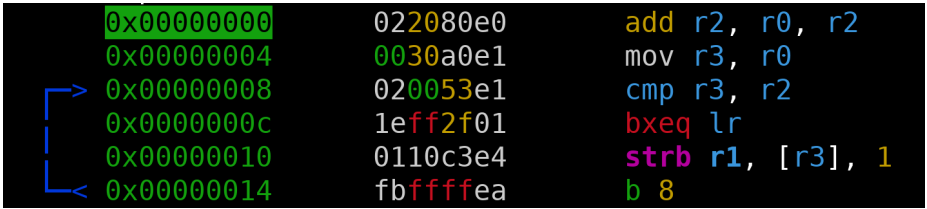
ENGR-E 399/599: Embedded Systems Reverse Engineering

Lecture 5: Dynamic analysis of embedded Linux systems

Austin Roach
ahroach@iu.edu

February 10, 2022

Mystery function



- How many arguments does the function take?
- What are the types of the arguments?
- What does the function do?
- What does the function return?

Mystery function revealed

```
void *memset(void *s, int c, size_t n);
```

Description

The `memset()` function fills the first `n` bytes of the memory area pointed to by `s` with the constant byte `c`.

Return value

The `memset()` function returns a pointer to the memory area `s`.

Today's plan

You will often have some system access, but not quite the access that you want...

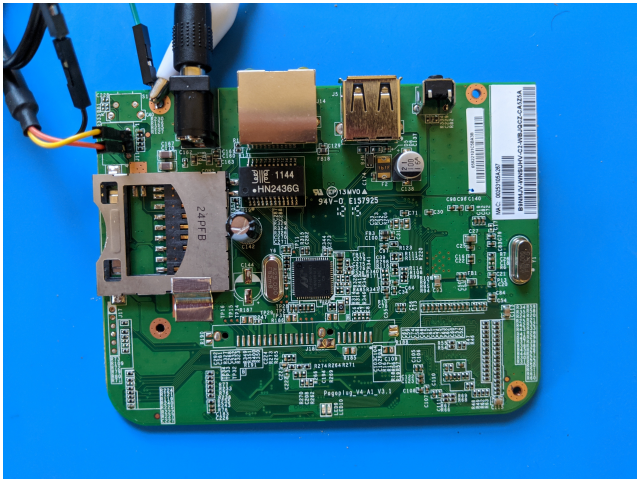
- Getting a shell
  - Bootloader interaction: kernel command-line parameters
  - Filesystem modifications
  - Providing a remote shell
- Dynamic analysis
  - Debugging with QEMU system-mode debugger
  - Adding useful tools to your target
  - Debugging with gdbserver
  - Other dynamic analysis tools
- Reverse-engineering a system service, part 2
  - Interacting with binary protocols

A real physical embedded system!

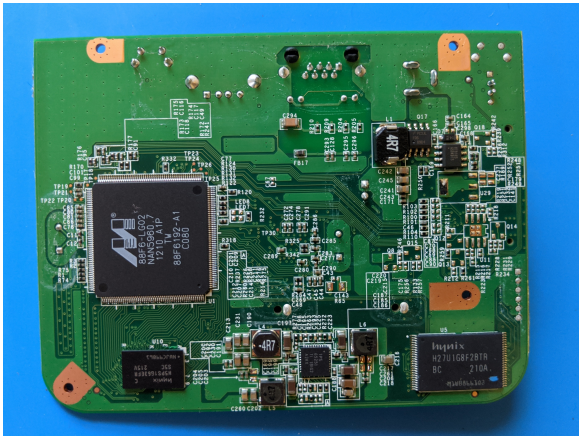
PogoPlug

- Cloud storage IoT device
  - Out of business
- Marvell ARM processor
- U-Boot/Linux/busybox

PogoPlug PCB: top



PogoPlug PCB: bottom



PogoPlug explorations

- Connect to UART for shell
  - From left to right: 3.3V, Tx, Rx, GND
  - Baud rate 115200
- Interrupt U-Boot with space
- Modify kernel command line arguments
  - Add `single` to bootargs (`printenv`, `setenv`, and `boot` commands)
- Explore filesystem
  - Busybox
  - Init scripts
  - Memory device access
- Explore execution environment
  - ps
  - mount
  - netstat
  - ip addr

Kernel command line parameters for QEMU

- Modify launch-emulated-system.sh to include `init=/bin/sh`
- Run elements of /etc/inittab manually
- Run /etc/init.d scripts to bring up system services

Filesystem modifications

If you can control the filesystem, you can accomplish a lot

- Change bootloader parameters
- Change init scripts
- Change credentials on system
- Add tools
- Modify programs
- ...

