Fast and Vulnerable: A Story of Telematic Failures

Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage
Department of Computer Science and Engineering
University of California, San Diego
{idfoster,aprudhomme,supersat,savage}@cs.ucsd.edu

Abstract

Modern automobiles are complex distributed systems in which virtually all functionality — from acceleration and braking to lighting and HVAC — is mediated by computerized controllers. The interconnected nature of these systems raises obvious security concerns and prior work has demonstrated that a vulnerability in any single component may provide the means to compromise the system as a whole. Thus, the addition of new components, and especially new components with external networking capability, creates risks that must be carefully considered.

In this paper we examine a popular aftermarket telematics control unit (TCU) which connects to a vehicle via the standard OBD-II port. We show that these devices can be discovered, targeted, and compromised by a remote attacker and we demonstrate that such a compromise allows arbitrary remote control of the vehicle. This problem is particularly challenging because, since this is aftermarket equipment, it cannot be well addressed by automobile manufacturers themselves.

1 Introduction

Telematic control units (TCU) are the embedded devices that underlie the "connected car" concept. They interconnect existing in-vehicle electronic control units (ECUs) with outside systems to provide new services and features. These TCUs can be divided into those sold and integrated by the OEM itself (e.g., such as GM's OnStar, Ford's Sync, etc.) and those that serve the aftermarket (e.g., Progressive Snapshot's, Automatic Lab's Automatic, Delphi's Connect, etc.) In prior work we demonstrated vulnerabilities in the former class of such devices, showing that certain OEM provided TCUs allowed both local and remote compromise and subsequent takeover of virtually all automotive systems [1, 4]. More recently, this result was duplicated by Miller and Valasek on the Jeep Cherokee and its UConnect telematics system [3]. However, at least for our own research, working with the associated manufacturers to fix those problems was relatively straightforward. Since the TCUs were tightly coupled into their vehicles, they could be updated during scheduled vehicle service and mitigated through changes in the OEM-operated service. In this paper we have turned our attention to the thornier problem of aftermarket devices, which are typically purchased directly by consumers or through a third-party service offering (e.g., insurance or fleet management), are loosely coupled with the vehicle in which they are installed, and are maintained independent of normal automotive service channels.

Most of such aftermarket devices take the form of small "dongles" that interface with automotive systems through the mandated On-Board Diagnostics port (OBD-II in the US, EOBD in Europe, JOBD in Japan) typically located under the driver's side dashboard. A series of standard protocols and schemas allow an OBD-II peer to access an array of low-level sensor information about the car's operation (most of this is defined in SAE standards J1979, J2012, and J2178). In addition, it is common that the physical OBD-II port also provides access to the vehicle's internal networks (typically one or more CAN buses) for monitoring additional, vehicle specific, information. In spite of the fact that most aftermarket TCUs are designed for monitoring only, CAN is a multi-master bus and thus any device with a CAN transceiver is able to send messages as well as receive. This presents a key security problem since as we, and others, have shown, transmit access to the CAN bus is frequently sufficient to obtain arbitrary control over all key vehicular systems (including throttle and brakes).

Further, it is common that these devices, in addition to a microprocessor and a CAN transceiver, also provide a number of additional functionality including a GPS, accelerometers and, critically, external networking connectivity. This latter feature may be as simple as a short-distance Bluetooth interface (e.g., as with Automatic Lab's dongle), but more commonly is a full cel-

lular modem (2G or 3G) that provides remote data connectivity via the Internet. Taken together, these pieces of functionality place tremendous weight on the security of aftermarket TCU software. Should such software be vulnerable to external compromise, this would allow an attacker to control a wide array of vehicles at arbitrary distance.

In this paper, we explore this issue via a case study of one such aftermarket TCU device. We show that the device is vulnerable via a range of both local and remote vectors. Moreover, post-compromise the device has complete access to the vehicle's CAN bus and is able to inject arbitrary payloads to automotive ECUs and thus interactively control vehicular systems at arbitrary distance.

In the remainder of this paper, we provide some of the contextual and technical background for this problem, explain our threat model, and describe our experimental methodology in identifying TCU vulnerabilities. Then, in addition to describing our findings, we discuss the clear set of security controls that would reduce the ease with which our attacks succeeded.

2 Background

In this section, we briefly review the aftermarket TCU ecosystem, describe the device we have studied, and highlight key related work.

