Cyber Apocalypse HTB 2022 Crypto - Memory Acceleration

deuterium — 2022-05-21

Canonical HTML: https://deuterium.github.io/WriteUps/2022/cyber_apocalypse/crypto/memory_acceleration/2022-05-21-HTB-Cyber-Apcalypse-2022-Memory-Acceleration.html

Abstract

CTF writeup: HTB Cyber Apocalypse 2022's Memory Acceleration puts Z3 on a diet: brute-force part of a custom proof-of-work hash and model the smaller remainder.

Challenge Description

> Memory Acceleration While everyone was asleep, you were pushing the capabilities of your technology to the max. Night after night, you frantically tried to repair the encrypted parts of your brain, reversing custom protocols implemented by your father, wanting to pinpoint exactly what damage had been done and constantly keeping notes because of your inability of forming new memories. On one of those nights, you had a flashback. Your father had always talked about a new technology and how it would change the galaxy. You realized that he had used it on you. This technology dealt with a proof of a work function and decentralized networks. Along with Virgil's help, you had a "Eureka!" moment, but his approach, brute forcing, meant draining all your energy. Can you find a quicker way to validate new memory blocks?

Source Files

> source.py
> pofwork.py

Source Analysis

From `source.py`

```
import socketserver
import signal
from pofwork import phash


DEBUG_MSG = "DEBUG MSG - "
WELCOME_MSG = """Virgil says:
Klaus I'm connecting the serial debugger to your memory.
Please stay still. We don't want anything wrong to happen.
Ok you should be able to see debug messages now..\n\n"""
```

```
with open('memories.txt', 'r') as f:
    MEMORIES = [m.strip() for m in f.readlines()]


class Handler(socketserver.BaseRequestHandler):
    def handle(self):
        signal.alarm(0)
        main(self.request)


class ReusableTCPServer(socketserver.ForkingMixIn, socketserver.TCPServer):
    pass


def sendMessage(s, msg):
    s.send(msg.encode())


def recieveMessage(s, msg):
    sendMessage(s, msg)
    return s.recv(4096).decode().strip()


def main(s):
    block = ""
    counter = 0
    sendMessage(s, WELCOME_MSG)

    while True:
        block += MEMORIES[counter]

        sendMessage(s, DEBUG_MSG +
                    f"You need to validate this memory block: {block}\n")

        first_key = recieveMessage(s, DEBUG_MSG + "Enter first key: ")
        second_key = recieveMessage(s, DEBUG_MSG + "Enter second key: ")

        try:
            first_key, second_key = int(first_key), int(second_key)
            proof_of_work = phash(block, first_key, second_key)
        except:
            sendMessage(s, "\nVirgil says: \n"
                        "Be carefull Klaus!! You don't want to damage yourself.\n"
                        "Let's start over.")
            exit()

        if proof_of_work == 0:
            block += f" ({first_key}, {second_key}). "
            sendMessage(s, "\nVirgil says: \nWow you formed a new memory!!\n")
            counter += 1
            sendMessage(
                s, f"Let's try again {4 - counter} times just to be sure!\n\n")
        else:
            sendMessage(s, DEBUG_MSG + f"Incorect proof of work\n"
                        "\nVirgil says: \n"
                        "You calculated something wrong Klaus we need to start over.")
            exit()

        if counter == 4:
            sendMessage(s, "It seems that everything are working fine.\n"
```

```
                        "Wait what is that...\n"
                        "Klaus this is important!!\n"
                        "This can help you find your father!!\n"
                        f"{MEMORIES[-1]}")
            exit()

if __name__ == '__main__':
    socketserver.TCPServer.allow_reuse_address = True
    server = ReusableTCPServer(("0.0.0.0", 1337), Handler)
    server.serve_forever()
```

We have to provide two integers `first_key` and `second_key` such that `phash(block, first_key, second_key) == 0`. `block` will be presented by the challenge server. If we do it 4 times, we get our flag.
It's roughly like how we validate blocks with Proof-of-Work in blockchains

