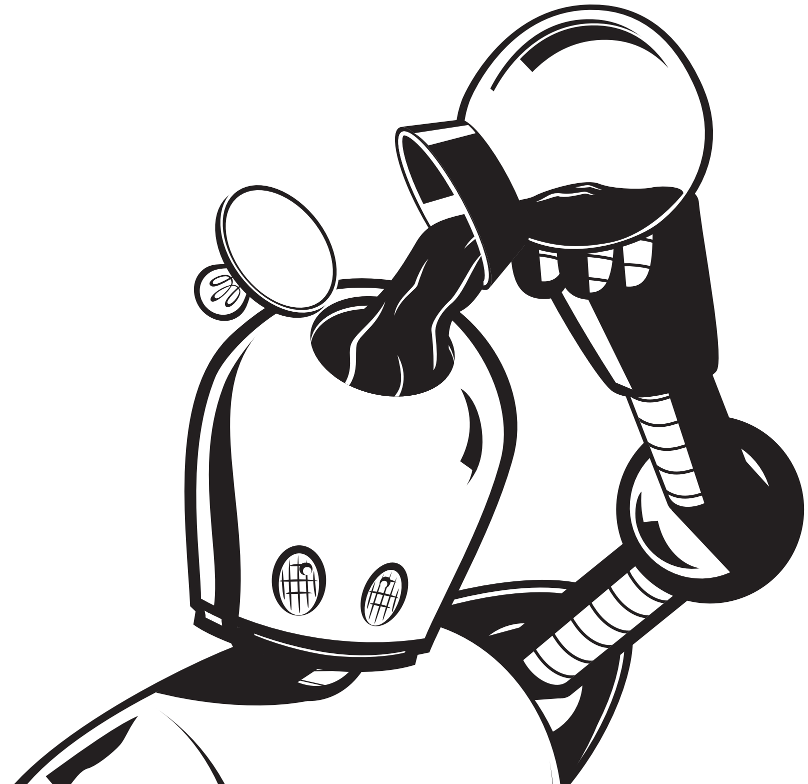
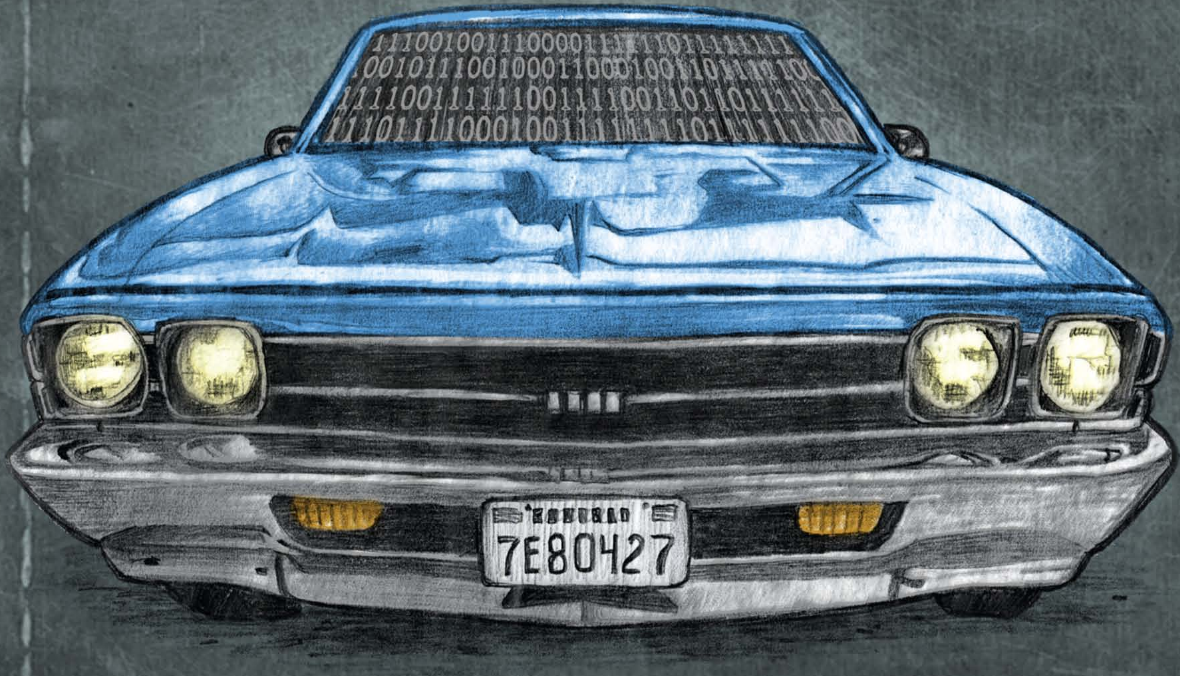


FEATURING EXCERPTS FROM

The Car Hacker's Handbook

A Guide for the Penetration Tester



Craig Smith

Foreword by Chris Evans

no starch press

5

REVERSE ENGINEERING THE CAN BUS

In order to reverse engineer the CAN bus, we first have to be able to read the CAN packets and identify which packets control what. That said, we don't need to be able to access the official diagnostic CAN packets because they're primarily a read-only window. Instead, we're interested in accessing *all* the other packets that flood the CAN bus. The rest of the nondiagnostic packets are the ones that the car actually uses to perform actions. It can take a long time to grasp the information contained in these packets, but that knowledge can be critical to understanding the car's behavior.

Locating the CAN Bus

Of course, before we can reverse the CAN bus, we need to locate the CAN. If you have access to the OBD-II connector, your vehicle's connector pinout map should show you where the CAN is. (See Chapter 2 for common locations of the OBD connectors and their pinouts.) If you don't have access to the OBD-II connector or you're looking for hidden CAN signals, try one of these methods:

- Look for paired and twisted wires. CAN wires are typically two wires twisted together.
- Use a multimeter to check for a 2.5V baseline voltage. (This can be difficult to identify because the bus is often noisy.)
- Use a multimeter to check for ohm resistance. The CAN bus uses a 120-ohm terminator on each end of the bus, so there should be 60 ohms between the two twisted-pair wires you suspect are CAN.
- Use a two-channel oscilloscope and subtract the difference between the two suspected CAN wires. You should get a constant signal because the differential signals should cancel each other out. (Differential signaling is discussed in "The CAN Bus" on page 122.)

NOTE *If the car is turned off, the CAN bus is usually silent, but something as simple as inserting the car key or pulling up on the door handle will usually wake the vehicle and generate signals.*

Once you've identified a CAN network, the next step is to start monitoring the traffic.

Reversing CAN Bus Communications with can-utils and Wireshark

First, you need to determine the type of communication running on the bus. You'll often want to identify a certain signal or the way a certain component talks—for example, how the car unlocks or how the drivetrain works. In order to do so, locate the bus those target components use, and then reverse engineer the packets traveling on that bus to identify their purpose.

To monitor the activity on your CAN, you need a device that can monitor and generate CAN packets, such as the ones discussed in Appendix A. There are a *ton* of these devices on the market. The cheap OBD-II devices that sell for under $20 technically work, but their sniffers are slow and will miss a lot of packets. It's always best to have a device that's as open as possible because it'll work with the majority of software tools—open source hardware and software is ideal. However, a proprietary device specifically designed to sniff CAN should still work. We'll look at using `candump`, from the `can-utils` suite, and Wireshark to capture and filter the packets.

Generic packet analysis won't work for CAN because CAN packets are unique to each vehicle's make and model. Also, because there's so much noise on CAN, it's too cumbersome to sort through every packet as it flows by in sequence.

Using Wireshark

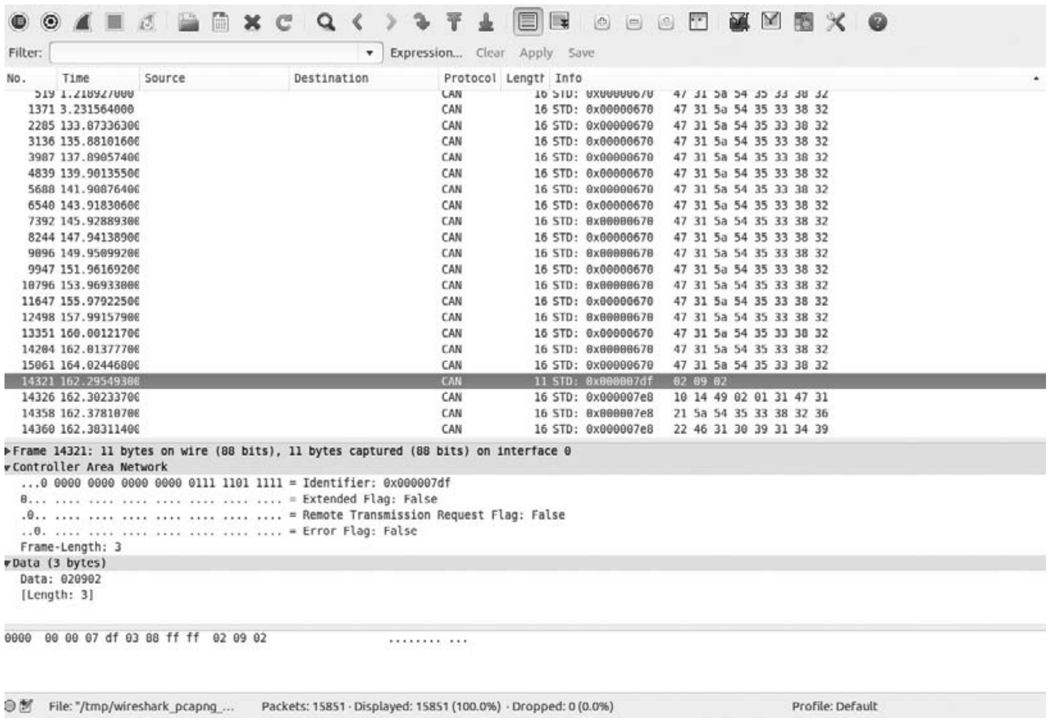
Wireshark (*https://www.wireshark.org/*) is a common network monitoring tool. If your background is in networking, your first instinct may be to use Wireshark to look at CAN packets. This technically works, but we will soon see why Wireshark is not the best tool for the job.

If you want to use Wireshark to capture CAN packets, you can do so together with SocketCAN. Wireshark can listen on both canX and vcanX devices, but not on slcanX because serial-link devices are not true netlink devices and they need a translation daemon in order for them to work. If you need to use a slcanX device with Wireshark, try changing the name from *slcanX* to *canX*. (I discuss CAN interfaces in detail Chapter 2.)

If renaming the interface doesn't work or you simply need to move CAN packets from an interface that Wireshark can't read to one it can, you can bridge the two interfaces. You'll need to use `candump` from the `can-utils` package in bridge mode to send packets from `slcan0` to `vcan0`.

```
$ candump -b vcan0 slcan0
```

Notice in Figure 5-1 that the data section isn't decoded and is just showing raw hex bytes. This is because Wireshark's decoder handles only the basic CAN header and doesn't know how to deal with ISO-TP or UDS packets. The highlighted packet is a UDS request for VIN. (I've sorted the packets in the screen by identifier, rather than by time, to make it easier to read.)



```
No.   Time          Source   Destination   Protocol Length Info
  519 1.218927000                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 1371 3.231564000                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 2285 133.8733630                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 3136 135.8810160                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 3987 137.8905740                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 4839 139.9013550                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 5688 141.9087640                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 6540 143.9183060                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 7392 145.9288930                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 8244 147.9413890                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 9096 149.9509920                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
 9947 151.9616920                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
10796 153.9693300                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
11647 155.9792250                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
12498 157.9915790                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
13351 160.0012170                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
14204 162.0137770                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
15061 164.0244680                          CAN      16 STD: 0x00000670   47 31 5a 54 35 33 38 32
14321 162.2954930                          CAN      11 STD: 0x000007df   02 09 02
14326 162.3023370                          CAN      16 STD: 0x000007e8   10 14 49 02 01 31 47 31
14358 162.3781070                          CAN      16 STD: 0x000007e8   21 5a 54 35 33 38 32 36
14360 162.3831140                          CAN      16 STD: 0x000007e8   22 46 31 30 39 31 34 39

▸Frame 14321: 11 bytes on wire (88 bits), 11 bytes captured (88 bits) on interface 0
▾Controller Area Network
  ...0 0000 0000 0000 0000 0111 1101 1111 = Identifier: 0x000007df
  0... .... .... .... .... .... .... .... = Extended Flag: False
  .0.. .... .... .... .... .... .... .... = Remote Transmission Request Flag: False
  ..0. .... .... .... .... .... .... .... = Error Flag: False
  Frame-Length: 3
▾Data (3 bytes)
  Data: 020902
  [Length: 3]

0000  00 00 07 df 03 00 ff ff  02 09 02           ........ ...

File: "/tmp/wireshark_pcapng_...   Packets: 15851 · Displayed: 15851 (100.0%) · Dropped: 0 (0.0%)   Profile: Default
```

Figure 5-1: Wireshark on the CAN bus

Using candump

As with Wireshark, `candump` doesn't decode the data for you; that job is left up to you, as the reverse engineer. Listing 5-1 uses `slcan0` as the sniffer device.

```
$ candump slcan0
  slcan0❶  388❷  [2]❸  01 10❹
  slcan0   110   [8]   00 00 00 00 00 00 00 00
  slcan0   120   [8]   F2 89 63 20 03 20 03 20
  slcan0   320   [8]   20 04 00 00 00 00 00 00
  slcan0   128   [3]   A1 00 02
  slcan0   7DF   [3]   02 09 02
  slcan0   7E8   [8]   10 14 49 02 01 31 47 31
  slcan0   110   [8]   00 00 00 00 00 00 00 00
  slcan0   120   [8]   F2 89 63 20 03 20 03 20
  slcan0   410   [8]   20 00 00 00 00 00 00 00
  slcan0   128   [3]   A2 00 01
  slcan0   380   [8]   02 02 00 00 E0 00 7E 0E
  slcan0   388   [2]   01 10
  slcan0   128   [3]   A3 00 00
  slcan0   110   [8]   00 00 00 00 00 00 00 00
  slcan0   120   [8]   F2 89 63 20 03 20 03 20
  slcan0   520   [8]   00 00 04 00 00 00 00 00
  slcan0   128   [3]   A0 00 03
  slcan0   380   [8]   02 02 00 00 E0 00 7F 0D
  slcan0   388   [2]   01 10
  slcan0   110   [8]   00 00 00 00 00 00 00 00
  slcan0   120   [8]   F2 89 63 20 03 20 03 20
  slcan0   128   [3]   A1 00 02
  slcan0   110   [8]   00 00 00 00 00 00 00 00
  slcan0   120   [8]   F2 89 63 20 03 20 03 20
  slcan0   128   [3]   A2 00 01
  slcan0   380   [8]   02 02 00 00 E0 00 7C 00
```

Listing 5-1: candump of traffic streaming through a CAN bus

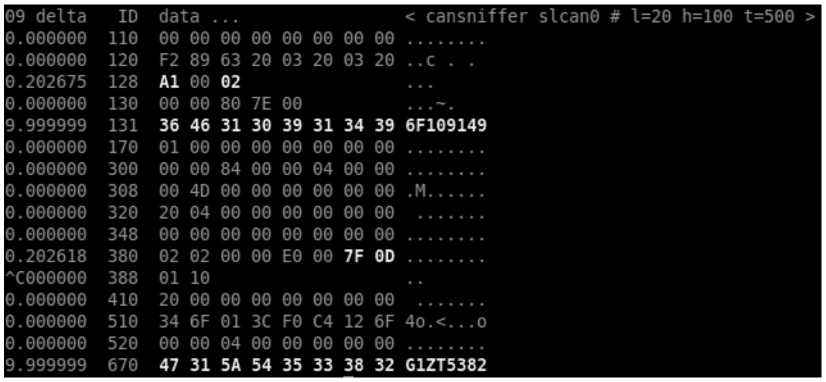
The columns are broken down to show the sniffer device ❶, the arbitration ID ❷, the size of the CAN packet ❸, and the CAN data itself ❹. Now you have some captured packets, but they aren't the easiest to read. We'll use filters to help identify the packets we want to analyze in more detail.

Grouping Streamed Data from the CAN Bus

Devices on a CAN network are noisy, often pulsing at set intervals or when triggered by an event, such as a door unlocking. This noise can make it futile to stream data from a CAN network without a filter. Good CAN sniffer software will group changes to packets in a data stream based on their arbitration ID, highlighting only the portions of data that have changed since the last time the packet was seen. Grouping packets in this way makes it easier to spot changes that result directly from vehicle manipulation, allowing you to actively monitor the tool's sniffing section and watch for color changes that correlate to physical changes. For example, if each time you unlock a door you see the same byte change in the data stream, you know that you've probably identified at least the byte that controls the door-unlocking functions.

Grouping Packets with cansniffer

The `cansniffer` command line tool groups the packets by arbitration ID and highlights the bytes that have changed since the last time the sniffer looked at that ID. For example, Figure 5-2 shows the result of running `cansniffer` on the device `slcan0`.



```
09 delta   ID  data ...                < cansniffer slcan0 # l=20 h=100 t=500 >
0.000000  110  00 00 00 00 00 00 00 00 ........
0.000000  120  F2 89 63 20 03 20 03 20 ..c . .
0.202675  128  A1 00 02                ...
0.000000  130  00 00 80 7E 00          ...~.
9.999999  131  36 46 31 30 39 31 34 39 6F109149
0.000000  170  01 00 00 00 00 00 00 00 ........
0.000000  300  00 00 84 00 00 04 00 00 ........
0.000000  308  00 4D 00 00 00 00 00 00 .M......
0.000000  320  20 04 00 00 00 00 00 00  .......
0.000000  348  00 00 00 00 00 00 00 00 ........
0.202618  380  02 02 00 00 E0 00 7F 0D ........
^C000000  388  01 10                   ..
0.000000  410  20 00 00 00 00 00 00 00  .......
0.000000  510  34 6F 01 3C F0 C4 12 6F 4o.<...o
0.000000  520  00 00 04 00 00 00 00 00 ........
9.999999  670  47 31 5A 54 35 33 38 32 G1ZT5382
```

Figure 5-2: `cansniffer` example output

You can add the `-c` flag to colorize any changing bytes.

```
$ cansniffer -c slcan0
```

The `cansniffer` tool can also remove repeating CAN traffic that isn't changing, thereby reducing the number of packets you need to watch.

Filtering the Packets Display

One advantage of `cansniffer` is that you can send it keyboard input to filter results as they're displayed in the terminal. (Note that you won't see the commands you enter while `cansniffer` is outputting results.) For example, to see only IDs 301 and 308 as `cansniffer` collects packets, enter this:

```
-000000
+301
+308
```

Entering `-000000` turns off all packets, and entering `+301` and `+308` filters out all except IDs 301 and 308.

The `-000000` command uses a *bitmask*, which does a bit-level comparison against the arbitration ID. Any binary value of 1 used in a mask is a bit that has to be true, while a binary value of 0 is a wildcard that can match anything. A bitmask of all 0s tells `cansniffer` to match any arbitration ID. The minus sign (-) in front of the bitmask removes all matching bits, which is every packet.

You can also use a filter and a bitmask with `cansniffer` to grab a range of IDs. For example, the following command adds the IDs from 500 through 5FF to the display, where 500 is the ID applied to the bitmask of 700 to define the range we're interested in.

```
+500700
```

To display all IDs of 5*XX*, you'd use the following binary representation:

```
ID  Binary Representation
500  101 0000 0000
700  111 0000 0000
------------------
     101 XXXX XXXX
      5    X    X
```

You could specify F00 instead of 700, but because the arbitration ID is made up of only 3 bits, a 7 is all that's required.

Using 7FF as a mask is the same as not specifying a bitmask for an ID. For example

```
+3017FF
```

is the same as

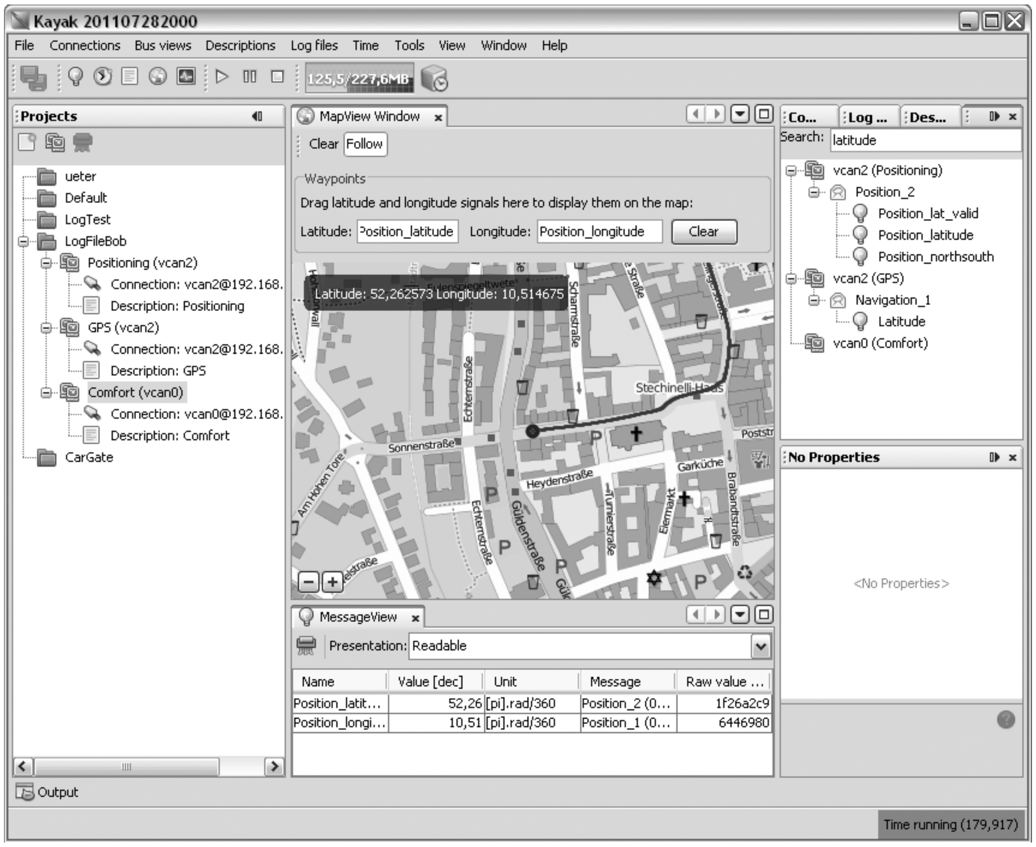
```
+301
```

This mask uses binary math and performs an `AND` operation on the two numbers, 0x301 and 0x7FF:

```
ID    Binary Representation
301   011  0000  0001
7FF   111  1111  1111
-------------------------
      011  0000  0001
      3    0     1
```

For those not familiar with `AND` operations, each binary bit is compared, and if *both* are a 1 then the output is a 1. For instance, `1 AND 1 = 1`, while `1 AND 0 = 0`.

If you prefer to have a GUI interface, Kayak, which we discussed in "Kayak" on page 126, is a CAN bus–monitoring application that also uses socketcand and will colorize its display of capture packets. Kayak won't remove repeating packets the way `cansniffer` does, but it offers a few unique capabilities that you can't easily get on the command line, such as documenting the identified packets in XML (*.kcd* files), which can be used by Kayak to display virtual instrument clusters and map data (see Figure 5-3).



Search: latitude

Latitude: Position_latitude Longitude: Position_longitude

Presentation: Readable

Name	Value [dec]	Unit	Message	Raw value ...
Position_latit...	52,26	[pi].rad/360	Position_2 (0...	1f26a2c9
Position_longi...	10,51	[pi].rad/360	Position_1 (0...	6446980

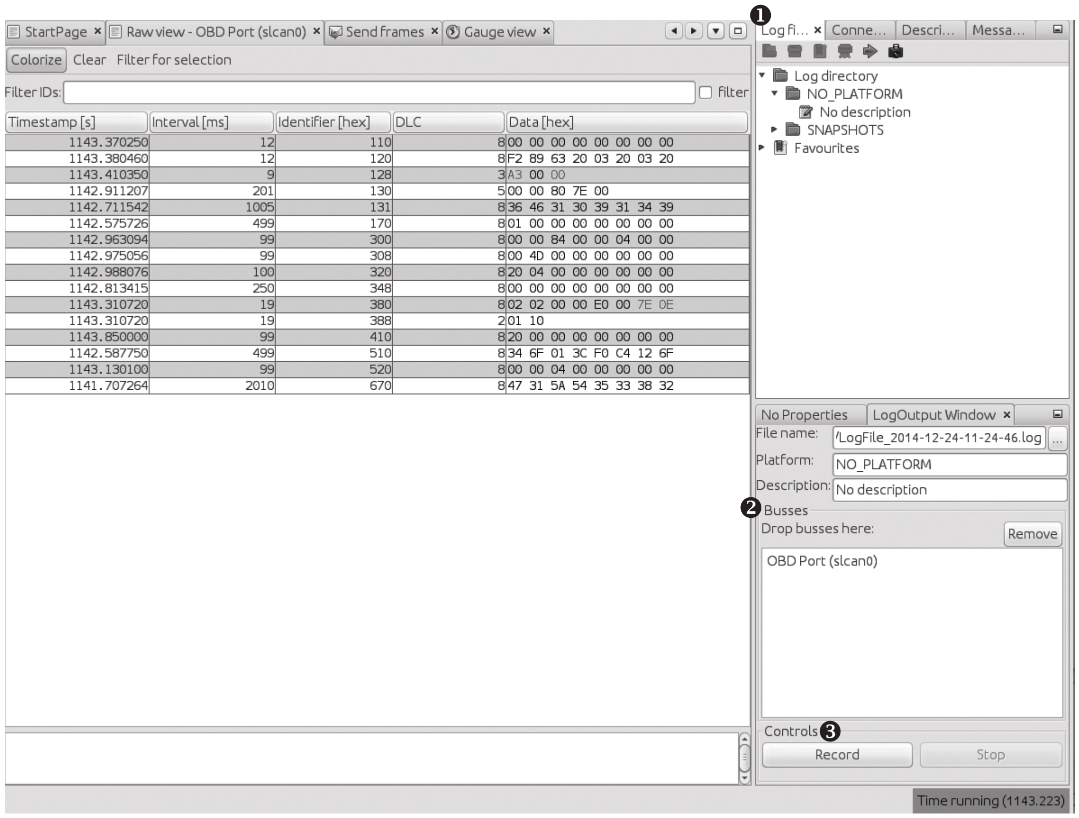
Figure 5-3: Kayak GUI interface

Using Record and Playback

Once you've used `cansniffer` or a similar tool to identify certain packets to focus on, the next step is to record and play back packets so you can analyze them. We'll look at two different tools to do this: `can-utils` and Kayak. They have similar functionality, and your choice of tool will depend on what you're working on and your interface preferences.

The `can-utils` suite records CAN packets using a simple ASCII format, which you can view with a simple text editor, and most of its tools support this format for both recording and playback. For example, you can record with `candump`, redirect standard output or use the command line options to record to a file, and then use `canplayer` to play back recordings.

Figure 5-4 shows a view of the layout of Kayak's equivalent to `cansniffer`.



Timestamp [s]	Interval [ms]	Identifier [hex]	DLC	Data [hex]
1143.370250	12	110	8	00 00 00 00 00 00 00 00
1143.380460	12	120	8	F2 89 63 20 03 20 03 20
1143.410350	9	128	3	A3 00 00
1142.911207	201	130	5	00 00 80 7E 00
1142.711542	1005	131	8	36 46 31 30 39 31 34 39
1142.575726	499	170	8	01 00 00 00 00 00 00 00
1142.963094	99	300	8	00 00 84 00 00 04 00 00
1142.975056	99	308	8	00 4D 00 00 00 00 00 00
1142.988076	100	320	8	20 04 00 00 00 00 00 00
1142.813415	250	348	8	00 00 00 00 00 00 00 00
1143.310720	19	380	8	02 02 00 00 E0 00 7E 0E
1143.310720	19	388	2	01 10
1143.850000	99	410	8	20 00 00 00 00 00 00 00
1142.587750	499	510	8	34 6F 01 3C F0 C4 12 6F
1143.130100	99	520	8	00 00 04 00 00 00 00 00
1141.707264	2010	670	8	47 31 5A 54 35 33 38 32

File name: /LogFile_2014-12-24-11-24-46.log
Platform: NO_PLATFORM
Description: No description

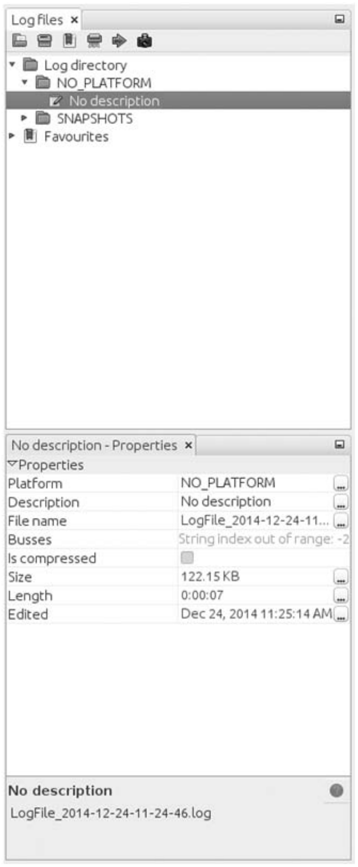
Figure 5-4: Kayak recording to a logfile

To record CAN packets with Kayak, first click the Play button in the Log files tab ❶. Then drag one or more buses from the Projects pane to the Busses field of the LogOutput Window tab ❷. Press the Record and Stop buttons at the bottom of the LogOutput window ❸ to start or stop recording. Once your packet capture is complete, the logging should show in the Log Directory drop-down menu (see Figure 5-5).

If you open a Kayak logfile, you'll see something like the code snippet in Listing 5-2. The values in this example won't directly correlate to those in Figure 5-4 because the GUI groups by ID, as in `cansniffer`, but the log is sequential, as in `candump`.

```
PLATFORM NO_PLATFORM
DESCRIPTION "No description"
DEVICE_ALIAS OBD Port slcan0
(1094.141850)❶ slcan0❷ 128#a20001❸
(1094.141863)   slcan0   380#02020000e0007e0e
(1094.141865)   slcan0   388#0110
(1094.144851)   slcan0   110#0000000000000000
(1094.144857)   slcan0   120#f289632003200320
```

Listing 5-2: Contents of Kayak's logfile



Log files ×

- Log directory
  - NO_PLATFORM
    - No description
  - SNAPSHOTS
- Favourites

No description - Properties ×

Properties

Platform	NO_PLATFORM
Description	No description
File name	LogFile_2014-12-24-11...
Busses	String index out of range: -2
Is compressed	☐
Size	122.15 KB
Length	0:00:07
Edited	Dec 24, 2014 11:25:14 AM

No description
LogFile_2014-12-24-11-24-46.log

Figure 5-5: Right pane of Log files tab settings

Other than some metadata (`PLATFORM`, `DESCRIPTION`, and `DEVICE_ALIAS`), the log is pretty much the same as the one captured by the `can-utils` package: ❶ is the timestamp, ❷ is your bus, and ❸ is your arbitration ID and data separated by a `#` symbol. To play back the capture, right-click the **Log Description** in the right panel, and open the recording (see Figure 5-5).

