Can you find me now?

Unlocking the Verizon Wireless xv6800 (HTC Titan) GPS

10/2008

Skywing
skywing_uninformed@valhallalegends.com

Contents

1 Foreword

Abstract: In August 2008 Verizon Wireless released a firmware upgrade for their xv6800 (rebranded HTC Titan) line of Windows Mobile smartphones that provided a number of new features previously unavailable on the device on the initial release firmware. In particular, support for accessing the device's built-in Qualcomm gpsOne assisted GPS chipset was introduced with this update. However, Verizon Wireless elected to attempt to lock down the GPS hardware on xv6800 such that only applications authorized by Verizon Wireless would be able to access the device's built-in GPS hardware and perform location-based functions (such as GPS-assisted navigation). The mechanism used to lock down the GPS hardware is entirely client-side based, however, and as such suffers from fundamental limitations in terms of how effective the lock down can be in the face of an almost fully user-programmable Windows Mobile-based device. This article outlines the basic philosophy used to prevent unauthorized applications from accessing the GPS hardware and provides a discussion of several of the flaws inherent in the chosen design of the protection mechanism. In addition, several pitfalls relating to debugging and reverse engineering programs on Windows Mobile are also discussed. Finally, several suggested design alterations that would have mitigated some of the flaws in the current GPS lock down system from the perspective of safeguarding the privacy of user location data are also presented.

2 Introduction

The Verizon Wireless xv6800 (which is in and of itself a rebranded version of the HTC Titan, with a carrier-customized firmware loadout) is a recently released Windows Mobile-based smartphone. A firmware update released during August 2008 enabled several new features on the device. For the purposes of this article, the author has elected to focus on the embedded Qualcomm gpsOne chipset, which provides assisted GPS facilities to applications running on the device.

With the official firmware upgrade (known as MR1), the assisted GPS support on the device, which had previously remained inaccessible when using carrier-supported firmware, was activated, albeit with a catch; only applications that were approved by Verizon Wireless were able to access the built-in GPS hardware present on the device. Although third-party applications could access an externally connected (for example, Bluetooth-enabled) GPS device, the Qualcomm gpsOne chipset embedded in the phone itself remained inaccessible. Coinciding with the public release of the xv6800 MR1 firmware, Verizon Wireless also began making available a subscription-based application (called "VZ Navigator"), which provides voice-based turn-by-turn navigation on the xv6800 via the usage of the device's built-in GPS hardware.

There have been a variety of third-party firmware images released for the xv6800 that mix-and-match portions of official firmware releases from other carriers supporting their own rebranded versions of xv6800 (HTC Titan). Some of these custom firmware images enable access to the gpsOne hardware, albeit with several caveats. In particular, until recently, assisted GPS mode, wherein the cellular network aids the device in acquiring a GPS fix, was not available on Verizon Wireless's network with custom firmware images; only standalone GPS mode (which requires waiting for a "cold lock" on three GPS satellites, a process that may take many minutes after device boot) was enabled. In addition, installing these custom firmware images requires patching out a signature check in the software loader on the device. This procedure may be considered dangerous if one wishes to retain hardware warranty support (which may be desirable, given the steep unsubsidized cost of the device).

Furthermore, should one install the official Verizon Wireless MR1 firmware upgrade, the gpsOne hardware on the device would remain locked down even if one switched to a currently available third-party firmware images. This is likely due to a sticky setting written to the firmware during the carrier provisioning process at the completion of the MR1 firmware upgrade. As the presently available third-party ROM images do not wipe the area of the device's firmware which seems to control the GPS hardware's lockdown state, it becomes difficult to unlock the GPS hardware after having upgraded to the MR1 firmware image. A lengthy process is available to undo this change, but it involves the complete reset of most provisioning settings on the device, such that the phone must be partially manually reprovisioned, as opposed to utilizing the over-the-air provisioning support.

Given the downsides of relying on custom firmware images for enabling the built-in GPS hardware on the xv6800, the official firmware release does pose a reasonable attraction. However, the locking down of the GPS hardware to only Verizon Wireless authorized applications is undesirable should one wish to use third-party location-enabled applications with the built-in GPS hardware, such as Google Maps or Microsoft's Live Search.

Verizon Wireless indicates that third-party application usage of the GPS hardware on their devices is subject to Verizon Wireless-dictated policies and procedures[1]. In particular, the security of user location information is often cited[2] as a reason for requiring location-enabled applications to be certified by Verizon Wireless. Unfortunately, the mechanism deployed to lock built-in GPS hardware on the xv6800 provides very little in the way of true security against third-party programs (malicious or otherwise) from accessing location information. In fact, given Windows Mobile 6's lack of "hard" process isolation, it is questionable as to whether it is even technically feasible to provide a truly secure protection mechanism on a device that allows user-supplied programs to be loaded and executed.

While there may be golden intentions in attempting to protect users from ma-

licious programs designed to harvest their location information on-the-fly, the protection system as implemented to control access to the gpsOne chipset on the xv6800 is unfortunately relatively weak. This is at odds with Verizon Wireless's stated goals of attemting to protect the security of a user's location information, and thus may place users at risk.