2.1 Aftermarket TCU ecosystem

Aftermarket TCUs are employed for a wide array of purposes. For example, TCUs offered by Dash and Automatic Labs provide a "smart driving assistant" that gives input (via a connection with your smartphone) about how to improve fuel efficiency, real-time stats about the car's performance (e.g., engine temperature) as well as automatic crash reporting. Delphi's Connect incorporates a cellular modem and provides geo-fencing (e.g., for teen drivers), remote engine start, trip tracking, and explaining vehicle health (i.e., a "scan tool" for consumers). Other such consumer-focused vendors including Zubie, Fuse, Carvoyant, and Kiwi. A related market revolves around consumer security and surveillance. For example, Carlock sells devices focused on vehicle theft, Splitsecnd on automatic crash reporting, and an array of vendors sell tracking TCUs marketed to private investigators and suspicious partners. Variants of these are also marketed for commercial fleet management purposes (to track the location, driving habits, and vehicular health of large vehicle fleets).

However, perhaps the most interesting sector experimenting with TCUs is automotive insurance. Several insurance carriers are using TCU analytics to offer "safe driving" discounts (e.g., Progressive Snapshot), validated low-mileage discounts (Travellers Intellidrive), or for offering pay-per-mile insurance products (Metromile and Allstate's Drive Wise). This sector is particularly interesting because the TCU device is supplied by the insurance provider as a condition of the issued policy and, due to their broad footprint, will connect millions of customers.[1] Moreover, while individual carriers frequently write or specialize the presentation and management software, it is common that the underlying hardware and embedded software is provided by a smaller set of common OEM providers (e.g., Progressive's Snapshot is a rebranded version of the Xirgo XT-3000, Metromile's service is based on C4E dongles from Mobile Devices, etc.)

Most of these products are built around a small number of embedded SoC solutions, typically using some form of low-power ARM core, a sensor package (gyroscope, accelerometers, GPIO), a GPS chip, and a cellular and/or Bluetooth modem. They are typically able to draw power directly from the OBD-II port (although some also incorporate rechargeable batteries) and incorporate a CAN transceiver chip for sending and receiving signals in accordance with the CAN protocol.

2.2 Related work

Quite a bit of automotive security research has focused on the limitations of the CAN bus. The CAN bus standard was designed by the auto industry to provide a way for the various electronic control units (ECU) in an automobile to communicate. It forms a simple bus topology network in which messages are tagged with identifiers that are associated with particular ECUs or functions (the layout of a sample CAN frame is shown in Figure 1). However, there is no enforced addressing or message source identification and thus any device on the CAN network can send any message in a manner indistinguishable to the target ECU.

In previous work, Koscher et al. [4] and Miller and Valasek [6] have shown that this property allows trivial replay attacks, activating a range of automotive functions. Moreover, both groups showed that many ECUs provide a reflashing protocol via the CAN bus (typically following the SAE J2534 standard), allowing a peer to overwrite the code with arbitrary functionality. Taken together, these results have been shown to allow control over safety critical vehicle functionality including the drivetrain and brakes on three different makes and models of vehicles. However, all these attacks required some physical access to the CAN bus (e.g., via the OBD-II port) thus increasing the cost and risk to the attacker.

In follow-on work to [4], Checkoway et al. showed that a variety of completely remote exploits were pos-

[1] Snapshot alone serves over two million customers according to 2014 press materials from Progressive

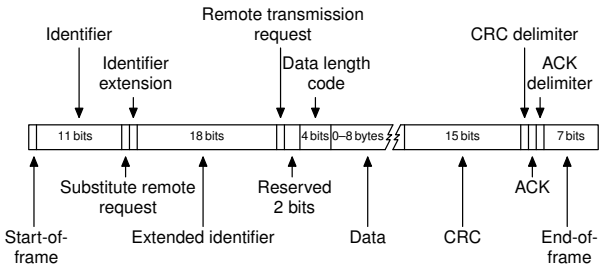

Figure 1: The CAN frame message format showing the message identifier, packet size, message data, and CRC sections.

sible, including by remotely subverting software in the TCU designed to demodulate and frame digital packets on an underlying audio channel [1]. It is our understanding that Miller and Valasek will present related results at the 2015 BlackHat USA Briefings as well. However, as mentioned previously, these TCUs are "built-ins" supplied by the vehicle manufacturer and thus there are a number of avenues available for addressing such problem (e.g., updating software during regular service visits) that would not be feasible for aftermarket products.

