ENGR-E 399/599: Embedded systems reverse engineering
Indiana University — Spring 2022
Prepared by: Austin Roach — Assigned: February 3, 2022

Assignment 2: Services on an embedded Linux system

Due date: March 4, 2022

For this assignment, you will analyze system services for an embedded Linux system. You will extract and analyze artifacts from compressed filesystems, and will interact dynamically with emulated versions of these systems.

Below is some background information that you may find useful.

0.1 File system analysis

The file systems that you will be provided are squashfs filesystems. Squashfs is a compressed, read-only filesystem. On Ubuntu, the `squashfs-tools` package provides `unsquashfs`, a program that can extract the content of squashfs filesystems, and `mksquashfs`, a program that can pack a directory structure into a squashfs filesystem.

0.2 Emulation

You will interact with the target services on an emulated system. To do this, you will use QEMU's full system emulation mode. The QEMU ARM system emulator is provided by the `qemu-system-arm` package on Ubuntu. There are a few pieces required to make this emulated system work:

- A kernel, compiled for the target instruction set architecture, that supports the emulated peripherals of a specified QEMU system. This is provided for you in the `resources` directory by `linux-5.10.4.zImage`.
- A network connection to the emulated system. The `launch-emulated-system.sh` script in the resources directory establishes and configures a TAP interface to create a network connection to the emulated system.
- A disk image. This is provided to you for each exercise. You could choose to modify it if doing so would help your analysis.

0.2.1 Launching the emulated environment

To launch an emulated system, you can simply navigate to the resources directory and run

```
sudo ./launch-emulated-system.sh path/to/disk-image.squashfs
```

The script expects to find the kernel in the current working directory. You can also specify a different path to the kernel by setting the `KERNEL` environment variable.

When the system starts, you will find that you current terminal window is used to connect to a virtualized UART interface. QEMU will also launch a second window that can be used to control the emulated system. This interface provides a lot of functionality, but for this assignment the only functionality that you are likely to need is the ability to shut down the emulated system. You can do this by typing `quit` in the QEMU window.

0.3 Network interactions

For each part of this assignment you will be interacting with a service provided over a TCP socket. For some services, you might be able to use an existing client program to interact with that service. (For example, the SSH backdoor service.) For others, there is no existing client. You will need to implement your own client using information that you recover through your analysis of the service. Almost every programming language has some library functionality for reading from and writing to TCP sockets. Some useful tools for interacting with TCP sockets are described below.

The emulated system has a hard-coded IP address of 169.254.15.2. The IP address for the TAP interface on your machine will be set to 169.254.15.1 by the `launch-emulated-system.sh` script. Each emulated system includes a web server to demonstrate correct network functionality. You can access it by navigating to `http://169.254.15.2` in a web browser on the system that is hosting the emulated embedded system.

0.3.1 netcat

`netcat` is a very versatile tool for communicating over sockets. If can be used to pipe information from stdin to a network socket, and from the network socket to stdout. For example, the following invocation would send the string "Hello, world!\n" to IP address 169.254.15.2 on TCP port 12345:

```
$ echo "Hello, world!" | nc 169.254.15.2 12345
```

Running `nc` by itself in your terminal will produce an interactive session, forwarding whatever you type into stdin over the network connection, and displaying received results:

```
$ nc 169.254.15.2 12345
```

`netcat` can be very useful when you're initially probing a network service, but it doesn't lend itself well to programmatic interactions.

0.3.2 Python socket module

The Python `socket` module[1] provides a relatively easy interface to interact programmatically over network sockets. To send the string "Hello, world!\n" to port 12345 and receive a response, you might execute:

[1] `https://docs.python.org/3/library/socket.html`

```
In [1]: import socket

In [2]: sock = socket.socket()

In [3]: sock.connect(('169.254.15.2', 12345))

In [4]: sock.sendall(b"Hello, world!\n")
Out[4]: 13

In [5]: print(sock.recv(1024))
```

Note that the `send()` and `recv()` methods use bytes-type objects (for example bytestrings or bytearrays), rather than strings.

0.4 Debugging

0.4.1 QEMU GDB stub

QEMU provides a GDB stub for system-level debugging. This is enabled by `-s` flag that is provided to `qemu-system-arm` in `launch-emulated-system.sh`. This starts the GDB server on TCP port 1234.

After installing the `gdb-multiarch` package in Ubuntu, you can connect with:

```
$ gdb-multiarch
(gdb) target remote localhost:1234
```

Keep in mind that this is a system-level debugger. You can set breakpoints at the virtual memory addresses that you wish to examine, but this debugger is unaware of process-level separation on the emulated system. Your breakpoint might trip when a different process than you expect executes an instruction at the virtual address that you choose.

0.4.2 gdbserver

If you have command-line access to the system and can load your own programs onto the system, it may be more convenient to use `gdbserver`. A statically linked `gdbserver` for 32-bit ARM is provided in the `resources` directory.

You can connect the GDB server to a running process using the `--attach` command line argument, or you can start a binary under `gdbserver`'s control. `gdbserver` will request that you specify an IP address and port number to bind to. Running

```
$ gdbserver --attach 0.0.0.0:12345 126
```

for example would attach to PID 126, providing the server on all network interfaces on TCP port 12345. You could then attach to this GDB server using `gdb` on your host machine:

```
$ gdb-multiarch
(gdb) target remote 169.254.15.2:12345
```

1 Simple admin server

The disk image `simple-admin-rootfs.squashfs` includes a service that provides a simple administrative interface to the system using human-readable commands. Such administrative interfaces are rather common in embedded systems, and often provide very powerful control mechanisms.