3 Overview of Protection Mechanisms

There are multiple levels of protection mechanisms built-in to both the MR1 firmware image for the xv6800, as well as the GPS-enabled subscription VZ Navigator software that Verizon Wireless supports as the sole officially sanctioned location-based application (at the time of this article's writing). The protection mechanisms can be broken up into those that exist on the device firmware itself, and those that exist in the VZ Navigator software.

3.1 Firmware-based Protection Mechanisms

The MR1 firmware provides the underlying foundation of the built-in GPS hardware lockdown logic. There are several built-in software components that are "baked into" the firmware image and support the GPS lockdown system. The principle design underpinning the firmware-based protection system, however, is a fairly run of the mill security-through-obscurity based approach. In particular, GPS location information obtained by the built-in gpsOne hardware (specifically, latitude and longitude) is encrypted. Only programs that understand how to decrypt the position information are able to make sense of any data returned by the gpsOne chipset.

Furthermore, in order to initiate a location fix via the built-in gpsOne hardware, an application must continually answer correctly to a series of challenge-response interactions with the gpsOne chipset driver (and thus the radio firmware on the device). The reason for implementing both a challenge-response mechanism as well as obfuscating the actual GPS location will become apparent after further discussion.

The firmware-based protected gpsOne interface has several constituent layers, with supporting code present at radio-firmware level, kernel driver level, and user mode application level.

At the lowest level, the radio firmware for the device chipset would appear to have a hand in obfuscating returned GPS positioning data. This assumption is logically based on a strings dump of radio firmware images indicating the presence of AES-related calls in GPS-related code (AES is used to encrypt the returned location information), and the fact that switching to a custom firmware

image after installing the MR1 update does not re-enable the plaintext gpsOne interface).

Between the radio firmware (which executes outside the context of Windows Mobile) and the OS itself, there exists a kernel mode Windows Mobile driver known as the GPS intermediate driver. This module (gpsid_qct.dll) provides an interface between user mode callers and the GPS hardware on the device. It also provides support for multiplexing a single piece of GPS hardware across multiple user mode applications concurrently (a standard feature of Windows Mobile's GPS support). However, Verizon Wireless has broken this support with the locked down GPS logic that has been placed in the xv6800's implementation of the GPS intermediate driver.

Beneath the GPS intermediate driver, there are two different interfaces that are supported for the collection of location data on Windows Mobile-based devices[4]. The first of these is an emulated serial port that is exposed to user mode, and implements a standard NMEA-compatible text-based interface for accessing location information. This interface has also been broken by the GPS intermediate driver used by Verizon Wireless on the xv6800, for reasons that will become clear upon further discussion.

The second interface for retrieving location information via the GPS intermediate driver is a set of IOCTLs implemented by the GPS intermediate driver to retrieve parsed (binary) GPS data from the currently-active GPS hardware (returned as C-style structures). User mode callers do not typically call these IOCTLs directly from their code, but instead indirect through a set of thin C API wrappers in a system-supplied module called gpsapi.dll. This interface is also broken by the GPS lockdown logic in the GPS intermediate driver, although an extended version of this IOCTL-based interface is used by GPS-enabled applications that support the locked down mode of operation on the xv6800.

Verizon Wireless ships a custom module parallel to gpsapi.dll on the xv6800, named oemgpsOne.dll. This module exports a superset of the APIs provided by the standard gpsapi.dll (although there are slight differences in function names). Additionally, new APIs (which are, as in gpsapi.dll, simply thin wrappers around IOCTL requests sent to the GPS intermediate driver) are provided to manage the challenge-response and encrypted GPS location aspects of the gpsOne lockdown system present on the xv6800. Through correct usage of the APIs exported by oemgpsOne.dll, a program with knowledge of the GPS lock down system can retrieve valid positioning data from the gpsOne chipset on the device.

Applications that are approved by Verizon Wireless for location-enabled operation make calls to a library developed by Verizon Wireless and Autodesk, named LBSDriver.dll, which is itself a client of oemgpsOne.dll. LBSDriver.dll and its security measures are discussed later, along with VZ Navigator.

3.1.1 Application Authorization via Challenge-response

In order to activate the gpsOne hardware on the xv6800 and request a GPS location fix, an application must receive a challenge data block from the gpsOne driver and perform a secret transform on the given data in order to create a well-formed response. Until this process is completed, the gpsOne hardware will not attempt to return a location fix. Furthermore, a location-enabled application using the built-in gpsOne hardware must continually complete additional challenge-response sequences (using the same underlying algorithms) as it continues to acquire updated location fixes from the gpsOne hardware.

The first step in connecting to the GPS intermediate driver to retrieve valid position information is to open a handle to a GPS intermediate driver instance. This is accomplished with a call to an oemgpsOne.dll export by the name of oGPSOpenDevice. The parameters and return value of this function are analogous to the standard Windows Mobile GPSOpenDevice routine[5].

```
HANDLE
oGPSOpenDevice(
   __in HANDLE NewLocationData,
   __in HANDLE DeviceStateChange,
   __in const WCHAR *DeviceName,
   __in DWORD Flags
   );
```