Listing 5-3 shows the logfile created by `candump` using the `-l` command line option:

```
(1442245115.027238) slcan0 166#D0320018
(1442245115.028348) slcan0 158#0000000000000019
(1442245115.028370) slcan0 161#000005500108001C
(1442245115.028377) slcan0 191#010010A141000B
```

Listing 5-3: `candump` logfile

Notice in Listing 5-3 that the `candump` logfiles are almost identical to those displayed by Kayak in Figure 5-4. (For more details on different `can-utils` programs, see "The CAN Utilities Suite" on page 129.)

Creative Packet Analysis

Now that we've captured packets, it's time to determine what each packet does so we can use it to unlock things or exploit the CAN bus. Let's start with a simple action that'll most likely toggle only a single bit—the code to unlock the doors—and see whether we can find the packet that controls that behavior.

Using Kayak to Find the Door-Unlock Control

There's a ton of noise on the CAN bus, so finding a single-bit change can be very difficult, even with a good sniffer. But here's a universal way to identify the function of a single CAN packet:

1. Press **Record**.
2. Perform the physical action, such as unlocking a door.
3. Stop **Record**.
4. Press **Playback**.
5. See whether the action was repeated. For example, did the door unlock?

If pressing Playback didn't unlock the door, a couple of things may have gone wrong. First, you may have missed the action in the recording, so try recording and performing the action again. If you still can't seem to record and replay the action, the message is probably hardwired to the physical lock button, as is often the case with the driver's-side door lock. Try unlocking the passenger door instead while recording. If that still doesn't work, the message for the unlock action is either on a CAN bus other than the one you're monitoring—you'll need to find the correct one—or the playback may have caused a collision, resulting in the packet being stomped on. Try to replay the recording a few times to make sure the playback is working.

Once you have a recording that performs the desired action, use the method shown in Figure 5-6 to filter out the noise and locate the exact packet and bits that are used to unlock the door via the CAN bus.

Now, keep halving the size of the packet capture until you're down to only one packet, at which point you should be able figure out which bit or bits are used to unlock the door. The quickest way to do this is to open your sniffer and filter on the arbitration ID you singled out. Unlock the door, and the bit or byte that changed should highlight. Now, try to unlock the car's back doors, and see how the bytes change. You should be able to tell exactly which bit must be changed in order to unlock each door.

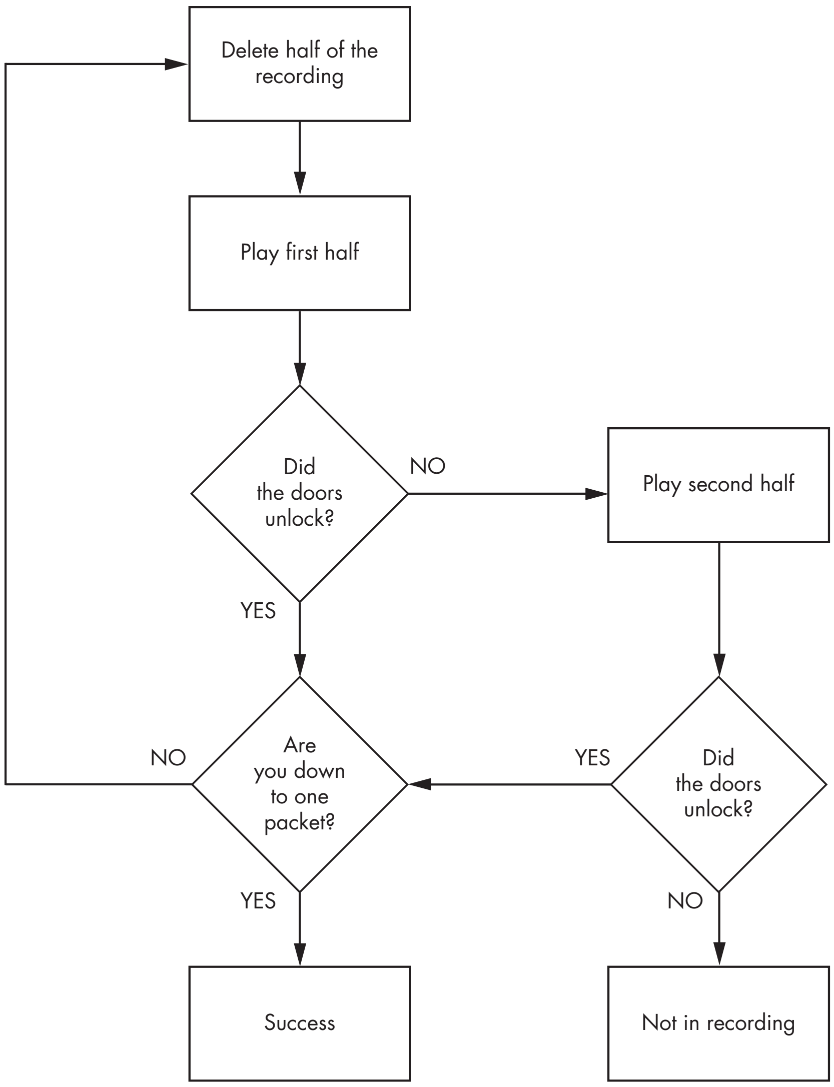


Figure 5-6: Sample unlock reversing flow

Using can-utils to Find the Door-Unlock Control

To identify packets via `can-utils`, you'd use `candump` to record and `canplayer` to play back the logfile, as noted earlier. Then, you'd use a text editor to whittle down the file before playback. Once you're down to one packet, you can then determine which byte or bits control the targeted operation with the help of `cansend`. For instance, by removing different halves of a logfile, you can identify the one ID that triggers the door to unlock:

```
slcan0  300   [8]  00 00 84 00 00 0F 00 00
```

Now, you could edit each byte and play back the line, or you could use `cansniffer` with a filter of +300 to single out just the 300 arbitration ID and monitor which byte changes when you unlock the door. For example, if the byte that controls the door unlock is the sixth byte—0x0F in the preceding example—we know that when the sixth byte is 0x00, the doors unlock, and when it's 0x0F, the doors lock.

NOTE *This is a hypothetical example that assumes we've performed all the steps listed earlier in this chapter to identify this particular byte. The specifics will vary for each vehicle.*

We can verify our findings with `cansend`:

```
$ cansend slcan0 300#00008400000F0000
```

If, after sending this, all the doors lock, we've successfully identified which packets control the door unlock.

Now, what happens when you change the 0x0F? To find out, unlock the car and this time send a 0x01:

```
$ cansend slcan0 300#0000840000010000
```

Observe that only the driver's-side door locks and the rest stay open. If you repeat this process with a 0x02, only the front passenger's-side door locks. When you repeat again with a 0x03, both the driver's-side door and the front passenger's-side door lock. But why did 0x03 control two doors and not a different third door? The answer may make more sense when you look at the binary representation:

```
0x00 = 00000000
0x01 = 00000001
0x02 = 00000010
0x03 = 00000011
```

The first bit represents the driver's-side door, and the second represents the front passenger's-side door. When the bit is a 1, the door locks, and when it's a 0, it unlocks. When you send an 0x0F, you're setting all bits that could affect the door lock to a binary 1, thereby locking all doors:

```
0x0F =  00001111
```

What about the remaining four bits? The best way to find out what they do is to simply set them to 1 and monitor the vehicle for changes. We already know that at least some of the 0x300 signal relates to doors, so it's fairly safe to assume the other four bits will, too. If not, they might control different door-like behavior, such as unlatching the trunk.

NOTE *If you don't get a response when you toggle a bit, it may not be used at all and may simply be reserved.*

Getting the Tachometer Reading

Obtaining information on the tachometer (the vehicle's speed) can be achieved in the same way as unlocking the doors. The diagnostic codes report the speed of a vehicle, but they can't be used to set how the speed displays (and what fun is that?), so we need to find out what the vehicle is using to control the readings on the instrument cluster (IC).

To save space, the RPM values won't display as a hex equivalent of the reading; instead, the value is shifted such that 1000 RPM may look like 0xFA0. This value is often referred to as "shifted" because in the code, the developers use bit shifting to perform the equivalent of multiplying or dividing. For the UDS protocol, this value is actually as follows:

$$\frac{(\textit{first byte} \times 256) + \textit{second byte}}{4}$$

To make matters worse, you can't monitor CAN traffic and query the diagnostic RPM to look for changing values at the same time. This is because vehicles often compress the RPM value using a proprietary method. Although the diagnostic values are set, they aren't the actual packets and values that the vehicle is using, so we need to find the real value by reversing the raw CAN packets. (Be sure to put the car in park before you do this, and even lift the vehicle off the ground or put it on rollers first to avoid it starting suddenly and crushing you.)

Follow the same steps that you used to find the door unlock control:

1. Press **Record**.
2. Press the gas pedal.
3. Stop **Record**.
4. Press **Playback**.
5. See whether the tachometer gauge has moved.

You'll probably find that a lot of engine lights flash and go crazy during this test because this packet is doing a lot more than just unlocking the car door. Ignore all the blinking warning lights, and follow the flowchart shown in Figure 5-6 to find the arbitration ID that causes the tachometer to change. You'll have a much higher chance of collisions this time than when trying to find the bit to unlock the doors because there's a lot more going on. Consequently, you may have to play and record more traffic than before. (Remember the value conversions mentioned earlier, and keep in mind that more than one byte in this arbitration ID will probably control the reported speed.)

Putting Kayak to Work

To make things a bit easier, we'll use Kayak's GUI instead of `can-utils` to find the arbitration IDs that control the tachometer. Again, make sure that the car is immobilized in an open area, with the emergency brake on, and maybe even up on blocks or rollers. Start recording and give the engine a good rev. Then, stop recording and play back the data. The RPM gauge should move; if it doesn't, you may be on the wrong bus and will need to locate the correct bus, as described earlier in this chapter.

Once you have the reaction you expect from the vehicle, repeat the halving process used to find the door unlock, with some additional Kayak options.

Kayak's playback interface lets you set the playback to loop infinitely and, more importantly, set the "in" and "out" packets (see Figure 5-7). The slider represents the number of packets captured. Use the slider to pick which packet you start and stop with during playback. You can quickly jump to the middle or other sections of the recording using the slider, which makes playing back half of a section very easy.

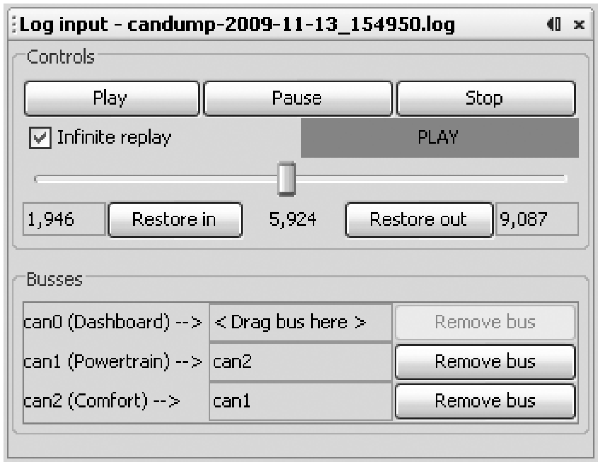


Figure 5-7: Kayak playback interface

As for testing, you won't be able to send only a single packet as you did when you tried to unlock the car because the vehicle is constantly reporting its current speed. To override this noise, you need to talk even faster than the normal communication to avoid colliding all the time. For instance, if you play your packets right after the real packet plays, then the last seen update will be the modified one. Reducing noise on the bus results in fewer collisions and cleaner demos. If you can send your fake packet immediately after the real packet, you often get better results than you would by simply flooding the bus.

To send packets continuously with `can-utils`, you can use a `while` loop with `cansend` or `cangen`. (When using Kayak's Send Frame dialog to transmit packets, make sure to check the Interval box.)

Creating Background Noise with the Instrument Cluster Simulator

The instrument cluster simulator (ICSim) is one of the most useful tools to come out of Open Garages, a group that fosters open collaboration between mechanics, performance tuners, and security researchers (see Appendix A). ICSim is a software utility designed to produce a few key CAN signals in order to provide a lot of seemingly "normal" background CAN noise—essentially, it's designed to let you practice CAN bus reversing without having to tinker around with your car. (ICSim is Linux only because it relies on the virtual CAN devices.) The methods you'll learn playing with ICSim will directly translate to your target vehicles. ICSim was designed as a safe way to familiarize yourself with CAN reversing so that the transition to an actual vehicle is as seamless as possible.

Setting Up the ICSim

Grab the source code for the ICSim from *https://github.com/zombieCraig/ICSim* and follow the README file supplied with the download to compile the software. Before you run ICSim, you should find a sample script in the README called *setup_vcan.sh* that you can run to set up a `vcan0` interface for the ICSim to use.

ICSim comes with two components, `icsim` and `controls`, which talk to each other over a CAN bus. To use ICSim, first load the instrument cluster to the vcan device like this:

```
$ ./icsim vcan0
```

In response, you should see the ICSim instrument cluster with turn signals, a speedometer, and a picture of a car, which will be used to show the car doors locking and unlocking (see Figure 5-8).

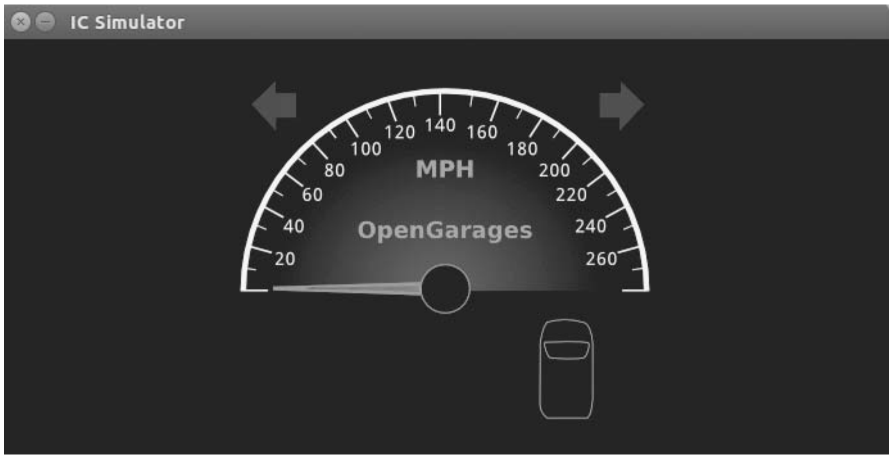


Figure 5-8: ICSim instrument cluster

The `icsim` application listens only for CAN signals, so when the ICSim first loads, you shouldn't see any activity. In order to control the simulator, load the CANBus Control Panel like this:

```
$ ./controls vcan0
```

The CANBus Control Panel shown in Figure 5-9 should appear.

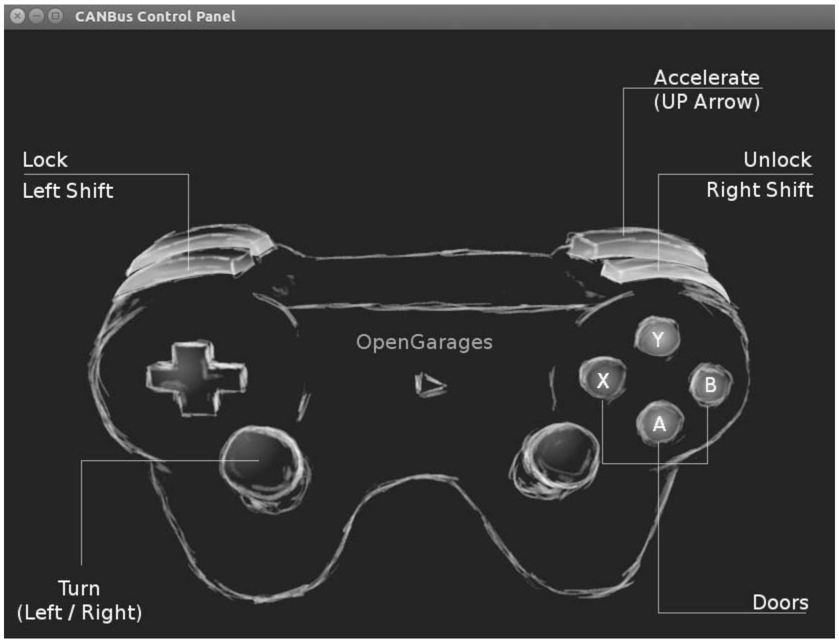


Figure 5-9: ICSim control interface

The screen looks like a game controller; in fact, you can plug in a USB game controller, and it should be supported by ICSim. (As of this writing, you can use `sixad` tools to connect a PS3 controller over Bluetooth as well.) You can use the controller to operate the ICSim in a method similar to driving a car using a gaming console, or you can control it by pressing the corresponding keys on your keyboard (see Figure 5-9).

NOTE *Once the control panel is loaded, you should see the speedometer idle just above 0 mph. If the needle is jiggling a bit, you know it's working. The control application writes only to the CAN bus and has no other way to communicate with the* `icsim`*. The only way to control the virtual car is through the CAN.*

The main controls on the CANBus Control Panel are as follows:

Accelerate (up arrow) Press this to make the speedometer go faster. The longer you hold the key down, the faster the virtual vehicle goes.

Turn (left/right arrows) Hold down a turn direction to blink the turn signals.

Lock (left SHIFT), Unlock (right SHIFT) This one requires you to press two buttons at once. Hold down the left SHIFT and press a button (A, B, X, or Y) to lock a corresponding door. Hold down the right SHIFT and press one of the buttons to unlock a door. If you hold down left SHIFT and then press right SHIFT, it will *unlock* all the doors. If you hold down right SHIFT and press left SHIFT, you'll *lock* all the doors.

Make sure you can fit both the ICSim and the CANBus Control Panel on the same screen so that you can see how they influence each other. Then, select the control panel so that it's ready to receive input. Play around with the controls to make sure that the ICSim is responding properly. If you don't see a response to your controls, ensure that the ICSim control window is selected and active.

Reading CAN Bus Traffic on the ICSim

When you're sure everything is working, fire up your sniffer of choice and take a look at the CAN bus traffic, as shown in Figure 5-10. Try to identify which packets control the vehicle, and create scripts to control ICSim without using the control panel.

Most of the changing data you see in Figure 5-10 is caused by a replay file of a real CAN bus. You'll have to sort through the messages to determine the proper packets. All methods of replay and packet sending will work with ICSim, so you can validate your findings.

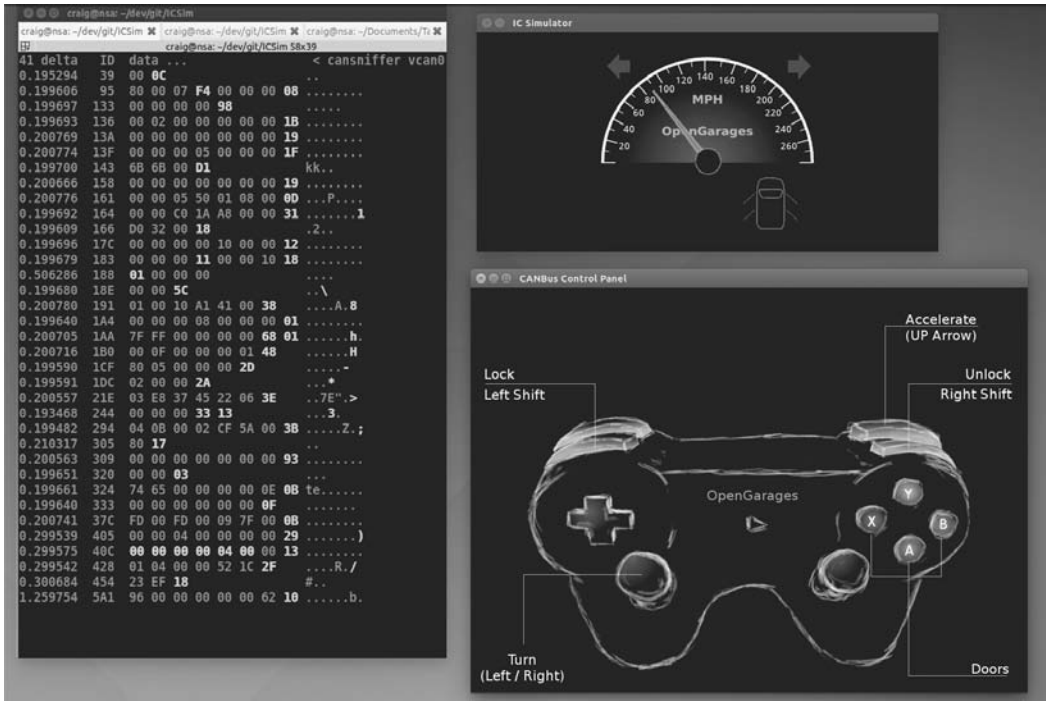

```
41 delta   ID  data ...                < cansniffer vcan0
0.195294   39  00 0C                   ..
0.199606   95  80 00 07 F4 00 00 00 08 ........
0.199697  133  00 00 00 00 98          .....
0.199693  136  00 02 00 00 00 00 00 1B ........
0.200769  13A  00 00 00 00 00 00 00 19 ........
0.200774  13F  00 00 00 05 00 00 00 1F ........
0.199700  143  6B 6B 00 D1             kk..
0.200666  158  00 00 00 00 00 00 00 19 ........
0.200776  161  00 00 05 50 01 08 00 0D ...P....
0.199692  164  00 00 C0 1A A8 00 00 31 .......1
0.199609  166  D0 32 00 18             .2..
0.199696  17C  00 00 00 00 10 00 00 12 ........
0.199679  183  00 00 00 11 00 00 10 18 ........
0.506286  188  01 00 00 00             ....
0.199680  18E  00 00 5C                ..\
0.200780  191  01 00 10 A1 41 00 38    ....A.8
0.199640  1A4  00 00 00 08 00 00 00 01 ........
0.200705  1AA  7F FF 00 00 00 00 68 01 ......h.
0.200716  1B0  00 0F 00 00 00 01 48    ......H
0.199590  1CF  80 05 00 00 00 2D       .....-
0.199591  1DC  02 00 00 2A             ...*
0.200557  21E  03 E8 37 45 22 06 3E    ..7E".>
0.193468  244  00 00 00 33 13          ...3.
0.199482  294  04 0B 00 02 CF 5A 00 3B .....Z.;
0.210317  305  80 17                   ..
0.200563  309  00 00 00 00 00 00 00 93 ........
0.199651  320  00 00 03                ...
0.199661  324  74 65 00 00 00 00 0E 0B te......
0.199640  333  00 00 00 00 00 00 0F    .......
0.200741  37C  FD 00 FD 00 09 7F 00 0B ........
0.299539  405  00 00 04 00 00 00 00 29 .......)
0.299575  40C  00 00 00 00 04 00 00 13 ........
0.299542  428  01 04 00 00 52 1C 2F    ....R./
0.300684  454  23 EF 18                #..
1.259754  5A1  96 00 00 00 00 00 62 10 ......b.
```

Figure 5-10: Screen layout for using ICSim

Changing the Difficulty of ICSim

One of the great things about ICSim is that you can challenge yourself by making it harder to find the target CAN traffic. ICSim supports four difficulty levels—0 through 3, with level 1 as the default. Level 0 is a super simple CAN packet that does the intended operation without any background noise, while level 3 randomizes all the bytes in the packet as well. To have the simulator choose different IDs and target byte positions, use ICSim's randomize option:

```
$ ./icsim -r vcan0
Using CAN interface vcan0
Seed: 1419525427
```

This option prints a randomized seed value to the console screen.

Pass this value into the CANBus Control Panel along with your choice of difficulty level:

```
$ ./controls -s 1419525427 -l 3 vcan0
```

You can replay or share a specific seed value as well. If you find one you like or if you want to race your friends to see who can decipher the packets first, launch ICSim with a set seed value like this:

```
$ ./icsim -s  1419525427 vcan0
```

Next, launch the CANBus Control Panel using the same seed value to sync up the randomized control panel to the ICSim. If the seed values aren't the same, they won't be able to communicate.

It may take you a while to locate the proper packets the first time using ICSim, but after a few passes, you should be able to quickly identify which packets are your targets.

Try to complete the following challenges in ICSim:

1. Create "hazard lights." Make both turn signals blink at the same time.
2. Create a command that locks only the back two doors.
3. Set the speedometer as close as possible to 220 mph.

Reversing the CAN Bus with OpenXC

Depending on your vehicle, one solution to reverse engineering the CAN bus is OpenXC, an open hardware and software standard that translates proprietary CAN protocols into an easy-to-read format. The OpenXC initiative was spearheaded by the Ford Motor Company—and as I write this, OpenXC is supported only by Ford—but it could work with any auto manufacturer that supports it. (Visit *http://openxcplatform.com/* for information on how to acquire a pre-made dongle.)

Ideally, open standards for CAN data such as OpenXC will remove the need for many applications to reverse engineer CAN traffic. If the rest of the automotive industry were to agree on a standard that defines how their vehicles work, it would greatly improve a car owner's ability to tinker and build on new innovative tools.

Translating CAN Bus Messages

If a vehicle supports OpenXC, you can plug a vehicle interface (VI) in to the CAN bus, and the VI should translate the proprietary CAN messages and send them to your PC so you can read the supported packets without having to reverse them. In theory, OpenXC should allow access to any CAN packet via a standard API. This access could be read-only or allow you to transmit packets. If more auto manufacturers eventually support OpenXC, it could provide third-party tools with more raw access to a vehicle than they would have with standard UDS diagnostic commands.

NOTE *OpenXC supports Python and Android and includes tools such as* `openxc-dump` *to display CAN activity.*

The fields from OpenXC's default API are as follows:

- `accelerator_pedal_position`
- `brake_pedal_status`
- `button_event` (typically steering wheel buttons)
- `door_status`
- `engine_speed`
- `fuel_consumed_since_last_restart`
- `fuel_level`
- `headlamp_status`
- `high_beam_status`
- `ignition_status`
- `latitude`
- `longitude`
- `odometer`
- `parking_brake_status`
- `steering_wheel_angle`
- `torque_at_transmission`
- `transmission_gear_position`
- `vehicle_speed`
- `windshield_wiper_status`

Different vehicles may support different signals than the ones listed here or no signals at all.

OpenXC also supports JSON trace output for recording vehicle journey. JSON provides a common data format that's easy for most other modern languages to consume, as shown in Listing 5-4.

```
{"metadata": {
    "version": "v3.0",
    "vehicle_interface_id": "7ABF",
    "vehicle": {
        "make": "Ford",
        "model": "Mustang",
        "trim": "V6 Premium",
        "year": 2013
    },
    "description": "highway drive to work",
    "driver_name": "TJ Giuli",
    "vehicle_id": "17N1039247929"
}
```

Listing 5-4: Simple JSON file output

Notice how the metadata definitions in JSON make it fairly easy for both humans and a programming language to read and interpret. The above JSON listing is a definition file, so an API request would be even smaller. For example, when requesting the field `steering_wheel_angle`, the translated CAN packets would look like this:

```
{"timestamp": 1385133351.285525, "name": "steering_wheel_angle", "value": 45}
```

You can interface with the OpenXC with OBD like this:

```
$ openxc-diag -message-id 0x7df -mode 0x3
```

Writing to the CAN Bus

If you want to write back to the bus, you *might* be able to use something like the following line, which writes the steering wheel angle back to the vehicle, but you'll find that the device will resend only a few messages to the CAN bus.

```
$ openxc-control write -name steering_wheel_angle -value 42.0
```

Technically, OpenXC supports raw CAN writes, too, like this:

```
$ openxc-control write -bus 1 -id 42 -data 0x1234
```

This brings us back from translated JSON to raw CAN hacking, as described earlier in this chapter. However, if you want to write an app or embedded graphical interface to only read and react to your vehicle and you own a new Ford, then this may be the quickest route to those goals.

Hacking OpenXC

If you've done the work to reverse the CAN signals, you can even make your own VI OpenXC firmware. Compiling your own firmware means you don't have any limitations, so you can read and write whatever you want and even create "unsupported" signals. For example, you could create a signal for `remote_engine_start` and add it to your own firmware in order to provide a simple interface to start your car. Hooray, open source!

Consider a signal that represents `engine_speed`. Listing 5-5 will set a basic configuration to output the `engine_speed` signal. We'll send RPM data with a 2-byte-long message ID 0x110 starting at the second byte.

```
{  "name" : "Test Bench",
    "buses": {
       "hs": {
           "controller": 1,
           "speed": 500000
       }
   },
   "messages": {
      "0x110": {
         "name": "Acceleration",
         "bus", "hs",
         "signals": {
             "engine_speed_signal": {
                "generic_name": "engine_speed",
                "bit_position": 8,
                "bit_size": 16
             }
          }
       }
   }
}
```

Listing 5-5: Simple OpenXC config file to define `engine_speed`

The OpenXC config files that you want to modify are stored in JSON. First, we define the bus by creating a JSON file with a text editor. In the example, we create a JSON config for a signal on the high-speed bus running at 500Kbps.

Once you have the JSON config defined, use the following code to compile it into a CPP file that can be compiled into the firmware:

```
$ openxc-generate-firmware-code –message-set ./test-bench.json > signals.cpp
```

Then, recompile the VI firmware with these commands:

```
$ fab reference build
```

If all goes well, you should have a *.bin* file that can be uploaded to your OpenXC-compatible device. The default bus is set up in raw read/write mode that sets the firmware to a cautionary read-only mode by default, unless signals or a whole bus is set up to support writing. To set those up, when defining the bus, you can add `raw_can_mode` or `raw_writable` and set them to true.

By making your own config files for OpenXC, you can bypass the restrictions set up in prereleased firmware and support other vehicles besides Ford. Ideally, other manufacturers will begin to support OpenXC, but adoption has been slow, and the bus restrictions are so strict you'll probably want to use custom firmware anyhow.

Fuzzing the CAN Bus

Fuzzing the CAN bus can be a good way to find undocumented diagnostic methods or functions. Fuzzing takes a random, shotgun-like approach to reversing. When *fuzzing*, you send random-ish data to an input and look for unexpected behavior, which in the case of a vehicle could be physical changes, such as IC messages, or component crashes, such as shutdowns or reboots.

The good news is that it's easy to make a CAN fuzzer. The bad news is that it's rarely useful. Useful packets are often part of a collection of packets used to cause a particular change, such as a diagnostic service that is active only after a successful security token has been passed to it, so it's difficult to tell which packet to focus on when fuzzing. Also, some CAN packets are visible only from within a moving vehicle, which would be very dangerous. Nevertheless, don't rule out fuzzing as a potential method of attack because you can sometimes use it to locate undocumented services or crashes to a target component you want to spoof.

Some sniffers support fuzzing directly—a feature usually found in the transmission section and represented by the tool's ability to transmit packets with incrementing bytes in the data section. For example, in the case of SocketCAN, you can use `cangen` to generate random CAN traffic. Several other open source CAN sniffing solutions allow for easy scripting or programming with languages such as Python.

A good starting point for fuzzing is to look at the UDS commands, specifically the "undocumented" manufacturer commands. When fuzzing undocumented UDS modes, we typically look for any type of response from an unknown mode. For instance, when targeting the UDS diagnostics of the ECU, you might send random data to ID 0x7DF and get an error packet from an unexpected mode. If you use brute-forcing tools such as CaringCaribou, however, there are often cleaner ways of accomplishing the same thing, such as monitoring or reversing the diagnostic tools themselves.