Lets take a look at `phash` from `pofwork.py`

```
from hashlib import md5
from Crypto.Util.number import long_to_bytes, bytes_to_long


sbox = [
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
0x54, 0xbb, 0x16
]
```

```
def rotl(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff


def sub(b):
    b = long_to_bytes(b)
    return bytes([sbox[i] for i in b])


def phash(block, key1, key2):
    block = md5(block.encode()).digest()
    block = 4 * block
    blocks = [bytes_to_long(block[i:i+4]) for i in range(0, len(block), 4)]

    m = 0xffffffff
    rv1, rv2 = 0x2423380b4d045, 0x3b30fa7ccaa83
    x, y, z, u = key1, 0x39ef52e9f30b3, 0x253ea615d0215, 0x2cd1372d21d77

    for i in range(13):
        x, y = blocks[i] ^ x, blocks[i+1] ^ y
        z, u = blocks[i+2] ^ z, blocks[i+3] ^ u
        rv1 ^= (x := (x & m) * (m + (y >> 16)) ^ rotl(z, 3))
        rv2 ^= (y := (y & m) * (m + (z >> 16)) ^ rotl(x, 3))
        rv1, rv2 = rv2, rv1
        rv1 = sub(rv1)
        rv1 = bytes_to_long(rv1)

    h = rv1 + 0x6276137d7 & m
    key2 = sub(key2)

    for i, d in enumerate(key2):
        a = (h << 1) & m
        b = (h << 3) & m
        c = (h >> 4) & m
        h ^= (a + b + c - d)
        h += h
        h &= m

    h *= u * z
    h &= m

    return h
```

Few things to note here -

1. `rotl` is 32-bit rotate left
2. `sbox` is AES sbox, so that we dont try linear/differential cryptanalysis XD
3. Every operation in `phash` can be roughly thought on working on 32 bit `uints` since each operation is preceded by `&m (0xffffffff)` which makes everything operate mod 2^{32}
4. Which means `rv1`, `rv2`, `x`, `y`, `z`, `u` are all 32bit values including our keys, i.e 'rv1 = 0x2423380b4d045 & m = 0x80b4d045
5. Insted of a block, `md5(block)` is hashed, so we have little to no control over the message and we have to actually expoit the keys.

Finding relevant `key1` and `key2` seems difficult by bare logic, dont worry since the first block provides a subtle hint

> `"You don't have to add the z3 solver to your firmware ever again. Now you can use it forever.`

We can make an SMT model in z3 and let it do its wonders!
But first let us demarcate the function so it's easier to refer

function setup

```
def phash(block, key1, key2):
    block = md5(block.encode()).digest()
    block = 4 * block
    blocks = [bytes_to_long(block[i:i+4]) for i in range(0, len(block), 4)]

    m = 0xffffffff
    rv1, rv2 = 0x2423380b4d045, 0x3b30fa7ccaa83
    x, y, z, u = key1, 0x39ef52e9f30b3, 0x253ea615d0215, 0x2cd1372d21d77
```

key1 loop

```
    for i in range(13):
        x, y = blocks[i] ^ x, blocks[i+1] ^ y
        z, u = blocks[i+2] ^ z, blocks[i+3] ^ u
        rv1 ^= (x := (x & m) * (m + (y >> 16)) ^ rotl(z, 3))
        rv2 ^= (y := (y & m) * (m + (z >> 16)) ^ rotl(x, 3))
        rv1, rv2 = rv2, rv1
        rv1 = sub(rv1)
        rv1 = bytes_to_long(rv1)
```

key2 substitution

```
    h = rv1 + 0x6276137d7 & m
    key2 = sub(key2)
```

key2 loop

```
    for i, d in enumerate(key2):
        a = (h << 1) & m
        b = (h << 3) & m
        c = (h >> 4) & m
        h ^= (a + b + c - d)
        h += h
        h &= m
```

final multiplication

```
    h *= u * z
    h &= m
```

```
    return h
```

Enter Z3

Since we are dealing with 32 bits entities only, we will use theory of bitvectors. Where each variable is simply considered a collection of bits and all the operations are treated as symbolic computation upon those sets of bits. Pretty much like a hardware circuit, where each component is say a 32 bit register.

Representing function setup

```
block = md5(message).digest()
block = 4*block
blocks = [int.from_bytes(block[i:i+4],'big') for i in range(0, len(block), 4)]
# we will let the blocks be the way they are in the real function
# or we could declare them as 32-bit constants, either suffices
# blocks = [BitVecVal(i,32) for i in blocks] is treated same as above
rv1, rv2 = BitVecVal(0x2423380b4d045,32), BitVecVal(0x3b30fa7ccaa83,32)
# note that they will automatically be truncated to 32 bits
key1 = BitVec('key1', 32)
key2 = BitVec('key2', 32)
# key1 and key2 treated as 32-bit unknowns
m = 0xffffffff #can be written as -1 as well ;)
x, y, z, u = key1, *[BitVecVal(i,32) for i in (0x39ef52e9f30b3, 0x253ea615d0215,
0x2cd1372d21d77)]
# bitvecs can be used almost like python variables! I love z3 API
```