After a handle to the GPS intermediate driver instance is available, the next step in preparing for the challenge-response sequence is to issue a call to a second function implemented by oemgpsOne.dll, named oGPSGetBaseSSD. This routine returns a session-specific blob of data that is later used in the challenge-response process. In the current implementation, the returned blob appears to always contain the same data across every invocation.

```
DWORD
oGPSGetBaseSSD(
   __in HANDLE Device,
   __out unsigned char *Buf, // sizeof = 0x10
   __out unsigned long *BufLength, // 0x10
   __out unsigned short *Buf2 // sizeof = 0x10
   );
```

Next, the GPS intermediate driver must be provided with a valid event handle to signal when a new challenge cycle has been requested by the driver. This is accomplished via a call to the oGPSEnableSecurity function in oemgpsOne.dll.

```
DWORD
oGPSEnableSecurity(
   __in HANDLE Device,
   __in HANDLE SecurityChangeEvent
   );
```

After the session-specific blob has been retrieved, and an event handle for new challenge requests has been provided to the GPS intermediate driver, the next step is to receive a challenge block from the GPS intermediate driver and compute a valid response. The application must wait until the GPS intermediate driver signals the challenge request event before requesting the current challenge data block. Once the driver signals the event that was passed to oGPSEnableSecurity, the application must execute one challenge-response cycle.

Challenge data blocks are retrieved from the gpsOne driver via a call to a routine exported from oemgpsOne.dll, named oGPSReadSecurityConfig. As per the prototype, this routine takes a handle to the GPS intermediate driver instance, and returns a blob of data used to generate a challenge response.

```
DWORD
oGPSReadSecurityConfig(
   __in HANDLE Device,
   __out unsigned char *Buf // On return, 0x4 + 1 + 1 + Buf[0x6] (max length 0x1c total)
   );
```

After the challenge data blob has been retrieved via a call to oGPSReadSecurityConfig, the GPS lockdown-aware application must perform a series of secret transformations on it before indicating a companion response blob down to the GPS intermediate driver. The transformation function consists of some bit-shuffling of the challenge blob, followed by a SHA-1 hash of the shuffled challenge blob concatenated with the session-specific data blob. This process yields the bulk of the response data less a two-byte header that is prepended prior to indication down to the GPS intermediate driver.

The process of sending the computed challenge-response is accomplished via a call to another function in oemgpsOne.dll, by the name of oGPSWriteSecurityConfig.

```
DWORD
oGPSWriteSecurityConfig(
   __in HANDLE Device,
   __in unsigned char *Buf // 0x1C
   );
```

The GPS intermediate driver will continue to periodically challenge the application while it requests updated position fixes from the gpsOne chipset. This is accomplished by signaling the event passed to oGPSEnableSecurity, which indicates to the application that it should retrieve a new challenge and create a new response, using the mechanism outlined above.

3.1.2 Location Information Encryption

Without passing the challenge-response scheme previously described, the GPS intermediate driver will refuse to return a set of position information from the gpsOne hardware. Even after the challenge-response system has been implemented, however, a secondary layer of security must be addressed. This security layer takes the form of the encryption of the latitude and longitude values returned by the gpsOne chipset.

While this second layer of security may appear superfluous at first glance, there exists a valid reason for it. Recall that the GPS intermediate driver multiplexes a single piece of GPS hardware across multiple applications. In the implementation of the current GPS intermediate driver for the xv6800, the challenge-response scheme appears to map directly to the gpsOne chipset itself.

Thus, once a single program has passed the challenge-response mechanism, and as long as that program continues to respond correctly to challenge-response requests, any program on the system can call any of the standard Windows Mobile GPS interfaces to retrieve location data. This presents the obvious security hole wherein a Verizon Wireless-approved GPS application is started, and then a third-party application using the standard Windows Mobile GPS API is loaded, in effect "piggy-backing" on top of the challenge-response code residing in the approved application to allow access to the embedded gpsOne hardware.

For reasons unclear to the author, the designers of the GPS lockdown system did not choose to simply disable GPS requests not associated with the program that has passed the challenge-response scheme. Instead, a different approach is taken, such that the GPS intermediate driver encrypts the location information that it returns via either serial port or gpsapi.dll interfaces.

In order to make sense of the returned latitude and longitude values, a program must be able to decrypt them. While the GPS intermediate driver provides the decryption key in plaintext equivalent to any program that knows how to request it, this information is not available to clients of the standard Windows Mobile NMEA-compatible virtual serial port or gpsapi.dll interfaces. Aside from latitude and longitude data, however, all other information returned by the standard Windows Mobile GPS interface is unadulterated and valid (this includes altitude and timing information, primarily).

Thus, the first step to decoding valid position values is to call an extended version of the standard Windows Mobile GPSGetPosition routine[6]. This extended routine is named oGPSGetPosition, and it, too, is implemented in oemgpsOne.dll. The prototype matches that of the standard GPSGetPosition, although an extended version of the GPS_POSITION structure containing additional information (including a blob needed to derive the decryption key required to decrypt the longitude and latitude values) is returned.

```
DWORD
oGPSGetPosition(
   __in HANDLE Device,
   __out PGPS_POSITION GPSPosition,
   __in DWORD MaximumAge,
   __in DWORD Flags
   );
```

Decryption of the latitude and longitude information is fairly straight- forward, involving a transform (via the same transformation process described previously) of the challenge data returned as a part of the extended GPS_POSITION structure. This yields an AES key, which is imported into a CryptoAPI key object, and then used in ECB mode to decrypt the latitude and longitude values.