Troubleshooting When Things Go Wrong

The CAN bus and its components are fault-tolerant, which limits the damage you can do when reversing the CAN bus. However, if you're fuzzing the CAN bus or replaying a large amount of CAN data back on a live CAN bus network, things can go wrong. Here are a few common problems and solutions.

Flashing IC Lights

It's common for the IC lights to flash when sending packets to the CAN bus, and you can usually reset them by restarting the vehicle. If restarting the vehicle still doesn't fix the lights, try disconnecting and reconnecting the battery. If that still doesn't fix the problem, make sure that your battery is properly charged since a low battery can also make the IC lights flash.

Car Not Turning On

If your car shuts off and won't turn back on, it's usually because you've drained the battery by working with the CAN bus while the car is not fully running. This can drain a battery much faster than you might think. To restart it, jump the vehicle with a spare battery.

If you've tried jumping the vehicle and it still won't turn on, you may need to pull a fuse and plug it back in to restart the car. Locate the engine fuses in the car's manual and begin by pulling the ones you most suspect are the culprits. The fuse probably isn't blown, so just pull it out and put it back in to force the problem device to restart. The fuses you choose to pull will depend on your type of vehicle, but if your engine isn't starting, you will want to locate major components to disconnect and check. Look for main fuses around major electronics. The fuses that control the headlamps probably are not the culprits. Use a process of elimination to determine the device that is causing the issue.

Car Not Turning Off

You might find that you're unable to shut the car down. This is a bad, but fortunately rare, situation. First, check that you aren't flooding the CAN bus with traffic; if you are, stop and disconnect from the CAN bus. If you're already disconnected from the CAN bus and your car still won't turn off, you'll need to start pulling fuses until it does.

Vehicle Responding Recklessly

This will only occur if you're injecting packets in a moving vehicle, which is a terrible idea and should never be done! If you must audit a vehicle while it's wheels are moving, raise it off the ground or on rollers.

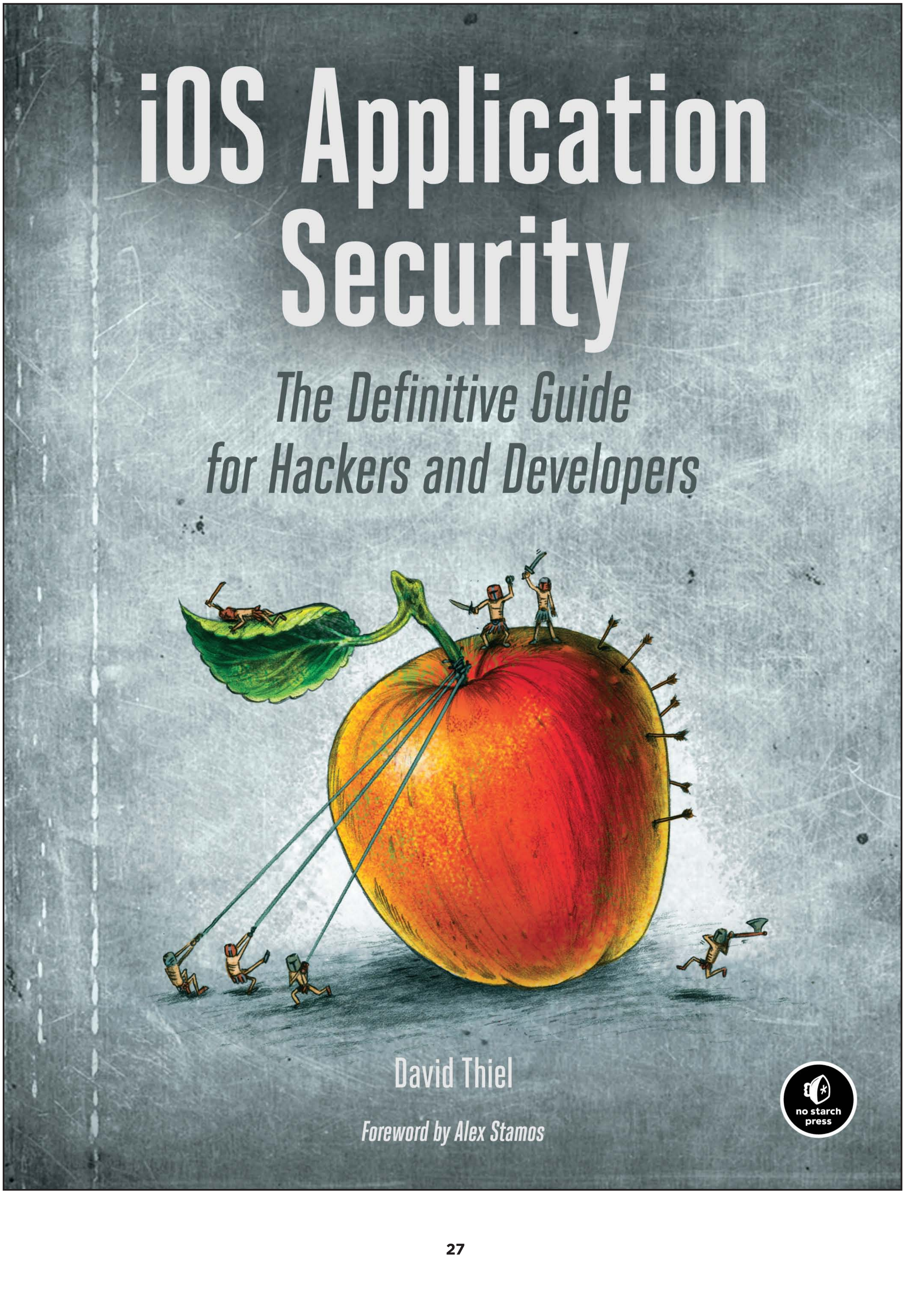
Bricking

Reverse engineering the CAN bus should never result in bricking—that is, breaking the vehicle so completely that it can do nothing. To brick a vehicle, you would need to mess around with the firmware, which would put the vehicle or component out of warranty and is done at your own risk.

Summary

In this chapter, you learned how to identify CAN wires from the jumble of wires under the dash, and how to use tools like `cansniffer` and Kayak to sniff traffic and identify what the different packets were doing. You also learned how to group CAN traffic to make changes easier to identify than they would be when using more traditional packet-sniffing tools, such as Wireshark.

You should now be able to look at CAN traffic and identify changing packets. Once you identify these packets, you can write programs to transmit them, create files for Kayak to define them, or create translators for OpenXC to make it easy to use dongles to interact with your vehicle. You now have all the tools you need to identify and control the components of your vehicle that run on CAN.



iOS Application Security

The Definitive Guide for Hackers and Developers

David Thiel

Foreword by Alex Stamos

no starch press

4
BUILDING YOUR TEST PLATFORM

In this chapter, I'll outline the tools you need to review your code and test your iOS applications, and I'll show you how to build a robust and useful test platform. That test platform will include a properly set up Xcode instance, an interactive network proxy, reverse engineering tools, and tools to bypass iOS platform security checks.

I'll also cover the settings you need to change in Xcode projects to make bugs easier to identify and fix. You'll then learn to leverage Xcode's static analyzer and compiler options to produce well-protected binaries and perform more in-depth bug detection.

Taking Off the Training Wheels

A number of behaviors in a default OS X install prevent you from really digging in to the system internals. To get your OS to stop hiding the things you need, enter the following commands at a Terminal prompt:

```
$ defaults write com.apple.Finder AppleShowAllFiles TRUE
$ defaults write com.apple.Finder ShowPathbar -bool true
$ defaults write com.apple.Finder _FXShowPosixPathInTitle -bool true
$ defaults write NSGlobalDomain AppleShowAllExtensions -bool true
$ chflags nohidden ~/Library/
```

These settings let you see all the files in the Finder, even ones that are hidden from view because they have a dot in front of their name. In addition, these changes will display more path information and file extensions, and most importantly, they allow you to see your user-specific *Library*, which is where the iOS Simulator will store all of its data.

The `chflags` command removes a level of obfuscation that Apple has put on directories that it considers too complicated for you, such as */tmp* or */usr*. I'm using the command here to show the contents of the iOS Simulator directories without having to use the command line every time.

One other thing: consider adding *$SIMPATH* to the Finder's sidebar for easy access. It's convenient to use *$SIMPATH* to examine the iOS Simulator's filesystem, but you can't get to it in the Finder by default. To make this change, browse to the following directory in the Terminal:

```
$ cd ~/Library/Application\ Support
$ open .
```

Then, in the Finder window that opens, drag the iPhone Simulator directory to the sidebar. Once you're riding without training wheels, it's time to choose your testing device.

Suggested Testing Devices

My favorite test device is the Wi-Fi only iPad because it's inexpensive and easy to jailbreak, which allows for testing iPad, iPhone, and iPod Touch applications. Its lack of cellular-based networking isn't much of a hindrance, given that you'll want to intercept network traffic most of the time anyway.

But this configuration does have some minor limitations. Most significantly, the iPad doesn't have GPS or SMS, and it obviously doesn't make phone calls. So it's not a bad idea to have an actual iPhone of some kind available.

I prefer to have at least two iPads handy for iOS testing: one jailbroken and one stock. The stock device allows for testing in a legitimate, realistic end-user environment, and it has all platform security mechanisms still intact. It can also register properly for push notifications, which has proven problematic for jailbroken devices in the past. The jailbroken device allows you to more closely inspect the filesystem layout and more detailed workings of iOS; it also facilitates black-box testing that wouldn't be feasible using a stock device alone.

Testing with a Device vs. Using a Simulator

Unlike some other mobile operating systems, iOS development uses a *simulator* rather than an emulator. This means there's no full emulation of the iOS device because that would require a virtualized ARM environment. Instead, the simulators that Apple distributes with Xcode are compiled for the x64 architecture, and they run natively on your development machine, which makes the process significantly faster and easier. (Try to boot the Android emulator inside a virtual machine, and you'll appreciate this feature.)

On the flip side, some things simply don't work the same way in the iOS Simulator as they do on the device. The differences are as follows:

Case-sensitivity Unless you've intentionally changed this behavior, OS X systems operate with case-insensitive HFS+ filesystems, while iOS uses the case-sensitive variant. This should rarely be relevant to security but can cause interoperability issues when modifying programs.

Libraries In some cases, iOS Simulator binaries link to OS X frameworks that may behave differently than those on iOS. This can result in slightly different behavior.

Memory and performance Since applications run natively in the iOS Simulator, they'll be taking full advantage of your development machine's resources. When gauging the impact of things such as PBKDF2 rounds (see Chapter 13), you'll want to compensate for this or test on a real device.

Camera As of now, the iOS Simulator does not use your development machine's camera. This is rarely a huge issue, but some applications do contain functionality such as "Take a picture of my check stub or receipt," where the handling of this photo data can be crucial.

SMS and cellular You can't test interaction with phone calls or SMS integration with the iOS Simulator, though you can technically simulate some aspects, such as toggling the "in-call" status bar.

Unlike in older versions of iOS, modern versions of the iOS Simulator do in fact simulate the Keychain API, meaning you can manage your own certificate and store and manipulate credentials. You can find the files behind this functionality in *$SIMPATH/Library/Keychains*.

Network and Proxy Setup

Most of the time, the first step in testing any iOS application is to run it through a proxy so you can examine and potentially modify traffic going from the device to its remote endpoint. Most iOS security testers I know use BurpSuite[1] for this purpose.

1. *http://www.portswigger.net*

Bypassing TLS Validation

There's one major catch to running an app under test through a proxy: iOS resolutely refuses to continue TLS/SSL connections when it cannot authenticate the server's certificate, as well it should. This is, of course, the correct behavior, but your proxy-based testing will screech to a halt rather quickly if iOS can't authenticate your proxy's certificate.

For BurpSuite specifically, you can obtain a CA certificate simply by configuring your device or iOS Simulator to use Burp as a proxy and then browsing to *http://burp/cert/* in Mobile Safari. This should work either on a real device or in the iOS Simulator. You can also install CA certificates onto a physical device by either emailing them to yourself or navigating to them on a web server.

For the iOS Simulator, a more general approach that works with almost any web proxy is to add the fingerprint of your proxy software's CA certificate directly into the iOS Simulator trust store. The trust store is a SQLite database, making it slightly more cumbersome to edit than typical certificate bundles. I recommend writing a script to automate this task. If you want to see an example to get you started, Gotham Digital Science has already created a Python script that does the job. You'll find the script here: *https://github.com/GDSSecurity/Add-Trusted-Certificate-to-iOS-Simulator/*.

To use this script, you need to obtain the CA certificate you want to install into the trust store. First configure Firefox[2] to use your local proxy (127.0.0.1, port 8080 for Burp). Then attempt to visit any SSL site; you should get a familiar certificate warning. Navigate to **Add Exception** → **View** → **Details** and click the **PortSwigger CA** entry, as shown in Figure 4-1.

Click **Export** and follow the prompts. Once you've saved the CA certificate, open *Terminal.app* and run the Python script to add the certificate to the store as follows:

```
$ python ./add_ca_to_iossim.py ~/Downloads/PortSwiggerCA.pem
```

Unfortunately, at the time of writing, there isn't a native way to configure the iOS Simulator to go through an HTTP proxy without also routing the rest of your system through the proxy. Therefore, you'll need to configure the proxy in your host system's Preferences, as shown in Figure 4-2.

If you're using the machine for both testing and other work activities, you might consider specifically configuring other applications to go through a separate proxy, using something like FoxyProxy[3] for your browser.

2. I generally consider Chrome a more secure daily browser, but the self-contained nature of Firefox does let you tweak proxy settings more conveniently.

3. *http://getfoxyproxy.org*

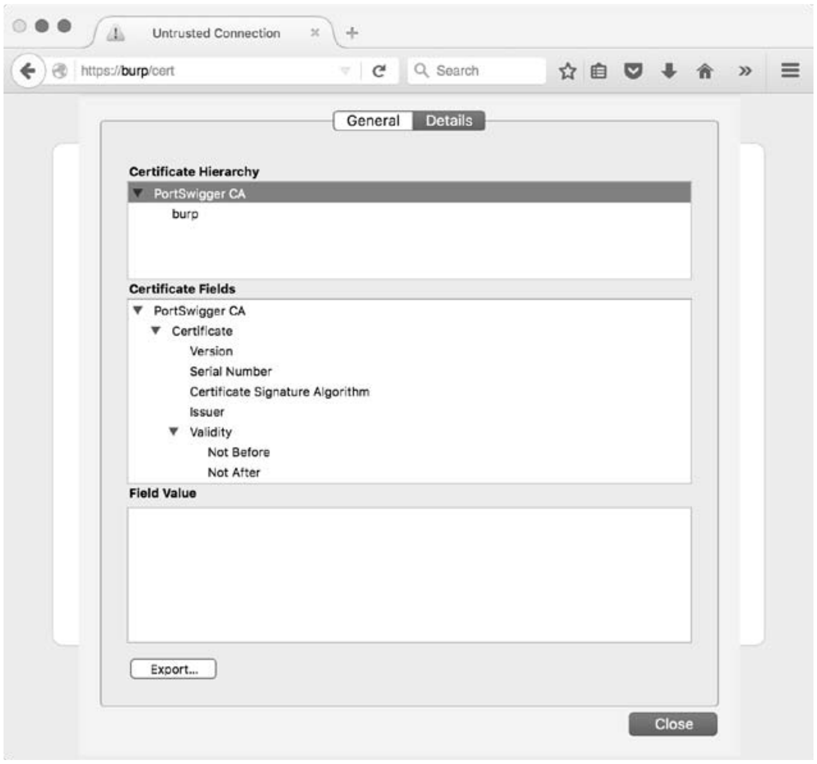
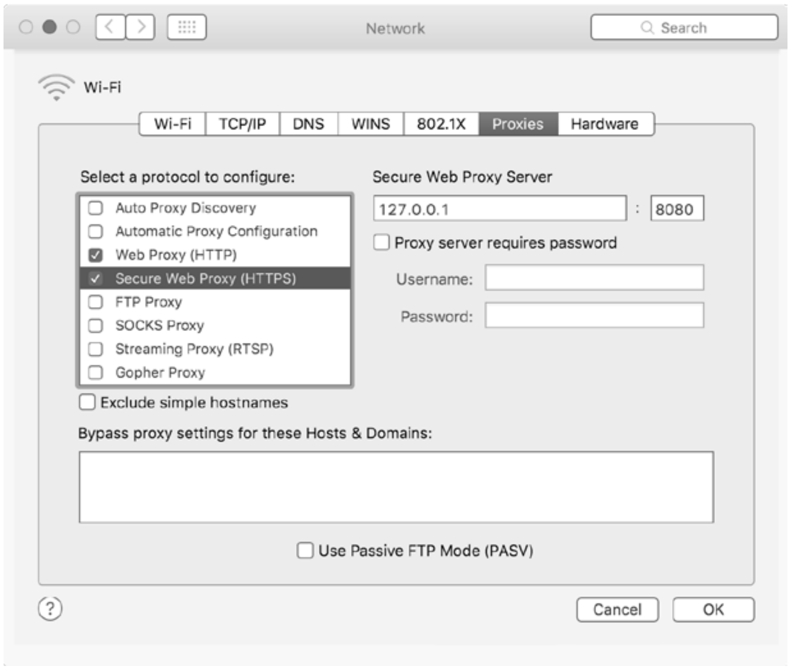


Figure 4-1: Selecting the PortSwigger CA for export



Select a protocol to configure:

- [ ] Auto Proxy Discovery
- [ ] Automatic Proxy Configuration
- [✓] Web Proxy (HTTP)
- [✓] Secure Web Proxy (HTTPS)
- [ ] FTP Proxy
- [ ] SOCKS Proxy
- [ ] Streaming Proxy (RTSP)
- [ ] Gopher Proxy

Secure Web Proxy Server: 127.0.0.1 : 8080

- [ ] Proxy server requires password

Username:

Password:

- [ ] Exclude simple hostnames

Bypass proxy settings for these Hosts & Domains:

- [ ] Use Passive FTP Mode (PASV)

Figure 4-2: Configuring the host system to connect via Burp

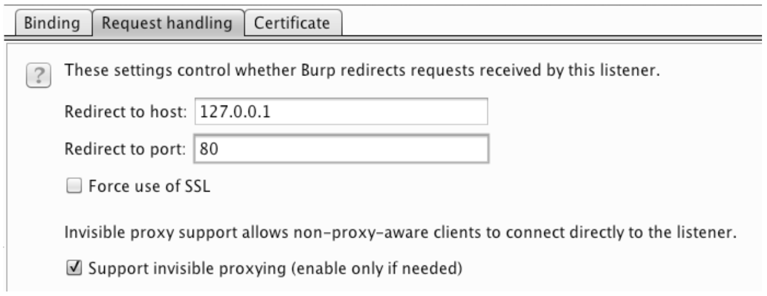
Bypassing SSL with stunnel

One method of bypassing SSL endpoint verification is to set up a termination point locally and then direct your application to use that instead. You can often accomplish this without recompiling the application, simply by modifying a plist file containing the endpoint URL.

This setup is particularly useful if you want to observe traffic easily in plaintext (for example, with Wireshark), but the Internet-accessible endpoint is available only over HTTPS. First, download and install stunnel,[4] which will act as a broker between the HTTPS endpoint and your local machine. If installed via Homebrew, stunnel's configuration file will be in */usr/local/etc/stunnel/stunnel.conf-sample*. Move or copy this file to */usr/local/etc/stunnel/stunnel.conf* and edit it to reflect the following:

```
; SSL client mode
client = yes

; service-level configuration
[https]
accept  = 127.0.0.1:80
connect = 10.10.1.50:443
TIMEOUTclose = 0
```

This simply sets up stunnel in client mode, instructing it to accept connections on your loopback interface on port 80, while forwarding them to the remote endpoint over SSL. After editing this file, set up Burp so that it uses your loopback listener as a proxy, making sure to select the **Support invisible proxying** option, as shown in Figure 4-3. Figure 4-4 shows the resulting setup.



These settings control whether Burp redirects requests received by this listener.

Redirect to host: 127.0.0.1

Redirect to port: 80

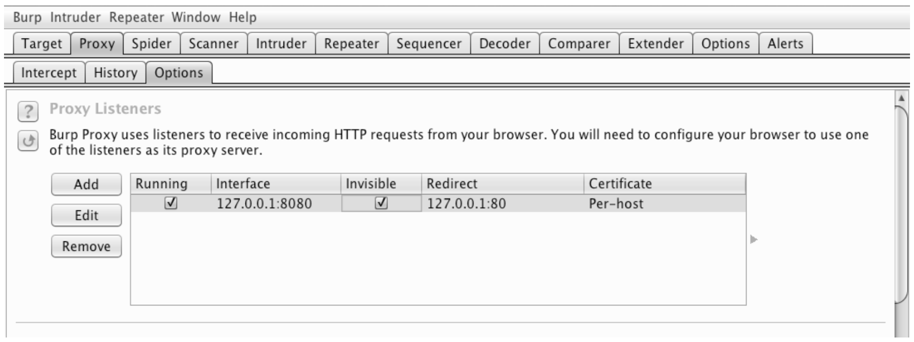
☐ Force use of SSL

Invisible proxy support allows non-proxy-aware clients to connect directly to the listener.

☑ Support invisible proxying (enable only if needed)

Figure 4-3: Setting up invisible proxying through the local stunnel endpoint

4. *http://www.stunnel.org/*


Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Options Alerts
Intercept History Options
Add
Edit
Remove

Proxy Listeners

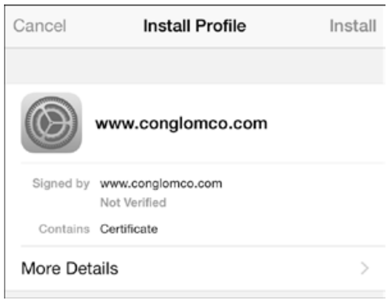
Burp Proxy uses listeners to receive incoming HTTP requests from your browser. You will need to configure your browser to use one of the listeners as its proxy server.

Running	Interface	Invisible	Redirect	Certificate
✓	127.0.0.1:8080	✓	127.0.0.1:80	Per-host

Figure 4-4: Final Burp/stunnel setup

Certificate Management on a Device

To install a certificate on a physical iOS device, simply email the certificate to an account associated with the device or put it on a public web server and navigate to it using Mobile Safari. You can then import it into the device's trust store, as shown in Figure 4-5. You can also configure your device to go through a network proxy (that is, Burp) hosted on another machine. Simply install the CA certificate (as described earlier) of the proxy onto the device and configure your proxy to listen on a network-accessible IP address, as in Figure 4-6.

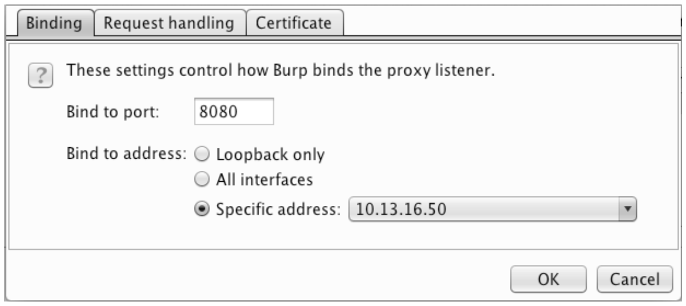


www.conglomco.com

Signed by www.conglomco.com
Not Verified

Contains Certificate

Figure 4-5: The certificate import prompt



These settings control how Burp binds the proxy listener.

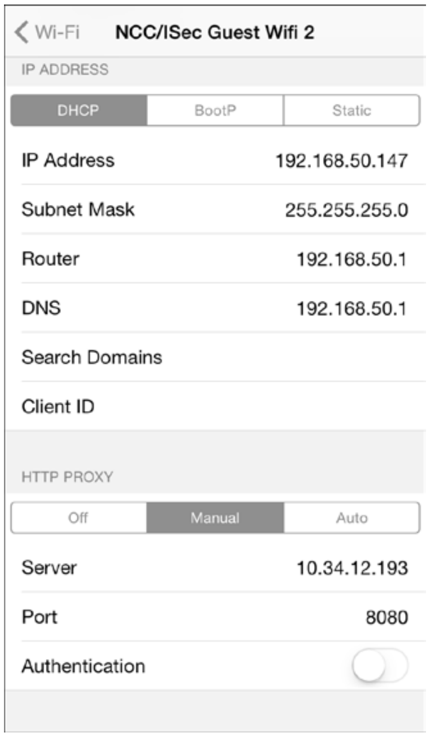
Bind to port: 8080

Bind to address: Loopback only / All interfaces / [✓ Specific address: 10.13.16.50]

Figure 4-6: Configuring Burp to use a nonlocalhost IP address

Proxy Setup on a Device

Once you've configured your certificate authorities and set up the proxy, go to **Settings** → **Network** → **Wi-Fi** and click the arrow to the right of your currently selected wireless network. You can enter the proxy address and port from this screen (see Figure 4-7).



IP Address 192.168.50.147
Subnet Mask 255.255.255.0
Router 192.168.50.1
DNS 192.168.50.1
Search Domains
Client ID

Server 10.34.12.193
Port 8080
Authentication

Figure 4-7: Configuring the device to use a test proxy on an internal network

Note that when configuring a device to use a proxy, only connections initiated by `NSURLConnection` or `NSURLSession` will obey the proxy settings; other connections such as `NSStream` and `CFStream` (which I'll discuss further in Chapter 7) will not. And of course, since this is an HTTP proxy, it works only for HTTP traffic. If you have an application using `CFStream`, you can edit the codebase with the following code snippet to route stream traffic through the same proxy as the host OS:

```
CFDictionaryRef systemProxySettings = CFNetworkCopySystemProxySettings();

CFReadStreamSetProperty(readStream, kCFStreamPropertyHTTPProxy, systemProxySettings
    );

CFWriteStreamSetProperty(writeStream, kCFStreamPropertyHTTPProxy,
    systemProxySettings);
```

If you're using `NSStream`, you can accomplish the same by casting the `NSInputStream` and `NSOutputStream` to their Core Foundation counterparts, like so:

```
CFDictionaryRef systemProxySettings = CFNetworkCopySystemProxySettings();

CFReadStreamSetProperty((CFReadStreamRef)readStream, kCFStreamPropertyHTTPProxy, (
    CFTypeRef)systemProxySettings);

CFWriteStreamSetProperty((CFWriteStreamRef)writeStream, kCFStreamPropertyHTTPProxy,
     (CFTypeRef)systemProxySettings);
```

If you're doing black-box testing and have an app that refuses to honor system proxy settings for HTTP requests, you can attempt to direct traffic through a proxy by adding a line to */etc/hosts* on the device to point name lookups to your proxy address, as shown in Listing 4-1.

```
10.50.22.11    myproxy api.testtarget.com
```

Listing 4-1: Adding a hosts file entry

You can also configure the device to use a DNS server controlled by you, which doesn't require jailbreaking your test device. One way to do this is to use Tim Newsham's dnsRedir,[5] a Python script that will provide a spoofed answer for DNS queries of a particular domain, while passing on queries for all other domains to another DNS server (by default, 8.8.8.8, but you can change this with the `-d` flag). The script can be used as follows:

```
$ dnsRedir.py 'A:www.evil.com.=1.2.3.4'
```

This should answer queries for *www.evil.com* with the IP address 1.2.3.4, where that IP address should usually be the IP address of the test machine you're proxying data through.

For non-HTTP traffic, things are a little more involved. You'll need to use a TCP proxy to intercept traffic. The aforementioned Tim Newsham has written a program that can make this simpler—the aptly named tcpprox.[6] If you use the `hosts` file method in Listing 4-1 to point the device to your proxy machine, you can then have tcpprox dynamically create SSL certificates and proxy the connection to the remote endpoint. To do this, you'll need to create a certificate authority certificate and install it on the device, as shown in Listing 4-2.

5. *https://github.com/iSECPartners/dnsRedir/*
6. *https://github.com/iSECPartners/tcpprox/*

```
$ ./prox.py -h
Usage: prox.py [opts] addr port

Options:
  -h, --help    show this help message and exit
  -6            Use IPv6
  -b BINDADDR   Address to bind to
  -L LOCPORT    Local port to listen on
  -s            Use SSL for incoming and outgoing connections
  --ssl-in      Use SSL for incoming connections
  --ssl-out     Use SSL for outgoing connections
  -3            Use SSLv3 protocol
  -T            Use TLSv1 protocol
  -C CERT       Cert for SSL
  -A AUTOCNAME  CName for Auto-generated SSL cert
  -1            Handle a single connection
  -l LOGFILE    Filename to log to

$ ./ca.py -c
$ ./pkcs12.sh ca
  (install CA cert on the device)
$ ./prox.py -s -L 8888 -A ssl.testtarget.com ssl.testtarget.com 8888
```

Listing 4-2: Creating a certificate and using tcpprox to intercept traffic

The *ca.py* script creates the signed certificate, and the *pkcs12.sh* script produces the certificate to install on the device, the same as shown in Figure 4-5. After running these and installing the certificate, your application should connect to the remote endpoint using the proxy, even for SSL connections. Once you've performed some testing, you can read the results with the *proxcat.py* script included with tcpprox, as follows:

```
$ ./proxcat.py -x log.txt
```

Once your application is connected through a proxy, you can start setting up your Xcode environment.

Xcode and Build Setup

Xcode contains a twisty maze of project configuration options—hardly anyone understands what each one does. This section takes a closer look at these options, discusses why you would or wouldn't want them, and shows you how to get Xcode to help you find bugs before they become real problems.

Make Life Difficult

First things first: treat warnings as errors. Most of the warnings generated by clang, Xcode's compiler frontend, are worth paying attention to. Not only do they often help reduce code complexity and ensure correct syntax, they also catch a number of errors that might be hard to spot, such as signedness issues or format string flaws. For example, consider the following:

```
- (void) validate:(NSArray*) someTribbles withValue:(NSInteger) desired {

    if (desired > [someTribbles count]) {
        [self allocateTribblesWithNumberOfTribbles:desired];
    }
}
```

The `count` method of `NSArray` returns an unsigned integer, (`NSUInteger`). If you were expecting the number of desired tribbles from user input, a submitted value might be –1, presumably indicating that the user would prefer to have an anti-tribble. Because `desired` is an integer being compared to an unsigned integer, the compiler will treat both as unsigned integers. Therefore, this method would unexpectedly allocate an absurd number of tribbles because –1 is an extremely large number when converted to an unsigned integer. I'll discuss this type of integer overflow issue further in Chapter 11.

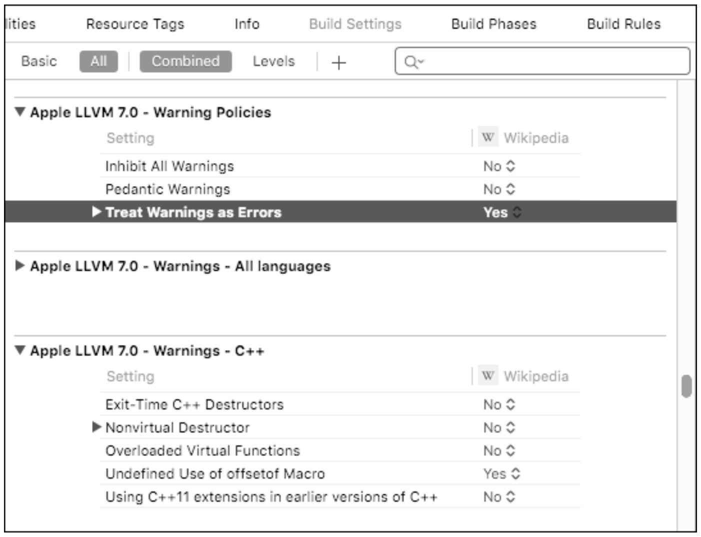
You can have clang flag this type of of bug by enabling most warnings and treating them as errors, in which case your build would fail with a message indicating `"Comparison of integers of different signs: 'int' and 'NSUInteger' (aka 'unsigned int')"`.