Representing key1 loop

```
for i in range(13):
        x,y = blocks[i] ^ x, blocks[i+1] ^ y
    # easy interop with xor
        z,u = blocks[i+2] ^ z, blocks[i+3] ^ u
        x,y,z,u = [simplify(i) for i in (x,y,z,u)]
    # simplify tries to evaluate the current symbolic computation of a variable
    # and tries to simplify as much as possible (no effect on truth, can skip)
        x = x*(m + LShR(y,16)) ^ RotateLeft(z, 3)
    # expanding the walrus (:=)
    # note that >> is replaced with LShR, this is because in theory of bitvecs
    # there are two kinds of shift rights, arithmetic and logical. logical
    # shift right shifts as is whereas arithmetic shift right also retains the
    # original sign. Python ints are infinite, so >> means logical shift by
    # shift by default but in z3 >> is arithmetic while LShR is logical
        rv1 ^= x
        y = y*(m + LShR(z,16)) ^ RotateLeft(x, 3)
        rv2 ^= y
        rv1, rv2 = rv2, rv1
        rv1 = sub(rv1)
```

Wait, how will `sub` work?
Good question, it wont the `sub` is a previously defined python function which expects python `int` to index the `SBOX` and return a value. It wont understand `BitVec` as index and so wont our model understand our function!

Theory of Arrays

SMT solvers are so mature, we can use multiple theories to create and solve a model!
With theory of arrays, we can essentially declare an array with arbitrary index and arbitrary stored value.

```
SBOX = Array('SBOX', BitVecSort(8), BitVecSort(8))
def sub(bitvec32):
    vec_bytes = [Extract(8*i+7, 8*i, bitvec32) for i in range(3,-1,-1)]
    # logical analogue of 32-bit int to 4-bytes in big-endian order
    # Extract(hi,lo,bitvec) takes the bits [lo,hi) and creates a new
    # bitvector of size hi-lo+1
    return Concat([SBOX[i] for i in vec_bytes])
    # i is index BitVec(8) to Array SBOX and SBOX[i] is BitVec(8) stored
    # Concat concatanates the 4 8-bitvectors to an 32-bitvector like it
    # is done in after the original sub in the key1 loop
```

Representing key2 substitution

```
h = simplify(rv1 + 0x6276137d7)
subkey2 = [Extract(8*i+7,8*i,key2) for i in range(3,-1,-1)]
# splitting to 8-bits again
subkey2 = [SBOX[i] for i in subkey2]
# note that we need to mention SBOX only once in the whole logic
subkey2 = [ZeroExt(24,i) for i in subkey2]
```

What is ZeroExt? Note that subkey2 after substitution is a list of 8-bit entities. Now this one wouldn't look much severe to a programmer since all programming languages dont bother much about adding two integer values c/c++ would give type warning but just add the smaller value to a bigger value without ranting. whereas python doesn't bother at all. But if we think like a hardware, you will be bothered when presented to add a 32-bit register to a 8-bit register. Since we require this value later, we make it a 32 bit value by Extending with 24 zeros in the front (if we were not dealing with uint32 we would have sign-extended these 8-bit values.

Representing key2 loop

```
for i,d in enumerate(subkey2):
        a = (h<<1)
        b = (h<<3)
        c = LShR(h,4)
        # note the LShR again, blindly missing an operator can cost you hours :)
```

```
        h ^= (a+b+c-d)
        h += h
h ^= u*z
```

Now after all this bizarre symbolic computation, we are not done yet, we are not here just to model but to ask the solver to find the values of `key1` and `key2` such that this symbolic function evaluates to 0

Calling a solver

```
solver = Solver()
solver.add(h==0) #the final h we have here should be 0
for i,v in enumerate(sbox): # the original AES sbox
    solver.add(SBOX[i]==v)
# specifing the exact substitution box
if solver.check() == sat:
    m = solver.model()
    # a desirable model
    return (m[key1].as_long(), m[key2].as_long())
# as_long converts bitvecs to python ints
```