Once decryption is complete, a scaling factor is then applied to the resultant coordinate values, in order to bring them in line with the unit system used by the standard Windows Mobile GPS interfaces.

3.1.3 VZ Navigator (Application-level) Protection Mechanisms

While many parts of the GPS lockdown system are implemented by radio firmware- level, or kernel mode-level code, portions are also implemented in user mode. An approved Verizon Wireless application accesses location information by calling through a module developed by Verizon Wireless and Autodesk, and named LBSDriver.dll. In an approved application, it is the responsibility of LBSDriver.dll to communicate with the GPS intermediate driver via oemgpsOne.dll, and implement the challenge-response and position decryption functionality. LBSDriver.dll then exports a subset of the standard Windows Mobile gpsapi.dll (with several custom additions), for usage by approved programs on the xv6800.

Additionally, LBSDriver.dll implements a user-controlled privacy policy on top of the gpsOne hardware. The user is allowed to specify at what times of day a particular program can access location information, and whether the user is prompted to confirm the request. The privacy policy configuration process is driven via a dialog box (implemented and created by LBSDriver.dll) that is shown on the device the first time an application runs, and subsequently via a Verizon Wireless-operated web site[7]. Privacy policy settings are obfuscated and stored in the registry, keyed off of a hash of the calling program's main process image fully-qualified filename.

Because LBSDriver.dll is a standard, loadable DLL, it is vulnerable to being loaded by untrusted code. There are several defenses implemented by the LBSDriver module which attempt to deter third-party programs that have not been approved by Verizon Wireless from successfully loading LBSDriver.dll and subsequently using it to access location information.

The first such protection embedded into LBSDriver.dll is a digital signature check on the main process executable corresponding to any program that attempts to load LBSDriver.dll. This check is ultimately triggered when the GPSOpenDevice export on LBSDriver.dll is called. Specifically, the calling process module is confirmed to be signed by a custom certificate. If this is not the case, then an error dialog is shown, and the GPSOpenDevice request is denied. This check is based on calling GetModuleFileName(NULL, ...)[8] to retrieve the path to the main process image, which is then run through the aforementioned signature check.

Additionally, LBSDriver.dll also connects to an Autodesk-operated server in order to determine if the calling program is authorized to use LBSDriver.dll. In addition to verifying that the calling program is approved as a GPS-enabled application, the Autodesk-operated server also appears to indicate back to the client whether or not the user's account has been provisioned for a subscription location-enabled application, such as VZ Navigator. A program hoping to utilize LBSDriver.dll must pass these checks in order to successfully acquire a location fix using the built-in gpsOne hardware.

The Autodesk-operated server also provides configuration information (such as Position Determining Entity (PDE) addresses) that is later used in the assisted GPS process. However, this configuration information appears to be more or less static, at least for the critical portions necessary to enable assisted GPS, and can thus be cached and reused by third-party programs without even needing to go through the Autodesk server.

4 Opening gpsOne on the xv6800 to Third-party Applications

Understanding the protection mechanisms that implement the locking down of the built-in GPS hardware is only part of the battle to enable third-party GPS-enabled programs to operate on the xv6800. Undocumented functions in oemgpsOne.dll with no equivalent in the standard Windows Mobile gpsapi.dll, and various quirks of Windows Mobile itself preclude a straightforward implementation to unlock the GPS for third-party programs.

Furthermore, third-party GPS-enabled programs are written to one (or commonly, both) of the standard Windows Mobile GPS interfaces. Because these interfaces are disabled on the xv6800, a solution to adapt third-party programs to the locked down GPS interface would be required (in lieu of modifying every single third-party application to support the locked down GPS interface). As many of these third-party applications are closed-source and frequently updated, any solution that required direct modification of a third-party program would be untenable from a maintenance perspective.

The solution chosen was to write an emulation layer for the standard Windows Mobile gpsapi.dll interface, which translates standard gpsapi.dll function calls into requests compatible with the locked down GPS interface.

4.1 Examining gpsOne Driver Interactions

The first step in implementing a layer to unlock the gpsOne hardware on the xv6800 involves discovering the correct sequence of oemgpsOne.dll calls (and thus calls to the GPS intermediate driver, as oemgpsOne.dll is merely a thin wrapper around IOCTL requests to the GPS intermediate driver, for the most part, with some minor exceptions).

The standard way that this would be done on a Windows-based system would be to run VZ Navigator under a debugger, but there exist several complications that prevent this from being an acceptable solution for monitoring oemgpsOne.dll requests.

First, the assisted GPS functionality of the gpsOne hardware requires that the device be connected to the cellular network, and operating with it as the default gateway, as a connection to a carrier-supplied server (known as a "Position Determining Entity", or PDE) must be made. The PDE servers that are operated by Verizon Wireless are firewalled off from outside their network, and in addition, it is possible that they use the IP address assigned to the user making a request for location assistance purposes.

Unfortunately, the debugger connection to a Windows Mobile-based device, for all the Windows Mobile debuggers that the author had access to (IDA Pro 5.1 and the Visual Studio 2005 debugger) require an ActiveSync link. While the ActiveSync link is enabled, it supersedes the cellular link for data traffic. Even when the computer on the other end of the ActiveSync link was connected to the cellular network via a separate cellular modem, the GPS functionality did not operate, due to an apparent check of whether the cellular link is the most-precedent data link on the device.