At least two researchers have specifically examined the security of aftermarket "dongle" TCUs. The most widely publicized is Corey Thuen's examination of the Progressive Snapshot device, wherein he extracted the firmware and performed a security analysis showing likely points of attack (although stopping short of identifying a particular vulnerability or demonstrating a working exploit) [8, 9]. Ron Ofir and Ofer Kapora's earlier work went further and examined the Zubie telematics device and showed that if one could mount a cellular man-in-the-middle attack (e.g., as feasible with a Stingray-like base station) then they could update the software and thereby inject CAN packets (e.g., unlocking doors) [7]. In our work, we show that such capabilities are unnecessary and that at least one popular TCU can be compromised in several different ways without any special capabilities.

3 TCU Attack Surface

Our reference TCU was purchased from eBay (shown in Figure 2) manufactured by Mobile Devices Ingenierie and known to be used for insurance purposes (Metromile's per-mile consumer policies and for their commercial driver on-demand policies offered in partnership with Uber) among other settings [5, 10].[2] Our reference

[2]There are several versions of this product, and slightly different configurations are used by different customers, but in our experience the software is substantially similar among all of them in the C4E family and to the best of our knowledge all of the vulnerabilities we identify exist across a range of product offerings and deployments.

Figure 2: Our TCU showing both the OBD-II and USB connectivity.

device included a GPS, 3D Accelerometer, and 3D Gyroscope. It is based around a 500MHz ARM11 CPU, 64MB RAM, with 256MB flash storage. For external connectivity it provides a USB port for programming, a 2G cellular data modem (3G in later models), and a CAN transceiver chip wired to the OBD-II pins.

In evaluating the device, we consider two threat models: local and remote. In the local threat model we evaluate how an attacker may be able to attack the TCU directly in an effort to gain control over the device (e.g., for compromising the device after intercepting it during shipping or after obtaining brief physical access to the vehicle such as a valet might have). In the remote model we assume that the attacker does not have physical access to the TCU, but the TCU is installed in an automobile of a victim and the attacker's goal is to compromise the vehicle.

3.1 Local attack surface

The local threat model assumes physical access to the TCU, and the goal of the attacker is to penetrate the TCU itself. We do not evaluate any automobile communications in this model as we assume an attacker with physical access already has access to the vehicle.

As mentioned before, our TCU includes a mini-USB connector for debugging purposes, which is configured to emulate a network adapter (i.e., once connected the TCU appears as a device on the network). If the debug console is enabled on the TCU then a web server is configured to listen on port 80 and a telnet console on port 23 (both without authentication).[3] The debugging interface also allows for minimal configuration changes and software updates to be installed. In addition, we identified several test points on the internal circuit board (requiring opening the packaging) and with physical access an attacker can also remove the NAND flash chip and dump or alter the contents (we discuss this attack in Section 4.1).

[3]Later we found that authentication could be enabled, however one could trivially bypass it by modifying the URL visited. None of the devices we tested had this authentication enabled.

3.2 Remote attack surface

In the remote context, we assume that the adversary has no physical access to the device or vehicle and may not even know where they are located geographically.

The attack surface for the remote threat model is a 2G modem (3G in later versions) that provides remote connectivity over cellular networks. Both SMS and IP data communications are used by the device for various functions. Both interfaces will respond directly to requests and thus can be remotely identified if the devices can be remotely addressed (we discuss this issue further in Section 4.3).

4 Evaluation

To evaluate the security of this TCU device, we explored a range of local and remote vectors, the difficulty in identifying such devices and the challenges in manipulating the device to control a vehicle. We describe our experience here (in roughly the order that our experiments took place).

4.1 Local attacks

Given physical access to the device, the most convenient means of connection is to interface with the USB port. When a USB cable was connected to a powered device, we were presented with a USB network interface. From the available device documentation, we were able to determine its subnet and IP address and then, using the network interface, probe for running services of interest. The device responded on standard ports for the telnet, web and SSH services.

