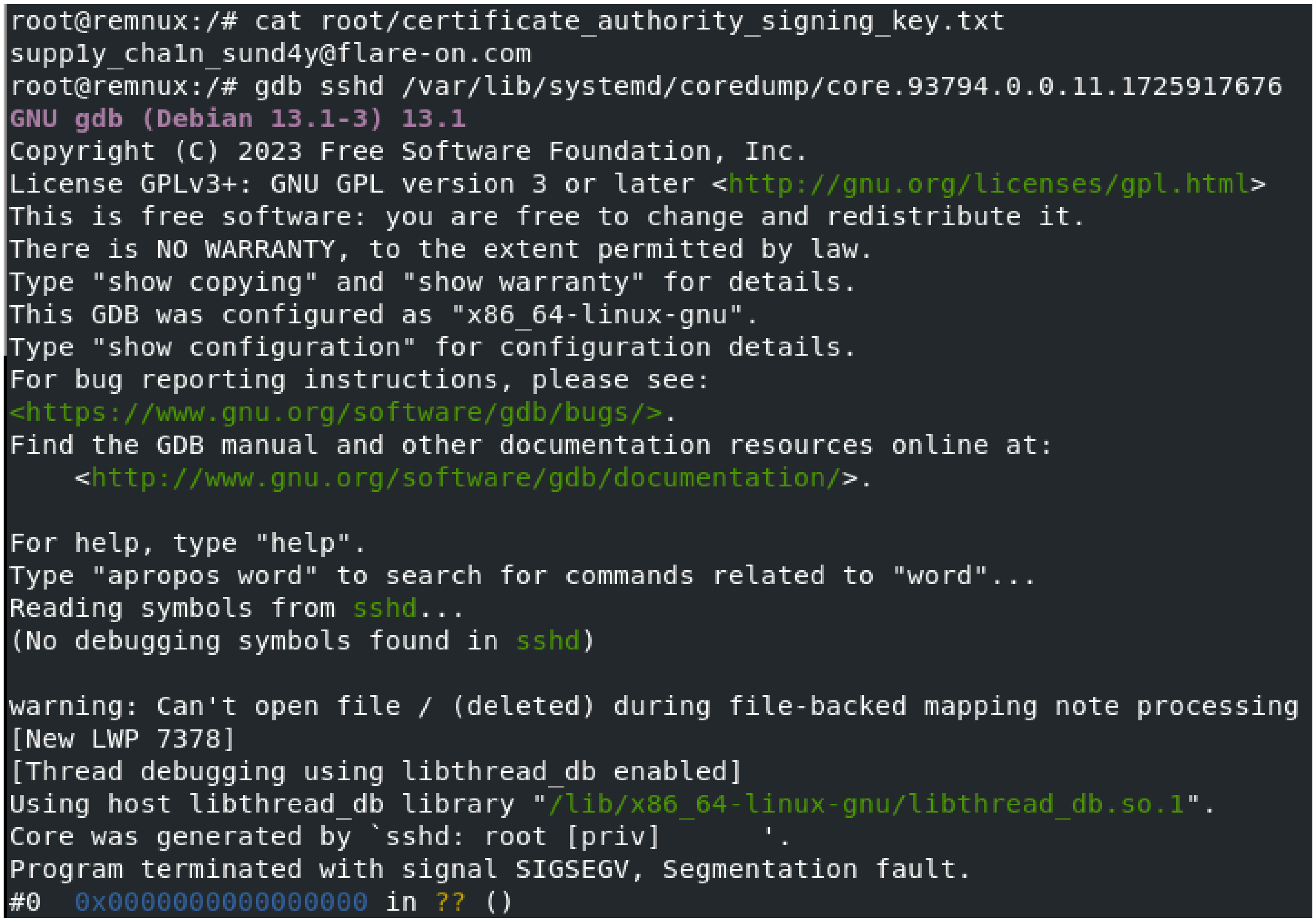
Flare-On 11 Challenge 5: sshd

By Christopher Gardner (@t00manybananas)