Putting it all together

```
def hack_proof_of_work(message):
        block = md5(message).digest()
        block = 4*block
        blocks = [int.from_bytes(block[i:i+4],'big') for i in range(0, len(block), 4)]
        rv1, rv2 = BitVecVal(0x2423380b4d045,32), BitVecVal(0x3b30fa7ccaa83,32)
        key1 = BitVec('key1', 32)
        key2 = BitVec('key2', 32)
        m = 0xffffffff
        x, y, z, u = key1, *[BitVecVal(i,32) for i in (0x39ef52e9f30b3,
0x253ea615d0215, 0x2cd1372d21d77)]
        for i in range(13):
                x,y = blocks[i] ^ x, blocks[i+1] ^ y
                z,u = blocks[i+2] ^ z, blocks[i+3] ^ u
                x,y,z,u = [simplify(i) for i in (x,y,z,u)]
                x = x*(m + LShR(y,16)) ^ RotateLeft(z, 3)
                rv1 ^= x
                y = y*(m + LShR(z,16)) ^ RotateLeft(x, 3)
                rv2 ^= y
                rv1, rv2 = rv2, rv1
                rv1 = sub(rv1)

        SBOX = Array('SBOX', BitVecSort(8), BitVecSort(8))
        def sub(bitvec32):
                vec_bytes = [Extract(8*i+7, 8*i, bitvec32) for i in range(3,-1,-1)]
                return Concat([SBOX[i] for i in vec_bytes])

        h = simplify(rv1 + 0x6276137d7)
        subkey2 = [Extract(8*i+7,8*i,key2) for i in range(3,-1,-1)]
        subkey2 = [SBOX[i] for i in subkey2]
        subkey2 = [ZeroExt(24,i) for i in subkey2]

        for i,d in enumerate(subkey2):
```

```
                a = (h<<1)
                b = (h<<3)
                c = LShR(h,4)
                h ^= (a+b+c-d)
                h += h
        h ^= u*z
        solver = Solver()
        solver.add(h==0)
        for i,v in enumerate(sbox):
                solver.add(SBOX[i]==v)
        if solver.check() == sat:
                m = solver.model()
                return (m[key1].as_long(), m[key2].as_long())
```

Lets go!

```
message_one = "You don't have to add the z3 solver to your firmware ever \
               again. Now you can use it forever."
key1, key2 = hack_proof_of_work(message_one)
```

Two hours later (external animation).

Well, no key yet?
I know, lets discuss a few problems and workarounds

Too complicated model

1. Too many multiplications. There are 13 loops and a lot of multiplications. And as one may know, factoring has never been easy.
2. We dont even have a tentative time by which the solver will spew a satisfying model. This is the general drawback of SMT/SAT solvers.
3. Not breaking the problem as (an actually intelligent) human

So lets analyze the problem carefully part by part.

Re-analysis

1. The final value is `h(final) = h(part 2)*u*z (part 1)` and we need it to be 0 final `h` will be 0 if sum of least significant `0` of `h`, `u` and `z` exceeds
2. as overflows are ignored in 32-bit multiplication.
3. What if we can get `h` of key2 loop to 0 by its own?

Reversing only key2 loop

```
def hack_only_key2(h,nbits=0):
    # note its post substitution for less complexity and speed
    # nbits is the number of nonzero most significant bits we can tolerate
    h = BitVecVal(h,32)
    key2 = BitVec('key2',32)
    subkey2 = [Extract(8*i+7,8*i,key2) for i in range(3,-1,-1)]
```

```
    subkey2 = [ZeroExt(24,i) for i in subkey2]
    for i,d in enumerate(subkey2):
        a = (h<<1)
        b = (h<<3)
        c = LShR(h,4)
        h ^= (a+b+c-d)
        h += h
    solver = Solver()
    # solver.add(Extract(31-nbits,0,h)==0)
    solver.add(h==0)
    if solver.check() == sat:
        m = solver.model()
        return m[key2].as_long()

print(try_only_key2(1337,0))
# None
print(try_only_key2(1337,1))
# None
print(try_only_key2(1337,2))
# None
print(try_only_key2(1337,4))
# None
print(try_only_key2(1337,8))
# 1311637496
print(try_only_key2(1,0))
# 75586596
print(try_only_key2(2,0))
# 276293996
print(try_only_key2(3,0))
# 283836416
```

It seems to be working but only for a limited number of values, lets see how frequently can it work independent of `h` from first loop.

```
from tqdm import tqdm
from random import randint
num_samples = 4096
validkey2 = [try_only_key2(randint(0,2**32,0)) for i in tqdm(range(num_samples))]
print("Number of suitable h", num_samples-validkey2.count(None))
# 5
validkey2 = [try_only_key2(randint(0,2**32,1)) for i in tqdm(range(num_samples))]
print("Number of suitable h", num_samples-validkey2.count(None))
# 11
validkey2 = [try_only_key2(randint(0,2**32,4)) for i in tqdm(range(num_samples))]
print("Number of suitable h", num_samples-validkey2.count(None))
# 72
```

It appears that if we entirely ignore `key1`, and let `h` be whatever it desires to be i.e. random, we can have our luck with finding `key2` with roughly 1 in 400 chance (ignoring the zeros for `u*z` entirely)

So we can bruteforce for `key1`, try solving for `key2` and this should take a couple of seconds and lo we are done.