Web/telnet console access

When contacted, the web and telnet servers both presented a specialized interface for querying and setting device parameters, as well as retrieving status information. This includes privacy-sensitive information such as the device's current location as indicated by GPS. Additionally, a set of more advanced commands were exposed including an interface to send a SMS message to a specified number. By using this feature, it was straightforward to identify the phone number used in the SIM chip for subsequent SMS-based testing. We also identified an interface for retrieving the version and state for all the TCU's internal components as well as the log files of the underlying Linux kernel (both of which were helpful in subsequent analysis). Finally, all configuration variables for software modules running were exposed and changeable..

NAND dump

To attempt to get more information on the software running on the device, we removed the NAND flash chip and extracted the data using a hardware reader. Since the TCU is a simple embedded device, the NAND flash does not have a controller to give it a block abstraction, such as with a desktop flash drive. Instead it exposes access to the raw erase blocks and depends on the file system to fully manage their use.

Thus, to access the extracted data, we used the Linux nandsim kernel module. This module creates a simulated raw NAND flash chip and is typically used for development of file systems that will be deployed over these types of chips. We configured this module to simulate a clone of the NAND present on the TCU.[4]

Replicating this structure and appropriately copying the dumped data gave us a perfect mountable copy of the original chip. The Unsorted Block Image File System (UBIFS) manages error detection and correction on raw flash chips and keeps track of bad blocks. Using the UBI file system module allowed us to mount some of the TCU partitions for reading.

One of these partitions contained the main third party software running on the TCU. This includes a collection of scripts and binaries related to various system actions, such as moving between sleep modes and running updates. We also found the main Java framework that manages interaction with the TCU and sensor information collection and transmission. Most importantly, the data contained a number of public and private cryptographic keys and certificates.

SSH keys

Initially, we had no access to the SSH service running on the device, or even knowledge of what users existed or what credentials to supply. This changed after we explored the dump of the NAND flash and identified the private key for the root user. This key gave us the ability to authenticate to the device over SSH and directly obtain a root shell. From here we could read and write any file, execute arbitrary commands and download and install additional software to create arbitrary functionality. At first blush, this capability did not seem particular threatening, since it first depended on obtaining the root private key via physical access to the device. However, upon testing this SSH key on several other TCU devices from the same manufacturer (including in production contexts) we found that the same key was in use. Thus, it appears that the key we obtained is the default (and potentially singular) key for all such devices. This

[4]The Linux logs taken from the web interface indicated the NAND type, its block size and how it was partitioned.

in turn was one of the enabling findings for our exploration of remote attacks. [5]

4.2 Remote attacks

After obtaining full control of the TCU via the USB interface, we shifted our focus to the software exposed via wireless interfaces available on the device: a cellular data interface providing Internet connectivity and an SMS interface.

Internet-based

In examining the code on the device, we discovered that the web, telnet console, and SSH servers were bound to all of the network interfaces and not simply USB. Since this includes an Internet-connected cellular data modem, an outside attacker could simply login to the TCU directly using the previously identified SSH key (assuming the IP address were known). However, the particular devices we tested were indirectly protected from this attack vector as their cellular carriers made pervasive use of network address translation (NAT) and thus we could not directly address them. We note that this implementation is entirely a property of the carrier used, and if the device used a cellular provider allowing direct addressing then this means of access would be effective. Indeed, as we discuss later, we have found well over a thousand such devices that are Internet exposed.

SMS-based

In addition to the cellular data connection, the device is SMS capable and thus, if the phone number of a TCU is known then an outside party can send it SMS messages.

Searching online revealed documentation for an SMS administration interface which we discovered was also enabled on all the devices we tested. Such commands include retrieving or setting the same properties exposed in the local debug interfaces, retrieval of sensor status information such as modem, GPS position, and others, initiating a remote update and more.

Of these, the remote update capability is the most serious as it potentially provides a mechanism to obtain a reverse shell and thus arbitrary access.