1.1 Reconnaissance: 1 point

Using whatever means you prefer, such as unpacking the filesystem, establishing access to a terminal on the system, or scanning the network interface with `nmap`, answer the following the questions:

- What TCP port does the administrative interface listen on?
- What files on the disk image are relevant to reverse engineering this service? Include any relevant init scripts, binaries, or shared libraries.

1.2 Functionality: 2 points

Describe the commands that the administrative interface accepts. What does the server do upon accepting each command?

1.3 Demonstration: 2 points

For each command that you found, give an example of how you would provide that command over the network to the server. One of these commands should make it possible for you to access a privileged remote shell over the network. Include an example that demonstrates the process for doing this.

2 Binary admin server

The disk image `binary-admin-rootfs.squashfs` includes a service providing an administrative interface. In contrast to the simple admin server, this server uses a custom binary format to encode the commands and their arguments.

There are some elements that you might expect in a binary message protocol. Those include:

- Some pattern indicating the start of a new message
- A message number or identifier
- A length for the message

The data carried by the mesage could be encoded in a number of ways. One scheme that is frequently used is type-length-value encoding.[2]

2.1 Reconnaissance: 1 point

Using whatever means you prefer, such as unpacking the filesystem, establishing access to a terminal on the system, or scanning the network interface with `nmap`, answer the following the questions:

- What TCP port does the administrative interface listen on?
- What files on the disk image are relevant to reverse engineering this service? Include any relevant init scripts, binaries, or shared libraries.

2.2 Functionality: 1 point

Describe the commands that the administrative interface accepts. What arguments does each command take? What does the server do upon accepting each command?

2.3 Command format: 1 point

Describe the format of a command. Label fields as best you can, and describe the range of valid values that could occur within each field.

2.4 Demonstration: 2 points

For each command that you found, give an example of how you would provide that command over the network to the server. One of these commands should make it possible for you to access a privileged remote shell over the network. Include an example that demonstrates the process for doing this.

[2] https://en.wikipedia.org/wiki/Type-length-value

Once you understand the protocol, you may find it easiest to generate messages for the server using a script or program. If so, provide the source code for your script or program with your assignment submission.

3 Authenticated binary admin server

The disk image `binary-admin-auth-rootfs.squashfs` includes a service providing an administrative interface. This uses the same binary protocol as the binary admin server, but this server incorporates a challenge-response authentication system.

3.1 Reconnaissance: 1 point

Using whatever means you prefer, such as unpacking the filesystem, establishing access to a terminal on the system, or scanning the network interface with `nmap`, answer the following the questions:

- What TCP port does the administrative interface listen on?
- What files on the disk image are relevant to reverse engineering this service? Include any relevant init scripts, binaries, or shared libraries.

3.2 Functionality: 1 point

Describe the commands that the administrative interface accepts. What arguments does each command take? What does the server do upon accepting each command? Which of these commands require authentication?

3.3 Authentication: 1 point

Describe the authentication mechanism. How are challenges generated? What must the relationship be between a challenge and the response in order to authenticate to the server?

3.4 Demonstration: 2 points

For each command that you found, give an example of how you would provide that command over the network to the server. Executing authenticated commands will require some back-and-forth communication with the server. Implement the process of authenticating and issuing commands in a script or program to automate the interaction. Provide the source code for the script or program with your assignment submission.

One of these commands should make it possible for you to access a privileged remote shell over the network. Include an example that demonstrates the process for doing this.

4 SSH backdoor

The disk image `dropbear-backdoor-rootfs.squashfs` includes an SSH server that has been modified to include a backdoor. Even if passwords for accounts on the system are changed, someone with knowledge of the backdoor can use it to gain privileged access to the system.

4.1 Reconnaissance: 1 point

Using whatever means you prefer, such as unpacking the filesystem, establishing access to a terminal on the system, or scanning the network interface with `nmap`, answer the following the questions:

- What TCP port does the backdoored SSH service listen on?
- What files on the disk image are relevant to reverse engineering this service? Include any relevant init scripts, binaries, or shared libraries.

4.2 Narrowing the scope: 1 point

Find the function in the binary that includes the backdoor functionality. Provide the start address and end address for the function. A fact that may help here: when the backdoor is triggered, the following string appears in the system log:

"Login accepted for backdoor admin user."

Two notes about Ghidra that may help here:

- The 'Defined Strings' window, accessible by selecting Window→Defined Strings from the code browser's menu bar, can be used to search for identified strings in the program. This window will give you the location of the string. By navigating to the location of the string, you can see identified cross-references to that string.
- If Ghidra fails to disassemble some section of executable code that you think should be disassembled, you can manually launch disassembly by navigating to the start of that segment of code in the Listing pane and pressing 'd'.

4.3 Back to the source: 1 point

This backdoor was inserted in a piece of open-source software, which means that there is source code available that can help inform our investigation. Find the name of the source file in the Dropbear source repository[3] that provides the function that you previously identified. Provide the source file name in your assignment submission. Note that you can use Github's search feature to search for characteristic features within the dropbear repository. For example, you might search for other strings that you find in the code that you're examining.

[3] `https://github.com/mkj/dropbear`

Find the function in the source file that corresponds to the function that you identified, and provide the function name in your assignment submission. Be careful here–at least one other function was inlined into this function at compile time.

4.4 Demonstration: 2 points

Describe how the backdoor functionality works. How you can use this backdoor to gain privileged access to the system?