This means that observing much of the oemgpsOne.dll calls relating to position fixes would not be possible with the standard debugging tools available. The solution that was implemented for this problem was to write a proxy DLL that exports every symbol exported by oemgpsOne.dll, logs the parameters of any such API calls, and then forwards them on to the underlying oemgpsOne.dll implementation (logging return values and out parameters after the actual implementation function in question returned).

While potentially labor-intensive, in terms of creating the proxy DLL, such a technique is relatively simple on Windows. The usual procedure for such a task would be to create the proxy DLL, place it in the directory containing the main process image of the program to be hooked, and then load the real DLL with a

fully-qualified path name from inside the proxy DLL.

Unfortunately, Windows Mobile does not allow two DLLs with the same base name to be loaded, even if a fully-qualified path is specified with a call to LoadLibrary. Instead, the first DLL that happened to get loaded by any process on the entire system matching the requested base name is returned. This means that in order to load a proxy DLL, one of two approaches would need to be taken.

The first such option is to rename the the proxy DLL itself, along with the filename of the imported DLL in the desired target module, by modifying the actual desired target module itself on-disk. The second option is to rename the DLL containing the implementation of the proxied functionality, and then load that DLL by the altered name in the proxy DLL. Both approaches are functionally equivalent on Windows Mobile; the author chose the former in this case.

Through disassembly, a rough estimate of the prototypes of the various APIs exported by oemgpsOne.dll was created, and from there, a proxy module (oemgpsOneProxy.dll) was written to log specific API calls to a file for later analysis. This approach allowed for relatively quick identification of any arguments to oemgpsOne.dll calls which were not immediately obvious from static disassembly, despite the lack of a debugger on the target when many of the calls were made.

4.2 Implementing a Custom oemgpsOne.dll client

After discerning the prototypes for the various oemgpsOne.dll supporting APIs, the next step in unlocking the built-in GPS hardware on the xv6800 was to write a custom client program that utilized oemgpsOne.dll to retrieve decrypted location values from the gpsOne chipset.

Although one approach to this task might be to attempt to disable the various security checks present in LBSDriver.dll, it was deemed easier to re-implement an oemgpsOne.dll client from scratch. In addition, this approach also allowed the author to circumvent various implementation bugs and limitations present in LBSDriver.dll.

Given the information gleaned from analyzing LBSDriver.dll's implementation of the challenge-response and GPS decryption logic, and the API call logging from the oemgpsOne.dll proxy module, writing a client for oemgpsOne.dll is merely an exercise in writing the necessary code to connect all of the pieces together in the correct fashion.

After valid GPS position data can be retrieved from oemgpsOne.dll, all that remains is to write an adapter layer to connect programs written against the standard Windows Mobile gpsapi.dll to the custom oemgpsOne.dll client.

However, there are inherent design limitations in the locked down GPS interface that complicate the creation of a practical adapter to convert gpsapi.dll calls into oemgpsOne.dll calls. For example, a naive implementation that might involve creating a module to replace gpsapi.dll with a custom binary to make inline calls to oemgpsOne.dll would run aground of a number of pitfalls.

Specifically, as oemgpsOne.dll depends on gpsapi.dll, attempting to simply replace gpsapi.dll with a custom module will break the very oemgpsOne.dll functionality used to communicate with the GPS intermediate driver, due to the previously mentioned "one dll for a given base name" Windows Mobile limitation. In addition, it is not possible for two programs to simply simultaneously operate full clients of oemgpsOne.dll, as the challenge-response mechanism operates globally and will not operate correctly should two applications simultaneously attempt to engage it.

The most straightforward solution to the former issue is to simply rename a copy of the stock gpsapi.dll, and then modify oemgpsOne.dll to refer to the renamed gpsapi.dll. This opens the door to replacing the system-supplied gpsapi.dll with a custom replacement gpsapi.dll implementing a client for oemgpsOne.dll.

4.3 Multiplexing GPS Across Multiple Applications

The GPS intermediate driver supports multiplexing the GPS hardware present on a Windows Mobile-based device across multiple applications. However, as previously noted, the locked down GPS interface breaks this functionality, as no two programs can participate in the full challenge-response protocol for keeping the gpsOne hardware active simultaneously.

Although the first program to start could be designated the "master", and thus be responsible for challenge-response operations (with secondary programs merely decrypting position data locally), this introduces a great deal of extra complexity. Specifically, significant coordination issues arise relating to cleanly handling the fact that third-party GPS-enabled programs are typically unaware of each other. Thus, work must be done to handle the case where one program having previously activated the gpsOne hardware exits, leaving any remaining programs still using GPS with the problem of selecting a new "master" program to perform challenge-responses with the GPS intermediate driver.

Given the difficulties of such an approach, a different model was chosen, such that the replacement gpsapi.dll acts as a client of a server program which then mediates access to the locked down GPS interface on behalf of all active GPS-enabled programs. Although there exist synchronization and coordination issues with this model, they are simpler to deal with than the alternative implementation.