Limited documentation could be found on the update procedure for the TCU; however after examining logs created from initiating an update via SMS we were able to determine the full update procedure as described in Figure 3. The update procedure retrieves a special text file, which for the purposes of this paper we'll call `UpdateFile.txt`, from the server.[6] This file contains a file name, path, and hash of files to add or remove from the system. The steps for the update procedure are as follows:

1. The SMS command `rupd,USER,HOST,PORT,DIR` is sent to the TCU, which responds with `update,started`.
2. The TCU uses SCP to remote into the `HOST` on port `PORT` as user `USER` and retrieves `UpdateFile.txt` from `DIR`.
3. `UpdateFile.txt` is examined and files which have incorrect hashes or do not exist on the local system are then retrieved via SCP from the update server to a temporary directory on the TCU.
4. If the hashes of the new files match those found in `UpdateFile.txt`, the new files are moved to their target directories, otherwise the update process is restarted.
5. If `UpdateFile.txt` contains any of the following console commands, they are executed performing the appropriate action: `clear`, `identify`, `status`, or `reset`.

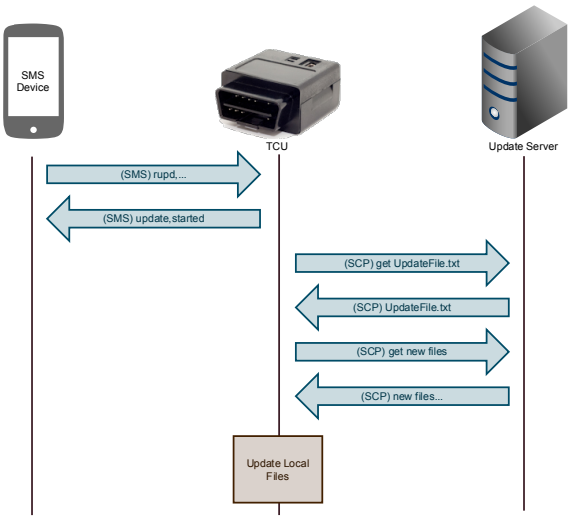

Figure 3: Remote update procedure

There are a number of unfortunate choices in this design. First, updates are not cryptographically signed in any way and thus it is easy to substitute arbitrary code in

[5] We also found that the devices have common root and user passwords as well and that ssh is configured to accept password login.

[6] The true filename is a minor secret and easy to determine with access to a device, but we do not use it here simply to filter out potential low-level war dialing attacks while the problems described here are being fixed.

an update. Second, while the server authenticates the device (somewhat uselessly since all devices also seem to share the same public and private update keypair) the devie does not authenticate the server. Finally, the interface allows an arbitrary update server to be chosen providing significant flexibility to the attacker.

To verify these limitations we created a rogue update server that serves an update which immediately spawns a reverse shell and SSH tunnel to the victim TCU. The attack, shown in Figure 4, consists of the following steps:

1. Initiate a remote update using our rogue server via telnet, web, or SMS.
2. The TCU downloads `UpdateFile.txt` containing `console.bak` (the original console binary), `console` (a shell script we created which contains our attack), and the command `clear`.
3. After the TCU downloads all the files and replaces the system console command with our console script it calls "`console clear`" to clear the logs.
4. Our console script starts and replaces itself with the original `console.bak`, starts a reverse shell and SSH tunnel, sends a SMS to us informing us the attack was successfully, and then calls the original console with the `clear` command.
5. Once we receive the SMS from our script, or get a notification from the update server that the reverse shell is ready, we can SSH or tunnel into the device to get a root shell and access to the telnet and web interfaces.

Finally, we also validated that this update procedure can be triggered via the web and telnet console interface as well if they are accessible.

4.3 Finding devices

All forms of remote compromise (remote login via ssh, or update via web, telnet console or SMS) require knowledge of the TCU addresses – either its globally reachable IP address or the phone number associated with the SIM card. We found that there were several means of finding this information in the wild.

If the TCU has a cellular network provider that does not use NAT, the provided built-in web server is accessible from the Internet and consequently can be indexed by search engines. Indeed, doing an Internet search from strings of words unique to the web interface revealed IP addresses for a range of likely TCUs. Similarly, it is possible to identify vulnerable TCUs based on the information exported from their telnet and SSH servers. The online service Shodan collects and indexes a large amount