On a physical device, might need to reprogram a memory

- Much easier on an emulated system

Filesystem modification demo: change passwords

```
$ unsquashfs demo2-rootfs.squashfs
$ openssl passwd -5

... Replace an entry in /etc/shadow
$ mksquashfs squashfs-root demo2-rootfs.squashfs.mod
```

Starting from single command execution

Scenario:

- You have a network connection to the system
- You have the ability to execute a shell command
  - Administration service
  - Security weakness
  - UART shell, but you prefer to interact over a network

Some options:

- `telnetd -l /bin/sh -p 1337`
- `nc -l -p 1337 -e /bin/sh`

Dynamic analysis with debuggers

QEMU system-mode debugger

- QEMU can provide a GDB server to the emulated system
  - This is the `-s` option in `launch-emulated-system.sh`
  - Provides the server on TCP port 1234 by default
- Provides debug access to the *full* system
  - Set break points on virtual memory addresses
  - Pros:
    - ★ Can debug everything, even the kernel
    - ★ Don't need a shell or any filesystem modifications
  - Cons:
    - ★ It can be difficult to tell different processes apart, especially if they have overlapping virtual memory addresses
    - ★ Don't have the process-level debugging features (memory maps, symbols, etc.)
- Similar to debug access provided by JTAG debug access ports

QEMU system-mode debugger demo

Started `demo2-service` QEMU image with `-s`, connected with `nc 169.254.15.2 31331` after setting breakpoint

```
$ gdb-multiarch
(gdb) target remote localhost:1234
(gdb) b *0x10bc8
Breakpoint 1 at 0x10bc8
(gdb) c
Continuing.

Breakpoint 1, 0x00010bc8 in ?? ()
(gdb) info reg
```

What we would really like for a debugger

It would be better to have a debugger that's aware of process separation

- Start process under debugger control
- No problem with overlapping virtual memory address spaces
- Get process information (memory maps, etc.)
- Set behavior for forking, keep track of threads, etc.

GDB can do all of that, but we don't have GDB on our system...

gdbserver

On a lightweight embedded system, running GDB can be difficult.

A better option is `gdbserver`

- Small and lightweight
- Can connect over a network connection to a controlling GDB instance

All we have to do is get `gdbserver` onto our system somehow

Adding gdbserver

The quick and dirty way:

```
On target: $ nc -l -p 5678 > /tmp/gdbserver
On controlling host: $ nc 169.254.15.2 5678 < gdbserver
On target: $ chmod 755 /tmp/gdbserver
```

If you have control of the filesystem:

```
$ unsquashfs demo-rootfs.squashfs
$ cp gdbserver squashfs-root/usr/bin
$ mksquashfs squashfs-root demo-rootfs.squashfs.mod
```

Using gdbserver

Connect to an already running process:

```
$ gdbserver --attach 0.0.0.0:5432 108
```

Start a new process under gdbserver control:

```
$ gdbserver 0.0.0.0:5432 demo-service
```

Then connect as before:

```
$ gdb-multiarch
(gdb) target remote 169.254.15.2:5432
(gdb) set follow-fork-mode child
(gdb) b *0x10bc8
```

Other useful dynamic analysis approaches

- strace
  - Show system calls
- `LDPRELOAD`
  - Modify interactions with libraries
- `preeny`
  - Pre-made preload libraries
  - `https://github.com/zardus/preeny`
- Frida
  - Dynamic analysis framework
  - `https://frida.re/`

Demo: dynamic analysis of a system service