdot (q-1)$.
6. Randomly choose a starting value a for the puzzle, $1 < a < n$.
7. Compute $a^{2^t} \pmod{n}$ via the following steps:
   7.1. compute $r \leftarrow 2^t \pmod{\varphi(n)}$;

7.2. compute $b \leftarrow a^r \pmod{n}$.

8. Encode the key k in the puzzle solution $c_k \leftarrow b + k \pmod{n}$.
9. Discard the puzzle shortcut parameters p, q and $\varphi(n)$, as well as encryption key k.

An example of the header file produced as output by our implementation is shown in Figure 3.

```
const unsigned char PUZZLE_N[] =
"f221f7b08027ec4259c73f27d983a8c1941fd2f9d7967a71262b3289a8f1d0c705fb7d2e4ba3b603b2d753a0ee895284b872422c
9ece2b4a34562bc092f3d6376edcb1827a3d1626100f83ddd55b79a45fd856c1a9140fbfdb3c1bc157ca00d5dc8a89f3531bb4e69
1298c6e572cc4ac93bf684ae412fc2a7b838c9b830e1989";

const unsigned char PUZZLE_A[] =
"4b261a61c5ce500d3f129192fe3ca635b44d654366986322504f2d43491afb018e974e6309190cc559902ad5bb861b1c754f79fd
3354e9dae38b6b8d2d5a5afcb9ba789a44b4ca94086bed9eede527e5d2b5bd77649faacb5dbdb52a3df3bf3a25f206f86dfba028f
6b012d72dc8d6f4df3e9350b929cb9f5fb140a1364d5f71";

const unsigned char PUZZLE_CK[] =
"6a34af450ea209369e69dfd9022284fd87f03ea5654700eeb258ffb29b0d002fa9106965890a1f8d01fa0265a281037cfb8790c2
f74336c71f4f43d63c6192a3504465cd741a82a004e2221e7016faa36f5f5080935cc7ab448786e91cb075ab009289c9b480ce31a
c624fd4ff1317cffbe7795e8179131047e4543b664c8f88";

const int PUZZLE_T = 10;

unsigned char PUZZLE_EXE[] = {
  0x90, 0x0a, 0xb7, 0x98, 0x6d, 0xa2, 0x3c, 0xcf,
  0xfe, 0xa4, 0x2e, 0xdf, 0x31, 0xef, 0x76, 0xb6,
  0x0f, 0x8d, 0x77, 0xe6, 0xf3, 0x10, 0x97, 0x84,
  0xb6, 0x56, 0x8b, 0x30, 0xb3, 0xad, 0x68, 0x80,
  0x94, 0x30, 0xaf, 0xd1, 0x04, 0x24, 0x04, 0xc4,
  0xed, 0x69, 0x75, 0xcc, 0x4d, 0x00, 0xff, 0xc3,
  0x9f, 0xaf, 0x1f, 0xf0, 0xbf, 0x86, 0xcd, 0xe2,
  0x29, 0x62, 0xd5, 0x41, 0x6b, 0xfd, 0xc0, 0xfb,
  0x48, 0xce, 0x72, 0xcf, 0x19, 0x24, 0xb6, 0xee,
  0xdd, 0x82, 0x9c, 0x11, 0x42, 0xca, 0x9b, 0xa7 };

const unsigned char PUZZLE_IV[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
```

Figure 3: Example "puzzle.h" header containing encrypted EICAR test file

Algorithm Solve a time-lock puzzle

INPUT: A timelock puzzle $\left(n, a, c_k, t, E_k(f)\right)$.