NOTE *In general, you should enable all warnings in your project build configuration and promote warnings to errors so that you are forced to deal with bugs as early as possible in the development cycle.*

You can enable these options in your project and target build settings. To do so, first, under Warning Policies, set Treat Warnings as Errors to **Yes** (Figure 4-8). Then, under the Warnings sections, turn on all the desired options.

Note that not every build warning that clang supports has an exposed toggle in the Xcode UI. To develop in "hard mode," you can add the `-Wextra` or `-Weverything` flag, as in Figure 4-9. Not all warnings will be useful, but it's best to try to understand exactly what an option intends to highlight before disabling it.

`-Weverything`, used in Figure 4-9, is probably overkill unless you're curious about clang internals; `-Wextra` is normally sufficient. To save you a bit of time, Table 4-1 discusses two warnings that are almost sure to get in your way (or that are just plain bizarre).



▼ Apple LLVM 7.0 - Warning Policies

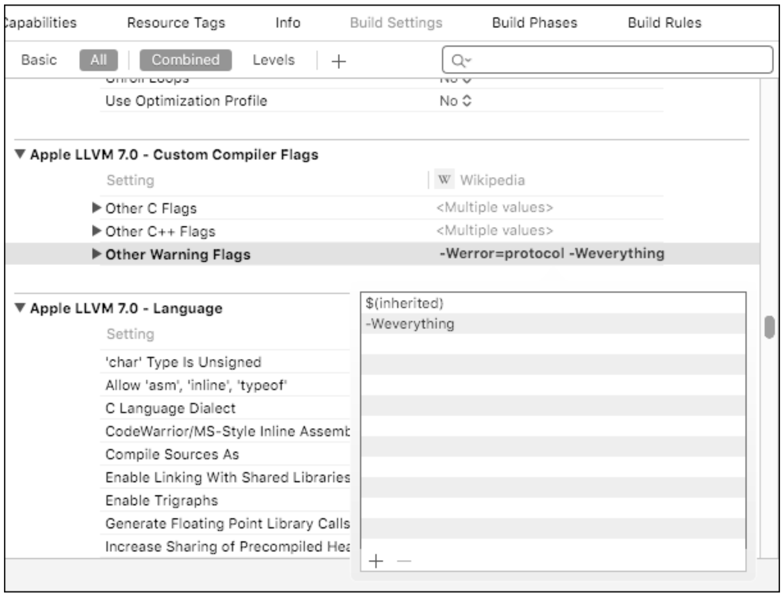
Setting	Wikipedia
Inhibit All Warnings	No
Pedantic Warnings	No
▶ **Treat Warnings as Errors**	**Yes**

▶ Apple LLVM 7.0 - Warnings - All languages

▼ Apple LLVM 7.0 - Warnings - C++

Setting	Wikipedia
Exit-Time C++ Destructors	No
▶ Nonvirtual Destructor	No
Overloaded Virtual Functions	No
Undefined Use of offsetof Macro	Yes
Using C++11 extensions in earlier versions of C++	No

Figure 4-8: Treating all warnings as errors



Use Optimization Profile	No

▼ Apple LLVM 7.0 - Custom Compiler Flags

Setting	Wikipedia
▶ Other C Flags	<Multiple values>
▶ Other C++ Flags	<Multiple values>
▶ **Other Warning Flags**	**-Werror=protocol -Weverything**

```
$(inherited)
-Weverything
```

▼ Apple LLVM 7.0 - Language

Setting
'char' Type Is Unsigned
Allow 'asm', 'inline', 'typeof'
C Language Dialect
CodeWarrior/MS-Style Inline Assemb
Compile Sources As
Enable Linking With Shared Libraries
Enable Trigraphs
Generate Floating Point Library Calls
Increase Sharing of Precompiled Hea

Figure 4-9: This setting enables all warnings, including options for which there is no exposed UI.

Table 4-1: Obnoxious Warnings to Disable in Xcode

Compiler warning	Justification for disabling
Implicit synthesized properties	Since property synthesis is now automatic, this isn't really an error unless your development guidelines require explicit synthesis.
Unused parameters/functions/variables etc.	These can be supremely irritating when writing code, since your code is obviously not completely implemented yet. Consider enabling these only for nondebug builds.

Enabling Full ASLR

In iOS 4.3, Apple introduced *address space layout randomization (ASLR)*. ASLR ensures that the in-memory structure of the program and its data (libraries, the main executable, stack and heap, and memory-mapped files) are loaded into less predictable locations in the virtual address space. This makes code execution exploits more difficult because many rely on referencing the virtual addresses of specific library calls, as well as referencing data on the stack or heap.

For this to be fully effective, however, the application must be built as a *position-independent executable (PIE)*, which instructs the compiler to build machine code that can function regardless of its location in memory. Without this option, the location of the base executable and the stack will remain the same, even across reboots,[7] making an attacker's job much easier.

To ensure that full ASLR with PIE is enabled, check that Deployment Target in your Target's settings is set to at least iOS version 4.3. In your project's Build Settings, ensure that Generate Position-Dependent Code is set to No and that the bizarrely named Don't Create Position Independent Executable is also set to No. So don't create position-independent executables. Got it?

For black-box testing or to ensure that your app is built with ASLR correctly, you can use otool on the binary, as follows:

```
$ unzip MyApp.ipa
$ cd Payload/MyApp.app
$ otool -vh MyApp

MyApp (architecture armv7):
Mach header
      magic cputype cpusubtype caps    filetype ncmds sizeofcmds            flags
   MH_MAGIC     ARM         V7 0x00     EXECUTE    21       2672 NOUNDEFS DYLDLINK
                                                                      TWOLEVEL PIE
```

7. *http://www.trailofbits.com/resources/ios4_security_evaluation_paper.pdf*

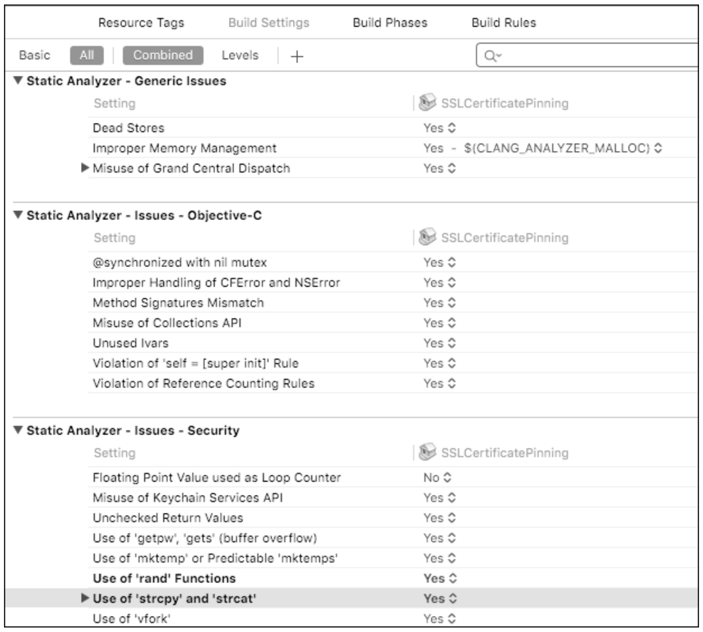
```
MyApp (architecture armv7s):
Mach header
      magic cputype cpusubtype caps    filetype ncmds sizeofcmds            flags
   MH_MAGIC     ARM        V7S 0x00     EXECUTE    21       2672  NOUNDEFS DYLDLINK
                                                                       TWOLEVEL PIE
```

At the end of each `MH_MAGIC` line, if you have your settings correct, you should see the `PIE` flag, highlighted in bold. (Note that this must be done on a binary compiled for an iOS device and will not work when used on iOS Simulator binaries.)

Clang and Static Analysis

In computer security, *static analysis* generally refers to using tools to analyze a codebase and identify security flaws. This could involve identifying dangerous APIs, or it might include analyzing data flow through the program to identify the potentially unsafe handling of program inputs. As part of the build tool chain, clang is a good spot to embed static analysis language.

Beginning with Xcode 3.2, clang's static analyzer[8] has been integrated with Xcode, providing users with a UI to trace logic, coding flaws, and general API misuse. While clang's static analyzer is handy, several of its important features are disabled by default in Xcode. Notably, the checks for classic dangerous C library functions, such as `strcpy` and `strcat`, are oddly absent. Enable these in your Project or Target settings, as in Figure 4-10.



Static Analyzer - Generic Issues

Setting	SSLCertificatePinning
Dead Stores	Yes
Improper Memory Management	Yes - $(CLANG_ANALYZER_MALLOC)
Misuse of Grand Central Dispatch	Yes

Static Analyzer - Issues - Objective-C

Setting	SSLCertificatePinning
@synchronized with nil mutex	Yes
Improper Handling of CFError and NSError	Yes
Method Signatures Mismatch	Yes
Misuse of Collections API	Yes
Unused Ivars	Yes
Violation of 'self = [super init]' Rule	Yes
Violation of Reference Counting Rules	Yes

Static Analyzer - Issues - Security

Setting	SSLCertificatePinning
Floating Point Value used as Loop Counter	No
Misuse of Keychain Services API	Yes
Unchecked Return Values	Yes
Use of 'getpw', 'gets' (buffer overflow)	Yes
Use of 'mktemp' or Predictable 'mktemps'	Yes
Use of 'rand' Functions	**Yes**
Use of 'strcpy' and 'strcat'	**Yes**
Use of 'vfork'	Yes

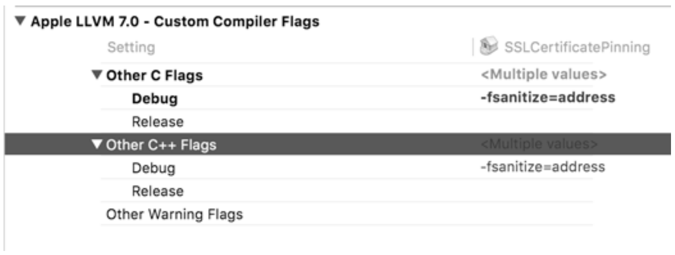
Figure 4-10: Enabling all clang static analysis checks in Xcode

8. *http://clang-analyzer.llvm.org/*

Address Sanitizer and Dynamic Analysis

Recent versions of Xcode include a version of clang/llvm that features the Address Sanitizer (ASan). ASan is a dynamic analysis tool similar to Valgrind, but ASan runs faster and has improved coverage.[9] ASan tests for stack and heap overflows and use-after-free bugs, among other things, to help you track down crucial security flaws. It does have a performance impact (program execution is estimated to be roughly two times slower), so don't enable it on your release builds, but it should be perfectly usable during testing, quality assurance, or fuzzing runs.

To enable ASan, add `-fsanitize=address` to your compiler flags for debug builds (see Figure 4-11). On any unsafe crashes, ASan should write extra debug information to the console to help you determine the nature and severity of the issues. In conjunction with fuzzing,[10] ASan can be a great help in pinning down serious issues that may be security-sensitive and in giving an idea of their exploitability.



▼ Apple LLVM 7.0 - Custom Compiler Flags	
Setting	SSLCertificatePinning
▼ Other C Flags	<Multiple values>
Debug	-fsanitize=address
Release	
▼ Other C++ Flags	<Multiple values>
Debug	-fsanitize=address
Release	
Other Warning Flags	

Figure 4-11: Setting the ASan compiler flags

Monitoring Programs with Instruments

Regardless of whether you're analyzing someone else's application or trying to improve your own, the DTrace-powered Instruments tool is extremely helpful for observing an app's activity on a fine-grained level. This tool is useful for monitoring network socket usage, finding memory allocation issues, and watching filesystem interactions. Instruments can be an excellent tool for discovering what objects an application stores on local storage in order to find places where sensitive information might leak; I use it in that way frequently.

Activating Instruments

To use Instruments on an application from within Xcode, hold down the **Run** button and select the **Build for Profiling** option (see Figure 4-12). After building, you will be presented with a list of preconfigured templates tailored for monitoring certain resources, such as disk reads and writes, memory allocations, CPU usage, and so on.

9. *http://clang.llvm.org/docs/AddressSanitizer.html*

10. *http://blog.chromium.org/2012/04/fuzzing-for-security.html*

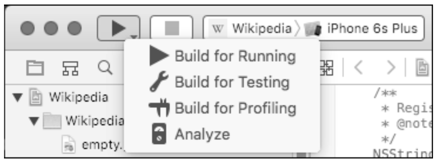
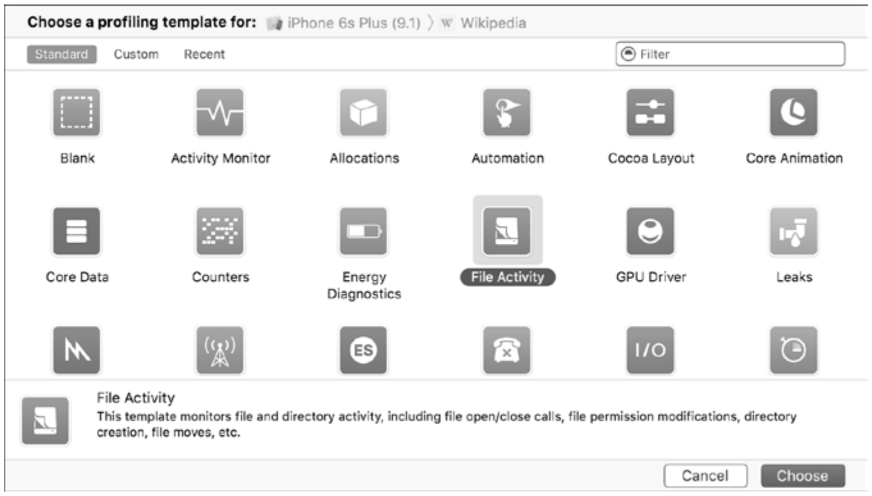


Figure 4-12: Selecting the Build for Profiling option

The File Activity template (shown in Figure 4-13) will help you monitor your application's disk I/O operations. After selecting the template, the iOS Simulator should automatically launch your application and begin recording its activity.



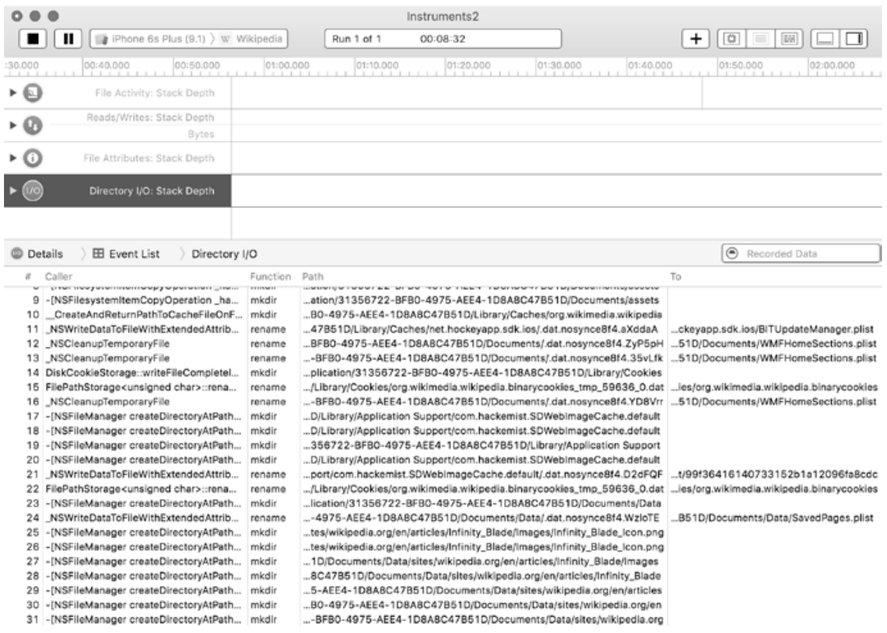
File Activity
This template monitors file and directory activity, including file open/close calls, file permission modifications, directory creation, file moves, etc.

Figure 4-13: Selecting the File Activity profiling template

There are a few preset views in Instruments for monitoring file activity. A good place to start is Directory I/O, which will capture all file creation or deletion events. Test your application the way you normally would and watch the output here. Each event is listed with its Objective-C caller, the C function call underlying it, the file's full path, and its new path if the event is a rename operation.

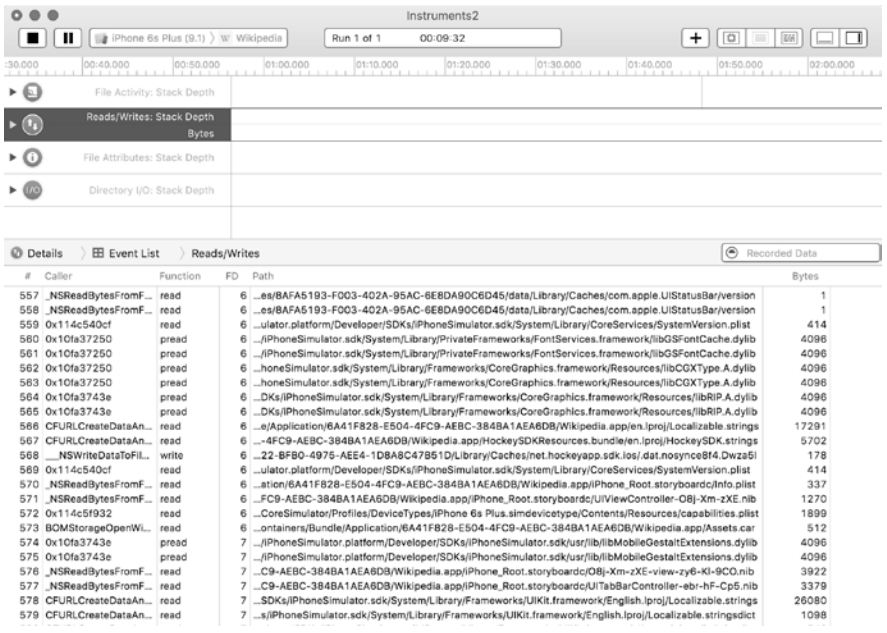
You'll likely notice several types of cache files being written here (see Figure 4-14), as well as cookies or documents your application has been asked to open. If you suspend your application, you should see the application screenshot written to disk, which I'll discuss in Chapter 10.

For a more detailed view, you can select the Reads/Writes view, as shown in Figure 4-15. This will show any read or write operations on files or sockets, along with statistics on the amount of data read or written.


Instruments2
iPhone 6s Plus (9.1) 〉 Wikipedia
Run 1 of 1 00:08:32
File Activity: Stack Depth
Reads/Writes: Stack Depth
Bytes
File Attributes: Stack Depth
Directory I/O: Stack Depth
Details 〉 Event List 〉 Directory I/O
Recorded Data

#	Caller	Function	Path	To
9	-[NSFilesystemItemCopyOperation _ha...	mkdir	...ation/31356722-BFB0-4975-AEE4-1D8A8C47B51D/Documents/assets	
10	__CreateAndReturnPathToCacheFileOnF...	mkdir	...B0-4975-AEE4-1D8A8C47B51D/Library/Caches/org.wikimedia.wikipedia	
11	_NSWriteDataToFileWithExtendedAttrib...	rename	...47B51D/Library/Caches/net.hockeyapp.sdk.ios/.dat.nosynce8f4.aXddaA	...ckeyapp.sdk.ios/BITUpdateManager.plist
12	_NSCleanupTemporaryFile	rename	...BFB0-4975-AEE4-1D8A8C47B51D/Documents/.dat.nosynce8f4.ZyP5pH	...51D/Documents/WMFHomeSections.plist
13	_NSCleanupTemporaryFile	rename	...-BFB0-4975-AEE4-1D8A8C47B51D/Documents/.dat.nosynce8f4.35vLfk	...51D/Documents/WMFHomeSections.plist
14	DiskCookieStorage::writeFileCompletel...	mkdir	...plication/31356722-BFB0-4975-AEE4-1D8A8C47B51D/Library/Cookies	
15	FilePathStorage<unsigned char>::rena...	rename	.../Library/Cookies/org.wikimedia.wikipedia.binarycookies_tmp_59636_0.dat	...ies/org.wikimedia.wikipedia.binarycookies
16	_NSCleanupTemporaryFile	rename	...-BFB0-4975-AEE4-1D8A8C47B51D/Documents/.dat.nosynce8f4.YD8Vrr	...51D/Documents/WMFHomeSections.plist
17	-[NSFileManager createDirectoryAtPath...	mkdir	...D/Library/Application Support/com.hackemist.SDWebImageCache.default	
18	-[NSFileManager createDirectoryAtPath...	mkdir	...D/Library/Application Support/com.hackemist.SDWebImageCache.default	
19	-[NSFileManager createDirectoryAtPath...	mkdir	...356722-BFB0-4975-AEE4-1D8A8C47B51D/Library/Application Support	
20	-[NSFileManager createDirectoryAtPath...	mkdir	...D/Library/Application Support/com.hackemist.SDWebImageCache.default	
21	_NSWriteDataToFileWithExtendedAttrib...	rename	...port/com.hackemist.SDWebImageCache.default/.dat.nosynce8f4.D2dFQF	...t/99f36416140733152b1a12096fa8cdc
22	FilePathStorage<unsigned char>::rena...	rename	.../Library/Cookies/org.wikimedia.wikipedia.binarycookies_tmp_59636_0.dat	...ies/org.wikimedia.wikipedia.binarycookies
23	-[NSFileManager createDirectoryAtPath...	mkdir	...lication/31356722-BFB0-4975-AEE4-1D8A8C47B51D/Documents/Data	
24	_NSWriteDataToFileWithExtendedAttrib...	rename	...-4975-AEE4-1D8A8C47B51D/Documents/Data/.dat.nosynce8f4.WzloTE	...B51D/Documents/Data/SavedPages.plist
25	-[NSFileManager createDirectoryAtPath...	mkdir	...tes/wikipedia.org/en/articles/Infinity_Blade/Images/Infinity_Blade_Icon.png	
26	-[NSFileManager createDirectoryAtPath...	mkdir	...tes/wikipedia.org/en/articles/Infinity_Blade/Images/Infinity_Blade_Icon.png	
27	-[NSFileManager createDirectoryAtPath...	mkdir	...1D/Documents/Data/sites/wikipedia.org/en/articles/Infinity_Blade/Images	
28	-[NSFileManager createDirectoryAtPath...	mkdir	...8C47B51D/Documents/Data/sites/wikipedia.org/en/articles/Infinity_Blade	
29	-[NSFileManager createDirectoryAtPath...	mkdir	...5-AEE4-1D8A8C47B51D/Documents/Data/sites/wikipedia.org/en/articles	
30	-[NSFileManager createDirectoryAtPath...	mkdir	...B0-4975-AEE4-1D8A8C47B51D/Documents/Data/sites/wikipedia.org/en	
31	-[NSFileManager createDirectoryAtPath...	mkdir	...-BFB0-4975-AEE4-1D8A8C47B51D/Documents/Data/sites/wikipedia.org	

Figure 4-14: Directory I/O view showing files created or deleted


Instruments2
iPhone 6s Plus (9.1) 〉 Wikipedia
Run 1 of 1 00:09:32
File Activity: Stack Depth
Reads/Writes: Stack Depth
Bytes
File Attributes: Stack Depth
Directory I/O: Stack Depth
Details 〉 Event List 〉 Reads/Writes
Recorded Data

#	Caller	Function	FD	Path	Bytes
557	_NSReadBytesFromF...	read	6	...es/BAFA5193-F003-402A-95AC-6E8DA90C6D45/data/Library/Caches/com.apple.UIStatusBar/version	1
558	_NSReadBytesFromF...	read	6	...es/BAFA5193-F003-402A-95AC-6E8DA90C6D45/data/Library/Caches/com.apple.UIStatusBar/version	1
559	0x114c540cf	read	6	...ulator.platform/Developer/SDKs/iPhoneSimulator.sdk/System/Library/CoreServices/SystemVersion.plist	414
560	0x10fa37250	pread	6	.../iPhoneSimulator.sdk/System/Library/PrivateFrameworks/FontServices.framework/libGSFontCache.dylib	4096
561	0x10fa37250	pread	6	.../iPhoneSimulator.sdk/System/Library/PrivateFrameworks/FontServices.framework/libGSFontCache.dylib	4096
562	0x10fa37250	pread	6	...honeSimulator.sdk/System/Library/Frameworks/CoreGraphics.framework/Resources/libCGXType.A.dylib	4096
563	0x10fa37250	pread	6	...honeSimulator.sdk/System/Library/Frameworks/CoreGraphics.framework/Resources/libCGXType.A.dylib	4096
564	0x10fa3743e	pread	6	...DKs/iPhoneSimulator.sdk/System/Library/Frameworks/CoreGraphics.framework/Resources/libRIP.A.dylib	4096
565	0x10fa3743e	pread	6	...DKs/iPhoneSimulator.sdk/System/Library/Frameworks/CoreGraphics.framework/Resources/libRIP.A.dylib	4096
566	CFURLCreateDataAn...	read	6	...e/Application/6A41F828-E504-4FC9-AEBC-384BA1AEA6DB/Wikipedia.app/en.lproj/Localizable.strings	17291
567	CFURLCreateDataAn...	read	6	...-4FC9-AEBC-384BA1AEA6DB/Wikipedia.app/HockeySDKResources.bundle/en.lproj/HockeySDK.strings	5702
568	__NSWriteDataToFil...	write	6	...22-BFB0-4975-AEE4-1D8A8C47B51D/Library/Caches/net.hockeyapp.sdk.ios/.dat.nosynce8f4.Dwza5l	178
569	0x114c540cf	read	6	...ulator.platform/Developer/SDKs/iPhoneSimulator.sdk/System/Library/CoreServices/SystemVersion.plist	414
570	_NSReadBytesFromF...	read	6	...ation/6A41F828-E504-4FC9-AEBC-384BA1AEA6DB/Wikipedia.app/iPhone_Root.storyboardc/Info.plist	337
571	_NSReadBytesFromF...	read	6	...FC9-AEBC-384BA1AEA6DB/Wikipedia.app/iPhone_Root.storyboardc/UIViewController-O8j-Xm-zXE.nib	1270
572	0x114c5f932	read	6	...CoreSimulator/Profiles/DeviceTypes/iPhone 6s Plus.simdevicetype/Contents/Resources/capabilities.plist	1899
573	BOMStorageOpenWi...	read	6	...ontainers/Bundle/Application/6A41F828-E504-4FC9-AEBC-384BA1AEA6DB/Wikipedia.app/Assets.car	512
574	0x10fa3743e	pread	7	.../iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/lib/libMobileGestaltExtensions.dylib	4096
575	0x10fa3743e	pread	7	.../iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/lib/libMobileGestaltExtensions.dylib	4096
576	_NSReadBytesFromF...	read	7	...C9-AEBC-384BA1AEA6DB/Wikipedia.app/iPhone_Root.storyboardc/O8j-Xm-zXE-view-zy6-KI-9CO.nib	3922
577	_NSReadBytesFromF...	read	7	...C9-AEBC-384BA1AEA6DB/Wikipedia.app/iPhone_Root.storyboardc/UITabBarController-ebr-hF-Cp5.nib	3379
578	CFURLCreateDataAn...	read	7	...SDKs/iPhoneSimulator.sdk/System/Library/Frameworks/UIKit.framework/English.lproj/Localizable.strings	26080
579	CFURLCreateDataAn...	read	7	...s/iPhoneSimulator.sdk/System/Library/Frameworks/UIKit.framework/English.lproj/Localizable.stringsdict	1098

Figure 4-15: Profiling results showing detailed file reads and writes

Watching Filesystem Activity with Watchdog

Instruments should catch most iOS Simulator activity, but some file writes or network calls may actually be performed by other system services, thereby escaping the tool's notice. It's a good idea to manually inspect the iOS Simulator's directory tree to get a better feel for the structure of iOS and its applications and to catch application activity that you might otherwise miss.

One easy way to automate this is to use the Python watchdog module.[11] Watchdog will use either the kqueue or FSEvents API to monitor directory trees for file activity and can either log events or take specific actions when these events occur. To install watchdog, use the following:

```
$ pip install watchdog
```

You can write your own scripts to use watchdog's functionality, but you'll find a nice command line tool already included with watchdog called watchmedo. If you open a Terminal window and navigate to the Simulator directory, you should be able to use watchmedo to monitor all file changes under the iOS Simulator's directory tree, as follows:

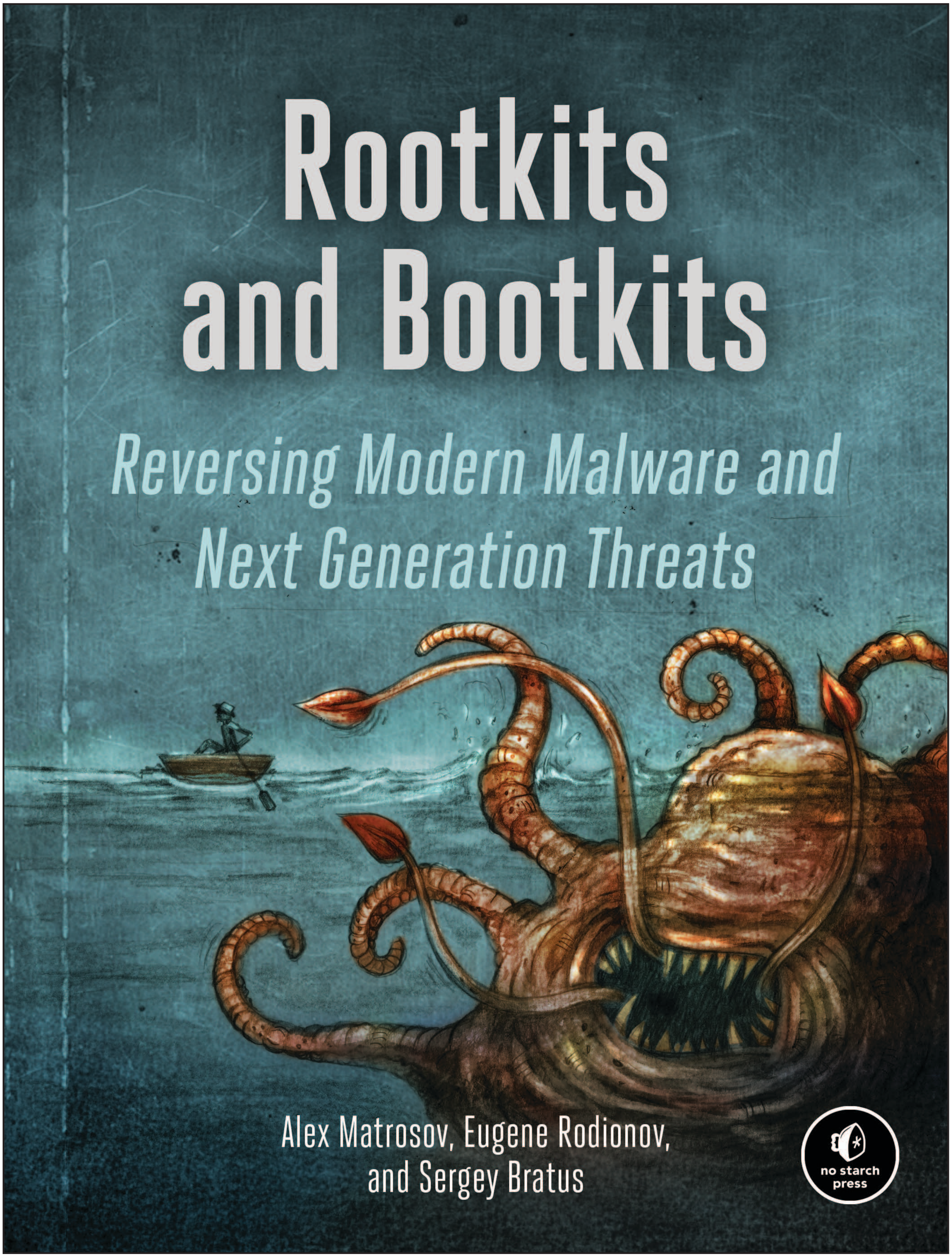
```
$ cd ~/Library/Application\ Support/iPhone\ Simulator/6.1
$ watchmedo log --recursive .
on_modified(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
    DirModifiedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
    Simulator/6.1/Library/Preferences>)
on_created(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
    FileCreatedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
    Simulator/6.1/Applications/9460475C-B94A-43E8-89C0-285DD036DA7A/Library/Caches
    /Snapshots/com.yourcompany.UICatalog/UIApplicationAutomaticSnapshotDefault-
    Portrait.png>)
on_modified(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
    DirModifiedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
    Simulator/6.1/Applications/9460475C-B94A-43E8-89C0-285DD036DA7A/Library/Caches
    /Snapshots>)
on_created(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
    DirCreatedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
    Simulator/6.1/Applications/9460475C-B94A-43E8-89C0-285DD036DA7A/Library/Caches
    /Snapshots/com.yourcompany.UICatalog>)
on_modified(self=<watchdog.tricks.LoggerTrick object at 0x103c9b190>, event=<
    DirModifiedEvent: src_path=/Users/dthiel/Library/Application Support/iPhone
    Simulator/6.1/Library/SpringBoard>)
```

11. *https://pypi.python.org/pypi/watchdog/*

Entries that start with on `on_modified` indicate a file was changed, and entries that start with `on_created` indicate a new file. There are several other change indicators you might see from watchmedo, and you can read about them in the Watchdog documentation.