Overview

`sshd` is distributed as a tar archive that is clearly a dump of an entire Linux filesystem (specifically a Docker container as evidenced by the presence of the `.dockerenv` file but that's irrelevant to the challenge). As the challenge description mentions that the ssh server has crashed, it's logical to begin looking for coredumps on the system. There's one at `/var/lib/systemd/coredump/sshd.core.93794.0.0.11.1725917676`, which is the default folder for SSHD coredumps on Linux. To examine this coredump, it's best to use `gdb`, which is thankfully included in the container dump. We can chroot into the container dump and then debug the coredump:

```
gdb sshd /var/lib/systemd/coredump/core.93794.0.0.11.1725917676
```

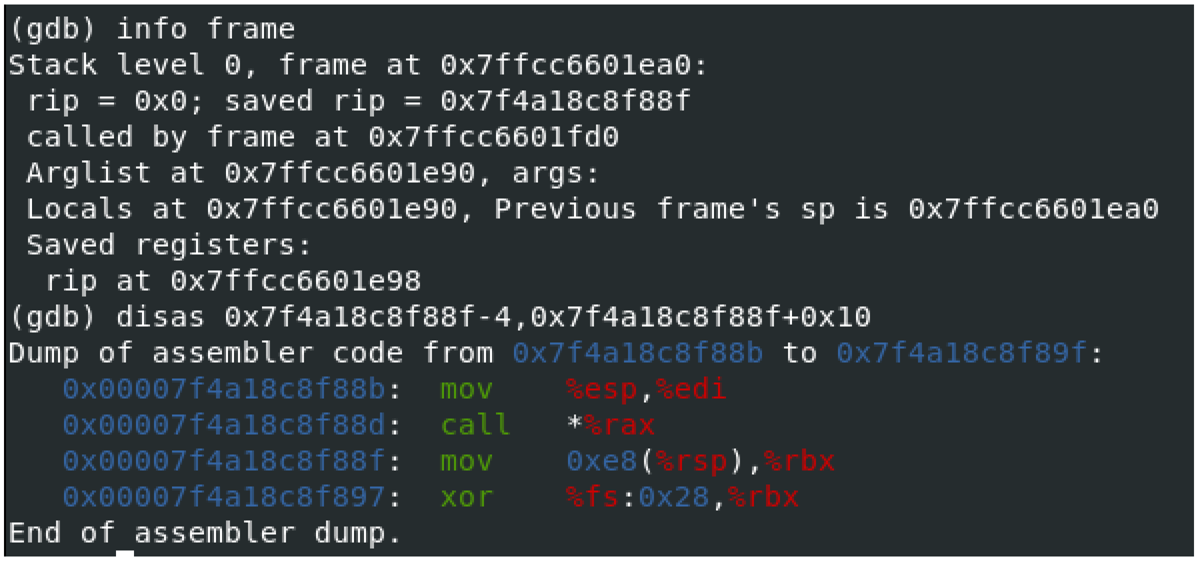


```
root@remnux:/# cat root/certificate_authority_signing_key.txt
supp1y_cha1n_sund4y@flare-on.com
root@remnux:/# gdb sshd /var/lib/systemd/coredump/core.93794.0.0.11.1725917676
GNU gdb (Debian 13.1-3) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sshd...
(No debugging symbols found in sshd)

warning: Can't open file / (deleted) during file-backed mapping note processing
[New LWP 7378]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Core was generated by `sshd: root [priv]      '.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000000000000000 in ?? ()
```

Figure 1: Debugging the coredump

Note that it is possible to examine the coredump outside of a `chroot`, but this will lead to incorrect memory addresses and make the challenge more difficult (this is because GDB will load shared library segments from the libraries present on disk, as the coredump does not actually contain the contents of the libraries).