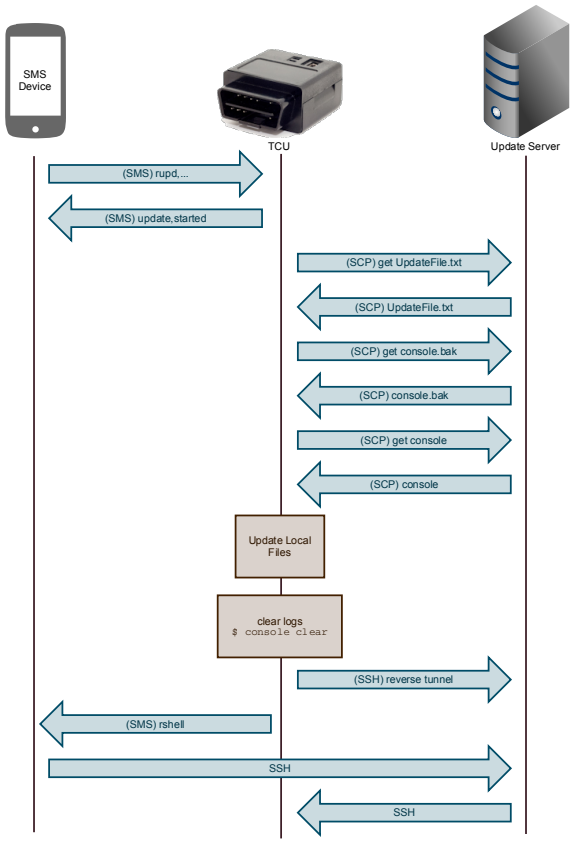

Figure 4: Remote exploitation via a malicious update server

of meta-data based on Internet scans.[7] Since all the TCUs from this manufacturer use the same SSH server key, the server fingerprint presented when connecting is also identical. Searching for this fingerprint yielded the IP addresses of about 1500 potential devices. Searching for the particular welcome message presented at the start of a telnet session found about 3000 unique IP address. The majority of the addresses were from Spanish ISPs. We suspect this is a result of NAT not being deployed by at least one wireless carrier in Spain.

We next assessed the possibility of finding devices by phone number. Since the TCU requires a data connection to send the sensor information to a remote server, the unit typically ships with a cellular connection with a prepaid data plan. The numbers for these connections were not random. We found evidence that many are sequentially assigned numbers from the 566 area code, which is reserved for "personal communication services". If the phone number of one of these devices can be determined, the nearby numbers have a high potential to also be TCUs. Moreover, simply sending a SMS admin-

[7] http://www.shodanhq.com/

istration command, such as "`status`," would confirm a TCU's identity, and makes implementing a "war dialer" for enumerating such devices relatively straightforward.

4.4 CAN bus capabilities

Having compromised a TCU, the next question is what capabilities does it have. For example, one potential TCU design would only receive CAN data from the OBD-II port and would be incapable, at a hardware level, of sending CAN messages. Alternatively, a TCU might allow arbitrary CAN access and thereby allow an attacker to directly communicate with the full range of ECUs on the vehicle.

In exploring this question we identified two different firmware versions in use on our family of TCUs. The most recent version includes SocketCAN, which is a Linux kernel module that presents the CAN bus as a network interface. Additionally, these newer devices shipped with the Linux can-utils package which includes tools for reading, saving, creating, and replaying CAN messages, much like the way one can do with packet captures for traditional network interfaces. With these tools we found we could send and receive arbitrary CAN packets.

The older version of the firmware implemented a custom interface to send commands from the main ARM CPU to a PIC micro-controller which controls the physical CAN controller on the TCU. After analyzing the serial line between the ARM and PIC chips, we were able to understand the protocol enough to send our own CAN packets to the PIC chip. However, the PIC chip would periodically query OBD-II and not send CAN packets if it detected that the vehicle was not in ACC (Accessory) mode and the ignition was off.

The TCUs shipped with a utility to flash the PIC firmware, which we were able to modify to also dump the existing firmware. After reverse engineering the firmware dump, we were able to identify the ACC and ignition checks, disable them, and reflash the "bypass" version of the PIC image. This modification allowed us to send CAN packets irrespective of what state the automobile is in.

4.5 Proof of Concept attack

Finally, having discovered how to remotely compromise the TCU and convince it to send arbitrary CAN packets, we constructed an end-to-end demonstration highlighting the potential seriousness of the vulnerability. In our demo, we assume that the phone number of the device is known and that an attack payload for the victim car is available (the combination of past experience and the increasing availability of public CAN "recipes" allowed us to construct our payload in only a few days even though we have never worked with the test car before).