Closing Thoughts

You should now have your build and test environment configured for running, modifying, and examining iOS apps. In Chapter 5, we'll take a closer look at how to debug and inspect applications dynamically, as well as how to change their behavior at runtime.



Rootkits and Bootkits

Reversing Modern Malware and Next Generation Threats

Alex Matrosov, Eugene Rodionov, and Sergey Bratus

no starch press

6
Bootkit Background and History

This chapter will introduce you to bootkits by looking at the history, evolution, and recent re-emergence of bootkit infection methods. A *bootkit* is a malicious program that infects the early stages of the system startup process before the operating system is fully loaded. They first emerged in the old days of MS-DOS (the non-graphical operating system that preceded Windows), when the default behavior of the PC BIOS was to attempt to boot from whatever disk was in the floppy drive. Infecting floppies was the simplest strategy for attackers to gain control: all it took was for the user to leave an infected floppy in the drive when powering up or rebooting the PC—which, back then, happened often. As more systems were implemented with BIOSes that allowed PC owners to change the boot order and bypass the floppy drive, the utility of infected floppies decreased. With Windows taking control of the boot process over from MS-DOS, and allowing ample opportunity for the attacker to infect drivers, executables, DLLs, and other system resources post-boot without messing with the trickier Windows boot process, bootkits became a rare and exotic option among more practical threats, to be replaced by rootkits as the primary malware threat.

This situation changed when Microsoft introduced the Kernel-Mode Code Signing Policy on 64-bit operating systems, starting with Windows Vista. Suddenly, easy loading of arbitrary code into the kernel no longer worked for the attackers. Anticipating that, attackers returned to the older methods of compromising a PC before its operating system could load—bringing bootkits back into prominence.

Bootkits have made an impressive comeback after their prominence waxed and waned (and then rebounded) with the changes in the boot process of a typical PC. The modern bootkit is making use of variations on really old approaches to stealth and persistence—the ability for malware to remain active on the targeted system for as long as possible and without the system user's knowledge. In this chapter, we'll look at the resurgence of boot-infecting malware, trace the history of their spectacular comeback, and then briefly review the history of early viruses and original methods of bootkit infection.

A New Boot Process, a New Beginning for Bootkits

The introduction of Microsoft's Kernel-Mode Code Signing Policy in Windows Vista and later 64-bit Windows turned the tables on the attackers by incorporating a new strategy for the distribution of system drivers. No longer able to inject their code into the kernel once the OS was fully loaded, attackers turned to the old BSI tricks. These tricks evolved—or, rather, co-evolved alongside boot process defenses--into new types of attacks on operating system boot loaders; a co-evolution that shows no signs of slowing down any time soon. In this section we'll look at how the Kernel-mode Code Signing Policy determined the direction of new bootkits, and then examine the timeline of the co-evolution of bootkits and bootkit Proofs-of-Concepts. In the following chapters, we'll go on to describe the details of bootkit attacks.

Bypassing Kernel-mode Code Signing Policy

The development of modern bootkits was heavily influenced by the necessity of bypassing integrity checks in modern computer systems. All known tricks for bypassing the digital signature checks introduced with Microsoft's Kernel-mode Code Signing Policy can be divided into three groups, as illustrated in Figure 6-1. The first group works entirely within user mode and is based on the system-provided methods for legitimately disabling the signing policy. The second group targets the process of booting the operating system in order to manipulate kernel-mode memory: this currently appears to be the most popular approach to bootkit development. The third group of methods is based on exploiting vulnerabilities in system firmware. In particular, there are only two ways for an unsigned driver to be loaded into the kernel: either by using an exploitable vulnerability in the system kernel or third-party driver, or by compromising the boot process and thus the entire system via a bootkit infection. In practice, malware typically makes use of the latter technique as it creates a more permanent way for penetrating into the system: once a vulnerability in a driver is patched it cannot be no longer exploited by malware while flaws in the boot processes last longer. But as more computers ship with Secure Boot protection enabled and supported by the OS, we expect to see the landscape changing once again, in the near future.

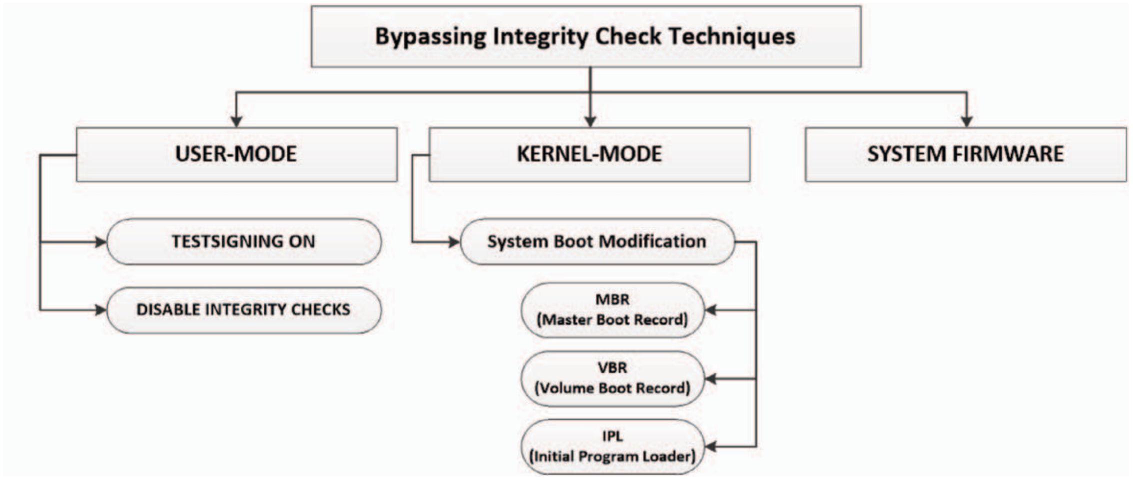


Figure 6-1: Kernel-Mode Code Signing Policy bypassing techniques

Thus, modern bootkits have taken the form of infectors that target and compromise the OS booting process.

Co-evolution of Bootkit Research and Malware

The harbinger of the first modern bootkits is generally considered to be the eEye's Proof of Concept (PoC) BootRoot[1], presented at the BlackHat conference in 2005. The BootRootKit code was an NDIS (Network Driver Interface Specification) backdoor by Derek Soeder and Ryan Permeh. It demonstrated for the first time how it was possible to use the original concepts behind boot virus infection as a model for modern operating system attacks. However, while the eEye presentation was an important step toward the development of bootkit malware, it was two years before any new malicious samples with bootkit functionality were detected in the wild.

[1] eEye BootRoot, BlackHat 2005 // http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf

The first modern bootkit detected in the wild was Mebroot[2], in 2007. The detection of Mebroot coincided with the presentation of another Proof-of-Concept, Vbootkit[3], at the BlackHat conference that same year. This Proof-of-Concept code demonstrated possible attacks on Microsoft's Windows Vista kernel by modifying the boot sector. The authors of Vbootkit released its code as an open-source project.

Mebroot was one of the most sophisticated malicious threat seen at this time. It offered a real challenge to antivirus companies because this malware used new stealth techniques for surviving after reboot. At the same time, and also at BlackHat, another Proof of Concept was released - the Stoned bootkit[4], named so in homage to the much earlier but very successful Stoned boot sector virus (BSV, an alternative acronym to BSI).

We must emphasize that these Proof-of-Concept bootkits are not the reason for the coinciding releases of malicious bootkits such as Mebroot. Rather, emergence of these Proofs-of-Concept enabled timely detection of such malware, by showing the industry what to look for. Malware developers had already been searching for new and stealthy ways to push the moment a system could be actively infected to earlier into the boot process, before security software was able to detect the presence of the infection. Had the researchers hesitated to publish their results, malware authors would have succeeded in pre-empting the system's ability to detect the new bootkit malware.

[2] Stoned Bootkit, BlackHat 2009 // http://www.blackhat.com/presentations/bh-usa-09/KLEISSNER/BHUSA09-Kleissner-StonedBootkit-PAPER.pdf
[3] Vbootkit, BlackHat 2007 // https://www.blackhat.com/presentations/bh-europe-07/Kumar/Whitepaper/bh-eu-07-Kumar-WP-apr19.pdf
[4] The Rise of MBR Rootkits // http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/your_computer_is_now_stoned.pdf

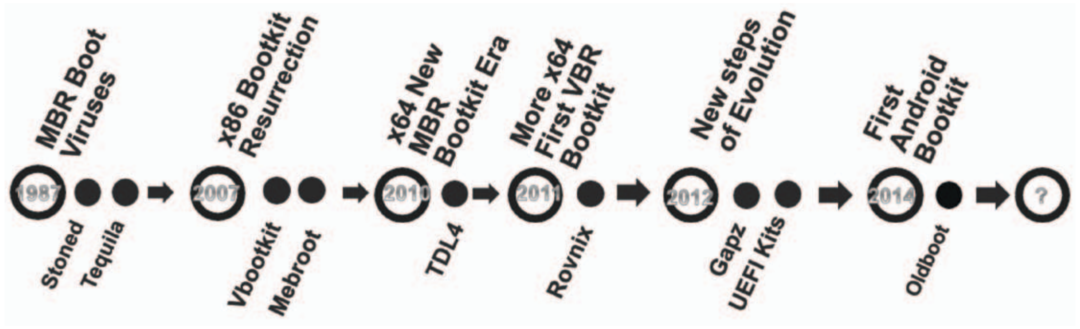


Figure 6-2: Bootkit resurrection timeline

With bootkits, as in other fields of computer security, we see the co-evolution of Proofs-of-Concept that enable us to understand and detect the threats presented by real malware samples found in the wild. The former category is developed by security researchers to demonstrate that the threats are real and should be looked for; the latter consists of unequivocally malicious threats developed by cybercriminals. Table 6.1 and Figure 6-2 show the evolution of such Proofs-of-Concepts and real malware threats side-by-side, from 2005 to 2014.

Proof of Concept Bootkits Evolution	Bootkit Threats Evolution
eEye Bootroot – 2005 The first MBR–based bootkit for MS Windows operating systems.	**Mebroot – 2007** The first MBR-based bootkit for MS Windows operating systems in the wild.
Vbootkit – 2007 The first bootkit to abuse Microsoft Windows Vista.	**Mebratix – 2008** The other malware family based on MBR infection.
Vbootkit5 x64 – 2009 The first bootkit to bypass the digital signature checks on MS Windows 7.	**Mebroot v2 – 2009** The evolved version of Mebroot malware.

5 VBootkit 2.0 – Attacking Windows 7 via Boot Sectors, HiTB 2009 // http://conference.hitb.org/hitbsecconf2009dubai/materials/D2T2%20-%20Vipin%20and%20Nitin%20Kumar%20-%20vbootkit%202.0.pdf

Stoned Bootkit – 2009 Another example of MBR-based bootkit infection.	**Olmarik (TDL4) - 2010/11** The first 64-bit bootkit in the wild.
Stoned Bootkit x64 – 2011 MBR-based bootkit supporting the infection of 64-bit operating systems.	**Olmasco (TDL4 modification) - 2011** The first VBR-based bootkit infection.
DeepBoot6 – 2011 Used interesting tricks to switch from real-mode to protected mode.	**Rovnix – 2011** The evolution of VBR based infection with polymorphic code.
Evil Core7 - 2011 Concept bootkit that used SMP (symmetric multiprocessing) for booting into protected-mode	**Mebromi – 2011** The first exploration of the concept of BIOSkits seen In the Wild.
VGA Bootkit8 – 2012 VGA based bootkit concept.	**Gapz9 – 2012** The next evolution of VBR infection
DreamBoot10 – 2013 The first public concept of UEFI bootkit.	**OldBoot11 - 2014** The first bootkit for the Android operating system in the wild.

Table 1-1: The chronological evolution of PoC bootkits versus real world bootkit threats

Bootkits on this timeline are classified by the stage of the initial boot process they subvert, as well as by the data structure they abuse for this subversion. The first such subdivision starts with the Master Boot Record (MBR), the first sector of the bootable hard drive. The MBR consists of the boot

[6] DeepBoot, Ekoparty 2011 // http://www.ekoparty.org//archive/2011/ekoparty2011_Economou-Luksenberg_Deep_Boot.pdf
[7] Evil Core Bootkit, NinjaCon 2011 // http://downloads.ninjacon.net/downloads/proceedings/2011/Ettlinger_Viehboeck-Evil_Core_Bootkit.pdf
[8] VGA Persistent Rootkit, Ekoparty 2012 // http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=publication&name=vga_persistent_rootkit
[9] Mind the Gapz: The most complex bootkit ever analyzed?// http://www.welivesecurity.com/wp-content/uploads/2013/05/gapz-bootkit-whitepaper.pdf
[10] UEFI and Dreamboot, HiTB 2013 // http://www.quarkslab.com/dl/13-04-hitb-uefi-dreamboot.pdf
[11] Oldboot: the first bootkit on Android // http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/

code and a partition table that describes the hard drive's partitioning scheme. At the very beginning of the bootup process the BIOS code reads the MBR and transfers execution flow to the code located there--if it finds the MBR correctly formatted. The main purpose of the MBR code is to locate an active partition on the disk and read its very first sector – the Volume Boot Record (VBR). The VBR contains file system-specific boot code, which is needed to load the OS boot loader's components. In fact, in Windows systems there are 15 consecutive sectors following the VBR that contain bootstrap code for the New Technology File System (NTFS) partition. These 15 sectors are refered to as Initial Program Loader (IPL). The IPL parses the NTFS file system and locates the OS boot loader components (for instance, *BOOTMGR*, the Windows Boot Manager).

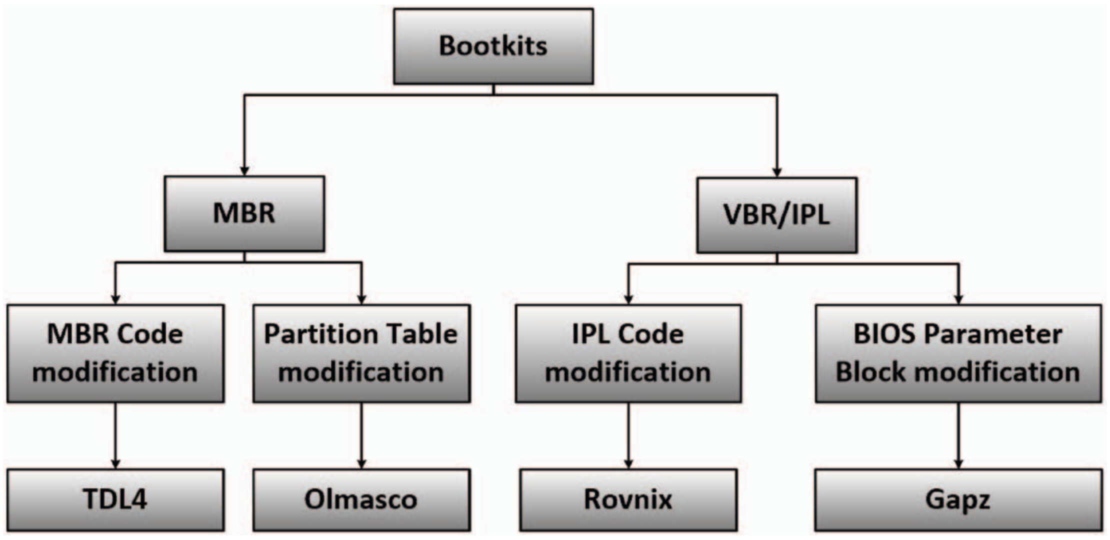


Figure 6-3: Bootkit classification by type of boot sector infection

Modern bootkits can be classified into two groups according to the type of boot sector infection employed: MBR and VBR bootkits (as shown in Figure 6-

3). The more sophisticated and stealthier bootkits we see are based on VBR infection techniques.

The control flow of the bootstrap code from the MBR to the full Windows system initialization is shown in Figure 6-4.

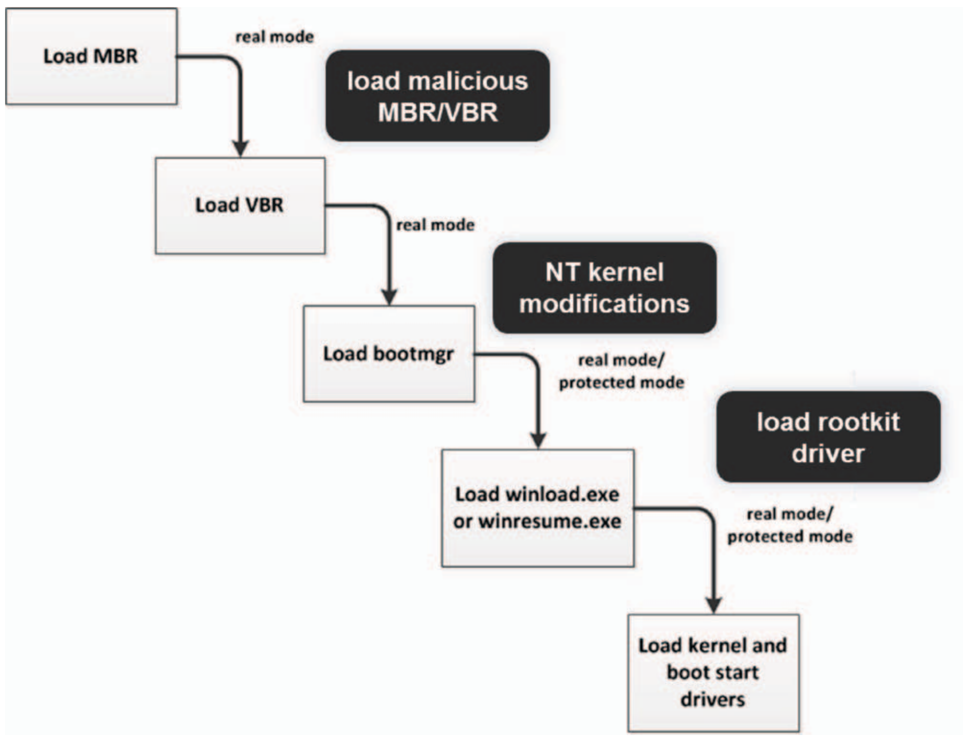

Load MBR
real mode
load malicious MBR/VBR
Load VBR
real mode
NT kernel modifications
Load bootmgr
real mode/ protected mode
load rootkit driver
Load winload.exe or winresume.exe
real mode/ protected mode
Load kernel and boot start drivers

Figure 6-4: Booting scheme of compromised operating system

Microsoft Windows operating system versions before Windows 8.x do not check the integrity of firmware, such as BIOS or UEFI, that are responsible for booting the operating system in its early stages. Before the Windows 8 operating system became available, the firmware that booted the system was by default assumed to be trustworthy—obviously, an unwarranted assumption considering the complexity the boot process has reached. Windows 8 onwards incoporated Secure Boot technology, intended to work in cooperation with modern BIOS software, in order to prevent or mitigate bootkit infections—but as any complex security technology, it has its vulnerbilities. In chapters 11 and 12 of this book we will discuss ways to bypass Secure Boot by using BIOS vulnerabilities.

The History of Bootkits and its Lessons

The history of bootkits goes back a long way, to the early IBM-PC days and even earlier. Ironically, the first IBM-PC-compatible boot sector viruses from 1987 use the same concepts and approaches as modern threats: infecting boot loaders so that malicious code is launched even before the operating system is booted.

In fact, attacks on the PC boot sector were already known from even before the days of MS-DOS. Indeed, early versions of Windows essentially ran under MS-DOS rather than running as the core operating system, and were often referred to as an operating environment rather than as an operating system. While it's unlikely that any of those prehistoric viruses are still 'in the wild' today in any meaningful sense, they have a part to play in our understanding of the development of approaches to taking over a system by compromising and hijacking the boot process.

Bootkit Pre-History

Boot Sector Infectors (BSIs) were certainly among the earliest bootkit contenders, and the first to be seen on microcomputers, but they weren't the very earliest forms of malware.

The honor of being the first virus is usually bestowed upon Creeper (1971-72), a self-replicating program running under the TENEX networked operating system on VAX PDP-10s at BBN Technologies. The first “antivirus” was a program called Reaper, dedicated to the removal of Creeper infections.

It could be argued that as these were experimental/Proof-of-Concept programs, the term ‘malware’ (MAL-icious soft-WARE) isn't really appropriate, but in fact many of the earliest viruses now unequivocally regarded as malware did no deliberate harm and were written by way of experimentation and out of curiosity, so we tend not to discriminate. Bear in mind that software doesn't really need to be consciously malicious to be illegal: software that deliberately accesses and/or modifies a system that isn’t the property of its author without permission from the system’s owner contravenes modern anti-malware legislation in many countries and jurisdictions.

Legally, this could include programs like Reaper and other software intended to counteract earlier malicious software—it's not uncommon for unequivocally malicious software to disinfect other malware, though the motivation in such cases has usually more to do with eliminating competition than concern for the wellbeing of the target system.

After Creeper came PERVADE (1975), a subroutine in the ANIMAL game, running on a UNIVAC 1100//42 mainframe that copied ANIMAL to any directory the current user had access to.

PC Floppy Flotsam & the Original PC Boot Process

Before going into the history and evolution of bootkits, we’ll look at how boot sector infectors work. In these days of optical disks and USB thumb drives it may be difficult to comprehend that early operating systems could be contained on such low capacity media as floppies, so we’ll summarize the architecture of floppy disks in order to understand the boot process better, and to see how it was manipulated by original bootkits.

Every formatted diskette had a boot sector, located in its first physical sector. Unlike hard drives, diskettes were not partitioned. On a hard drive, the boot sector is located in first logical sector. The Master Boot Record (MBR) in that first logical sector contains the partition table, specifying the hard disk type and how it is partitioned.

At bootup, the BIOS program looked for a bootable diskette to start from in drive A and ran whatever code it found in the appropriate sector. In the case of an unbootable diskette (that is, one not capable of loading the operating system), the boot sector code would simply display a *”Not a bootable disk”* message. It was all too easy to leave a diskette in the drive, and if it happened to be infected with a BSI, the diskette would infect the system even if the disk wasn't bootable, which goes some way to accounting for the early success of the boot sector infector (BSI).

'Pure' BSIs were hardware-specific and not OS-specific: if an infected floppy found itself in the drive at bootup it attempted to infect IBM-compatible PCs irrespective of what operating system was being run. This made its effect upon the targeted system somewhat unpredictable. However, malware droppers using BIOS and DOS services to install malware into the MBR were (and are) unable to do so in a Windows NT or NT-derived system (Windows 2000 and onward) unless it was set up to multiboot a less secure OS. An MBR infector that succeeded in installing on an NT or NT-derived system could locate itself in memory, but once the OS had loaded, the direct disk services provided by the BIOS were no longer available, due to NT's use of protected mode drivers, so secondary infection of diskettes was stymied.

There were other potential problems, too. If the virus didn't preserve the original boot record it could prevent the system from booting at all. BSIs that infected the DOS Boot Record (DBR) rather than the MBR (as did Form, another highly successful BSI) could prevent booting from a new technology file system (NTFS) partition, too.

The rate of BSI infection first began to decline when it became possible to change the boot order in setup so that the system would boot from the hard disk and ignore any floppy that happened to have been left in the drive. It was with the increasing take-up of modern Windows versions and the virtual disappearance of the floppy drive that the old-school BSI was finally killed off.

Apple Disorder

The first microcomputer affected by viral software seems to have been the Apple II. At that time, Apple II (sometimes written Apple][) diskettes normally contained the disk operating system. Around 1981, according to Robert Slade[12] in his first book on viruses and malware, there were versions of a viral DOS circulating after discussions about 'evolution' and 'natural selection' in pirated games at Texas A&M. In general, though, the credit for the first Apple II virus is given to Rich Skrenta's Elk Cloner (1982-3), as noted in *Viruses Revealed*[13] and in a more research-oriented book by Peter Szor [14].

Though Elk Cloner preceded PC boot sector viruses by several years, it's usually described as a boot sector infector as its method of infection was very similar. Elk Cloner modified the loaded OS by hooking itself, and stayed resident in RAM in order to infect other floppies, intercepting disk accesses and

[12] Robert Slade's Guide to Computer Viruses, Robert Slade, Springer. http://www.amazon.com/Robert-Slades-Guide-Computer-Viruses/dp/0387946632
[13] Viruses Revealed; David Harley, Robert Slade and Urs Gattiker, Osborne http://www.amazon.com/Viruses-Revealed-David-Harley/dp/B007PMOWTQ
[14] The Art of Computer Virus Research and Defense, Peter Szor, Addison Wesley http://books.google.co.uk/books/about/The_Art_of_Computer_Virus_Research_and_D.html?id=XE-ddYF6uhYC&redir_esc=y

overwriting their system boot sectors with its own code. At every 50th bootup it displayed a message (sometimes generously described as a poem):

ELK CLONER:

THE PROGRAM WITH A PERSONALITY

IT WILL GET ON ALL YOUR DISKS
IT WILL INFILTRATE YOUR CHIPS
YES, IT'S CLONER!

IT WILL STICK TO YOU LIKE GLUE
IT WILL MODIFY RAM TOO
SEND IN THE CLONER!

As David Harley wrote in an article[15] for Infosecurity Magazine, after Skrenta was interviewed for The Register in 2012[16]: "I guess it's as well that Skrenta subsequently went into the IT industry rather than embarking on a career in literature. As verse goes, that's really shaggy doggerel." Still, no verse that Harley wrote when he was in his teens has stood the test of time, either.

The later (1989) Load Runner malware, affecting Apple IIGS and ProDOS, is rarely mentioned nowadays, but it does have an interesting extra wrinkle. Apple users frequently needed to reboot to change operating systems, or sometimes to boot a 'special' disk. Load Runner's specialty was trapping the reset command triggered by the key combination CONTROL+COMMAND+RESET

[15] http://www.infosecurity-magazine.com/blog/2012/12/17/send-in-the-clones/735.aspx
[16] http://www.theregister.co.uk/2012/12/14/first_virus_elk_cloner_creator_interviewed/

and taking it as a cue to write itself to the current diskette, so that it would survive a reset. This may not be the earliest example of malware persistence, but it's certainly a precursor to more sophisticated attempts to maintain its presence.

PC © Brain Damage

We have to look ahead to 1986 for the first PC virus, usually considered to be Brain . Brain was a fairly bulky BSI, occupying the first two sectors of a diskette with its own code and marking the sectors as 'bad' so that the space wouldn't be overwritten. This meant that the boot code was moved from the first sector to the third. The version usually taken to be the 'original' did not infect hard disks, only 360k diskettes.

However, Brain had features that prefigured some of the characterizing features of modern bootkits. Firstly, the use of a hidden storage area in which to keep its own code, though on an infinitely more basic level than TDSS and its contemporaries and successors. Secondly, the use of 'bad' sectors to protect that code from legitimate housekeeping by the operating system. Thirdly, the use of a stealth technique: if the virus was active when an infected sector was accessed, it hooked the disk interrupt handler to ensure that the original, legitimate boot sector stored in sector three was displayed.

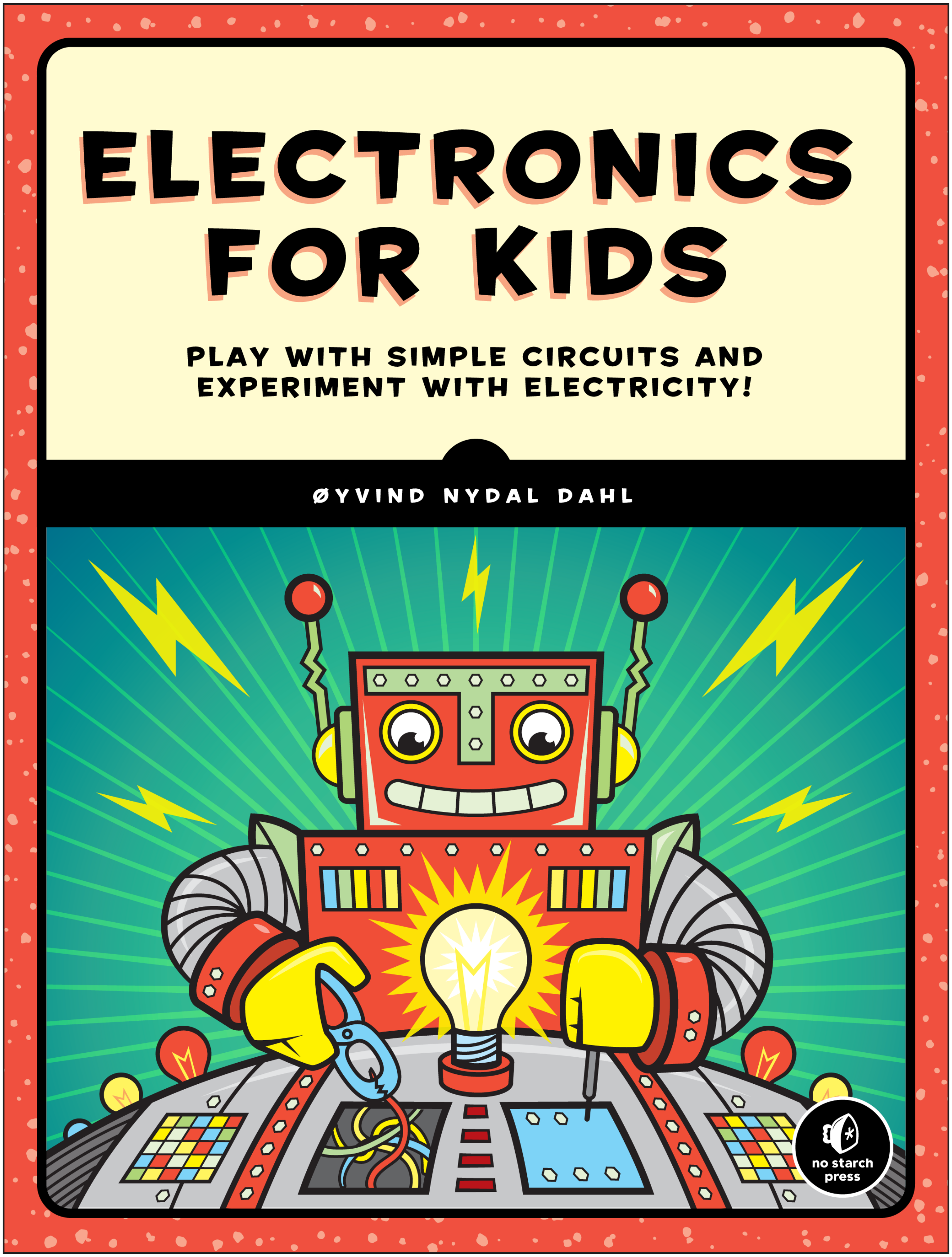
Characteristically, a boot sector virus would allocate a memory block for the use of its own code and hook the execution of the code flow there in order to infect new files or system areas (in the case of a BSI). Occasionally, multi-stage malware would use a combination of these methods; these were known as *Multipartites*.