Apparently the program has crashed by jumping to a null pointer, which most commonly happens after executing a `call` or `jmp` to a NULL function pointer. Indeed, examining `info frame` shows that the saved `rip` is `0x7f4a18c8f88f`, and that the instruction right before it is a call to `rax` (which is NULL).



```
(gdb) info frame
Stack level 0, frame at 0x7ffcc6601ea0:
 rip = 0x0; saved rip = 0x7f4a18c8f88f
 called by frame at 0x7ffcc6601fd0
 Arglist at 0x7ffcc6601e90, args:
 Locals at 0x7ffcc6601e90, Previous frame's sp is 0x7ffcc6601ea0
 Saved registers:
  rip at 0x7ffcc6601e98
(gdb) disas 0x7f4a18c8f88f-4,0x7f4a18c8f88f+0x10
Dump of assembler code from 0x7f4a18c8f88b to 0x7f4a18c8f89f:
   0x00007f4a18c8f88b:  mov    %esp,%edi
   0x00007f4a18c8f88d:  call   *%rax
   0x00007f4a18c8f88f:  mov    0xe8(%rsp),%rbx
   0x00007f4a18c8f897:  xor    %fs:0x28,%rbx
End of assembler dump.
```

Figure 2: `info frame` dump and disassembly at `rip`

From here, the next goal is to find out what module `0x7f4a18c8f88f` belongs to. GDB makes this very annoying to do, but scrolling through `info files` shows that this address appears in the `.text` section of `/lib/x86_64-linux-gnu/liblzma.so.5`. We can then look through `info proc mappings` to determine that the base address of this file is `0x7f4a18c86000` (interestingly, the name of this is `(deleted)` in `info proc mappings`). We can then grab this file from the archive and load it into IDA.

Part 2: The Library

Once we rebase the program to the base address in memory we can begin statically analyzing the crash in IDA. Our crashing instruction is contained within the function `sub_7F4A18C8F820`, and the crash is due to the program attempting to resolve and call the symbol `RSA_public_decrypt` (with a space at the end), which is invalid. Only sub_7F4A18C8ADD0 references this function indirectly. This function calls `sub_7F4A18C8F1B0` with the arguments `RSA_public_decrypt` and the address of our crashing function. It's easy to guess here that this function is hooking `RSA_public_decrypt`, and examining `sub_7F4A18C8F1B0` yields some strings that lead us to the open source plthook library, available at https://github.com/kubo/plthook.

The hook function is relatively simple, it first checks that the current process is running as root, and that the first integer of the second argument is equal to `0xC5407A48`. If that succeeds it uses parts of the second argument as the ChaCha20 key (a2 + 4) and nonce (a2 + 36) to decrypt a blob, and then executes that blob as shellcode.

For this part of the challenge, it's quite easy to extract the key and nonce from the coredump since the address of the second argument is stored in `rbp` and left intact at the time of the crash. This provides the key 943df638a81813e2de6318a507f9a0ba2dbb8a7ba63666d08d11a65ec914d66f and nonce f236839f4dcd711a52862955 for us to decrypt the shellcode offline.

Part 3: The Shellcode

If we analyze the shellcode in IDA, we can see that it is quite simple. It establishes a connection to `10.0.2.15:1337`, receives a key, nonce, and filename, then encrypts the contents of that file and sends that to the server. The encryption algorithm here is extremely similar to ChaCha20, the only difference is that the constant used to initialize the cipher state is changed from `expand 32-byte k` to `expand 32-byte K` (note the capitalized K at the end). Possible strategies to deal with this include emulation or just having a sharp eye.

From here, all that's left is to extract the ciphertext, key, and nonce, and then decrypt them. It's possible to manually calculate all the offsets from the value of `rsp` at the time of the crash, but the easiest way is to find something that looks like a filename (in this case `/root/certificate_authority_signing_key.txt`) and calculate relative offsets from that. Extracting these offsets gives us the following data:

ChaCha20 key: `8d ec 91 12 eb 76 0e da 7c 7d 87 a4 43 27 1c 35 d9 e0 cb 87 89 93 b4 d9 04 ae f9 34 fa 21 66 d7`

ChaCha20 nonce: `11 11 11 11 11 11 11 11 11 11 11 11`

Ciphertext: `A9 F6 34 08 42 2A 9E 1C 0C 03 A8 08 94 70 BB 8D AA DC 6D 7B 24 FF 7F 24 7C DA 83 9E 92 F7 07 1D 02 63 90 2E C1 58`

Decrypting this with the custom ChaCha20 implementation gives us the final flag.

Final Flag

```
Unset
supp1y_cha1n_sund4y@flare-on.com
```