4.4 Caveats

While the resultant GPS adapter system supports third-party programs that utilize gpsapi.dll, any programs using the virtual NMEA serial port interface will not operate successfully. Unfortunately, the same approach towards the replacement of gpsapi.dll is not feasible with the APIs utilized in communication with a serial port, by virtue of the sheer number of function calls present in coredll.dll that would need to be forwarded on to the real coredll.dll via a proxy module.

5 Bugs in the Verizon Wireless xv6800 gpsOne Lock Down Logic

Few programs designed to lockdown portions of a system via security through obscurity are bug-free, and the GPS lockdown logic on the xv6800 is certainly no exception. The lockdown code has a number of localized and systemic issues pervading the current implementation.

5.1 Thread Safety Issues

There are a number of threading related issues present throughout the locked down GPS interface.

- The GPS intermediate driver does not properly synchronize the case of multiple simultaneous callers using the extended IOCTLs not present on a stock GPS intermediate driver implementation.
- LBSDriver.dll utilizes a dedicated thread for performing challenge-response processing with the GPS intermediate driver. However, there is no synchronization provided between the challenge-response thread and the thread that retrieves and decrypts GPS position data, leading to a race condition in which it might be possible for decryption to return garbage data.

5.2 API Mis-use

In several cases, LBSDriver.dll fails to use standard Windows APIs correctly.

- LBSDriver.dll performs dangerous operations in DllMain, such as loading other DLLs, despite such operations being long-documented as blatantly illegal and prone to difficult to diagnose deadlocks (particularly on a device with extremely limited debugging support).

- When LBSDriver.dll performs the AES decryption on the latitude/longitude values returned by oemgpsOne.dll, it creates a CryptoAPI key blob, in order to import the derived AES key into a CryptoAPI key object (via the use of the CryptImportKey routine). However, the length of the key blob passed to CryptImportKey is actually too short. This would appear to make LBSDriver.dll seemingly dependent on a bug in the Windows Mobile 6 implementation of CryptoAPI. Specifically, the key blob format for a symmetric key includes a count in bytes of key material, and the data passed to CryptImportKey is such that the key blob structure claims to extend beyond the length of bytes that LBSDriver.dll specifies for the key blob structure itself. It might even be the case that this represents a security problem in CryptoAPI due to apparently non-functional length checking in this case, as key blobs are documented to be transportable across an untrusted medium.

To illustrate second issue, consider the following code fragment:

```
//
// Initialize the header.
//

BlobHeader = (BLOBHEADER *)KeyBlob;

BlobHeader->bType    = PLAINTEXTKEYBLOB;
BlobHeader->bVersion = 2;
BlobHeader->reserved = 0;
BlobHeader->aiKeyAlg = CALG_AES_128;

//
// Initialize the key length in the BLOB payload.
//

*(DWORD *)(&KeyBlob[ 0x08 ] ) = KeyLength;

//
// Initialize the key material in the BLOB payload.
//

memcpy( KeyBlob + 0x0C, KeyData, KeyLength );

//
// Generate a CryptoAPI AES-128 key object from our key material.
//

if (!CryptImportKey(
   CryptProv,
   KeyBlob,
   KeyLength, // BUGBUG: Should really be KeyLength + 0x0C...
   NULL,
   0,
   &Key))
{
```

```
   break;
}
```

Contrary to the Microsoft-supplied documentation[9] for CryptImportKey, the third parameter passed to CryptImportKey ("dwDataLen", as "KeyLength" in this example) is too short for the key blob specified, as the length field in the blob header itself describes the key material as being "KeyLength" bytes. Thus, the LBSDriver.dll module would appear to depend upon either CryptoAPI or the default Microsoft cryptographic provider on Windows Mobile not validating blob header key material lengths properly, as the supplied blob header claims that the key material extends outside the provided blob buffer (given the length passed to CryptImportKey).

Microsoft-supplied sample code[10] illustrates the correct construction of a symmetric key blob, and does not suffer from this deficiency.

6 Suggested Countermeasures

Although several attempts were made throughout the GPS lockdown system on the xv6800 to deter third party programs from successfully communicating with the integrated gpsOne hardware, the bulk of these checks were relatively easy to overcome. In fact, the principle barriers to the GPS unlocking projects were a lack of viable debugging tools for the platform, and an unfamiliarity with Windows Mobile on the part of the author.

Nevertheless, several improvements could have been made to improve the resilience of the lockdown system.

- Deny assisted GPS availability at the PDE if the user's account is not provisioned for GPS, or if the privacy policy configured time of day restrictions are not met. Because the security and lockdown checks are implemented client-side on the xv6800, they are relatively easily bypassable by third party applications. However, if the device is capable of performing a standalone GPS location fix, blocking assisted GPS access will not provide a hard defense.
- Require code signing from a Verizon Wireless CA for all applications loaded on the device. Users are, however, unlikely to purchase a device configured in such a matter, as expensive smartphone-class devices are often sold under the expectation that third party programs will be easily loadable.
- Moving enforcement checks for operations such as time of day requirements for the user's desired location privacy policy into the radio firmware and

out of the operating system environment. The radio firmware environment is significantly closer to a "black box" than the operating system which runs on the application core of the xv6800. Furthermore, if the software loader on the xv6800 were secured and locked down, the radio firmware could be made significantly more proof against unauthorized modifications. One could envision a system wherein the radio firmware communicates with the carrier's network out-of-band (with respect to the general-purpose operating system loaded on the device) to determine when it had been authorized by the user to provide location information to applications running on the device.