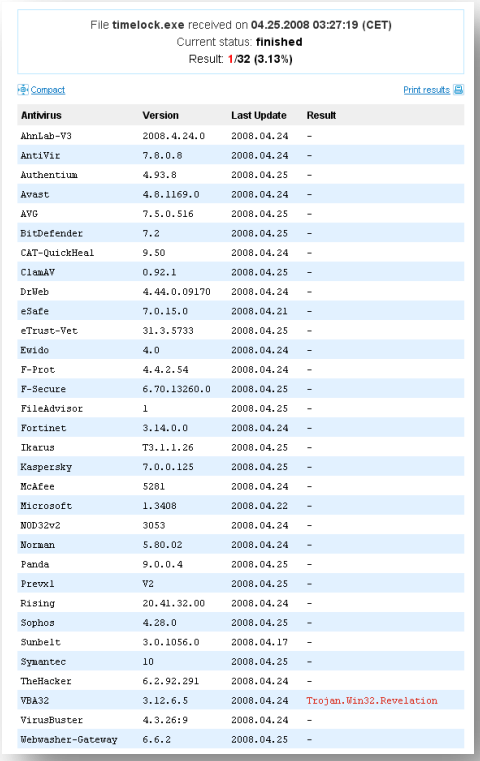
OUTPUT: The original file f.

1. $tmp \leftarrow a$
2. For i from 1 to t, do the following:
   2.1. $tmp \leftarrow tmp^2 \pmod{n}$.
3. $k \leftarrow c_k - tmp \pmod{n}$.
4. $f \leftarrow D_k\left(E_k(f)\right)$

Results

Although the EICAR test file is not designed to be used in this manner, we had hoped that AV scanners might anyway detect its presence in memory after it was decrypted. The EICAR file was not intended to be used in this manner, so it is not entirely surprising that this failed. Even for a puzzle that required only ten modular squarings, none of the scanners on VirusTotal indicated the timelock binary was suspicious (though there appeared to be one mis-detection, see Figure 4).

We reasoned that the only way to perhaps force a detection would be either to pack real malware, or to write a real packer and do some suspicious activity. We felt that we had already ethically taken this proof-of-concept as far as possible.



File **timelock.exe** received on **04.25.2008 03:27:19 (CET)**
Current status: **finished**
Result: **1/32 (3.13%)**

Compact | Print results

Antivirus	Version	Last Update	Result
AhnLab-V3	2008.4.24.0	2008.04.24	-
AntiVir	7.8.0.8	2008.04.24	-
Authentium	4.93.8	2008.04.25	-
Avast	4.8.1169.0	2008.04.24	-
AVG	7.5.0.516	2008.04.25	-
BitDefender	7.2	2008.04.25	-
CAT-QuickHeal	9.50	2008.04.24	-
ClamAV	0.92.1	2008.04.25	-
DrWeb	4.44.0.09170	2008.04.24	-
eSafe	7.0.15.0	2008.04.21	-
eTrust-Vet	31.3.5733	2008.04.25	-
Ewido	4.0	2008.04.24	-
F-Prot	4.4.2.54	2008.04.24	-
F-Secure	6.70.13260.0	2008.04.25	-
FileAdvisor	1	2008.04.25	-
Fortinet	3.14.0.0	2008.04.24	-
Ikarus	T3.1.1.26	2008.04.25	-
Kaspersky	7.0.0.125	2008.04.25	-
McAfee	5261	2008.04.24	-
Microsoft	1.3408	2008.04.22	-
NOD32v2	3053	2008.04.24	-
Norman	5.80.02	2008.04.24	-
Panda	9.0.0.4	2008.04.25	-
Prevx1	V2	2008.04.25	-
Rising	20.41.32.00	2008.04.24	-
Sophos	4.28.0	2008.04.25	-
Sunbelt	3.0.1056.0	2008.04.17	-
Symantec	10	2008.04.25	-
TheHacker	6.2.92.291	2008.04.24	-
VBA32	3.12.6.5	2008.04.24	Trojan.Win32.Revelation
VirusBuster	4.3.26:9	2008.04.24	-
Webwasher-Gateway	6.6.2	2008.04.25	-

Figure 4: VirusTotal results for a timelock puzzle that drops the EICAR test file

Mitigation

We conjecture that no anti-malware scanner would be able to afford the cycles to effectively unpack malware packed using time-lock puzzles.

We further conjecture that the main countermeasure against such a packer would be to blacklist it. This might be complicated if the time-lock packer itself had a polymorphic wrapper. Fortunately, there appears to be no incentive for commercial packers to adopt the time-lock technique, as the only effect is to delay execution, which would presumably annoy customers. The technique affords no increased protection against crackers.