In our demo, we used a two-stage attack. The initial payload is delivered via an SMS update command and takes a minute to apply due to the slow 2G connection. This first stage is very similar to the method described earlier, but does not restore the original `console` application back to the version saved as `console.bak`. Instead, it will start a reverse shell if we send the correct SMS command, otherwise it will just call `console.bak` directly. This allows for future reverse shells to be started without the need to re-download the `console` file every time. The second stage allows us to send another SMS command to the device to start another reverse shell. Starting a reverse shell during stage 2 takes a few seconds at most. Putting this together, we worked with a volunteer driver (operating at low speed to prevent injury) and demonstrated remote control over both body functions (remotely turning on windshield wipers) and brakes (selectively applying brakes and selectively disabling brakes). While this test was conducted at close distance for safety purposes, it was not limited by proximity and could have been carried out at arbitrary distance.

5 Proposed Solutions

In this section we provide some suggestions which, if implemented, we feel would prevent the types of attacks described in this paper.

5.1 Require update authentication

Once an update is initiated, it is performed entirely over SSH/SCP. While SSH provides strong privacy and integrity for all communication, the remote host is not verified. Instead, a device authenticates *itself* to the server and does not verify the server's fingerprint. Additionally, once the update is downloaded from the server, there is no cryptographic verification to ensure authenticity of the update. We suggest implementing code signing to verify that updates received are actually the intended update provided my the manufacture and written by a rogue third party.

5.2 Stronger SMS authentication

While we found that these devices do support a SMS whitelist and blacklist, we only found it enabled on a small subset of them. It is also possible to remotely disable or alter the SMS whitelist with the debug console, web interface, or via a SMS from a "valid" number. SMS whitelisting is not enough to perform remote authentication since telephone numbers are easily spoofable and

whitelists cannot be secret since they are distributed with the TCUs [2].[8] Remote SMS administration should either be disabled entirely or use an additional form of authentication on top of SMS.

5.3 Key management

Every device shipped with both the private and public keys for both the root SSH account and for logging in to the update server. There is no need to have the private key for root login on the device–only the public key should be stored on the device. By keeping the private key on the device the manufacturers are implicitly distributing the key to anyone who is able to gain access to the device's file system. Furthermore, we see no need for the device to authenticate itself to the update server–verifying the update server's fingerprint/public key should be sufficient. If, for some reason, the device needs to authenticate itself to the update server, each device should have a unique key (and whose generation should be done with sufficent entropy, a notorious problem with embedded systems).

5.4 Password management

After dumping `/etc/shadow` we discovered that the passwords were hashed with MD5, and of the two users (root and user) only the user account used a salt. Running a moderately sized word list against the hashes found the salted and hashed user password in under 10 seconds. The user password was 8 characters make up of only vowels and sequential numbers. It would have been better to run different processes as different users (each with the least permissions required to perform their intended task), not all as root, and disable root SSH access.

5.5 Disable WAN administration

The web, telnet, and SSH services listen on all available interfaces. This configuration means if the device has a WAN connection (via the 2G modem) it is exposing its debug interface to the world. This is especially bad when the wireless ISP does not implement NAT. In this case the devices can easily be found by scanning the web, which resulted in many being indexed by Google and Shodan as discussed in Section 4.3.

5.6 Require console authentication

Buried deep in the file system of the device was an option to require authentication for remote web or console access. We found no way to enable this form of authentication via any of the intended configuration interfaces, and none of the devices we examined had authentication enabled. Additionally, once enabled, it could be trivially bypassed for the web interface by altering the URL. This leads us to believe that the console authentication feature was never completed. We suggest requiring authentication for any sort of debugging access, local or remote. And for such authentication to be enforced in such a way that it cannot be bypassed. It would also be ideal if the authentication was different for each device so that if the authentication for one device was compromised it would not allow access to the debug console of others.

5.7 Maintain update server

The domain name specified in the debug console used to check for updates was found to be unregistered and available. We believe when the TCUs send collected data back to their cloud servers, the response may be able to initiate an update from an alternate host. However the default update domain is still configured and referenced in the documentation. It is unclear whether the devices periodically poll the default update server or if you need to instruct it to check for an available update. We did not attempt to register the domain, but if an attacker did they may be able to take over every TCU that checks for an update.