Solve Script

```
def zerocount(num):
    count = 0
    while num&1==0:
        count+=1
        num>>=1
    return count

def form_blocks(block):
    """making bruteforce faster by removing recomputation of md5"""
    block = md5(block.encode()).digest()
    block = 4 * block
    blocks = [bytes_to_long(block[i:i+4]) for i in range(0, len(block), 4)]
    return blocks


def phashk1(blocks, key1):
    """hash state till key1 part and before key2 substitution"""
    m = 0xffffffff
    rv1, rv2 = 0x2423380b4d045, 0x3b30fa7ccaa83
    x, y, z, u = key1, 0x39ef52e9f30b3, 0x253ea615d0215, 0x2cd1372d21d77

    for i in range(13):
        x, y = blocks[i] ^ x, blocks[i+1] ^ y
        z, u = blocks[i+2] ^ z, blocks[i+3] ^ u
        rv1 ^= (x := (x & m) * (m + (y >> 16)) ^ rotl(z, 3))
        rv2 ^= (y := (y & m) * (m + (z >> 16)) ^ rotl(x, 3))
        rv1, rv2 = rv2, rv1
        rv1 = sub(rv1)
        rv1 = bytes_to_long(rv1)
    h = rv1 + 0x6276137d7 & m
    # also return the number of zeros in u*z so our model finds it easier
    return h,zerocount(u*z)

def desubstitute(key2):
    """reverse the substitution on the found key2"""
    # we could have added the substitution to the model too, but since we
    # are bruteforcing, we appreciate a bit of extra speed
    return bytes([sbox.index(i) for i in int.to_bytes(key2,4,'big')])

def bruteforce_key1(block):
    blocks = form_blocks(block)
    for key1 in tqdm(range(2**32),total=-1):
        h, nbits = phashk1(blocks, key1)
        key2 = try_only_key2(h, nbits)
        if key2:
            return key1, desubstitute(key2)
```

Final Test

```
message_one = "You don't have to add the z3 solver to your firmware ever \
               again. Now you can use it forever."
key1, key2 = bruteforce_key1(message_one)
print(f"{key1=} {key2=}")
assert phash(message_one, key1, key2) == 0
```

```
# 909it [00:16, 54.92it/s]
# key1=909 key2=3711505522
```

Post solve wanderings

I solved the challenge manually by prompting bruteforce 4 times. I wanted to create a netcat script, but couldn't as Hack The Box terminated all instances
post the CTFs so I cant access the server.

I wonder why is there an uncanny resemblance between the hash function and a hash collision challenge I created last year for zh3r0 CTF v2

```
text = [int.from_bytes(text[i:i+4],'big') for i in range(0,len(text),4)]
M = 0xffff
x,y,z,u = 0x0124fdce, 0x89ab57ea, 0xba89370a, 0xfedc45ef
A,B,C,D = 0x401ab257, 0xb7cd34e1, 0x76b3a27c, 0xf13c3adf
RV1,RV2,RV3,RV4 = 0xe12f23cd, 0xc5ab6789, 0xf1234567, 0x9a8bc7ef
for i in range(0,len(text),4):
        X,Y,Z,U = text[i]^x,text[i+1]^y,text[i+2]^z,text[i+3]^u
        RV1 ^= (x := (X&0xffff)*(M - (Y>>16)) ^ ROTL(Z,1) ^ ROTR(U,1) ^ A)
        RV2 ^= (y := (Y&0xffff)*(M - (Z>>16)) ^ ROTL(U,2) ^ ROTR(X,2) ^ B)
        RV3 ^= (z := (Z&0xffff)*(M - (U>>16)) ^ ROTL(X,3) ^ ROTR(Y,3) ^ C)
        RV4 ^= (u := (U&0xffff)*(M - (X>>16)) ^ ROTL(Y,4) ^ ROTR(Z,4) ^ D)
```

Anyways, it was a fun challenge, I had a lot of fun and hope that some weird soul had its fun too reading this writeup :)

> Interactive controls are omitted from this PDF. See the canonical HTML version.