The client-side checks on the GPS lockdown system are likely a heritage of the fact that VZ Navigator and LBSDriver.dll appear to be more or less ports from BREW-based "dumb phones", where the application environment is more tightly controlled by code signing requirements. The Windows Mobile operating environment is significantly different in this respect, however.

Additionally, the author would submit that, from the perspective of attempting to safeguard users from unauthorized harvesting of their location data (a key reason cited by Verizon Wireless with respect to the certification process needed for an application to become approved for location-aware functionality), a hardware switch to enable or disable the GPS hardware on the device would be a far better investment. In fact, the xv6800 already possesses a hardware switch for 802.11 functionality; if this was instead changed to enable or disable the gpsOne chipset in future smartphone designs, users could be assured that their location information would be truly secure.

7 Debugging and Development Challenges on Windows Mobile and the xv6800

Windows Mobile has a severely reduced set of standard debugging tools as compared to the typically highly rich debugging environment available on most Windows-derived systems. This greatly complicated the process of understanding the underlying implementation details of the GPS lockdown system.

The author had access to two debuggers that could be used on the xv6800 at the time of this writing: the Visual Studio 2005 debugger, and the IDA Pro 5.1 debugger. Both programs have serious issues in and of their own respective rights.

Unfortunately, there does not appear to be any support for WinDbg, the author's preferred debugging tool, when using Windows CE-based systems, such as Windows Mobile. Although WinDbg can open ARM dump files (and ARM PE images as a dump file), and can disassemble ARM instructions, there is no

transport to connect it to a live process on an ARM system.

The relatively immature state of debugging tools for the Windows Mobile platform was a significant time consumer in the undertaking of this project.

7.1 Limitations of the Visual Studio Debugger

Visual Studio 2005 has integrated support for debugging Windows Mobile-based applications. However, this support is riddled with bugs, and the quality of the debugging experience rapidly diminishes if one does not have symbols and binaries for all images in the process being debugged present on the debugger machine. In particular, the Visual Studio 2005 debugger seems to be unable to disassemble at any location other than the current pc register value without having symbols for the containing binary available. (In the author's experience, attempting such a feat will fail with a complaint that no code exists at the desired address.)

Additionally, there seems to be no support for export symbols on the Windows Mobile debugger component of Visual Studio 2005. This, coupled with the lack of freely-targetable disassembly support, often made it difficult to identify standard API calls from the debugger. The author recommends falling back to static disassembly whenever possible, as available static disassembly tools, such as IDA Pro 5.1 Advanced or WinDbg provide a superior user experience.

7.2 Limitations of the IDA Pro 5.1 Debugger

Although IDA Pro 5.1 supports debugging of Windows Mobile-based programs, the debugger has several limitations that made it unfortunately less practical than the Visual Studio 2005 debugger. Foremost, it would appear that the debugger does not support suspending and breaking into a Windows Mobile target without the Windows Mobile target voluntarily breaking in (such as by hitting a previously defined breakpoint).

In addition, the default security policy configuration on the device needed to be modified in order to enable the debugger to connect at all (see note[3]).

7.3 Replacing a Firmware-baked Execute-in-place Module

Windows Mobile supports the concept of an execute in place (or XIP) module. Such an executable image is stored split up into PE subsections on disk (and does not contain a full image header). XIP modules are "baked" into the firmware image, and cannot be overwritten without flashing the OS firmware on the

device. Conversely, it is not possible to simply copy an XIP module off of the device and on to a conventional storage medium.

The advantage of XIP "baked" modules comes into play when one considers the limited amount of RAM available on a typical Windows Mobile device. XIP modules are pre-relocated to a guaranteed available base address, and do not require any runtime alterations to their backing memory when mapped. As a result, XIP modules can be backed entirely by ROM and not RAM, decreasing the (scarce) RAM that must be devoted to holding executable code.

It is possible to supersede an XIP "baked" module without flashing the OS image on the xv6800, however. This involves a rather convoluted procedure, which amounts to the following steps, for a given XIP module residing in a particular directory:

- First, rename the replacement module such that it has a filename which does not conflict with any files present in the directory containing the XIP module to supersede.
- Next, copy the renamed replacement module into the directory containing the desired XIP module to supersede.
- Finally, rename the replacement module to have the same filename as the desired XIP module.

Deleting the filename associated with the superseded XIP module will revert the device back to the ROM-supplied XIP module. This property proves beneficial in that it becomes easy to revert back to stock operating system-supplied modules after temporarily superseding them.

7.4 Import Address Table Hooking Limitations

One avenue considered during the development of the replacement gpsapi.dll module was to hook the import address tables (IATs) of programs utilizing gpsapi.dll.

Unfortunately, import table hooking is a significantly more complicated affair on Windows Mobile-based platforms than on standard Windows. The image headers for a loaded image are discarded after the image has been mapped, and the IAT itself is often relocated to be non-contiguous with the rest of the image.

This relocation is possible as there appears to be an implicit restriction that all references to an IAT address on ARM PE images must indirect through a global variable that contains the absolute address of the desired IAT address. As a result, there are no relative references to the IAT, and thus absolute address references may be fixed up via the aid of relocation information. It is not clear

to the author what the purpose for this relocation of the IAT outside the normal image confines serves on Windows Mobile for non-XIP modules that are loaded into device RAM.