Multipartites

Multipartite is a term used to describe malware that capable of infecting both boot sectors and files, though it isn't strictly correct to restrict the use of the term to 'file and boot' viruses. For example, there were instances of macro viruses that dropped file viruses, while there are also examples of malware that can spread both non-parasitically in worm fashion and also as file infectors. While the malware we see nowadays tends to a degree of sophistication, complexity, and modularity that would have been almost unimaginable in the 1980s and 1990s, the term has fallen largely into disuse in discussion of modern threats.

Conclusion

This chapter has been devoted to the history and evolution of boot compromises, with the intention of giving the reader a solid understanding of the basic concepts on which to build as we look at the detail of bootkit technology. In the next chapter we will be going deeper into Kernel-Mode Code Signing Policy and exploring the ways of bypassing this technology via bootkit infection, with particular reference to TDSS. The evolution of TDL3 and TDL4 neatly exemplifies the shift from user mode to kernel mode system compromise as a means of keeping the malware unnoticed but active for longer on a compromised system.

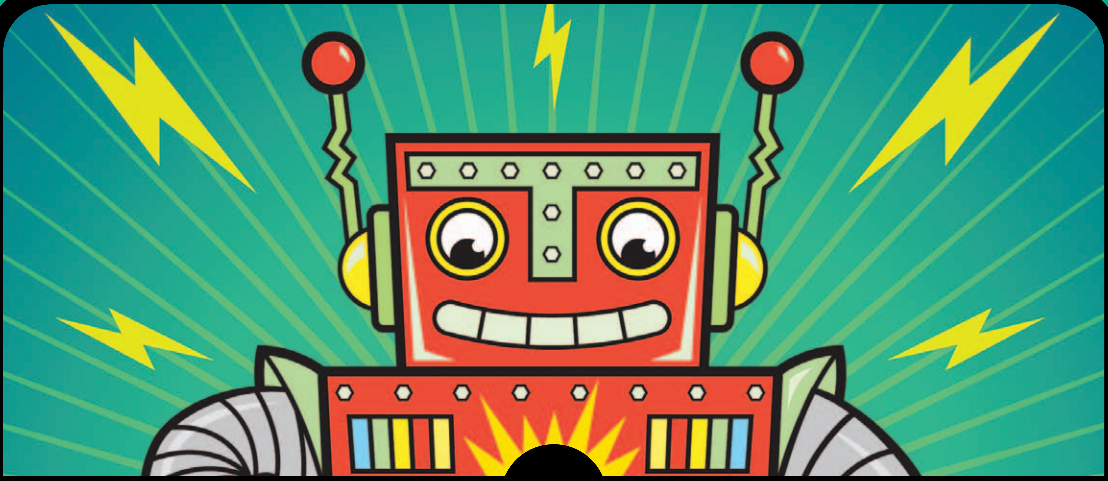


ELECTRONICS FOR KIDS

PLAY WITH SIMPLE CIRCUITS AND EXPERIMENT WITH ELECTRICITY!

ØYVIND NYDAL DAHL

no starch press



2 MAKING THINGS MOVE WITH ELECTRICITY AND MAGNETS

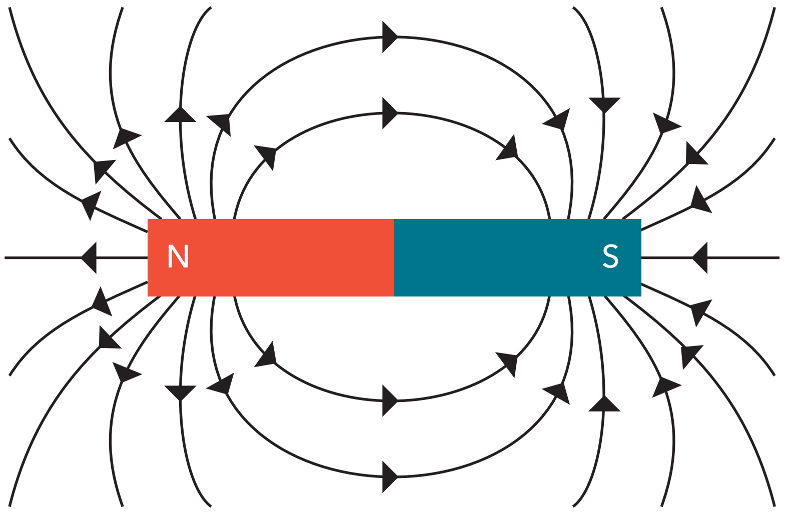
Big magnets attract small metal objects; small magnets stick to large metal objects. For example, refrigerator doors are usually big pieces of metal, so it's easy cover them with tiny, decorative magnets. You've probably seen magnets in cartoons, too: characters like to use giant horseshoe-shaped magnets to cause mischief. You can find magnets in nature or create them with electricity. A magnet created with electricity is called an *electromagnet*.

You can use an electromagnet to make things move, and you don't even have to be a superhero to do it! In fact, many things you see every day—like motors, loudspeakers, and the automatic doors in shops—work because electromagnets make something in them move.

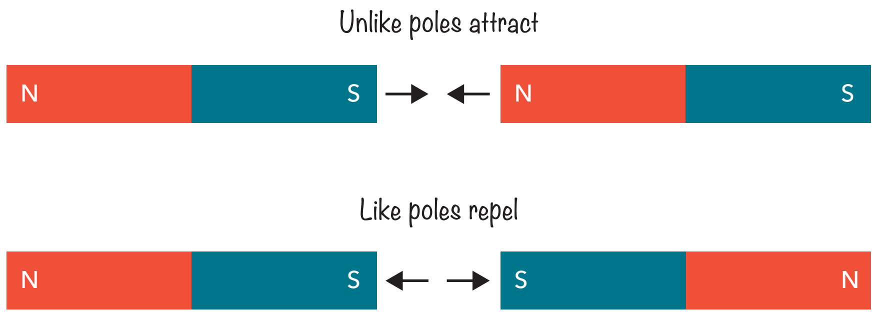
An electromagnet is very easy to make, and in this chapter, you'll build an electromagnet that you can turn on and off with a switch. Then, you'll use an electromagnet to build your very own motor!

HOW MAGNETS WORK

Magnets have two poles, the *north pole (N)* and *south pole (S)*, and they're surrounded by a magnetic field.



If you place two magnets side by side, the north pole of one magnet attracts the other magnet's south pole and repels that magnet's north pole. Try pushing two magnets together. If you don't force them, they should naturally attach to each other at their opposite poles. Now, try to force two of the same poles toward each other. That's harder, isn't it? Opposite poles are attracted to each other, and identical poles repel each other.



NOTE *Thin, flexible refrigerator magnets don't have two distinct poles. Instead they have many poles of opposite polarity next to each other, so it's harder to feel the magnets attract and repel.*

But magnets don't attract all materials. For example, plastic is unaffected by magnets. Try testing some metal objects around you!

TRY IT OUT: FIND SOME MAGNETIC OBJECTS!

Take any magnet and place it over objects made out of different materials, such as:

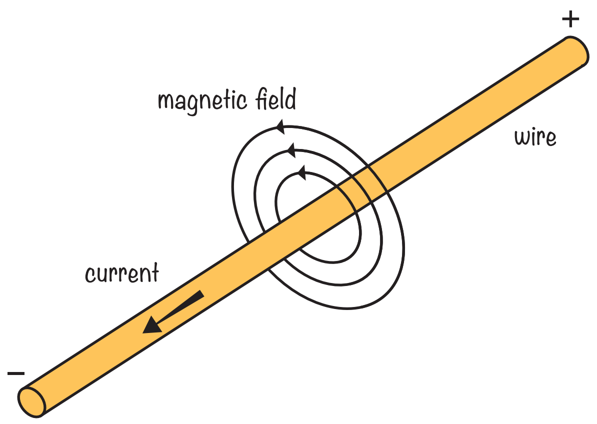
- Aluminum foil
- A stainless steel spoon
- A soda can
- An iron nail
- A piece of metal jewelry
- A few different coins

Which objects does the magnet attract or stick to? You should find that the magnet attracts some metals, but not all metals. What happens with aluminum foil?

It turns out that some metals can turn into magnets if you apply a little electricity. That's where electromagnets come in.

MEET THE ELECTROMAGNET

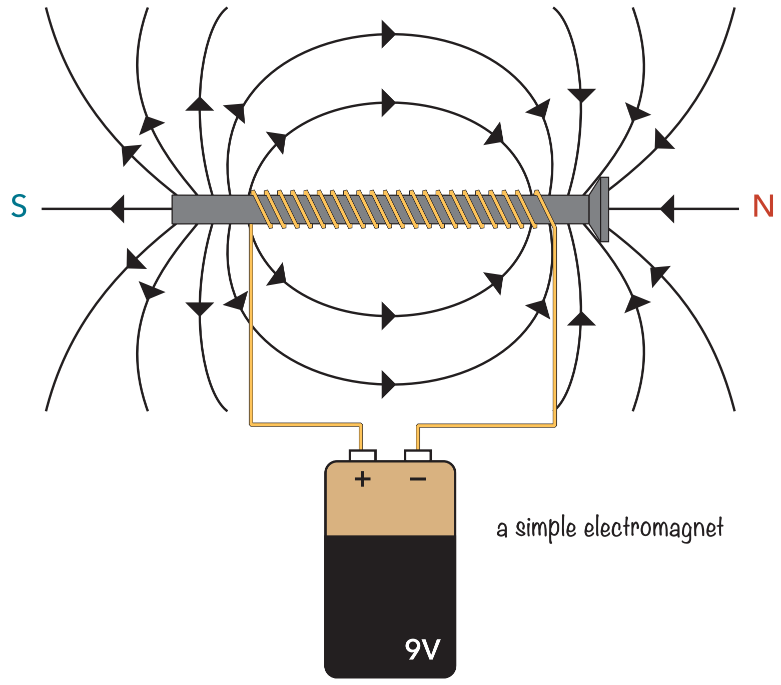
When current flows through a wire, something strange happens: the current creates a magnetic field around the wire.



The magnetic field of one wire, however, is very weak. To make a stronger magnetic field, you need to run current through lots of wires placed next to each other. But you still need only a single wire: you can just wind that wire into many loops to make a coil, and then send a current through it. The magnetic fields from each loop in the coil overlap and combine to create a stronger magnetic field. If you wind your wire around a piece of iron—like a nail, a bolt, or a screw—you'll get an even stronger magnetic field.

All you have to do to create an electromagnet is connect a battery to the ends of the coiled wire, making a closed circuit. When current flows through the wire, the piece of iron it's wrapped around starts to behave like a magnet, with the south pole at one end and the north pole at the other end. Which pole is which depends on the direction of the current, as well as the direction of the coil windings. When you disconnect the battery, the current stops and the magnetic field disappears.

Building an electromagnet will help you start to understand how you can use electricity to make things like a loudspeaker in the real world, so let's make one! With enough current, enough wire, and the right circuit, you could build a supermagnet straight out of your favorite cartoon, but for now, we'll start with a small one.

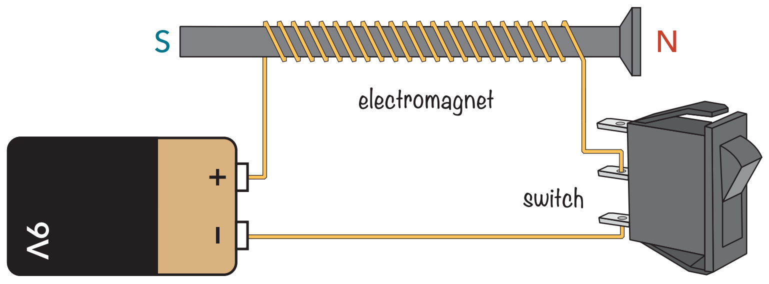


a simple electromagnet

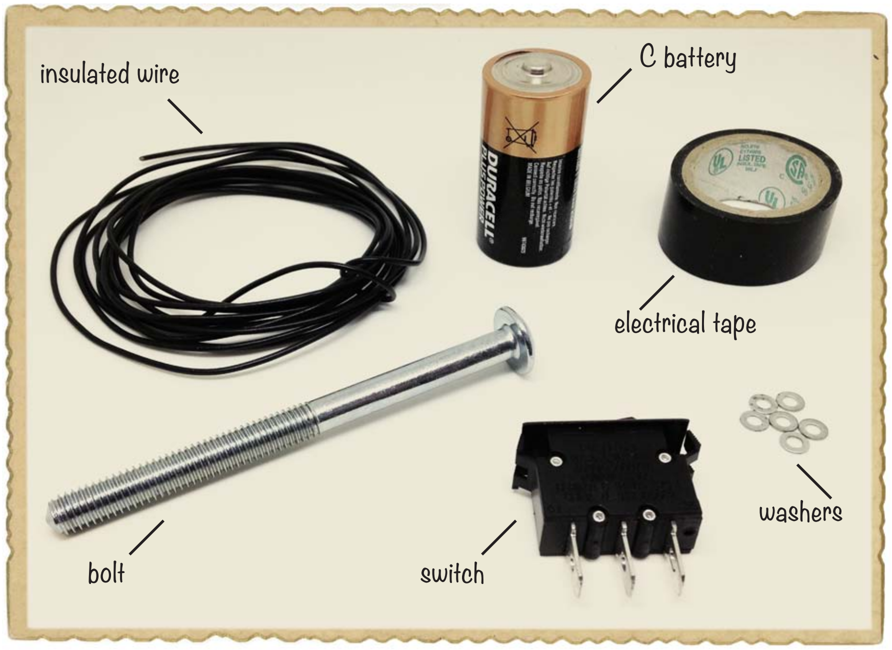
PROJECT #3: CREATE YOUR OWN ELECTROMAGNET

You know the theory behind how to build your own electromagnet. But reading the theory isn't the same as making something in real life, so it's time to have some fun!

You're going to build your own electromagnet with wire and a bolt. All you need to do is to wrap the wire around the bolt several times and connect the battery to the wire. To make it easy to turn the electromagnet on and off, you'll also add a switch to the circuit so that you can control whether or not current flows through the wires.

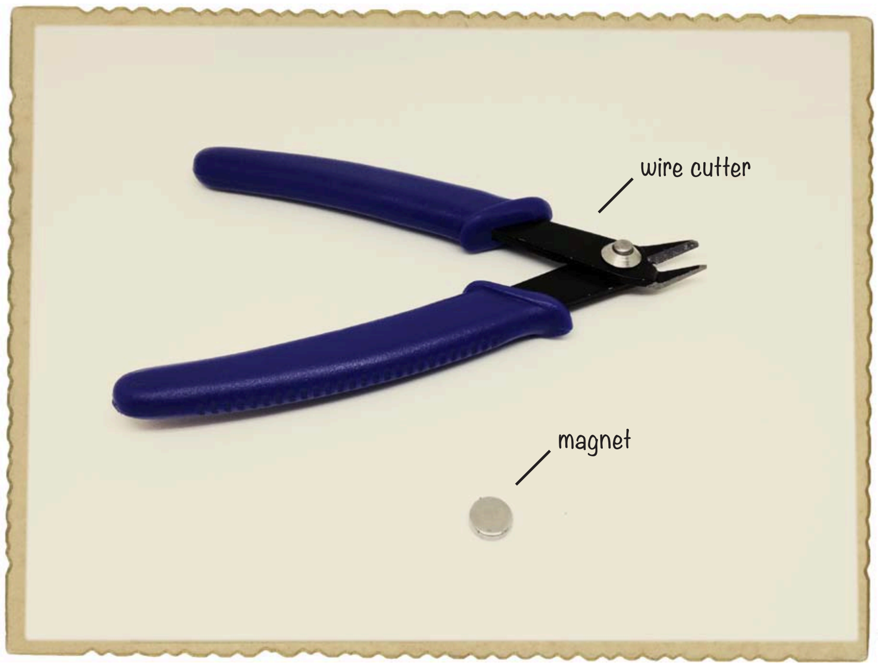


Shopping List



- **A 1.5 V alkaline (C) battery** (Jameco #2112428, Bitsbox.co.uk #BAT040), like the big round ones used in older flashlights. Don't use a rechargeable battery or plug-in power supply.
- **Insulated solid-core wire** (Jameco #36792, Bitsbox.co.uk #W106BK), about 7 feet. Standard hook-up wire works fine.
- **Tape** to fasten everything. You can use masking tape, electrical tape, or whatever you have.
- **Washers or paper clips**, or other small metal objects that your electromagnet can lift.
- **A bolt** to wind the wire around. Choose a big one to make room for many turns with the wire. The bolt I used was 0.3 inches thick and 4 inches long.
- **A switch** (Jameco #581685, Bitsbox.co.uk #SW018) to turn the electromagnet on and off.

Tools



- **A wire cutter** (Jameco #35482, Bitsbox.co.uk #TL008) to cut or remove the insulation from wire.
- **A standard magnet**

Step 1: Check Your Bolt

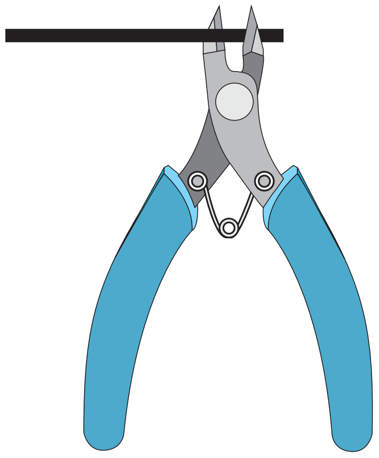
Your bolt is going to be the core of your electromagnet, making it stronger. But not all materials will work as an electromagnet's core! Most metal bolts should work, but if you're unlucky and find one that is made of nonmagnetic material, your electromagnet won't be very effective.

To check whether a bolt is okay to use in this project, just hold it close to any standard magnet. If the magnet attracts the bolt, then the bolt is a good one.

Step 2: Remove Insulation from One End of the Coil Wire

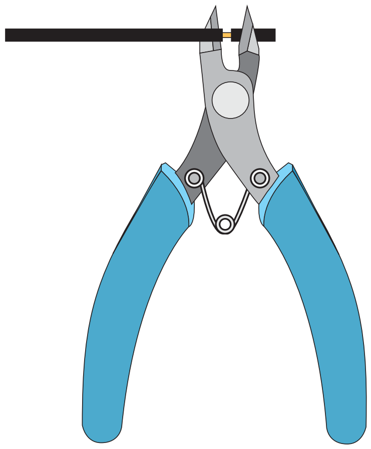
To connect the coil wire to the battery and the switch, you need to expose the metal of the wire at both ends. You'll use a wire cutter to strip away about 0.5 inches of insulation from the beginning of your wire. After you've wound the coil, you'll do the same with the end of your wire. Stripping wires can be a bit difficult if you've never done it before, so ask a parent or teacher for help to get started.

First, gently grasp the end of the wire with the cutters.



Apply just enough pressure with the wire cutter to cut the plastic around the wire, but not the wire itself. When you've cut through the insulation, your wire should look something like this:

Then, place the wire cutter in the cut you made. Squeeze the wire cutter enough to grip the loose plastic with the blades. Use the wire cutter to gently pull off the plastic without cutting into the metal of the wire.

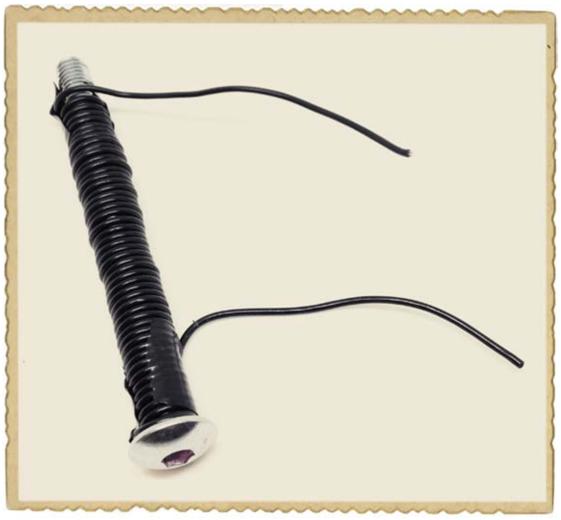


Now, you should have a wire with some exposed metal at the end, like this:

If stripping wires seems tricky in the beginning, don't worry: it becomes much easier with practice.

Step 3: Wind the Wire

Take the wire and wrap it around your bolt 50 to 100 times. Leave about 3 inches of each end of the wire hanging loose. Make sure you don't use all the wire; you'll need a piece of wire about 4 inches long in a later step.



Wrap the wire as tight as possible and tape the end to make sure the turns stay in place. We call this wound wire the *coil* of the electromagnet.

Repeat Step 2 to strip the insulation off the other end of your coil.

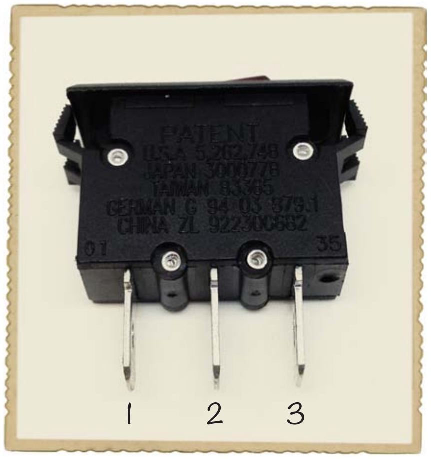
Step 4: Connect the Negative Battery Terminal to the Coil

Connect one end of the coil—it doesn't matter which—to the negative terminal of the battery. Fasten it to the battery with tape.

WARNING *Be sure you're using the recommended 1.5 V battery! Anything more powerful could send too much current through your coil, which could make both the battery and the coil hot enough to burn you.*

Step 5: Connect the Switch

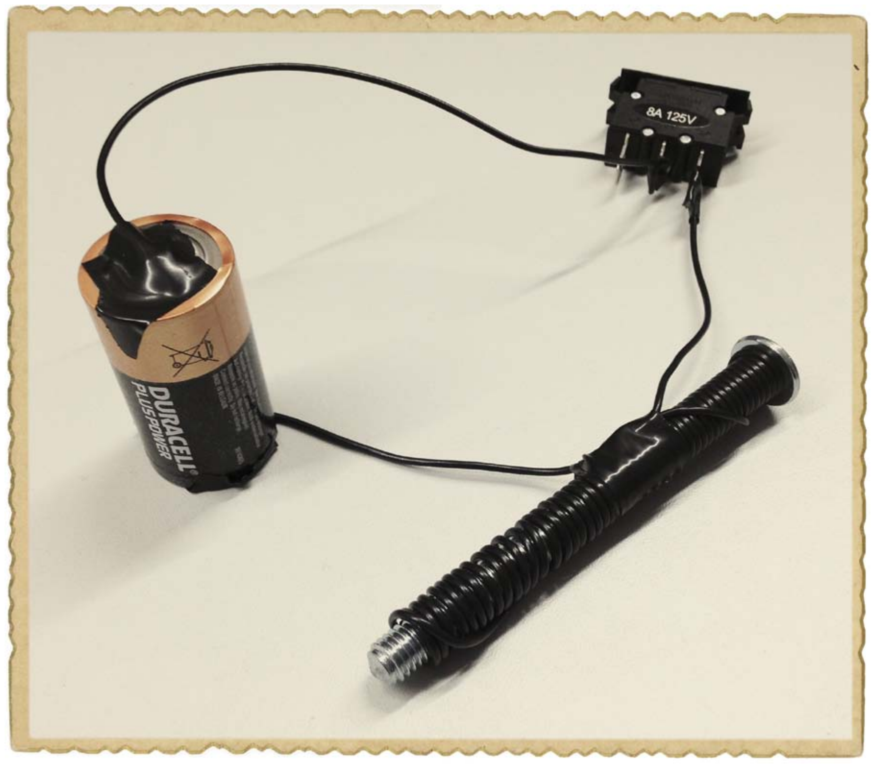
In Chapter 1, I showed you how to build your own switch and described how you can use one to turn something on and off. Now, you're going to connect a prebuilt switch to your electromagnet to turn it on and off. A switch often has three *pins* that you can connect to.



On the switch in this project's Shopping List, pin 2 is the *common pin*, which is connected to either pin 1 or pin 3, depending on the position of the button. If the button is pushed toward pin 1, then pins 2 and 1 are connected. If it's pushed toward pin 3, then pins 2 and 3 are connected.

Some switches have only two pins. In that case, the two pins are connected when the button is in one position, and not connected in the other—just like the switch you built in "Project #2: Intruder Alarm" on page 72.

Fasten the other end of the coil wire to pin 1 of the switch and make sure the button of the switch is pushed toward pin 3. Then, cut a brand-new piece of wire from your spool, about 4 inches long, and strip some insulation from both ends to expose the metal. Connect one end of the new wire to the positive battery terminal and one end to the middle pin of the switch. Use tape to make sure the wires are properly connected and stay in place.



Step 6: Test Your Super Electromagnet

That's it for building the circuit! Now, let's test it. If you've connected everything correctly, your electromagnet should be off now.

First, find a good piece of metal to attract with your electromagnet. A small metal paper clip should do the trick, though I used a little pile of steel washers. Magnets won't attract all metals—for example, aluminum foil is not magnetic—so hold a regular magnet next to the metal you want to attract first to make sure it's magnetic.

Then, flip your switch and place your electromagnet close to your paper clip or whatever other metal object you're using. If you've found the *on* position, the bolt should pull the metal object toward it.

If nothing happens, press your switch into the other position; the bolt should start to pull the metal object now.

The electromagnet consumes a lot of power, so if you keep the switch flipped on for too long, your battery will drain quickly. You might also notice that the battery and the coil become hot. Try to limit the time your electromagnet is on to only a few seconds, and always disconnect the battery before you leave your circuit.

Step 7: What If the Electromagnet Isn't Working?

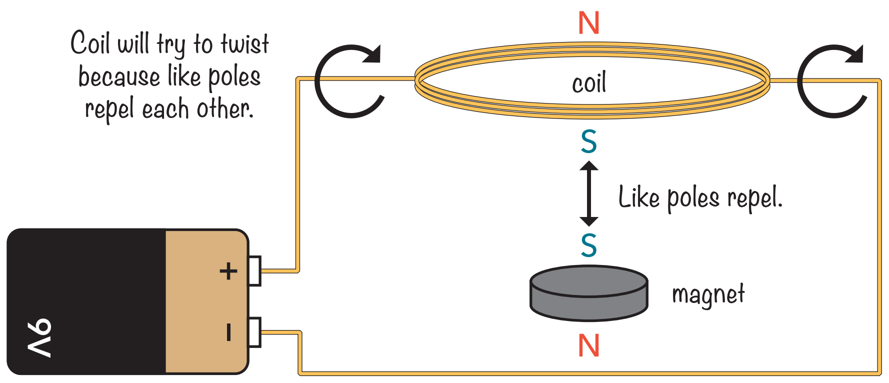
Make sure you used insulated wire to make the loops around the bolt. The wire must have some kind of insulating layer on the outside of the metal; otherwise, it won't work. The reason for this is that without the insulating layer, the electrons won't follow the wire loops around the bolt. Instead, they'll go through the bolt if the bolt is conductive or through to the neighboring wire if the loops of wire are touching. In either case, the electrons will function as if you had one thick wire.

Another possible problem is that your battery is dead. Try switching to a different battery that you're sure is working.

If you're sure you're using insulated wire and that the battery has power, check that the connections on the switch and battery are connected, as I described in Steps 4 and 5. If you're unsure, it might be a good idea to redo the connections.

MEET THE MOTOR

A wire with flowing current creates a magnetic field, as I described in "Meet the Electromagnet" on page 72. When powered, the coil from Project #3 will have a magnetic field with south and north poles, just like any other magnet. Like poles repel each other and opposite poles attract each other. So, if you put a magnetized coil of wire over a regular magnet with the same poles close to each other, the coil will try to twist itself around.



If you placed the wire coil on some kind of stand so that it could rotate freely over the magnet, it would flip back and forth without making a full spin. This is because when the coil has made a half spin, the opposite poles face and attract each other, which will force the coil in the opposite direction.

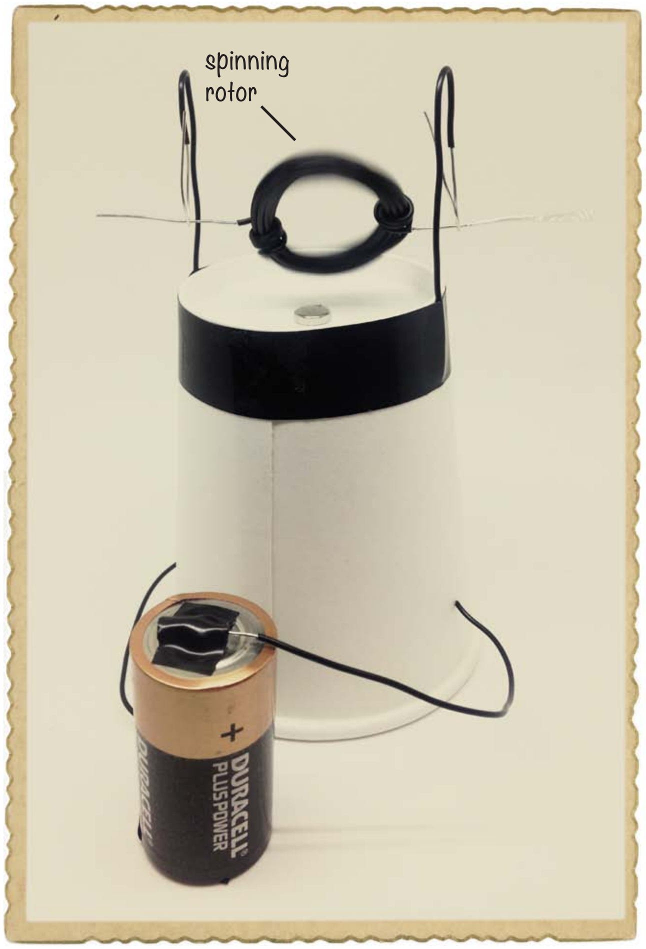
How can you make the coil continue to spin in one direction? You just need to find a way to disconnect the battery halfway around and turn the battery back on when the coil is back in its starting position. Then, here's what happens. The coil starts moving when it's powered and pushes the wire coil halfway through one round. Because you disconnect the battery halfway through, the existing motion keeps the coil moving forward. When it comes back to its original position, the battery gets reconnected and gives the coil another push forward, and it continues the same way.