6 Disclosure

We disclosed our understanding of the problems with the C4E class of TCUs to the vendor (Mobile Devices), to Metromile (a customer of theirs using that platform), and to Uber (a customer of Metromile).[9] All were supportive of our work, appreciative that we had informed them in advance, and intimated that the problems would be fixed (indeed, Metromile was concrete in its plans to disable all SMS access on its branded devices, consistent with our recommendation) or had already been fixed. However, we also experienced some of the challenges with this space arising from complex supply chain in which a device is customized for different markets.

In particular, while Mobile Devices indicated that many of the problems we found had been fixed in subsequent versions of software, we did not find those software updates on recent production devices we had tested (i.e., a newer software update that fixes bugs only helps if there is a mechanism for then deploying it across the legacy customers of the platform). Similarly, Mobile Devices suggested that our attacks should not work on production deployments of their devices since the consoles are only enabled in "developer mode" which should not

[8] Indeed, we found the same entries in the whitelist across multiple devices used by the same provider.

[9] We also recently informed DHS/US CERT, which created a team specifically tasked with dealing with various OBD-II vulnerability disclosures.

be deployed except for testing and that production deployments should not include SIM cards that allow SMS exchanges. While we have no way to evaluate these claims, they seem reasonable. Yet even if we take these statements at face value, they suggest a disconnect in the interface with customers since we identified these problems in a number of production devices directly (to say nothing of the several thousand we identified online).

This suggests that vendors of such devices (i.e., those that simultaneously are exposed to remote access and connect to safety-critical system) may need to anticipate efficient ways to roll out updates on short notice to a wide array of customers (from hobbyists to fleet managers).

7 Conclusion

In this paper, we presented a security analysis of a popular aftermarket telematics control unit. We were able to demonstrate both local and remote vulnerabilities, resulting from a combination of bad architectural decisions (e.g., the design of the update protocol) and particular configuration options (e.g., the use of SMS and debugging features in production deployments and the use of identical keys and passwords among such devices). We have experimentally validated that these vulnerabilities can be exploited; in particular, demonstrating a complete remote compromise via SMS. Once compromised, we have shown that the TCU can send arbitrary CAN packets, and that this is sufficient to remotely control safety-critical automobile features (e.g., the brakes). Finally, we provided some immediate suggestions for the manufacturer that would improve their security. Long-term we believe that the industry will require stronger mechanisms for code signing, authentication, and for limiting what kinds of communications a particular device can engage in (e.g., CAN firewalls), as well as efficient mechanisms for updating legacy vulnerable devices in the field.

Acknowledgments

We are grateful to Sid Karin for his assistance in our experiments and for the feedback from the anonymous reviewers. This work was supported in part by the National Science Foundation grant NSF-0963702 and by generous research, operational and/or in-kind support from the UCSD Center for Networked Systems (CNS).

References

[1] S. Checkoway, D. McCoy, D. Anderson, B. Kantor, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proceedings of the USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[2] N. Golde, K. Redon, and R. Borgaonkar. Weaponizing Femtocells: The Effect of Rogue Devices on Mobile Telecommunications. In *NDSS*, San Diego, CA, Feb. 2012. The Internet Society.

[3] A. Greenberg. Hackers Remotely Kill a Jeep on the Highway. http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway, Jul. 2015.

[4] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proceedings of the IEEE Symposium and Security and Privacy*, Oakland, CA, May 2010.

[5] Metromile: Per-mile insurance. https://www.metromile.com/insurance.

[6] C. Miller and C. Valasek. Adventures in automotive networks and control units. In *DEF CON 21*, Las Vegas, NV, Aug. 2013.

[7] R. Ofir and O. Kapora. A remote attack on an aftermarket telematics service. http://argus-sec.com/blog/remote-attack-aftermarket-telematics-service, Jul. 2014.

[8] Thomas Fox-Brewster. Hacker Says Attacks On 'Insecure' Progressive Insurance Dongle In 2 Million US Cars Could Spawn Road Carnage. http://www.forbes.com/sites/thomasbrewster/2015/01/15/researcher-says-progressive-insurance-dongle-totally-insecure, Jan. 2015.

[9] C. Thuen. Remote Control Automobiles. http://www.digitalbond.com/blog/2015/02/02/s4x15-video-remote-control-automobiles, Feb. 2015.

[10] Metromile To Offer Per-Mile Insurance Option To Uber Driver Partners. https://www.metromile.com/blog/2015128uber-partnership, Jan. 2015.