Furthermore, the HMODULE of an image does not equate to its load base address on Windows Mobile. One can retrieve the real load base address of a module on Windows Mobile via the GetModuleInformation API. This is a significant departure from standard Windows.

Due to these limitations, the author elected not to pursue IAT hooking for the purposes of the GPS unlocking project. Although there is code publicly available to cope with the relocation of an image's IAT, it appears to be dependent on kernel data structures that the author did not have a conveniently available and accurate definition for these structures corresponding to the Windows Mobile kernel shipping on the xv6800.

8 Conclusion

Locking down the gpsOne hardware on the xv6800 such that it can only be utilized by Verizon Wireless certified and approved applications can be seen in two lights. One could consider such actions an anti-competitive move, designed to lock out third party programs from having the opportunity to compete with VZ Navigator. However, such a reasoning is fairly questionable, given that other carriers in the United States (particularly GSM-based carriers) typically fully support third party GPS-enabled applications on their devices. As consumers expect more full-featured and advanced devices, locking down devices to only carrier-approved functionality is becoming an increasingly large competitive liability for companies seeking to differentiate their networks and devices in today's saturated mobile phone markets.

Furthermore, Verizon Wireless's currently shipping location-enabled application for the xv6800, VZ Navigator, remains competitive (by virtue of features such as turn-by-turn voice navigation, traffic awareness, and automatic re-routing) even if the built-in GPS hardware on the xv6800 were to be unlocked for general-purpose use. Freely available navigation programs lack these features, and commercial applications are based off of a different pricing model than the periodic monthly fee model used by VZ Navigator at the time of this article's writing.

A more reasonable (although perhaps misguided) rationale for locking down the gpsOne hardware is to protect users from having their location harvested or tracked by malicious programs. Unfortunately, the relatively open nature of Windows Mobile 6, and a lack of particularly effective privilege-level isolation on Windows Mobile 6 after any unsigned code is permitted to run both conspire to greatly diminish the effectiveness of the protection schemes that are implemented on the xv6800.

Whether this is a legitimate concern or not remains, of course, up for debate, but it is clear that the lockdown system as present on the xv6800 is not particularly effective against blocking access to un-approved third party applications.

Future releases of Windows Mobile claim support for a much more effective privilege isolation model that may provide true security from unprivileged, malicious programs. However, in currently shipping devices, the operating system cannot be relied upon to provide this protection. Relying on security through obscurity to implement lockdown and protection schemes may then seem attractive, but such mechanisms rarely provide true security.

As mobile phone advance to becoming more and more powerful devices, in effect becoming small general-purpose computers, privacy and security concerns begin to gain greater relevance. With the capability to record a user's location and audio and environment (via built-in microphones and cameras present on virtually all modern-day phones), there arises the chance for a serious privacy breeches, especially given modern day smartphones have historically not seen the more vigorous level of security review that is slowly becoming more commonplace on general purpose computers.

One simple and elegant potential solution to these privacy risks is to simply provide hardware switches to disable sensitive components, such as cameras or embedded GPS hardware. Keeping in mind with this philosophy, the author would encourage Verizon Wireless to fully open up their devices, and defer to simple and secure methods to allow users to manage their sensitive information, such as physical hardware switches.

References

[1] Verizon Wireless. Commercial Location Based Services. http://www.vzwdevelopers.com/aims/public/menu/lbs/LBSLanding.jsp; accessed October 10, 2008

[2] Verizon Wireless. LBS Application Questions ("What can I do to ensure that my application is accepted, and to ensure a smooth certification process?"). http://www.vzwdevelopers.com/aims/public/menu/lbs/LBSFAQ.jsp#LBSAppQues7; accessed October 10, 2008

[3] Daniel lvarez. Debugging Windows Mobile 6 Applications with IDA. http://dani.foroselectronica.es/debugging-windows-mobile-6-applications-with-ida-69/; accessed October 10, 2008

[4] Microsoft. GPS Intermediate Driver Reference. http://msdn.microsoft.com/en-us/library/ms850332.aspx; accessed October 10, 2008

[5] Microsoft. GPSOpenDevice. http://msdn.microsoft.com/en-us/library/bb202113.aspx; accessed October 10, 2008

[6] Microsoft. GPSGetPosition. http://msdn.microsoft.com/en-us/library/bb202050.aspx; accessed October 10, 2008

[7] Verizon Wireless. LBS Application Questions ("Can the user change their privacy settings?"). http://www.vzwdevelopers.com/aims/public/menu/lbs/LBSFAQ.jsp#GenQues16; accessed October 10, 2008

[8] Microsoft. GetModuleFileName Function (Windows). http://msdn.microsoft.com/en-us/library/ms683197(VS.85).aspx; accessed October 10, 2008

[9] Microsoft. CryptImportKey Function (Windows). http://msdn.microsoft.com/en-us/library/aa380207(VS.85).aspx; accessed October 11, 2008

[10] Microsoft. Example C program: Imprtoing a Plaintext Key (Windows). http://msdn.microsoft.com/en-us/library/aa382383(VS.85).aspx; accessed October 11, 2008