Electric motors are based on this basic principle of magnetic poles attracting and repelling each other.

PROJECT #4: CREATE A MOTOR

In this chapter, you've built your own electromagnet, and you've learned how motors work. Now, it's time to combine these two concepts. In this project, you'll build your very own motor from scratch!

You'll use a magnet together with a coil of wire. The coil will spin, and this spinning coil is called the *rotor* of the motor. You're going to build the motor so that the rotor coil has current through it for only half of the spin. The magnet should push the electromagnet for half of the spin, and the rotor coil should continue around the second half of its spin with the energy it gets from the first push.



Shopping List

- **A 1.5 V alkaline (C) battery** (Jameco #2112428, Bitsbox.co.uk #BAT040), like the big round ones used in older flashlights.
- **Insulated solid-core wire** (Jameco #36792, Bitsbox.co.uk #W106BK), about 13 feet. The stiff insulated wire will be used both for the coil and to support the coil.
- **Tape** to fasten everything. You can use masking tape, electrical tape, or whatever you have.
- **A paper or plastic cup** to hold everything in place.
- **Two disc magnets** (Jameco #2181319, Bitsbox.co.uk #HW145), the stronger the better.

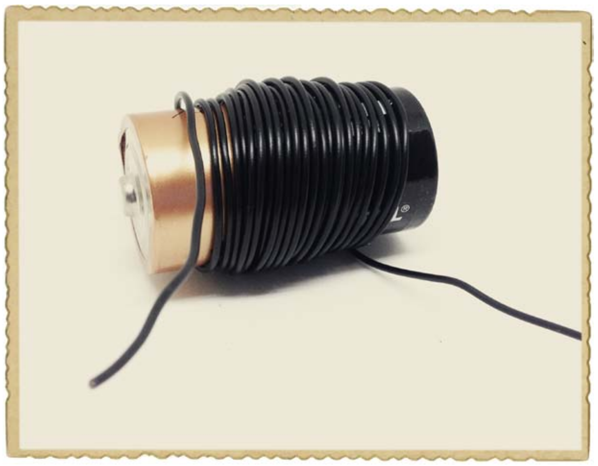
WARNING *Always keep small supermagnets like these away from babies and young children. These magnets are very dangerous if swallowed.*

Tools

- **A wire cutter** (Jameco #35482, Bitsbox.co.uk #TL008) to cut or remove the insulation from wire.

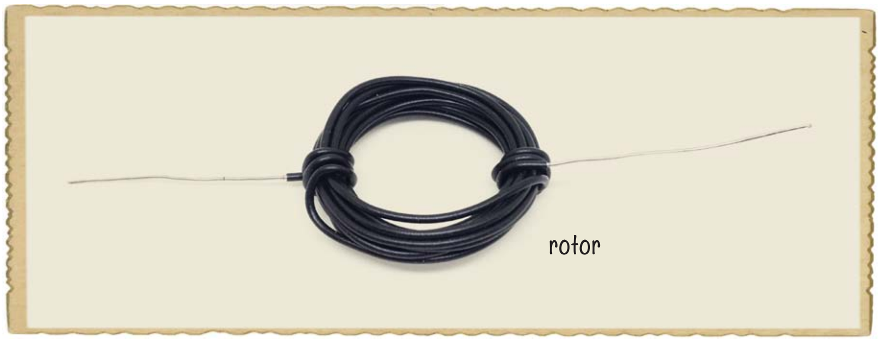
Step 1: Create the Rotor

First, we'll create a new coil of wire; this coil will be the rotor, or spinning part, of your motor. To create the rotor, first take your spool of wire and strip the insulation from about 1.5 inches of the free end. Then, wind the wire around the battery.



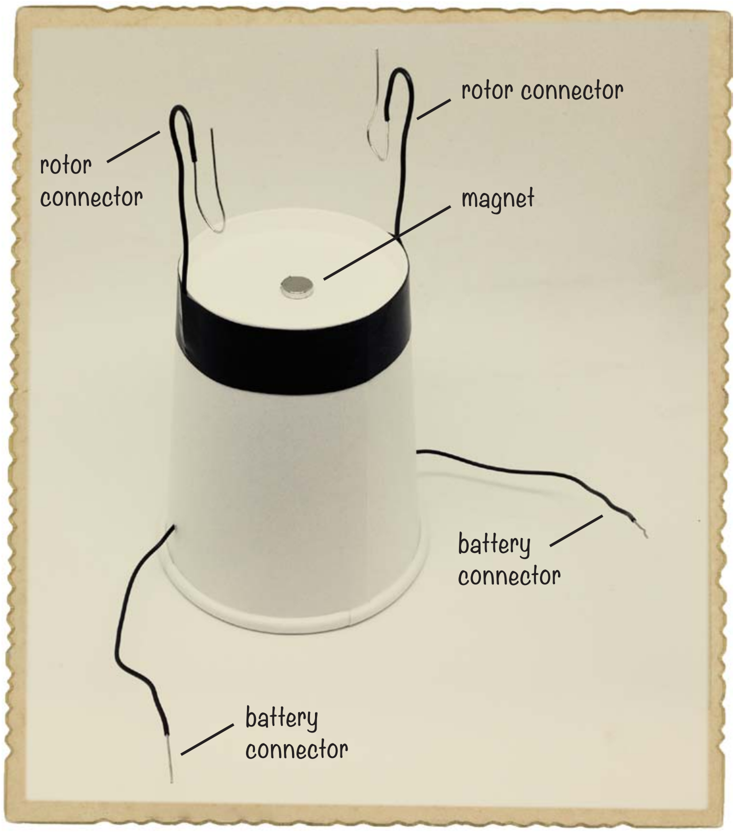
If you buy the wire I recommend in this project's Shopping List, try making around 30 windings; if you use thinner wire, wind it more. The point is to make the coil as magnetic as possible, without making it too heavy. More windings make the rotor more magnetic, but also heavier.

Carefully slide your coiled wire off the battery. Gather the windings into a loop and wrap the ends of the wire around your loop a few times on each side so that the coils stay together. Cut your loop from the spool of wire, leaving the other end about 1.5 inches long. Then, remove the insulation from this end, too, so that the metal inside is exposed. If you're using wires with plastic insulation, you can use a wire cutter, as described in "Step 2: Remove Insulation from One End of the Coil Wire" on page 76.



Step 2: Build the Motor's Structure

Set your coil aside for now and take out your paper cup. Punch a hole in one side of the cup about 0.4 inches from the top and another one about 0.4 inches from the bottom. Pull a piece of the stiff wire around 8 inches long through these two holes. Then, do the same on the other side of the cup. Turn the cup upside down, remove the insulation from the ends of both wires, and tape the wires to the cup to ensure they stay in place.



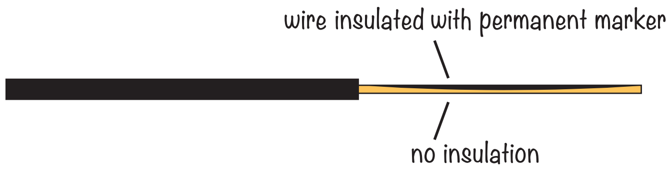
The ends that are now on the bottom will connect to the battery, and the top ends are going to make up the connection to the rotor and support it. Bend the top ends of the two wires into two U-shapes that can hold the rotor. Make sure the bottom part of each U has exposed metal so that it will touch the exposed wires of the rotor. This U-structure will be the battery's connection to the rotor.

Step 3: Place the Magnets

Place one magnet on top of the cup. Then place one magnet inside the cup so that the two magnets stick to each other through the cup. Place your rotor into the U-structure and adjust the position of the magnets to make sure they are at the center, just under the coil.

Step 4: Reinsulate Part of the Coil

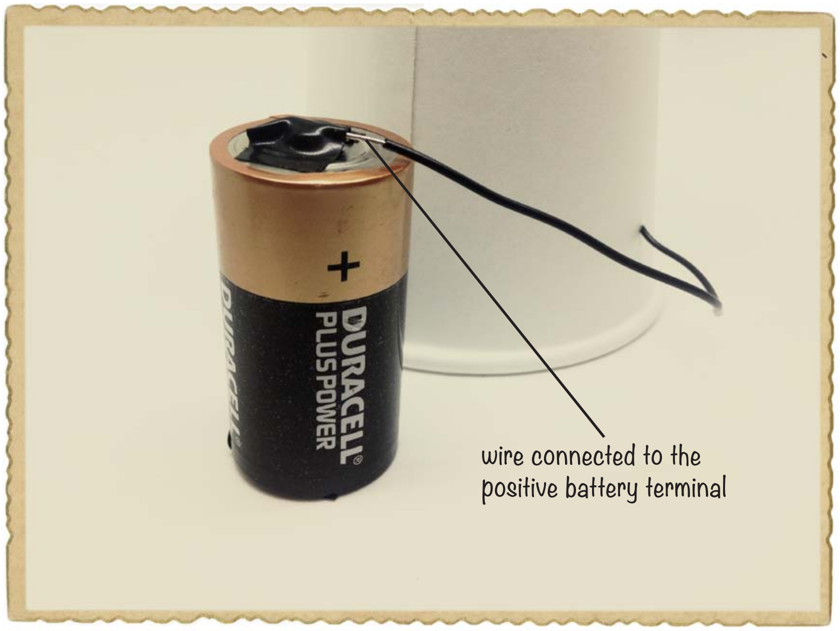
If you connected the battery now, the motor wouldn't work. With your coil rotor attached, you'd see movement, but the rotor would just be pushed back and forth in opposite directions because it's always connected to the battery. You need a way to disconnect the coil from the battery halfway through so that it's first pushed away from the magnet and then released until it has spun the rest of the way around. Then, it can reconnect with the magnet and get pushed again, and so on. You can make this happen by insulating the wire on one side with a permanent marker. Do this on only one arm of the rotor.



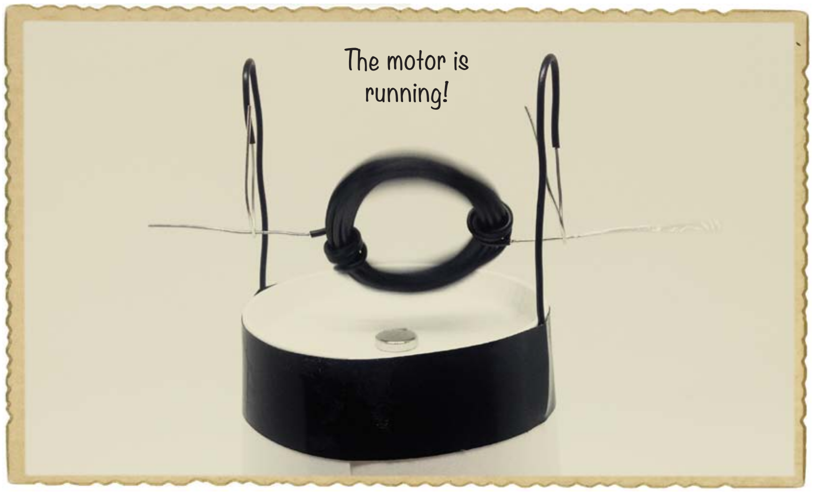
Lay your coil flat on the table and use a permanent marker to draw along the wire on one side to make it nonconductive. Draw your line so that the rotor disconnects from the battery when the loop lies horizontally above the magnet.

Step 5: Rev Up Your Motor

Let's get that motor running! Connect the battery by taping the two wires to the positive and negative terminals.



Now, place the rotor into the U-structure. The motor should start spinning. You might need to give it a little push. It won't run any cars, but if it works, then you definitely just made something move with electricity. Congratulations!



Step 6: What If the Motor Doesn't Work?

Can you see any movement? If you're very lucky, it'll work right away, but you'll most likely need to make some adjustments. Here are some places to start:

1. Make sure your coil is placed so that it starts with the exposed wire—that is, not the part you covered with the marker—touching the exposed wire of the U-shaped structure. That way, when you connect the battery, the coil becomes magnetic.
2. Figure out which way the battery should be connected. You might find that the rotor spins better in one direction than the other, so try to connect the battery the other way around to see what's best for your motor.
3. If your coil is a bit too heavy, the magnetism won't be enough to push the coil all the way around the loop. Try unwinding a few loops to make the coil lighter.
4. You might need to adjust the position of the magnets under your rotor. They should be as centered as possible.

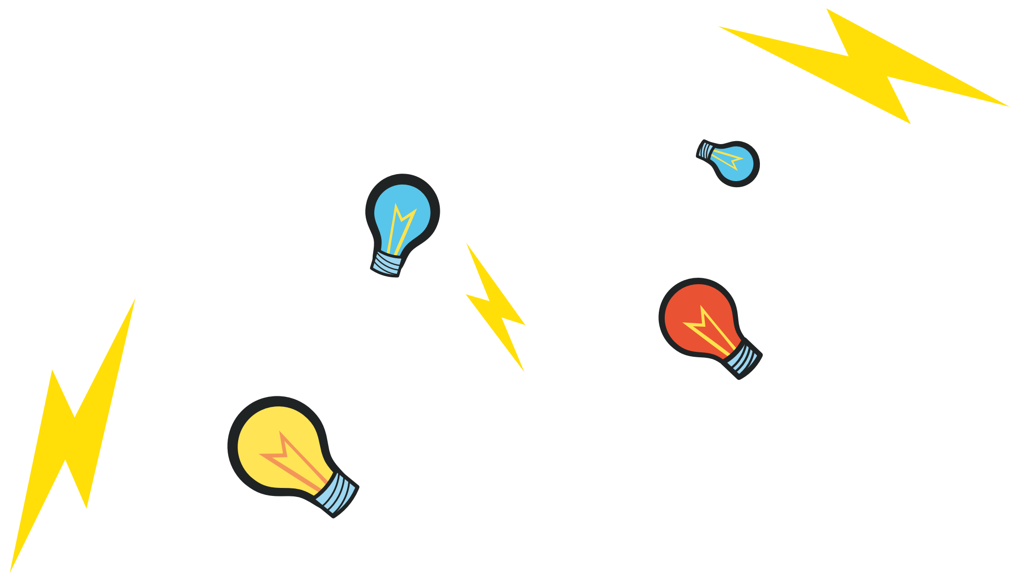
If your motor still doesn't run, your rotor may just need a little push to get started. Try tapping it lightly with your finger to see whether that unleashes a speed demon.

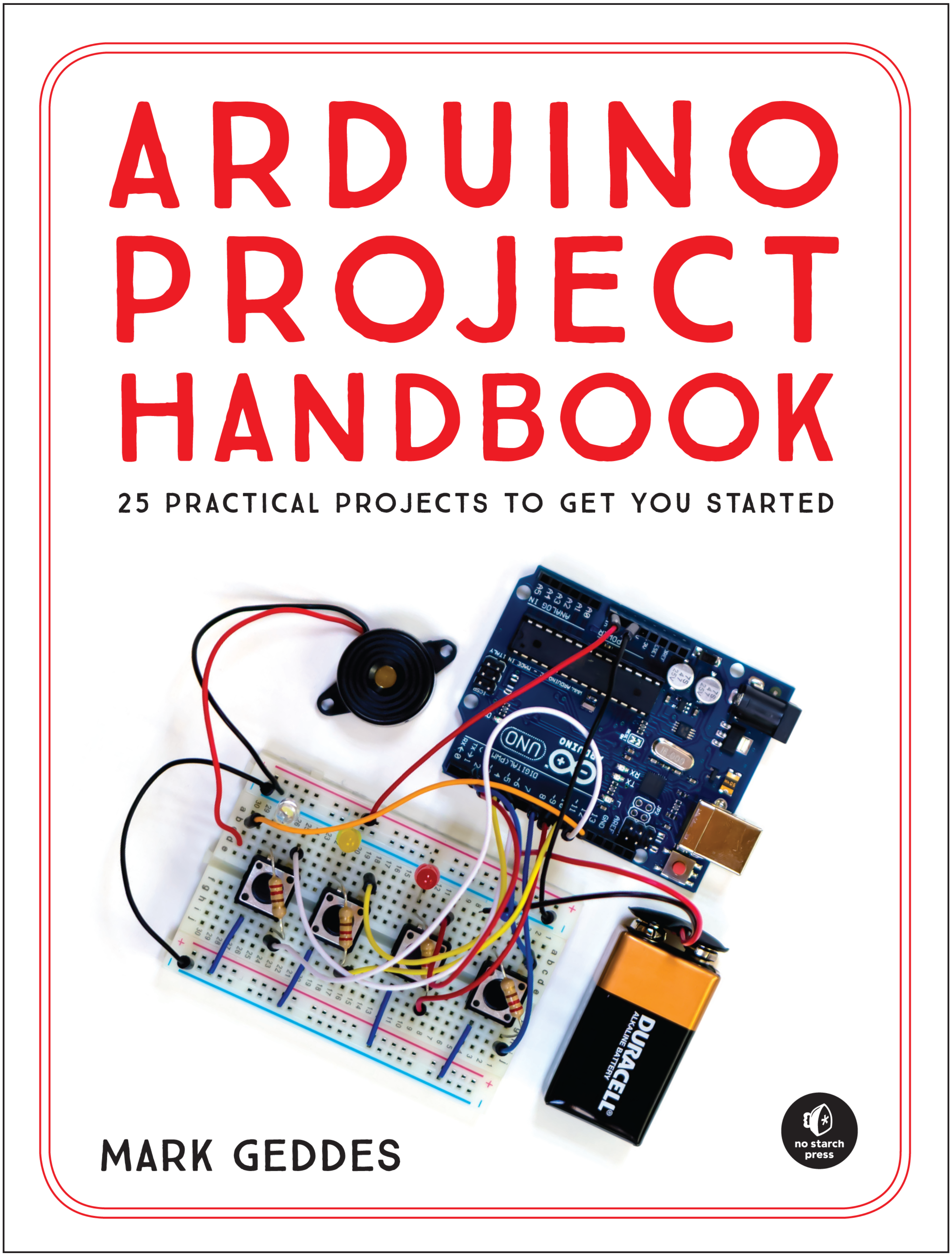
WHAT'S NEXT?

In this chapter, you've learned that magnets can be created by winding a wire around a bolt and connecting it to a battery, and you've tested this by building your own electromagnet. At the end, you learned how electric motors work, and you even built one for yourself. You really got things moving!

Now, take that knowledge and explore electricity a little further. Try adding even more magnets under the rotor of your motor. Then, wind a rotor coil that is twice as big or even bigger. You can create a much larger structure for the motor. How fast can you make your motor go?

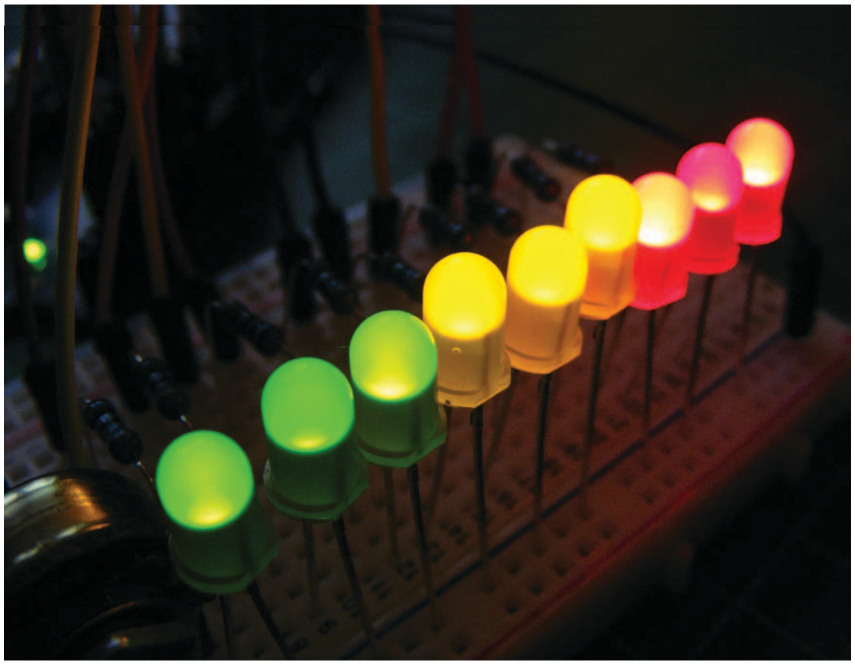
So far, you've only used electricity, but you can actually generate it, too. In the next chapter, you'll learn a couple of different ways to generate electricity, and you'll be playing around a bit more with magnets.

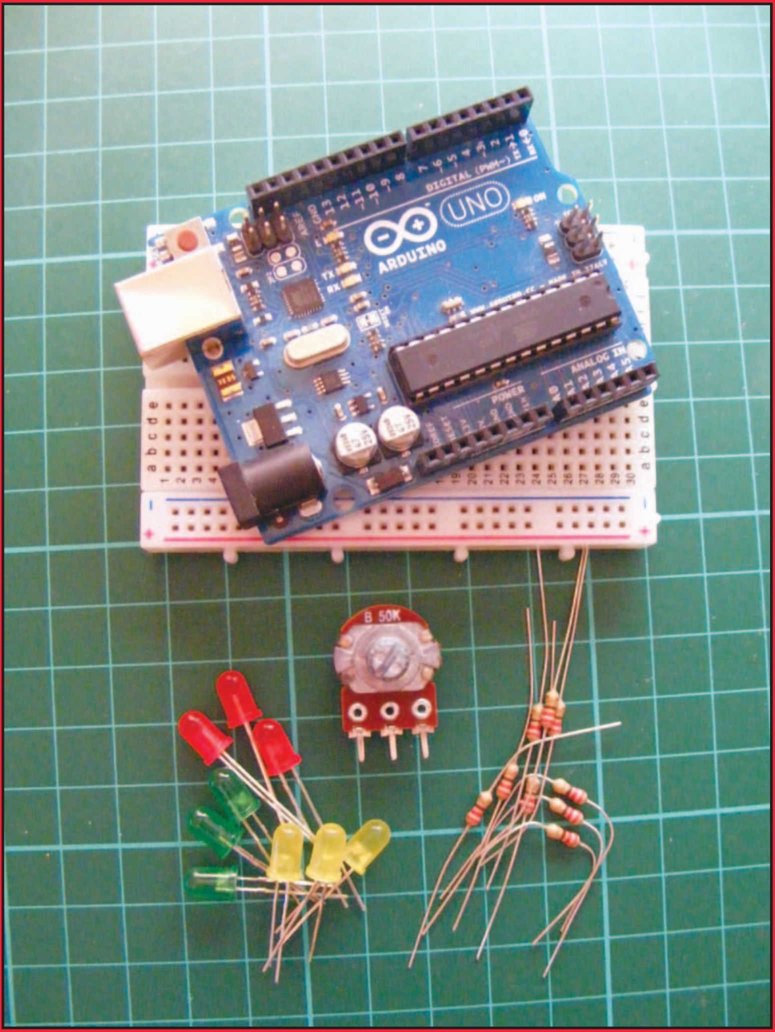




PROJECT 3: BAR GRAPH

IN THIS PROJECT, YOU'LL COMBINE WHAT YOU'VE LEARNED IN THE PREVIOUS LED PROJECTS TO CREATE AN LED BAR GRAPH THAT YOU CAN CONTROL WITH A POTENTIOMETER.





PARTS REQUIRED

- Arduino board
- Breadboard
- Wires
- 9 LEDs
- 50k-ohm potentiometer
- 9 220-ohm resistors

HOW IT WORKS

A bar graph is a series of LEDs in a line, similar to what you might see on an audio display. It's made up of a row of LEDs with an analog input, like a potentiometer or microphone. In this project, you use the analog signal from the potentiometer to control which LEDs are lit. When you turn the potentiometer one way, the LEDs light up one at a time in sequence, as shown in Figure 3-1(a), until they are all on, shown in Figure 3-1(b). When you turn it the other way, they turn off in sequence, as shown in Figure 3-1(c).

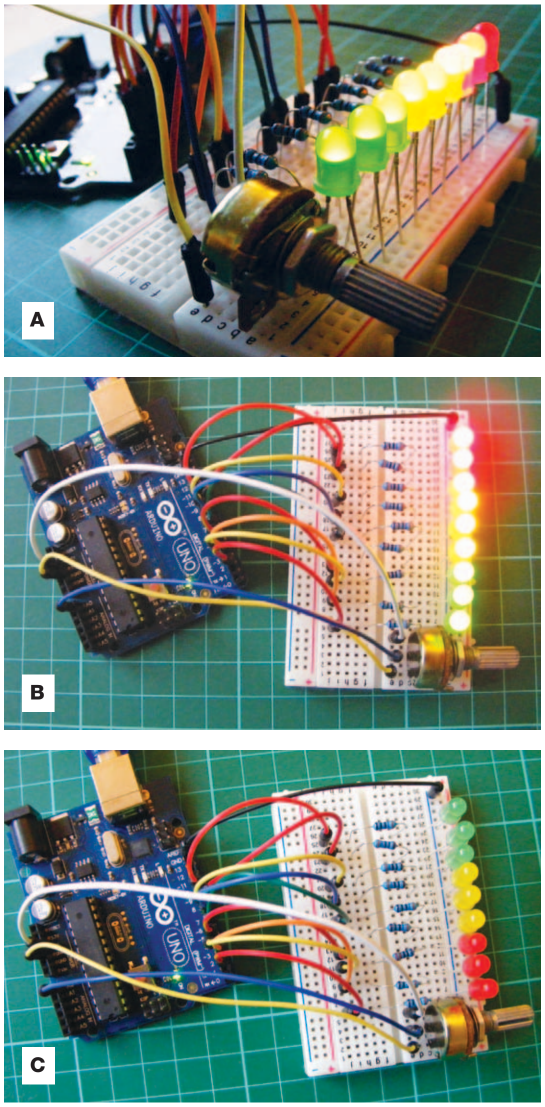
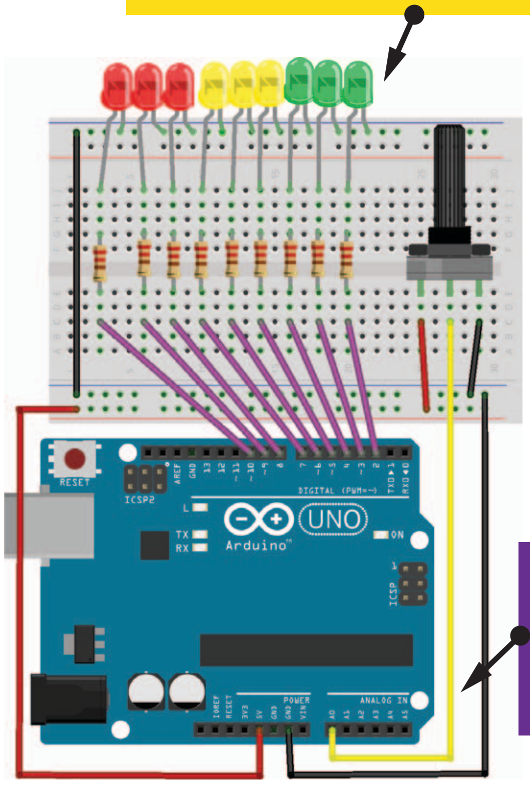


FIGURE 3-1: The LEDs light up and turn off in sequence as you turn the potentiometer.

THE BUILD

1. Insert the LEDs into the breadboard with their shorter, negative legs in the GND rail. Connect this rail to Arduino GND using a jumper wire.
2. Insert a 220-ohm resistor for each LED into the breadboard, with one resistor leg connected to the positive LED leg. Connect the other legs of the resistors to digital pins 2–10 in sequence, as shown in Figure 3-2. It's important that the resistors bridge the break in the breadboard as shown.

THE NEGATIVE LEGS OF THE LEDS ARE CONNECTED TO GND ON THE ARDUINO. THE POSITIVE LEGS ARE CONNECTED TO PINS 2–10.



THE CENTER PIN OF THE POTENTIOMETER IS CONNECTED TO ARDUINO A0.

FIGURE 3-2: Circuit diagram for the bar graph

LEDS	ARDUINO
Positive legs	Pins 2–10 via resistor
Negative legs	GND

NOTE

As mentioned in Project 2, it doesn't actually matter which way the outer potentiometer pins are connected, but I've given instructions here to reflect the images.

3. Place the potentiometer in the breadboard and connect the center pin to Arduino A0. Connect the right outer pin to +5v and the left potentiometer pin to GND.