The puzzle-solving code would be relatively easy to place a signature on, however, some care must be taken as the puzzle solving code is mostly just some basic mathematics, and would likely be done using a statically linked 3rd party library, as we used in our own proof-of-concept in C. It would be easy to accidentally false-alarm on some code snitched from an RSA library, such as OpenSSL.

Further work

EICAR 2.0

The effectiveness of the time-lock packer construction was difficult to test. It was hoped that using the EICAR test file would provide a means to test the emulation capabilities of various anti-virus engines without involving real malware, but this was not the case. The EICAR file was never designed to be used in such a manner, but we were unable to determine a suitable alternative.

We hope that this paper may stimulate work on a new EICAR test file, which can be used for testing the effectiveness of emulators.

Availability

In considering the potential anti-social uses of this technology, the author has opted not to make a full public release of the source code. Depending on interest, the author will make the code available to CARO members, or those who can find a CARO member to vouch for them. Please contact the author if you would like a copy.

Conclusion

We have presented and implemented what we conjecture is an effective anti-emulation technology that could be adopted by packers. This is an extension of activity that we have already seen in packers, though puts the basic idea on a rigorous theoretical footing.

The time-lock puzzle approach to packers would form an effective barrier against emulation by anti-virus scanners. There is little obfuscation benefit to such an approach, and due to the delay in execution it imposes, it is not expected that the technique will become popular in legitimate applications. If adopted by the underground community, the most effective remedy would likely be to blacklist the packer.

Acknowledgements

The author would like to thank Scott Molenkamp and Jakub Kaminski for their comments and advice on early versions of this work.

For the cryptography and large integer code in Python, we used Andrew Kuchling's excellent PyCrypto library. In C, we used the equally excellent MIRACL library from Shamus Software.

Bibliography

1. *Bad IDEA.* **Ször, Péter.** April 1998, Virus Bulletin, pp. 18-29.

2. **Rivest, R. L., Shamir, A. and Wagner, D. A.** *Time-lock Puzzles and Timed-release Crypto.* s.l. : LCS technical memo MIT/LCS/TR-684, 1996. p. 21.

3. **Rivest, Ron.** Description of the LCS35 Time Capsule Crypto-Puzzle. [Online] 4 April 1999. [Cited: 13 April 2008.] http://people.csail.mit.edu/rivest/lcs35-puzzle-description.txt.

4. *Timed-Release Cryptography.* **Mao, Wenbao.** s.l. : Springer-Verlag, 2001, Lecture Notes in Computer Science, Vol. 2259, pp. 342-357.

Appendix A – Time-lock puzzles in Python

```
#/usr/bin/python
# Requires Python Cryptography Toolkit
# http://www.amk.ca/python/code/crypto.html

from Crypto.Util import randpool
from Crypto.Util import number
import math
import sys

def makepuzzle(t):
    # Init PyCrypto RNG
    rnd = randpool.RandomPool()

    # Generate 512-bit primes
    p = number.getPrime(512, rnd.get_bytes)
    q = number.getPrime(512, rnd.get_bytes)

    n = p * q
    phi = (p - 1) * (q - 1)

    # AES key --- this is what we will encode into the puzzle solution
    key = number.getRandomNumber(128, rnd.get_bytes)

    # Need a random starting value for the puzzle, between 1 and n
    a = number.getRandomNumber(1025, rnd.get_bytes)
    a = a % n

    # *** puzzle shortcut ***
    # fast way to compute (a^2)^t (if you know phi)
    e = pow(2, t, phi)
    b = pow(a, e, n)

    # So b = (a^2)^t, and we encode the key into this solution
    ck = (key + b) % n

    return (key, (n, a, t, ck))

def solvepuzzle((n, a, t, ck)):
    tmp = a
    sys.stdout.write("Working")
    for i in range(t):
        tmp = pow(tmp, 2, n)
        if i % 10000 == 0:
            sys.stdout.write(".")
    print ""
    return (ck - tmp) % n

def main():
    # Generate a new puzzle requiting 100000 modular squarings to solve
    (key, puzzle) = makepuzzle(100000)

    # Use this key to encrypt a payload
    print "key = " + str(key)

    # Recover the key
```

```
    solution = solvepuzzle(puzzle)
    print "solution = " + str(solution)


if __name__ == "__main__":
    main()
```