POTENTIOMETER	ARDUINO
Left pin	GND
Center pin	A0
Right pin	+5v

4. Upload the code in "The Sketch" below.

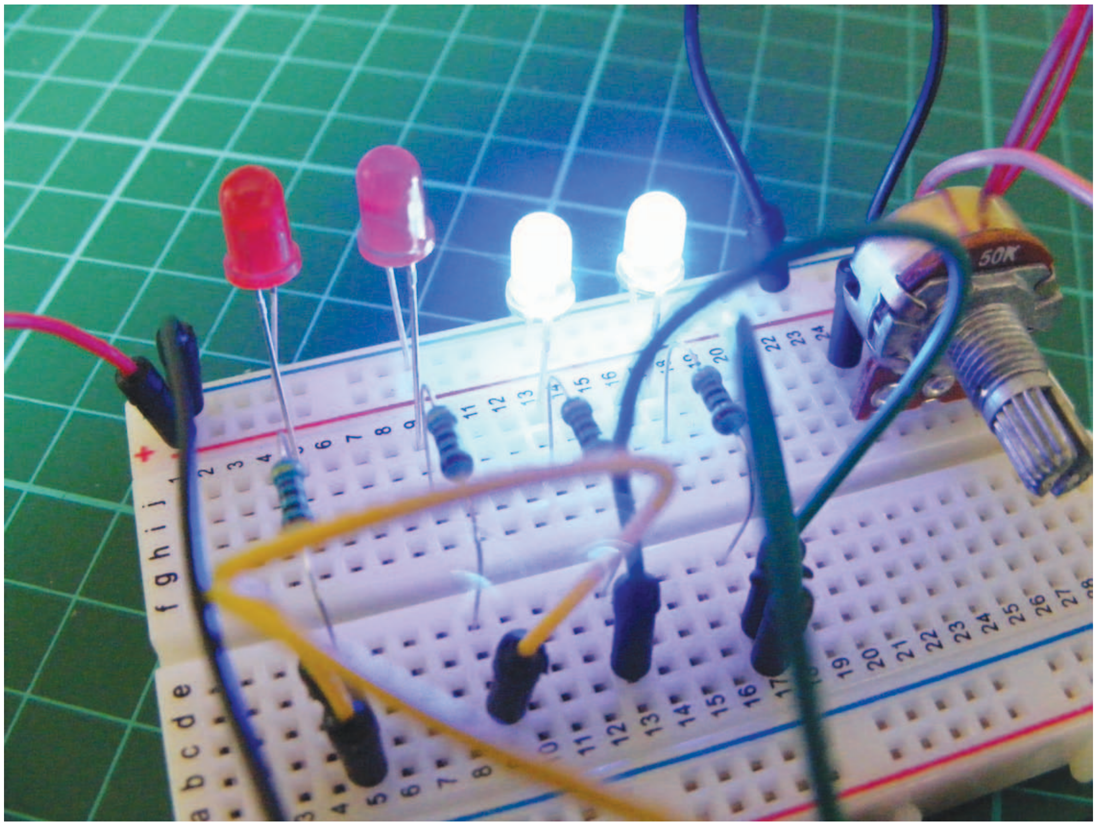
THE SKETCH

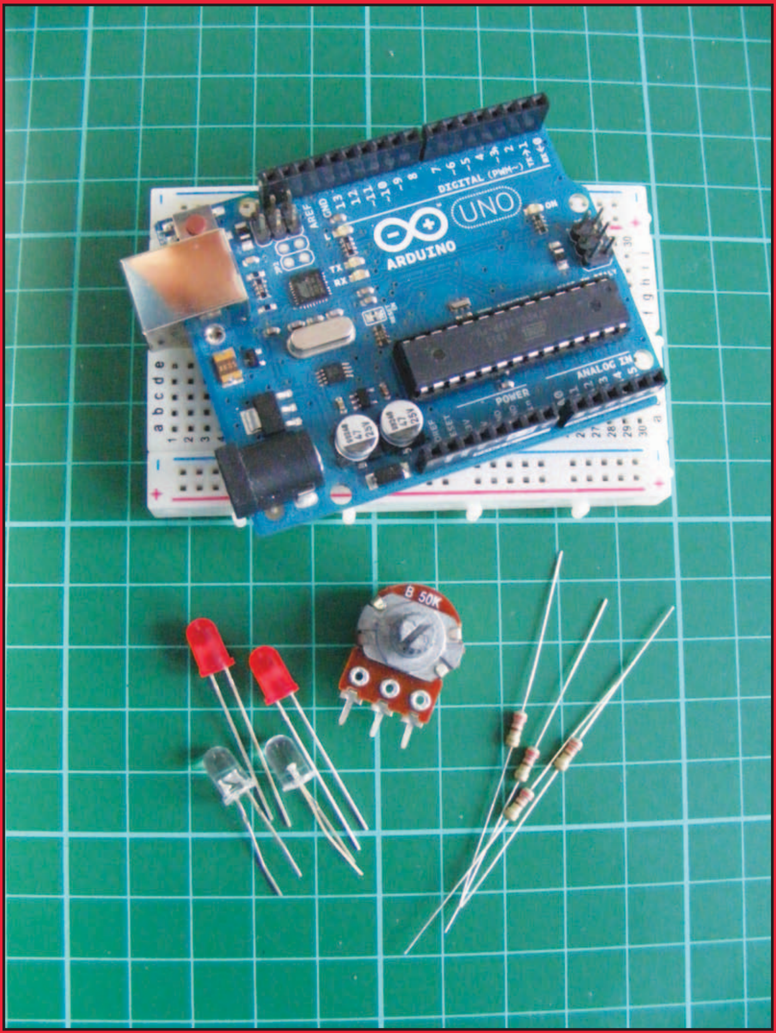
The sketch first reads the input from the potentiometer. It maps the input value to the output range, in this case nine LEDs. Then it sets up a `for` loop over the outputs. If the output number of the LED in the series is lower than the mapped input range, those LEDs turn on; if not, they turn off. See? Simple! If you turn the potentiometer to the right, the LEDs light up in sequence. Turn it to the left, and they turn off in sequence.

```
const int analogPin = A0; // Pin connected to the potentiometer
const int ledCount = 9;   // Number of LEDs
int ledPins[] = {2,3,4,5,6,7,8,9,10}; // Pins connected to the LEDs

void setup() {
  for (int thisLed = 0; thisLed < ledCount; thisLed++) {
    pinMode(ledPins[thisLed], OUTPUT); // Set the LED pins as output
  }
}

// Start a loop
void loop() {
  int sensorReading = analogRead(analogPin); // Analog input
  int ledLevel = map(sensorReading, 0, 1023, 0, ledCount);
  for (int thisLed = 0; thisLed < ledCount; thisLed++) {
    if (thisLed < ledLevel) { // Turn on LEDs in sequence
      digitalWrite(ledPins[thisLed], HIGH);
    }
    else { // Turn off LEDs in sequence
      digitalWrite(ledPins[thisLed], LOW);
    }
  }
}
```

PROJECT 4: DISCO STROBE LIGHT

IN THIS PROJECT, YOU'LL APPLY THE SKILLS YOU LEARNED IN PROJECT 3 TO MAKE A STROBE LIGHT DEVICE WITH ADJUSTABLE SPEED SETTINGS.





PARTS REQUIRED

- Arduino board
- Breadboard
- Wires
- 2 Blue LEDs
- 2 Red LEDs
- 50k-ohm potentiometer
- 4 220-ohm resistors

HOW IT WORKS

Turning the potentiometer up or down changes the speed of the flashing lights, creating a strobe effect. You can use red and blue LEDs for a flashing police light effect (see Figure 4-1). Connect the LEDs of the same color to the same Arduino pin so they'll always light together.
If you build a casing to house your LEDs, you'll have your own mobile strobe unit. You can add up to 10 LEDs and change the sketch code to include your output pins and the new number of LEDs.

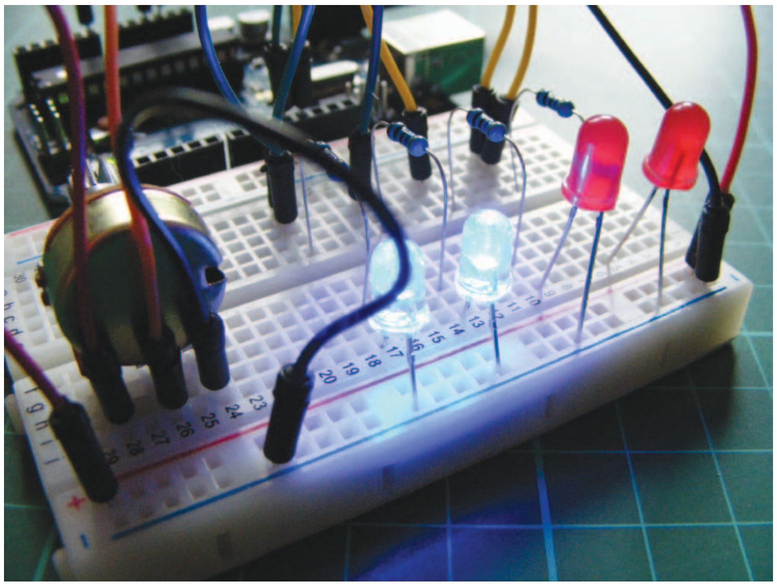


FIGURE 4-1: Red and blue LEDs mimic a police car siren.

THE BUILD

1. Place your LEDs into the breadboard with the short, negative legs in the GND rail, and then connect this rail to Arduino GND.
2. Insert the resistors into the board, connecting them to the longer, positive legs of the LEDs. Use jumper wires to connect the two red LEDs together and the two blue LEDs together via the resistors, as shown in Figure 4-2; this allows the LEDs of the same color to be controlled by a single pin.

NOTE

Remember to add power to the breadboard.

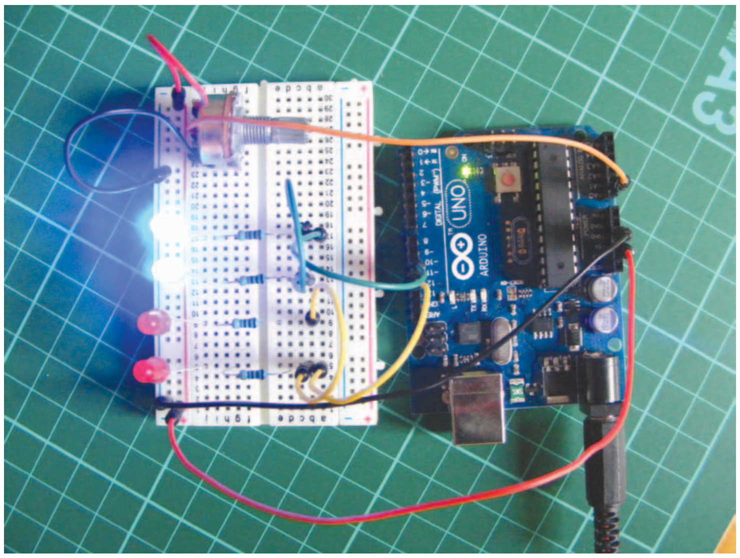


FIGURE 4-2: Connecting LEDs with jumper wires

3. Connect the red LEDs to Arduino pin 11 and the blue LEDs to Arduino pin 12.

LEDS	ARDUINO
Negative legs	GND
Positive leg (red)	Pin 11
Positive leg (blue)	Pin 12

4. Place the potentiometer in the breadboard and connect the center pin to Arduino A0, the left pin to GND, and the right pin to +5v.

POTENTIOMETER	ARDUINO
Left pin	GND
Center pin	A0
Right pin	+5v

5. Confirm that your setup matches that of Figure 4-3, and then upload the code in "The Sketch" on page 103.

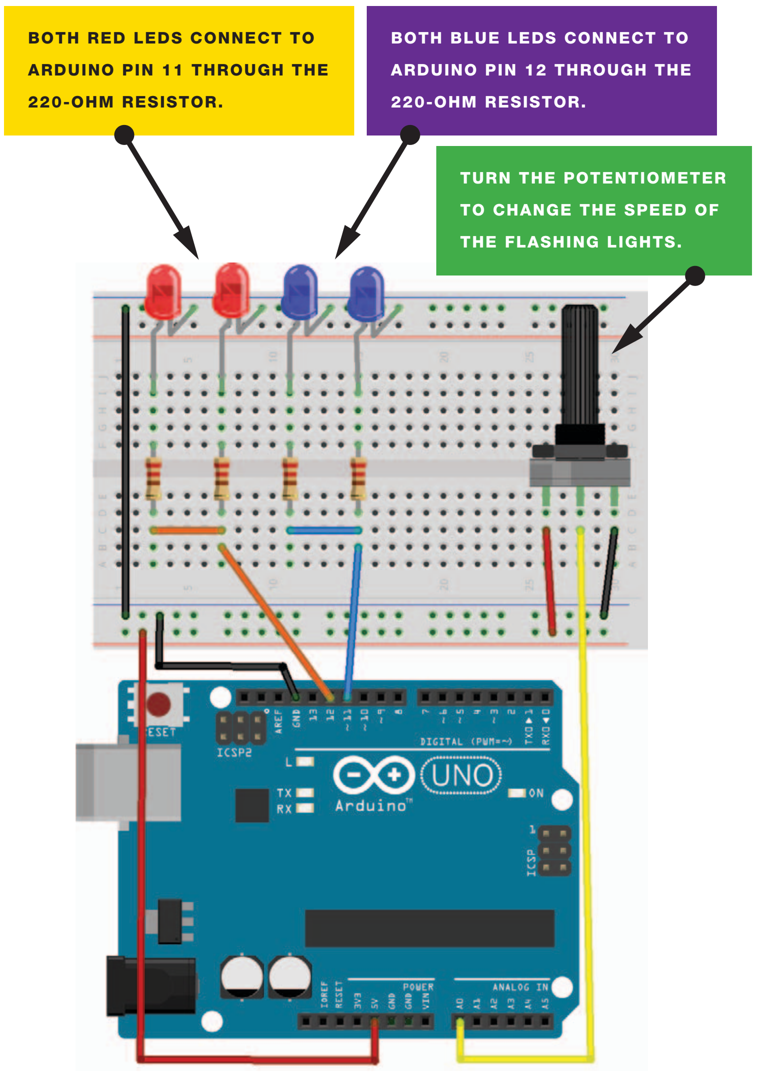

BOTH RED LEDS CONNECT TO ARDUINO PIN 11 THROUGH THE 220-OHM RESISTOR.
BOTH BLUE LEDS CONNECT TO ARDUINO PIN 12 THROUGH THE 220-OHM RESISTOR.
TURN THE POTENTIOMETER TO CHANGE THE SPEED OF THE FLASHING LIGHTS.

FIGURE 4-3: Circuit diagram for the disco strobe light

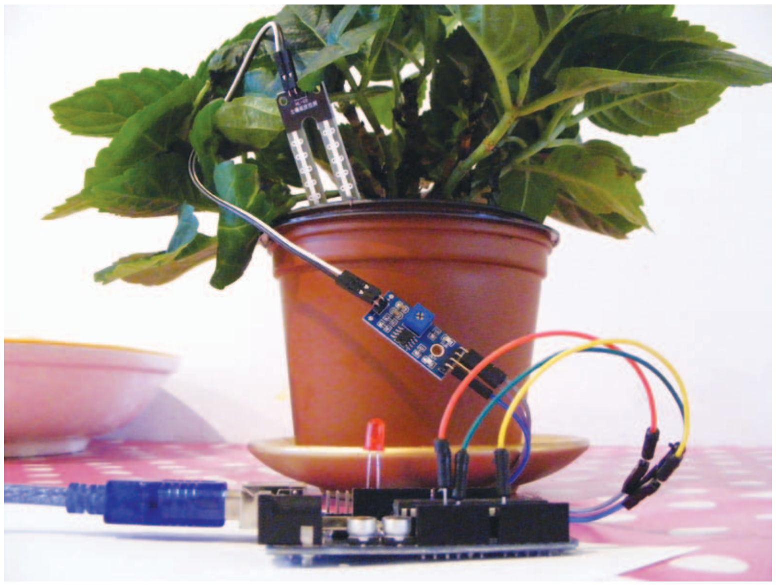
THE SKETCH

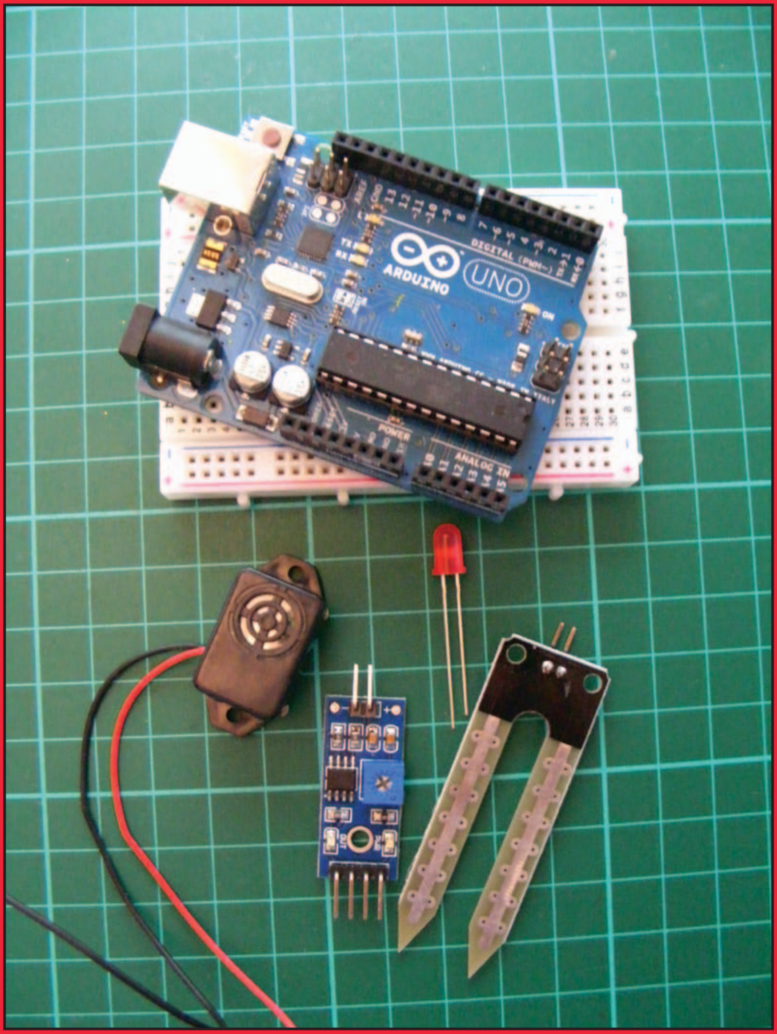
The sketch works by setting the analog signal from the potentiometer to the Arduino as an input and the pins connected to the LEDs as outputs. The Arduino reads the analog input from the potentiometer and uses this value as the *delay value*—the amount of time that passes before the LEDs change state (either on or off). This means that the LEDs are on and off for the duration of the potentiometer value, so changing this value alters the speed of the flashing. The sketch cycles through the LEDs to produce a strobe effect.

```
const int analogInPin = A0; // Analog input pin connected to the
                            // potentiometer
int sensorValue = 0;        // Value read from the potentiometer
int timer = 0;              // Delay value

// Set digital pins 12 and 11 as outputs
void setup() {
  pinMode(12, OUTPUT);
  pinMode(11, OUTPUT);
}

// Start a loop to turn LEDs on and off with a delay in between
void loop() {
  sensorValue = analogRead(analogInPin); // Read value from the
                                         // potentiometer
  timer = map(sensorValue, 0, 1023, 10, 500); // Delay 10ms to 500ms
  digitalWrite(12, HIGH); // LED turns on
  delay(timer);           // Delay depending on potentiometer value
  digitalWrite(12, LOW);  // LED turns off
  delay(timer);
  digitalWrite(12, HIGH);
  delay(timer);
  digitalWrite(12, LOW);
  digitalWrite(11, HIGH);
  delay(timer);
  digitalWrite(11, LOW);
  delay(timer);
  digitalWrite(11, HIGH);
  delay(timer);
  digitalWrite(11, LOW);
}
```

PROJECT 5: PLANT MONITOR

IN THIS PROJECT I'LL INTRODUCE A NEW TYPE OF ANALOG SENSOR THAT DETECTS MOISTURE LEVELS. YOU'LL SET UP A LIGHT AND SOUND ALARM SYSTEM TO TELL YOU WHEN YOUR PLANT NEEDS WATERING.





PARTS REQUIRED

- Arduino board
- Wires
- HL-69 moisture sensor
- LED
- Piezo buzzer

HOW IT WORKS

You'll use an HL-69 moisture sensor, readily available online for a few dollars or from some of the retailers listed in Chapter XX. The prongs of the sensor detect the moisture level in the surrounding soil by passing current through the soil and measuring the resistance. Damp soil conducts electricity easily, so it provides lower resistance, while dry soil conducts poorly and has a higher resistance.

The sensor consists of two parts, as shown in Figure 5-1: the actual prong sensor (a) and the controller (b). The two pins on the sensor need to connect to the two separate pins on the controller (connecting wires are usually supplied). The other side of the controller has four pins, three of which connect to the Arduino.

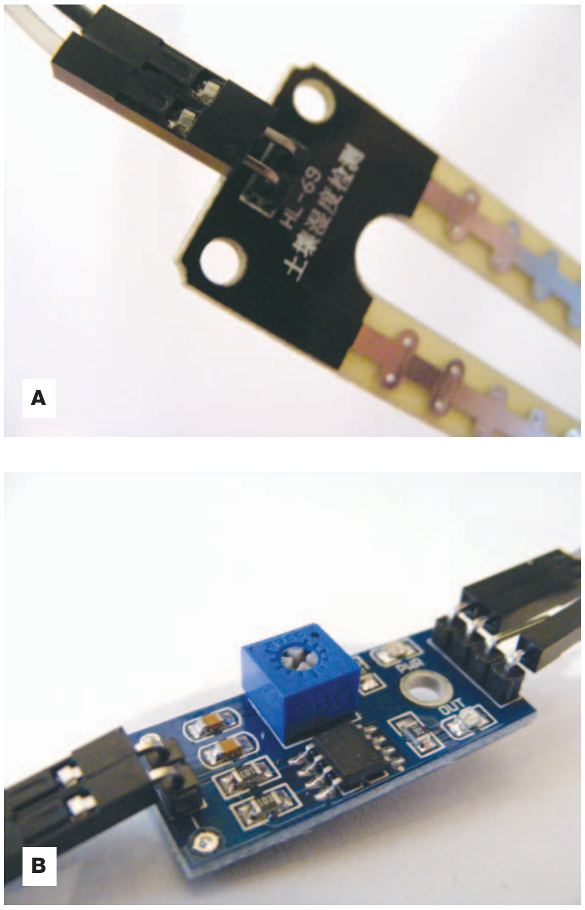


FIGURE 5-5: The HL-69 moisture sensor prong (a) and controller (b)

The four pins are, from left to right, AO (analog out), DO (digital out), GND, and VCC. You can read the values from the controller through the IDE when it's connected to your computer. This project doesn't use a breadboard, so the connections are all made directly to the Arduino.

Lower readings indicate that more moisture is being detected, and higher readings indicate dryness. If your reading is above 900, your plant is seriously thirsty.

THE BUILD

1. Connect the sensor's two pins to the + and – pins on the controller using the provided connecting wires, as shown in Figure 5-2.

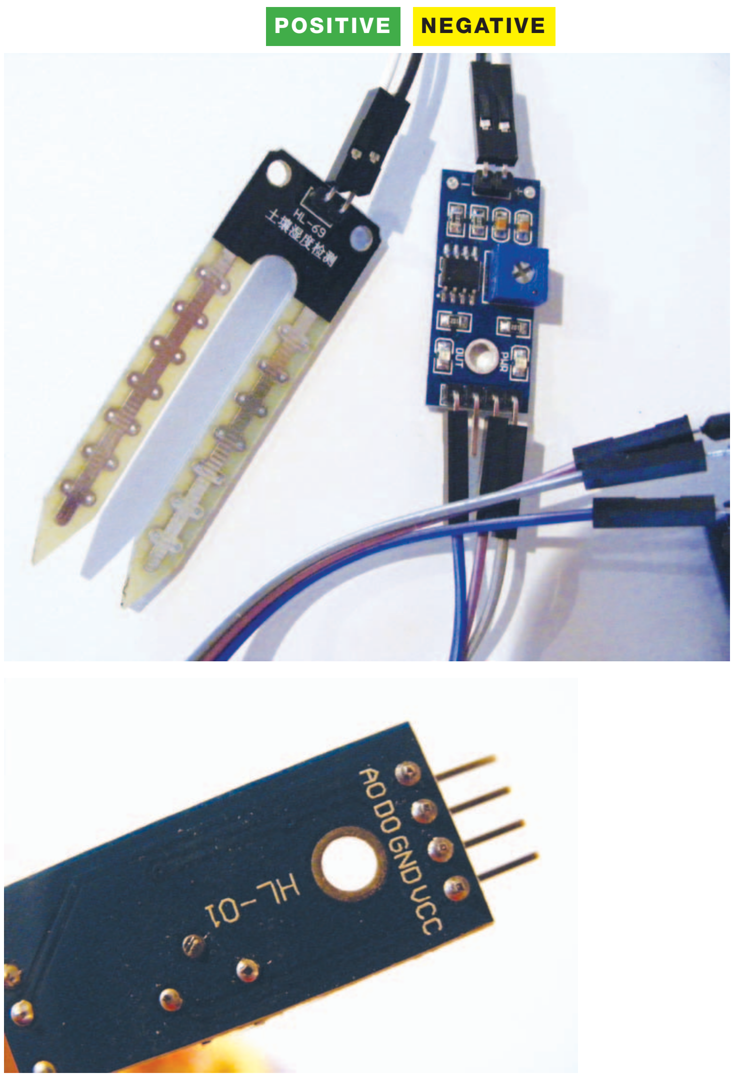


FIGURE 5-2: Connecting the sensor to the controller

2. Connect the three prongs from the controller to +5v, GND, and Arduino A0 directly on the Arduino, as shown in the following table. The DO pin is not used.

SENSOR CONTROLLER	ARDUINO
VCC	+5v
GND	GND
A0	A0
DO	Not used

3. Connect an LED directly to the Arduino with the shorter, negative leg in GND and the longer, positive leg in Arduino pin 13, as shown in Figure 5-3.

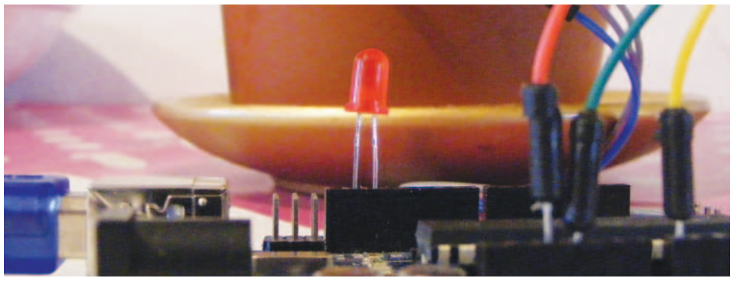


FIGURE 5-3: Connecting the LED to the Arduino

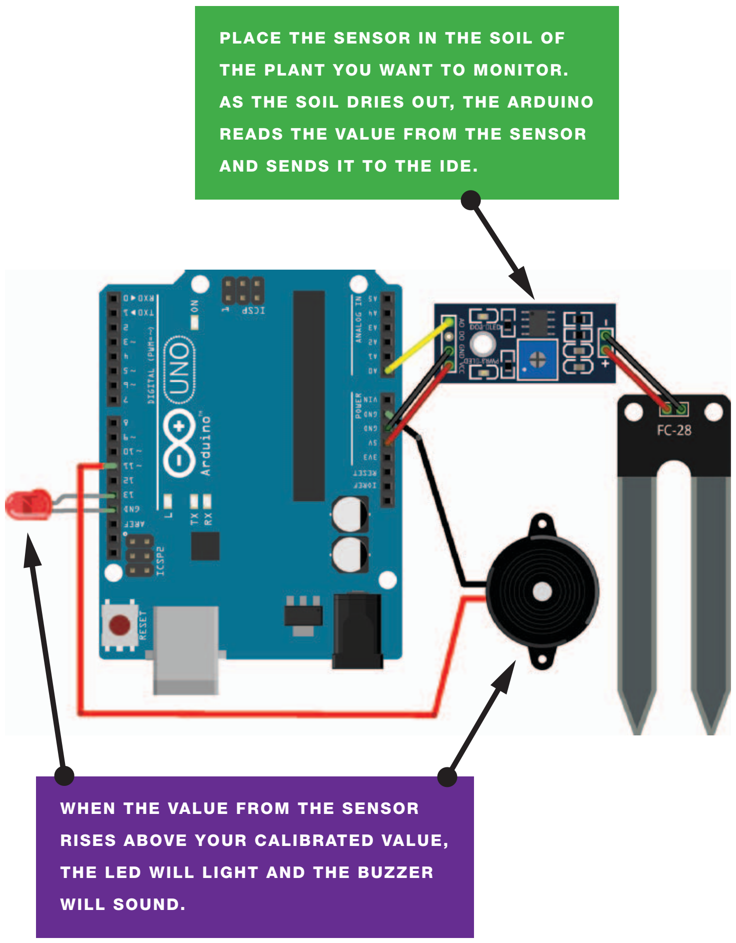
LED	ARDUINO
Positive leg	Pin 13
Negative leg	GND

4. Connect the piezo buzzer's black wire to GND and its red wire to Arduino pin 11.

PIEZO BUZZER	ARDUINO
Red	Pin 11
Black	GND

5. Check that your setup matches that of Figure 5-4, and then upload the code in "The Sketch" on page 111.

FIGURE 5-4: Circuit diagram for the plant monitor


PLACE THE SENSOR IN THE SOIL OF THE PLANT YOU WANT TO MONITOR. AS THE SOIL DRIES OUT, THE ARDUINO READS THE VALUE FROM THE SENSOR AND SENDS IT TO THE IDE.
FC-28
WHEN THE VALUE FROM THE SENSOR RISES ABOVE YOUR CALIBRATED VALUE, THE LED WILL LIGHT AND THE BUZZER WILL SOUND.

6. Connect the Arduino to your computer using the USB cable. Open the Serial Monitor in your IDE to see the values from the sensor—this will also help you to calibrate your plant monitor. The IDE will display the value of the sensor's reading. My value was 1000 with the sensor dry and not inserted in the soil, so I know this is the highest, and driest, value. To calibrate this value, turn the potentiometer on the controller clockwise to increase the resistance and counterclockwise to decrease it (see Figure 5-5).

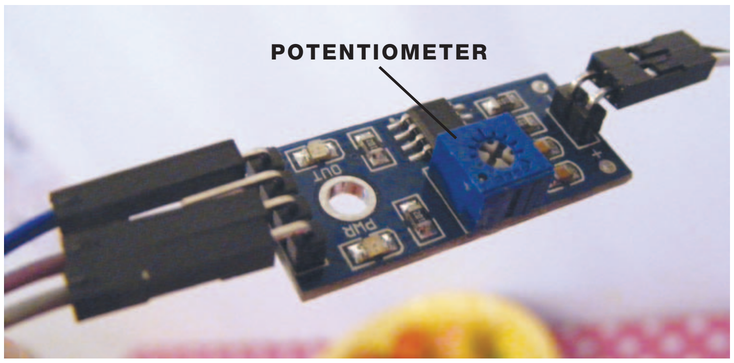


FIGURE 5-5: Turn the potentiometer to calibrate your plant monitor.

7. When the sensor is inserted into moist soil, the value will drop to about 400. As the soil dries out, the sensor value rises; when it reaches 900, the LED will light and the buzzer will sound.

THE SKETCH

The sketch first defines pin A0 so that it reads the moisture sensor value. It then defines pin 11 as output for the buzzer, and pin 13 as output for the LED. Use the `Serial.Println` function to send the reading from the sensor to the IDE, in order to see the value on the screen.

Change the value in the line

```
if(analogRead(0) > 900){
```

depending on the reading from the sensor when it is dry (here it's 900). When the soil is moist, this value will be below 900, so the LED and buzzer will remain off. When the value rises above 900, it means the soil is drying out, and the buzzer and LED will alert you to water your plant.

```
const int moistureAO = 0;
int AO = 0;       // Pin connected to AO on the controller
int tmp = 0;      // Value of the analog pin
int buzzPin = 11; // Pin connected to the piezo buzzer
int LED = 13;     // Pin connected to the LED

void setup () {
  Serial.begin(9600); // Send Arduino reading to IDE
  Serial.println("Soil moisture sensor");
  pinMode(moistureAO, INPUT);
  pinMode(buzzPin, OUTPUT); // Set pin as output
  pinMode(LED, OUTPUT);     // Set pin as output
}
```

```
void loop () {
  tmp = analogRead( moistureAO );
  if ( tmp != AO ) {
    AO = tmp;
    Serial.print("A = "); // Show the resistance value of the sensor
                          // in the IDE
    Serial.println(AO);
  }
  delay (1000);
  if (analogRead(0) > 900) { // If the reading is higher than 900,
    digitalWrite(buzzPin, HIGH); // the buzzer will sound
    digitalWrite(LED, HIGH);     // and the LED will light
    delay(1000); // Wait for 1 second
    digitalWrite(buzzPin, LOW);
    digitalWrite(LED, HIGH);
  }
  else {
    digitalWrite(buzzPin, LOW); // If the reading is below 900,
                                // the buzzer and LED stay off
    digitalWrite(LED, LOW);
  }
}
```